PUIGFLOWS

Guia tecnica - Nivel intermedio - 47 min de lectura

Bloqueos Distribuidos y Elección de Líder: Fencing Tokens, Leases de Kubernetes, Advisory Locks de PostgreSQL y por qué Redlock no basta para la corrección

Guía práctica del mecanismo que casi todo el mundo implementa mal: por qué todo lock con TTL tiene una ventana en la que dos procesos se creen dueños (pausas de GC, throttling de cgroups, particiones de red), la pregunta que hay que responder antes de elegir herramienta -¿bloqueas por eficiencia o por corrección?-, un lock en Redis bien hecho con SET NX PX y liberación CAS en Lua, la controversia de Redlock y qué compra de verdad, advisory locks de PostgreSQL a nivel de transacción (y por qué los de sesión se rompen con PgBouncer en modo transaction), colas con FOR UPDATE SKIP LOCKED, elección de líder con Leases de Kubernetes y etcd -incluido el hecho de que client-go no hace fencing y el campo leaseTransitions que sí sirve de token-, fencing de punta a punta con UPDATE ... WHERE fence_token < $1, y el capítulo más rentable: los cinco diseños que eliminan la necesidad del lock. Con códi

Contenido

Version en espanol

English version

puigflows.com - Publicado el 18 de septiembre de 2026 - Edicion ampliada. Distributed Locks and Leader Election: Fencing Tokens, Kubernetes Leases, PostgreSQL Advisory Locks, and Why Redlock Isn't Enough for Correctness

Version en espanol

Un equipo con el que trabajé tenía un proceso nocturno de facturación desplegado en tres réplicas. Sólo una debía ejecutarlo, así que pusieron un lock en Redis: SET con NX, un TTL de treinta segundos y un DEL al terminar. Funcionó durante catorce meses. La noche número cuatrocientos veintitrés, un cliente nuevo multiplicó por seis el volumen del lote, el proceso se comió la memoria, el recolector de basura hizo una pausa de 1,4 segundos, el TTL venció, otra réplica cogió el lock, y a las 03:11 se emitieron 812 facturas duplicadas.

El código del lock era correcto. Ese es el punto incómodo de este artículo: la mayoría de los bugs de exclusión mutua distribuida no están en el código del lock, están en la suposición de que un lock con TTL garantiza exclusión mutua. No la garantiza. Nunca la ha garantizado. Lo que sigue es el mapa completo: dónde está la ventana de fallo, cuándo puedes ignorarla, cómo se cierra de verdad cuando no puedes, y -el capítulo que más dinero ahorra- cómo diseñar el sistema para no necesitar el lock.

Antes de elegir herramienta: ¿eficiencia o corrección?

Martin Kleppmann planteó esta distinción en 2016 y sigue siendo la pregunta más útil que puedes hacerte antes de escribir una línea de código.

Bloqueas por eficiencia cuando una ejecución duplicada sólo desperdicia trabajo. Dos réplicas regeneran la misma miniatura, dos workers recalculan el mismo agregado, dos nodos refrescan la misma caché. El resultado final es idéntico; sólo has pagado CPU de más. Aquí un lock probabilístico es perfectamente razonable: una instancia de Redis, un TTL, y si una vez cada mil veces se cuela un doble trabajo, no pasa nada.

Bloqueas por corrección cuando una ejecución duplicada corrompe datos: cobrar dos veces, decrementar dos veces el mismo inventario, emitir dos facturas con el mismo número, escribir dos versiones incompatibles del mismo fichero. Aquí "casi siempre exclusivo" no vale nada. Un lock que falla una vez cada diez mil ejecuciones y que corre cada cinco minutos te da un incidente cada mes.

> *La regla práctica: si puedes escribir en un runbook "si esto se ejecuta dos veces, reintenta y ya está", es eficiencia. Si tienes que escribir "si esto se ejecuta dos veces, abre un ticket a finanzas", es corrección, y necesitas fencing.*

Por qué todo lock con TTL tiene una ventana de dos dueños

La cadena de razonamiento es corta y no tiene escapatoria. Un lock con TTL caduca por tiempo. Para que caducar por tiempo sea seguro, el proceso que lo tiene debe garantizar que termina -o al menos que deja de escribir- antes de que el TTL venza. Ningún proceso puede garantizar eso, porque ningún proceso controla cuándo se le retira la CPU.

Esta es la secuencia exacta del incidente de las facturas:

```
t=0.00  Worker A: SET lock:billing <token-a> NX PX 30000  -> OK
t=0.10  Worker A: empieza a procesar el lote
t=12.40 Worker A: pausa stop-the-world del GC (1.4 s)
t=13.80 Worker A: reanuda... pero el pod estaba en un nodo con CPU
```

```
                  throttling; el kernel no le devuelve la CPU
t=30.00 Redis:    el TTL vence, la clave desaparece
t=30.05 Worker B: SET lock:billing <token-b> NX PX 30000  -> OK
t=30.10 Worker B: empieza a procesar EL MISMO lote
t=31.20 Worker A: vuelve a ejecutarse, cree que sigue siendo el dueño,
                  y escribe en la base de datos
        ^^^ dos escritores simultáneos
```

Las causas de la pausa son más variadas de lo que la gente supone, y casi ninguna aparece en los tests:

- Pausas de GC stop-the-world. La JVM con heaps grandes, Go con un ciclo patológico, Python recolectando ciclos sobre millones de objetos.
- Throttling de cgroups. El caso más frecuente en Kubernetes y el que menos se sospecha: si el contenedor supera su cuota de CPU, el kernel lo congela hasta el siguiente periodo de 100 ms. Con un límite mal puesto acumulas segundos de congelación sin que ninguna métrica de la aplicación lo refleje.
- Fallos de página y swap. El proceso está "ejecutándose" mientras espera al disco.
- Migración en vivo de la VM o suspensión del hipervisor.
- Partición de red. El proceso sigue corriendo perfectamente; lo que no puede es hablar con el servicio de locks. Desde fuera es indistinguible de una caída, y esa es exactamente la razón por la que el TTL existe.
- Una llamada bloqueante sin timeout. Un PUT a S3 que tarda cuarenta segundos porque nadie configuró el timeout del cliente.

No puedes eliminar la ventana. Puedes hacerla improbable -TTL generoso, heartbeat, timeouts en todo- pero improbable no es imposible, y a suficiente escala improbable ocurre los jueves. La única salida real es que la capa de almacenamiento rechace las escrituras del dueño caducado.

Fencing tokens: la única defensa que cierra la ventana

La idea cabe en tres frases. Cada vez que el servicio de locks concede el lock, devuelve un entero estrictamente creciente: el fencing token. El titular incluye ese token en cada escritura. El almacenamiento guarda el token más alto que ha visto para ese recurso y rechaza cualquier escritura con un token menor o igual.

Con eso, el worker A del ejemplo vuelve del GC con el token 41, intenta escribir, y la base de datos -que ya vio el token 42 de B- lo rechaza. A no necesita saber que perdió el lock. El almacenamiento lo sabe por él.

```
Worker A obtiene el lock  -> token 41
Worker A se congela
Worker B obtiene el lock  -> token 42
Worker B escribe (42)     -> aceptado, fence_token pasa a 42
Worker A escribe (41)     -> 41 no supera a 42  -> RECHAZADO
```

En PostgreSQL el mecanismo es una columna y una condición en el WHERE:

```
ALTER TABLE cuentas ADD COLUMN fence_token bigint NOT NULL DEFAULT 0;

-- La escritura protegida. Si devuelve 0 filas, eres un líder caducado.
UPDATE cuentas
```

```
   SET saldo_centimos = saldo_centimos - $2,
       fence_token    = $3
 WHERE id = $1
   AND fence_token < $3;
```

Fíjate en que no hay ningún lock en esa sentencia. El token convierte la exclusión mutua en una comprobación de datos, que es algo que la base de datos ya sabe hacer de forma atómica. Ese es el truco entero.

La objeción habitual es que esto obliga a tocar todas las escrituras. Es cierto, y es la razón por la que casi nadie lo implementa. También es la razón por la que casi todo el mundo tiene el bug. Si el trabajo protegido escribe en cinco tablas, necesitas el token en las cinco; si escribe en S3, necesitas el token en la condición de la escritura condicional; si llama a una API de terceros, necesitas que esa API acepte una clave de idempotencia derivada del token.

Nivel 1 - Redis: un lock decente, hecho bien

Para bloqueo por eficiencia, una sola instancia de Redis es la respuesta correcta el 90% de las veces. Lo importante es no cometer los dos errores clásicos: adquirir sin TTL (si el proceso muere, el lock queda para siempre) y liberar con DEL a secas (borras el lock de otro cuando el tuyo ya había caducado).

La liberación tiene que ser un compare-and-delete atómico, y eso en Redis significa Lua:

```
-- release.lua: borra la clave sólo si sigue siendo mía
if redis.call('GET', KEYS[1]) == ARGV[1] then
  return redis.call('DEL', KEYS[1])
end
return 0
```

Y la renovación, igual: extiende el TTL sólo si sigues siendo el dueño.

```
-- extend.lua: renueva el TTL sólo si sigue siendo mío
if redis.call('GET', KEYS[1]) == ARGV[1] then
  return redis.call('PEXPIRE', KEYS[1], ARGV[2])
end
return 0
```

El cliente completo en Python, con heartbeat en un hilo aparte y un evento que avisa al trabajo si perdemos el lock:

```
import contextlib
import logging
import secrets
import threading
import redis

log = logging.getLogger(__name__)
r = redis.Redis(host="redis", socket_timeout=2, socket_connect_timeout=2)

_RELEASE = r.register_script(open("release.lua").read())
_EXTEND  = r.register_script(open("extend.lua").read())


@contextlib.contextmanager
def redis_lock(key: str, ttl_ms: int = 30_000):
```

```
    """Lock por EFICIENCIA. No lo uses donde una doble ejecución corrompa datos.

    Cede un threading.Event: si se activa, hemos perdido el lock y el
    trabajo debe abortar en cuanto pueda comprobarlo.
    """
    token = secrets.token_hex(16).encode()
    if not r.set(key, token, nx=True, px=ttl_ms):
        raise TimeoutError(f"lock ocupado: {key}")

    lost = threading.Event()
    stop = threading.Event()

    def heartbeat() -> None:
        # Renovamos a un tercio del TTL: tres intentos antes de caducar.
        interval = ttl_ms / 3000
        while not stop.wait(interval):
            try:
                if not _EXTEND(keys=[key], args=[token, ttl_ms]):
                    log.error("lock %s perdido: otro proceso lo tiene", key)
                    lost.set()
                    return
            except redis.RedisError:
                log.warning("renovación de %s fallida; reintentando", key)

    t = threading.Thread(target=heartbeat, daemon=True, name=f"hb:{key}")
    t.start()
    try:
        yield lost
    finally:
        stop.set()
        t.join(timeout=1)
        with contextlib.suppress(redis.RedisError):
            _RELEASE(keys=[key], args=[token])
```

Uso:

```
with redis_lock("lock:rebuild-search-index", ttl_ms=60_000) as lost:
    for lote in lotes():
        if lost.is_set():
            raise RuntimeError("lock perdido; abortando para no duplicar trabajo")
        procesar(lote)
```

Tres detalles que suelen faltar. El socket_timeout del cliente: sin él, una renovación puede quedarse colgada más que el propio TTL. El intervalo de heartbeat a un tercio del TTL, que te da tres oportunidades de renovar antes de perderlo, en vez de la única que te da renovar a la mitad. Y el evento lost: renovar no sirve de nada si el trabajo no comprueba nunca si sigue siendo el dueño.

Redlock y lo que compra de verdad

Redlock es el algoritmo que propone Redis para locks sobre N instancias independientes (habitualmente cinco): el cliente intenta adquirir en todas, se considera dueño si consigue la mayoría dentro de un presupuesto de tiempo, y descuenta el tiempo transcurrido de la validez restante del lock.

La crítica de Kleppmann, que sigue vigente diez años después, es que Redlock parece apto para corrección -tiene quórum, tiene mayoría, huele a consenso- y no lo es. El argumento en tres pasos:

1. La validez del lock depende de que los relojes de las cinco instancias avancen a un ritmo comparable. Un salto de reloj en una de ellas (NTP corrigiendo bruscamente, un administrador ajustando la hora, un reloj virtualizado que se desvía) puede hacer que expire antes de lo que el cliente calcula.

2. El descuento de tiempo transcurrido asume que el cliente puede medir cuánto tardó. Si el cliente se pausa entre el "adquirí" y el "calculo la validez", el cálculo ya es inválido.
3. Y sobre todo: ninguno de esos problemas desaparece con más réplicas, porque la ventana peligrosa está entre el titular y el almacenamiento, no entre el titular y el servicio de locks.

La conclusión práctica no es "Redlock es basura", es más aburrida: si bloqueas por eficiencia, cinco instancias de Redis no compran nada que no compre una, y sí cuestan cinco veces más en operación. Si bloqueas por corrección, Redlock tampoco basta, y lo que necesitas es un fencing token. Redis puede darte ese token -con un INCR sobre un contador dedicado-, pero entonces la garantía la está dando tu almacenamiento, no Redis.

```
-- acquire_fenced.lua
-- KEYS[1] = clave del lock, KEYS[2] = contador de fencing
-- ARGV[1] = identidad del dueño, ARGV[2] = TTL en ms
-- Devuelve el fencing token, o nil si el lock está ocupado.
if redis.call('SET', KEYS[1], ARGV[1], 'NX', 'PX', ARGV[2]) then
  return redis.call('INCR', KEYS[2])
end
return nil
```

Un apunte de durabilidad que casi nunca se menciona: con replicación asíncrona, un failover de Redis puede perder el SET que concedió el lock, y el nuevo primario lo entrega a otro cliente. El contador de INCR sufre exactamente lo mismo, lo que rompe la monotonía del token. Si vas a fencing en serio, el contador debe vivir en un almacén con consenso -etcd, o una secuencia de PostgreSQL- y no en la réplica que se acaba de promocionar.

Nivel 2 - PostgreSQL: advisory locks y SKIP LOCKED

Si ya tienes PostgreSQL -y lo tienes- probablemente no necesitas Redis para esto. Los advisory locks son locks sobre enteros arbitrarios que tú defines: no tocan el heap, no generan tuplas muertas, no crean entradas de MultiXact y se liberan solos.

Hay dos familias y elegir mal es el error número uno:

- Nivel de transacción (pg_try_advisory_xact_lock): se libera automáticamente al hacer COMMIT o ROLLBACK. Es lo que quieres el 95% de las veces.
- Nivel de sesión (pg_try_advisory_lock): vive hasta que lo sueltas explícitamente o se cierra la conexión. Si el worker muere sin cerrar limpio, el lock queda colgando hasta que el servidor detecta la conexión muerta, lo que con la configuración de TCP keepalive por defecto puede ser más de dos horas.

> *La trampa que se lleva más tardes: los advisory locks de sesión son incompatibles con PgBouncer en modo transaction. Con ese modo, la siguiente sentencia puede ir por otra conexión de servidor, así que adquieres el lock en una conexión y lo intentas soltar desde otra. El lock se queda puesto y el desbloqueo devuelve una advertencia que nadie lee. Si usas PgBouncer en modo transaction -y casi todo el mundo lo usa-, sólo los locks de transacción son seguros.*

La clave del lock es un bigint. Mucha gente usa hashtext(), que es una función interna sin garantías de estabilidad entre versiones mayores. Calcúlala en la aplicación y tendrás un valor reproducible y auditable:

```
import hashlib

def clave_lock(nombre: str) -> int:
    """bigint determinista y estable para un advisory lock de PostgreSQL."""
    digest = hashlib.blake2b(nombre.encode(), digest_size=8).digest()
    return int.from_bytes(digest, "big", signed=True)
```

Y el patrón completo con SQLAlchemy. Nótese que todo ocurre dentro de una única transacción: el lock protege exactamente el trabajo que se confirma con él.

```
from contextlib import contextmanager
from sqlalchemy import text
from sqlalchemy.orm import Session


@contextmanager
def advisory_xact_lock(session: Session, nombre: str):
    """Intenta el lock. Si otro lo tiene, lanza en vez de esperar.

    El lock se libera solo al terminar la transacción: no hay ruta de
    código, ni siquiera un kill -9, que lo deje colgado.
    """
    clave = clave_lock(nombre)
    obtenido = session.execute(
        text("SELECT pg_try_advisory_xact_lock(:k)"), {"k": clave}
    ).scalar_one()
    if not obtenido:
        raise TimeoutError(f"otro proceso tiene {nombre}")
    yield


with Session(engine) as s, s.begin():
    with advisory_xact_lock(s, "facturacion-nocturna"):
        generar_facturas(s)   # commit al salir del bloque; el lock cae con él
```

Si prefieres esperar en vez de rendirte, usa la variante bloqueante pg_advisory_xact_lock pero siempre con un lock_timeout puesto antes, o convertirás una contención en un cuelgue indefinido:

```
SET LOCAL lock_timeout = '5s';
SELECT pg_advisory_xact_lock(-6114512253460296197);
```

Cuando lo que quieres es una cola, no un lock

Un error de diseño muy común es usar un advisory lock global para "que sólo un worker procese la cola". Eso serializa el trabajo y desperdicia las réplicas. Si el trabajo es por elemento y no global, lo que necesitas es FOR UPDATE SKIP LOCKED, que deja que N workers cojan elementos distintos sin pisarse ni esperar:

```
WITH siguientes AS (
  SELECT id
    FROM trabajos
   WHERE estado = 'pendiente'
     AND ejecutar_tras <= now()
   ORDER BY prioridad DESC, ejecutar_tras
   LIMIT 10
   FOR UPDATE SKIP LOCKED
)
UPDATE trabajos t
   SET estado      = 'en_curso',
       iniciado_en = now(),
       intentos    = t.intentos + 1
```

```
  FROM siguientes s
 WHERE t.id = s.id
RETURNING t.id, t.payload;
```

Dos advertencias sobre esto. La primera: el trabajo real va fuera de esa transacción, no dentro; si lo metes dentro, una tarea de treinta segundos mantiene una transacción abierta treinta segundos, y las transacciones largas bloquean el autovacuum y engordan la tabla. La segunda: necesitas un reaper que devuelva a 'pendiente' las filas que llevan demasiado tiempo en 'en_curso', porque un worker que muere después del UPDATE deja el trabajo huérfano. Ese reaper es, literalmente, tu TTL.

Y el fencing token, si lo necesitas, sale gratis de una secuencia:

```
CREATE SEQUENCE fence_facturacion AS bigint;

-- En el arranque del trabajo, después de obtener el lock:
SELECT nextval('fence_facturacion');   -- monótono, duradero, con consenso
```

Nivel 3 - etcd y los Leases de Kubernetes

Cuando lo que necesitas no es un lock puntual sino un líder -un proceso que manda mientras esté vivo y cede cuando muere-, el mecanismo correcto es un lease sobre un almacén con consenso. En Kubernetes ya lo tienes: los objetos Lease del grupo coordination.k8s.io son la misma pieza que usan el kube-controller-manager y el scheduler para elegir su propio líder.

La implementación de referencia en Go, con client-go:

```
package main

import (
    "context"
    "os"
    "os/signal"
    "syscall"
    "time"

    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes"
    "k8s.io/client-go/rest"
    "k8s.io/client-go/tools/leaderelection"
    "k8s.io/client-go/tools/leaderelection/resourcelock"
    "k8s.io/klog/v2"
)

func main() {
    id := os.Getenv("POD_NAME") // valueFrom: fieldRef: metadata.name
    cfg, err := rest.InClusterConfig()
    if err != nil {
        klog.Fatal(err)
    }
    client := kubernetes.NewForConfigOrDie(cfg)

    lock := &resourcelock.LeaseLock{
        LeaseMeta: metav1.ObjectMeta{
            Name:      "planificador-facturacion",
            Namespace: os.Getenv("POD_NAMESPACE"),
        },
        Client:     client.CoordinationV1(),
        LockConfig: resourcelock.ResourceLockConfig{Identity: id},
    }
```

```
    ctx, cancel := signal.NotifyContext(
        context.Background(), syscall.SIGTERM, syscall.SIGINT)
    defer cancel()

    leaderelection.RunOrDie(ctx, leaderelection.LeaderElectionConfig{
        Lock: lock,
        // Invariante: RenewDeadline < LeaseDuration, y RetryPeriod
        // en torno a RenewDeadline/3. Si lo violas, client-go entra
        // en un ciclo de robo de liderazgo entre réplicas sanas.
        LeaseDuration:   15 * time.Second,
        RenewDeadline:   10 * time.Second,
        RetryPeriod:     2 * time.Second,
        ReleaseOnCancel: true, // cede el lease al recibir SIGTERM
        Callbacks: leaderelection.LeaderCallbacks{
            OnStartedLeading: func(ctx context.Context) {
                ejecutarPlanificador(ctx) // debe respetar ctx.Done()
            },
            OnStoppedLeading: func() {
                // Hemos perdido el lease. No sabemos si nuestro trabajo
                // sigue en vuelo, así que la única salida segura es morir
                // y dejar que el kubelet nos reinicie como seguidores.
                klog.Error("lease perdido; terminando el proceso")
                os.Exit(1)
            },
            OnNewLeader: func(identidad string) {
                if identidad != id {
                    klog.Infof("nuevo líder: %s", identidad)
                }
            },
        },
    })
}
```

Los permisos mínimos. Sin create la primera elección falla en un namespace limpio, y es el fallo de estreno más habitual:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: eleccion-lider
  namespace: facturacion
rules:
  - apiGroups: ["coordination.k8s.io"]
    resources: ["leases"]
    verbs: ["get", "list", "watch", "create", "update"]
```

Y aquí viene el detalle que casi nadie conoce: la elección de líder de client-go no hace fencing. Es explícito en su propia documentación. Si el líder se congela, deja vencer el lease y luego reanuda, seguirá creyendo que manda. OnStoppedLeading sólo se dispara cuando el proceso consigue ejecutarse y detecta la pérdida; si está congelado, no detecta nada. Por eso el os.Exit(1) del ejemplo no es pereza: es la manera más barata de garantizar que un líder degradado deja de escribir.

Si necesitas fencing de verdad, el propio objeto Lease te da el token: el campo spec.leaseTransitions es un entero que client-go incrementa sólo cuando el lease cambia de titular, no en cada renovación. Es exactamente la semántica de un fencing token.

```
# El token: incrementa en cada cambio de líder, no en cada renovación.
kubectl get lease planificador-facturacion -n facturacion \
  -o jsonpath='{.spec.leaseTransitions}'
```

En etcd directamente, el equivalente es la revisión de creación de la clave del mutex, que es globalmente monótona:

```
sess, err := concurrency.NewSession(cli, concurrency.WithTTL(10))
if err != nil {
    return err
}
defer sess.Close() // revoca el lease: el sucesor entra en milisegundos

mu := concurrency.NewMutex(sess, "/locks/reindexado")
if err := mu.TryLock(ctx); err != nil {
    if errors.Is(err, concurrency.ErrLocked) {
        return nil // otro lo tiene; nada que hacer en este ciclo
    }
    return err
}
defer mu.Unlock(context.WithoutCancel(ctx)) // desbloquea aunque ctx muera

fence := mu.Header().Revision // monótona en todo el clúster: úsala como token
```

Fencing de punta a punta: un ejemplo completo

Juntemos las piezas en algo que realmente puedas copiar. El escenario: un trabajo que ajusta saldos de cuentas, ejecutado por varias réplicas, donde una doble ejecución es un problema contable. El lock da coordinación; la base de datos da la garantía.

```
import logging
import secrets
import redis
from sqlalchemy import text
from sqlalchemy.orm import Session

log = logging.getLogger(__name__)
r = redis.Redis(host="redis", socket_timeout=2)

_ACQUIRE = r.register_script(open("acquire_fenced.lua").read())


class LiderCaducado(Exception):
    """Nuestro fencing token ya no es el más alto: alguien nos sustituyó."""


def adquirir_con_fence(clave: bytes, ttl_ms: int) -> tuple[bytes, int] | None:
    dueno = secrets.token_hex(16).encode()
    token = _ACQUIRE(keys=[clave, clave + b":fence"], args=[dueno, ttl_ms])
    return (dueno, int(token)) if token else None


def ajustar_saldo(s: Session, cuenta_id: int, delta: int, fence: int) -> None:
    """Escritura protegida. Rechazada si un líder posterior ya escribió."""
    filas = s.execute(
        text("""
            UPDATE cuentas
               SET saldo_centimos = saldo_centimos + :delta,
                   fence_token    = :fence
             WHERE id = :id
               AND fence_token < :fence
        """),
        {"delta": delta, "fence": fence, "id": cuenta_id},
    ).rowcount

    if filas == 0:
        # O la cuenta no existe, o nos han adelantado. Distinguirlo importa:
        # lo primero es un bug de datos, lo segundo es funcionamiento normal.
        actual = s.execute(
            text("SELECT fence_token FROM cuentas WHERE id = :id"),
            {"id": cuenta_id},
        ).scalar_one_or_none()
```

```
        if actual is None:
            raise LookupError(f"cuenta {cuenta_id} inexistente")
        raise LiderCaducado(f"nuestro token {fence} no supera a {actual}")


def ejecutar_ajustes(engine, ajustes: list[tuple[int, int]]) -> None:
    concedido = adquirir_con_fence(b"lock:ajustes", 30_000)
    if concedido is None:
        log.info("otra réplica tiene el lock; no hacemos nada")
        return

    _dueno, fence = concedido
    log.info("lock adquirido con fencing token %d", fence)

    try:
        with Session(engine) as s, s.begin():
            for cuenta_id, delta in ajustes:
                ajustar_saldo(s, cuenta_id, delta, fence)
    except LiderCaducado:
        # No es un error que haya que alertar: el sistema hizo su trabajo.
        # La transacción revierte y el líder vivo termina el lote.
        log.warning("líder caducado detectado; abortando sin escribir")
```

Lo que hace elegante a este diseño es que no hay ninguna comprobación de "¿sigo siendo el líder?" en la lógica de negocio. No hay carrera entre comprobar y escribir, porque la comprobación es la escritura. Ese es el criterio con el que juzgar cualquier implementación de fencing que te encuentres: si en algún punto aparece un if soy_lider(): escribir(), entre esas dos llamadas cabe una pausa de GC.

Un matiz sobre el token monótono por recurso frente a global. En el ejemplo el contador es global al lock y la columna fence_token es por cuenta. Funciona porque un token globalmente monótono también es monótono para cada cuenta individual. El efecto secundario es que una cuenta que no se toca en meses conserva un token viejo, lo cual es inofensivo. Lo que no puedes hacer es usar un contador distinto por lock y compartir la columna entre varios locks: los tokens de dos contadores no son comparables entre sí.

El mejor lock distribuido es el que no escribes

Este es el capítulo que debería leerse primero. En la práctica, más de la mitad de los locks distribuidos de un sistema típico se pueden eliminar cambiando el diseño, y cada uno que eliminas es una clase entera de incidentes que desaparece.

1. Restricciones de unicidad en vez de exclusión mutua

Si el objetivo del lock es "que no se cree dos veces", díselo a la base de datos y deja que ella lo imponga. Un índice único es exclusión mutua perfecta: sin TTL, sin relojes y sin ventana.

```
CREATE TABLE pagos (
  id                 bigserial PRIMARY KEY,
  clave_idempotencia text NOT NULL,
  pedido_id          bigint NOT NULL,
  importe_centimos   bigint NOT NULL,
  creado_en          timestamptz NOT NULL DEFAULT now()
);

CREATE UNIQUE INDEX pagos_idem ON pagos (clave_idempotencia);

-- El segundo intento no falla: no crea nada y devuelve cero filas,
-- momento en el que lees el pago original y lo devuelves tal cual.
INSERT INTO pagos (clave_idempotencia, pedido_id, importe_centimos)
```

```
VALUES ($1, $2, $3)
ON CONFLICT (clave_idempotencia) DO NOTHING
RETURNING id;
```

2. Concurrencia optimista en vez de pesimista

Lee la versión, calcula, y escribe condicionando a que la versión no haya cambiado. Si cambió, reintenta con el dato fresco. Cuesta un reintento ocasional y te ahorra toda la infraestructura de locks.

```
UPDATE inventario
   SET unidades = unidades - $2,
       version  = version + 1
 WHERE sku = $1
   AND version = $3
   AND unidades >= $2;
-- 0 filas: o alguien escribió antes que tú, o no hay stock suficiente.
-- Relee, decide, y reintenta con un tope de intentos.
```

3. Deja que la base de datos serialice

Muchos "necesito un lock" son en realidad "necesito que esta operación sea atómica", y eso ya lo hace una única sentencia. UPDATE saldos SET importe = importe - 10 WHERE id = 7 no necesita ningún lock externo: PostgreSQL bloquea la fila mientras dura la sentencia y las demás transacciones esperan. El lock externo sólo hace falta cuando entre la lectura y la escritura ocurre algo que la base de datos no ve.

4. Particiona por clave en vez de coordinar

Si los mensajes de un mismo agregado van siempre a la misma partición de Kafka, y cada partición la consume un único miembro del grupo, ya tienes exclusión mutua por agregado sin ningún lock. Es el mismo principio del sharding por hash de usuario: la coordinación se sustituye por enrutamiento.

5. Un único escritor por diseño

Un Deployment con replicas: 1 y strategy: Recreate, o un StatefulSet de una sola réplica, es la forma más barata de tener un único escritor. No es tolerante a fallos -durante el reinicio no hay nadie- pero para un trabajo nocturno que puede esperar dos minutos eso es una ventaja de operación, no un defecto. Ojo con RollingUpdate: deja una ventana en la que conviven el pod viejo y el nuevo, así que Recreate aquí es una decisión deliberada, no el valor por defecto.

Ocho errores que se repiten en todas las revisiones de código

1. Liberar con DEL en vez de compare-and-delete. Es el bug más extendido y el más silencioso: tu lock caducó, otro lo cogió, y tú le borras el suyo al terminar. A partir de ahí hay tres procesos convencidos de ser el dueño. Siempre Lua, siempre comparando el token.
2. Adquirir sin TTL "porque siempre liberamos en el finally". El finally no se ejecuta cuando el kernel mata el proceso por OOM, cuando el nodo se apaga, ni cuando alguien hace kill -9. Un lock sin caducidad es una avería que espera al peor momento.
3. Renovar el lock sin comprobar nunca si lo has perdido. El heartbeat te dice cuándo dejaste de ser el dueño, pero si el trabajo no consulta ese dato, la información no sirve de nada. Propaga la pérdida como una cancelación, no como un log.

4. Usar advisory locks de sesión detrás de PgBouncer en modo transaction. Adquieres en una conexión de servidor y liberas -crees- en otra. El lock se queda puesto hasta que la conexión muere. Usa siempre la variante _xact.
5. Elegir el TTL por intuición. El TTL debe salir del percentil 99 del tiempo de ejecución medido, con margen, y hay que revisarlo cuando el trabajo crece. El incidente de las facturas del principio fue exactamente esto: un TTL calibrado para un volumen que ya no existía.
6. Violar las invariantes de la elección de líder. Con RenewDeadline mayor o igual que LeaseDuration, o un RetryPeriod demasiado grande, client-go entra en un baile de liderazgo entre réplicas perfectamente sanas. La proporción sana es 15/10/2 segundos, y no se toca sin medir.
7. Tratar OnStoppedLeading como un log de aviso. Perder el lease significa que otro está trabajando ahora mismo. Si tu callback registra un mensaje y sigue, tienes dos líderes por diseño. Cancela el contexto, espera a que el trabajo pare, y si no puedes garantizar que paró, termina el proceso.
8. Bloquear alrededor de una llamada a un tercero. Un lock no puede proteger un efecto que ocurre fuera de tu sistema: si la pasarela de pago recibió la petición, ya cobró, aunque tú pierdas el lock un milisegundo después. Ahí la herramienta es la clave de idempotencia del proveedor, no el lock.

Observabilidad: qué medir para enterarte antes del incidente

Un lock distribuido mal instrumentado falla en silencio durante meses. Estas cuatro métricas detectan el problema antes de que lo detecte un cliente.

```
from prometheus_client import Counter, Histogram

LOCK_ESPERA = Histogram(
    "lock_adquisicion_segundos", "Tiempo hasta obtener el lock",
    ["nombre"], buckets=(0.001, 0.01, 0.1, 1, 5, 30),
)
LOCK_RETENCION = Histogram(
    "lock_retencion_segundos", "Tiempo que se mantuvo el lock",
    ["nombre"], buckets=(0.1, 1, 5, 15, 30, 60, 300),
)
LOCK_PERDIDO = Counter(
    "lock_perdido_total", "Veces que se perdió el lock mientras se trabajaba",
    ["nombre"],
)
FENCE_RECHAZOS = Counter(
    "fence_token_rechazado_total", "Escrituras rechazadas por token caducado",
    ["recurso"],
)
```

La métrica que de verdad importa es la relación entre lock_retencion_segundos y el TTL. Si el percentil 99 de retención se acerca al TTL, no tienes un problema potencial: tienes un incidente programado. La alerta correspondiente, con un TTL de 30 segundos:

```
groups:
  - name: locks-distribuidos
    rules:
      - alert: RetencionDeLockCercaDelTTL
        expr: |
          histogram_quantile(
            0.99,
            sum by (nombre, le) (rate(lock_retencion_segundos_bucket[30m]))
          ) > 0.6 * 30
        for: 15m
        labels:
          severity: warning
```

```
        annotations:
          summary: "El lock se retiene cerca de su TTL"
          description: >-
            El p99 de retención supera el 60% del TTL. Sube el TTL o
            reduce el trabajo protegido antes de que aparezcan dos dueños.

      - alert: EscriturasRechazadasPorFencing
        expr: increase(fence_token_rechazado_total[10m]) > 0
        labels:
          severity: critical
        annotations:
          summary: "Un líder caducado intentó escribir"
          description: >-
            El fencing hizo su trabajo, pero esto confirma que la ventana
            de dos dueños es real en este sistema. Investiga la pausa.
```

Y un detalle de logging que ahorra horas: registra el fencing token en cada línea de log del trabajo protegido. Cuando aparezca una escritura sospechosa, el token te dice inmediatamente qué instancia del líder la produjo y si era la vigente.

Checklist de producción

- Está escrito, en algún sitio, si este lock es por eficiencia o por corrección. Si es por corrección, hay fencing token y la capa de almacenamiento lo comprueba.
- El TTL sale del p99 medido del trabajo protegido, por al menos tres, y hay una alerta que avisa cuando la retención se acerca al TTL.
- La liberación es compare-and-delete atómico, nunca un borrado incondicional.
- El heartbeat renueva a un tercio del TTL y su fallo se propaga al trabajo como cancelación.
- El cliente del servicio de locks tiene timeouts de conexión y de socket menores que el TTL.
- En PostgreSQL se usan advisory locks de transacción, no de sesión, y las variantes bloqueantes van precedidas de SET LOCAL lock_timeout.
- Las colas usan FOR UPDATE SKIP LOCKED con el trabajo fuera de la transacción, y hay un reaper que rescata los elementos huérfanos.
- En la elección de líder se cumple RetryPeriod < RenewDeadline < LeaseDuration, y OnStoppedLeading detiene el trabajo de verdad.
- Existe un test que simula la pérdida del lock a mitad del trabajo (congelando el proceso o borrando la clave a mano) y comprueba que no hay doble escritura.
- Alguien se ha preguntado por escrito si este lock se puede eliminar con un índice único, una columna de versión o un particionado por clave.

Relojes: por qué un TTL medido con el reloj de pared es una promesa rota

Hasta aquí hemos hablado del lock como si hubiera un tiempo compartido entre tu proceso y el servidor que guarda la clave. No lo hay. Cuando pides un lock con TTL de treinta segundos, el servidor empieza a contar treinta segundos en su reloj, en el instante en que procesa el comando. Tú empiezas a contar en tu reloj, en el instante en que recibes la respuesta. Entre esos dos instantes hay una ida y vuelta de red que puede ser de dos milisegundos o de dos segundos, y durante toda la vida del lease esos dos relojes avanzan a ritmos ligeramente distintos.

La consecuencia práctica es que el margen de seguridad real de tu lock no es el TTL: es el TTL menos el tiempo de ida y vuelta de la adquisición, menos la deriva acumulada entre los dos relojes, menos lo que tardes en darte cuenta. Descontar el RTT completo y no la mitad no es pesimismo gratuito: no sabes si la latencia fue simétrica, y el caso malo es siempre el de asumir que tienes más tiempo del que tienes.

Y luego está la cuestión de qué reloj usas tú. Si mides el tiempo transcurrido con el reloj de pared -time.time() en Python, System.currentTimeMillis() en Java, Date.now() en Node- estás midiendo con un reloj que puede saltar. NTP puede dar un paso hacia atrás si la deriva es grande; un administrador puede corregir la hora a mano; una máquina virtual restaurada desde un snapshot despierta con la hora del snapshot; un leap smear reparte un segundo a lo largo de veinticuatro horas y durante ese día tu reloj corre al 99,9988% de velocidad. Un salto hacia atrás alarga el lease que crees tener. Ese es exactamente el fallo que quieres evitar.

El reloj monótono no salta. time.monotonic() en Linux se apoya en CLOCK_MONOTONIC, que no se ve afectado por ajustes de NTP ni por cambios manuales de hora. Tiene una limitación que conviene conocer: CLOCK_MONOTONIC no cuenta el tiempo que el sistema pasa suspendido. Si tu carga corre en una máquina virtual que puede ser pausada por el hipervisor, un lease medido con CLOCK_MONOTONIC se queda corto al despertar y creerás que aún te queda tiempo. CLOCK_BOOTTIME sí cuenta la suspensión, y en Python se llega a él con time.clock_gettime(time.CLOCK_BOOTTIME). Para un lock, esa es la elección correcta en cualquier entorno donde la suspensión sea posible.

```
import time, uuid
from dataclasses import dataclass

def _now() -> float:
    # CLOCK_BOOTTIME cuenta tambien el tiempo suspendido; CLOCK_MONOTONIC no.
    try:
        return time.clock_gettime(time.CLOCK_BOOTTIME)
    except (AttributeError, OSError):
        return time.monotonic()

@dataclass(frozen=True)
class Lease:
    key: str
    token: str          # identidad del duenyo, para liberar y renovar con CAS
    fence: int          # token monotonico que viaja hasta el almacen de datos
    ttl: float          # lo que el servidor prometio mantener la clave
    granted_at: float   # reloj monotonico JUSTO DESPUES de la respuesta
    rtt: float          # ida y vuelta medida de la adquisicion
    safety_margin: float = 1.0

    @property
    def deadline(self) -> float:
        # El servidor empezo a contar antes de que llegara la respuesta:
        # descontamos el RTT entero, no la mitad. No sabemos si fue simetrico.
        return self.granted_at + self.ttl - self.rtt - self.safety_margin

    def remaining(self) -> float:
        return self.deadline - _now()

    def good_for(self, seconds: float) -> bool:
        'True si queda margen suficiente para una operacion de esa duracion.'
        return self.remaining() > seconds


def acquire(redis, key: str, ttl: float = 30.0) -> Lease | None:
    token = uuid.uuid4().hex
    t0 = _now()
    ok = redis.set(key, token, nx=True, px=int(ttl * 1000))
    t1 = _now()
```

```
    if not ok:
        return None
    fence = redis.incr('fence:' + key)   # monotonico por clave
    return Lease(key=key, token=token, fence=fence, ttl=ttl,
                 granted_at=t1, rtt=t1 - t0)
```

Con esto en la mano aparece un concepto que casi nunca se nombra y que decide la corrección del código: el punto de no retorno. Antes de cada operación con efectos -escribir en la base de datos, enviar el correo, cobrar la tarjeta- no basta con preguntar si el lease sigue vivo. Hay que preguntar si queda margen para terminar esa operación. Un lease con doscientos milisegundos restantes es un lease caducado para una escritura que tarda medio segundo.

```
POR_ITEM = 0.8   # p99 medido de procesar un item, en segundos

for item in lote:
    if not lease.good_for(POR_ITEM):
        # No intentamos renovar aqui: renovar no devuelve la propiedad si
        # ya la perdimos. Paramos y dejamos que otro coja el resto.
        log.warning('lease agotado, quedan %d items', len(lote) - lote.index(item))
        break
    procesar(item, fence=lease.fence)
```

> *Regla que ahorra incidentes: el TTL no es el tiempo que tienes, es el tiempo que el servidor te prometió. El tiempo que tienes es el TTL menos el RTT, menos el margen de seguridad, menos la duración de lo que vas a hacer. Si ese número no es cómodamente positivo, no empieces.*

Renovar el lease no cierra la ventana: watchdog y liberación condicional

La reacción natural cuando el trabajo dura más que el TTL es alargar el TTL. La segunda reacción natural, que es mejor, es renovarlo periódicamente desde un hilo de fondo: el watchdog. Redisson lo hace por defecto con un lockWatchdogTimeout de treinta segundos que renueva cada diez; muchas implementaciones caseras hacen lo mismo. Conviene entender qué compra y qué no.

Lo que compra: que un proceso sano no pierda un lock por el simple hecho de que el trabajo dure más de lo previsto. Eso es real y es útil, y evita el antipatrón de poner un TTL de una hora "por si acaso", que es la forma más rápida de convertir una caída en un bloqueo de una hora.

Lo que no compra: seguridad. Un proceso pausado tiene el watchdog pausado. Mientras está congelado nadie renueva, el TTL expira, otro adquiere el lock y, cuando el primero despierta, su watchdog hace lo primero que tenía pendiente: renovar. Si la renovación es un PEXPIRE a secas, acabas de extender el lock de otro. El error es el mismo que el de liberar con DEL sin comprobar el token, y la solución es la misma: comparar y actuar de forma atómica.

```
-- renew.lua: extiende el TTL solo si seguimos siendo los duenyos
-- KEYS[1] = clave del lock, ARGV[1] = nuestro token, ARGV[2] = TTL en ms
if redis.call('GET', KEYS[1]) == ARGV[1] then
  return redis.call('PEXPIRE', KEYS[1], ARGV[2])
else
  return 0
end
```

El segundo detalle importante es qué hacer cuando la renovación devuelve cero. La respuesta que se ve en la mayoría del código es reintentar, y es la equivocada: un cero significa que la clave ya no es

nuestra, y ninguna cantidad de reintentos la devuelve. Lo correcto es señalar la pérdida y hacer que el trabajador la vea antes de su siguiente operación con efectos.

```
import threading

class Watchdog(threading.Thread):
    'Renueva el lease e informa de su perdida. No decide nada por su cuenta.'

    def __init__(self, redis, lease, renew_script):
        super().__init__(daemon=True)
        self.redis, self.lease, self.renew = redis, lease, renew_script
        self.lost = threading.Event()     # se activa si perdemos la propiedad
        self.stop = threading.Event()     # lo activa el trabajador al terminar
        self.deadline = lease.deadline    # lo unico que el trabajador debe leer

    def run(self):
        intervalo = self.lease.ttl / 3.0   # tres intentos antes de caducar
        while not self.stop.wait(intervalo):
            t0 = _now()
            try:
                ok = self.renew(keys=[self.lease.key],
                                args=[self.lease.token, int(self.lease.ttl * 1000)])
            except Exception:
                # Un fallo de red no prueba que hayamos perdido el lock, pero
                # tampoco prueba lo contrario: no movemos el deadline.
                continue
            if not ok:
                self.lost.set()            # perdida confirmada: no reintentamos
                return
            t1 = _now()
            self.deadline = t1 + self.lease.ttl - (t1 - t0) - self.lease.safety_margin
```

```
wd = Watchdog(redis, lease, renew_script); wd.start()
try:
    for item in lote:
        if wd.lost.is_set():
            raise LockPerdido(lease.key)
        if wd.deadline - _now() < POR_ITEM:
            break
        procesar(item, fence=lease.fence)
finally:
    wd.stop.set()
    release(redis, lease)     # CAS sobre el token, nunca un DEL a secas
```

Fíjate en lo que este diseño no hace: no pretende garantizar la exclusión mutua. El watchdog reduce la probabilidad de que dos procesos se solapen y, sobre todo, reduce el tiempo que un lock queda retenido de más. La garantía sigue estando donde estaba: en el fence que viaja con cada escritura y en el WHERE fence_token < $1 del almacén de datos. Si alguien te propone quitar el fencing porque "ahora renovamos el lease", la respuesta es que acaba de confundir una optimización con una demostración.

La aritmética de la elección de líder en Kubernetes

La elección de líder de client-go es probablemente el lock distribuido que más veces corre en la industria, porque lo lleva dentro cada controlador y cada operador. Tiene tres parámetros y casi nadie los toca, lo cual está bien hasta el día en que tu controlador cambia de líder cada minuto y nadie sabe por qué.

- LeaseDuration (por defecto 15 s): cuánto tiempo un candidato que no es líder espera, contando desde la última renovación que observó, antes de intentar tomar el relevo.

- RenewDeadline (por defecto 10 s): cuánto tiempo tiene el líder para conseguir renovar su Lease antes de rendirse y llamar a OnStoppedLeading.
- RetryPeriod (por defecto 2 s): cada cuánto se reintenta la adquisición o la renovación.

La invariante que impone la biblioteca es LeaseDuration > RenewDeadline > RetryPeriod * 1.2, y los valores por defecto la cumplen con holgura. Lo interesante es lo que significa la diferencia entre los dos primeros. Con 15 y 10 segundos, el líder se rinde cinco segundos antes de que cualquier otro candidato se atreva a tomar el relevo. Esos cinco segundos son todo el margen de seguridad del sistema.

Y ese margen depende por completo de que el líder note que ha fallado y pare. Si el proceso está congelado veinte segundos por una pausa de recolección de basura o por throttling de cgroups, no nota nada a los diez segundos: lo nota a los veinte, cuando ya lleva cinco segundos con un sucesor trabajando. Es exactamente la ventana de dos dueños del principio del artículo, con nombres de Kubernetes.

De ahí sale la regla operativa que más importa y que más se incumple: OnStoppedLeading no es un sitio para "limpiar y seguir". Es un sitio para salir del proceso. Cualquier código que intente apagar ordenadamente las gorutinas de reconciliación está corriendo, por definición, después de haber perdido el lease, y cada milisegundo que tarda es un milisegundo de solape.

```
lock := &resourcelock.LeaseLock{
    LeaseMeta:  metav1.ObjectMeta{Name: "mi-controlador", Namespace: ns},
    Client:     clientset.CoordinationV1(),
    LockConfig: resourcelock.ResourceLockConfig{Identity: podName},
}

leaderelection.RunOrDie(ctx, leaderelection.LeaderElectionConfig{
    Lock:            lock,
    ReleaseOnCancel: true,             // libera el Lease al apagar: relevo en ~1s
    LeaseDuration:   15 * time.Second,
    RenewDeadline:   10 * time.Second,
    RetryPeriod:     2 * time.Second,
    Callbacks: leaderelection.LeaderCallbacks{
        OnStartedLeading: func(ctx context.Context) {
            fence := leaseTransitions(ctx, clientset, ns, "mi-controlador")
            reconcileLoop(ctx, fence)
        },
        OnStoppedLeading: func() {
            // Ya no somos lider. No limpiamos: salimos. Kubernetes nos reinicia.
            klog.Error("lease perdido; terminando el proceso")
            os.Exit(1)
        },
        OnNewLeader: func(id string) {
            if id != podName { klog.Infof("nuevo lider: %s", id) }
        },
    },
})
```

Queda el fencing. client-go no lo hace por ti: te dice si eres líder, no te da un token que el almacén de datos pueda rechazar. Pero el propio objeto Lease lleva uno gratis. El campo spec.leaseTransitions es un contador que el algoritmo incrementa cada vez que el titular cambia, y no cuando el mismo titular renueva. Es decir: monótono, estable mientras tú sigas siendo líder, y distinto para tu sucesor. Eso es un fencing token en toda regla.

```
func leaseTransitions(ctx context.Context, cs kubernetes.Interface, ns, name string) int32 {
    l, err := cs.CoordinationV1().Leases(ns).Get(ctx, name, metav1.GetOptions{})
    if err != nil || l.Spec.LeaseTransitions == nil {
        // Sin token no se escribe. Preferimos no arrancar a arrancar sin fencing.
        klog.Fatalf("no se pudo leer leaseTransitions: %v", err)
```

```
    }
    return *l.Spec.LeaseTransitions
}
```

Con ese entero llegando hasta el UPDATE ... WHERE fence_token < $1 que ya vimos, el controlador pasa de "confío en que el lease funcione" a "si me equivoco, la base de datos me para". Para diagnosticar, la métrica que quieres es la tasa de cambios de líder: si leaseTransitions sube varias veces por hora, el problema no es el lock sino el apiserver, la latencia de etcd o los límites de CPU del pod, y subir RetryPeriod y LeaseDuration es la cura correcta mientras encuentras la causa.

Failover de Redis, minuto a minuto: cómo se pierde un lock sin que nadie se equivoque

Todos los fallos anteriores necesitan que algo salga mal: una pausa, un reloj torcido, un despliegue lento. Este no. Este ocurre con dos clientes correctos, relojes perfectos, ningún TTL expirado y un cluster de Redis funcionando exactamente como está diseñado.

- t = 0,000 s - El cliente A ejecuta SET lock:cuenta-42 tokenA NX PX 30000 contra el primario. El primario escribe la clave en memoria y responde OK.
- t = 0,001 s - A recibe el OK y empieza a trabajar. La replicación de Redis es asíncrona: la propagación a las réplicas se encola, no se espera.
- t = 0,010 s - El primario muere. Un fallo de hardware, un OOM kill, un cable. La clave nunca llegó a ninguna réplica.
- t = 5,2 s - Sentinel (o el propio cluster) declara el fallo y promueve una réplica. La réplica promovida nunca vio lock:cuenta-42.
- t = 5,3 s - El cliente B ejecuta el mismo SET ... NX. Para el nuevo primario la clave no existe. Responde OK.
- t = 5,3 s - A y B creen ser dueños del mismo lock, y lo creerán durante los próximos veinticinco segundos.

Nadie hizo nada mal. Es el modelo de replicación. Y por eso conviene ser preciso con lo que arreglan las herramientas que se proponen para esto.

WAIT numreplicas timeout bloquea hasta que un número dado de réplicas confirma haber recibido las escrituras anteriores. Estrecha la ventana, y estrecharla tiene valor. No la cierra, por tres razones: la confirmación de una réplica no significa durabilidad (con appendfsync everysec la réplica puede confirmar y perderlo al reiniciar); WAIT no puede obligar a que el failover elija precisamente a una de las réplicas que confirmaron; y WAIT añade una ida y vuelta a cada adquisición, que es justo el coste que estabas intentando evitar al no usar un sistema de consenso.

Los parámetros min-replicas-to-write y min-replicas-max-lag hacen que el primario rechace escrituras si no tiene suficientes réplicas al día. También estrechan la ventana, y además convierten una partición en un error visible en lugar de en una pérdida silenciosa, lo cual es una mejora real de operabilidad. Siguen sin ser una garantía.

La conclusión honesta es la que ya conocías, ahora con un ejemplo que no depende de ninguna suposición discutible: si el almacén del lock replica de forma asíncrona, la exclusión mutua no es una propiedad del lock. Es una propiedad del fencing. Redis es una elección perfectamente razonable para coordinar -es rápido, es operativamente simple y casi todo el mundo ya lo tiene- siempre que el argumento de corrección esté escrito en el WHERE de tus escrituras y no en la documentación del

cliente de Redis.

Otros dos backends que funcionan: DynamoDB y las sesiones de Consul

Redis, PostgreSQL y etcd no son las únicas opciones, y a veces no son las que tienes a mano. Dos alternativas merecen un sitio en el mapa porque resuelven el problema con primitivas distintas.

DynamoDB ofrece exactamente lo que hace falta: escrituras condicionales atómicas, lecturas fuertemente consistentes y contadores atómicos. Un lock es un ítem con dueño, caducidad y un contador de fencing que se incrementa en la misma operación que lo adquiere. La biblioteca de referencia de AWS, el DynamoDB Lock Client, lleva años funcionando dentro de Amazon con este modelo.

```
import time, uuid, boto3
from botocore.exceptions import ClientError

def acquire(table, key: str, owner: str, ttl_s: int = 30):
    'Devuelve el fencing token si adquirimos, None si el lock esta ocupado.'
    now_ms = int(time.time() * 1000)
    try:
        resp = table.update_item(
            Key={'lock_key': key},
            UpdateExpression=(
                'SET #own = :owner, lease_expiry = :exp, rvn = :rvn ADD fence :one'
            ),
            # Adquirimos si no existe, si ya somos nosotros, o si caduco.
            ConditionExpression=(
                'attribute_not_exists(lock_key) OR #own = :owner '
                'OR lease_expiry < :now'
            ),
            ExpressionAttributeNames={'#own': 'owner'},
            ExpressionAttributeValues={
                ':owner': owner,
                ':exp': now_ms + ttl_s * 1000,
                ':rvn': uuid.uuid4().hex,   # cambia en cada latido
                ':one': 1,
                ':now': now_ms,
            },
            ReturnValues='UPDATED_NEW',
        )
        return int(resp['Attributes']['fence'])
    except ClientError as e:
        if e.response['Error']['Code'] == 'ConditionalCheckFailedException':
            return None
        raise
```

Hay un detalle en ese código que conviene mirar con lupa, porque es el mismo problema de relojes de la sección anterior disfrazado: lease_expiry < :now se evalúa en el servidor, pero :now lo pone el cliente con su reloj de pared. Un cliente con el reloj adelantado roba locks que no han caducado. El DynamoDB Lock Client esquiva esto por completo con un truco elegante: en lugar de comparar marcas de tiempo, lee el ítem, apunta el rvn (el identificador que cambia con cada latido), espera localmente una duración de lease completa medida con su propio reloj monótono y vuelve a leer. Si el rvn no cambió, el dueño anterior lleva un lease entero sin dar señales de vida y el lock se considera abandonado. Ningún reloj se compara con ningún otro.

Consul aporta una pieza que ninguno de los otros tiene de serie: lock-delay. Cuando una sesión se invalida -por TTL vencido o por fallo de la comprobación de salud-, Consul se niega a conceder la clave a nadie durante un intervalo configurable, quince segundos por defecto. Es un margen de seguridad

incorporado al servidor: exactamente la ventana en la que el dueño anterior podría seguir creyéndose dueño. No es una demostración de corrección, pero es la única implementación popular que reconoce el problema en su diseño en lugar de dejártelo a ti.

```
# Sesion con TTL, borrado automatico de las claves al invalidarse, y lock-delay
SESSION=$(curl -s -X PUT http://consul:8500/v1/session/create -d '{
  "Name": "cierre-diario", "TTL": "30s",
  "Behavior": "delete", "LockDelay": "15s"
}' | jq -r .ID)

# Adquisicion: devuelve true o false
curl -s -X PUT "http://consul:8500/v1/kv/locks/cierre?acquire=$SESSION" -d 'worker-3'

# Renovacion del TTL (equivale al watchdog)
curl -s -X PUT "http://consul:8500/v1/session/renew/$SESSION"

# El ModifyIndex de la clave es monotonico en todo el cluster: sirve de fence
curl -s "http://consul:8500/v1/kv/locks/cierre" | jq '.[0].ModifyIndex'
```

El ModifyIndex de Consul y el leaseTransitions de Kubernetes son el mismo regalo: un entero monótono que el servidor ya mantiene y que puedes usar como fencing token sin escribir una línea de más. Cuando evalúes un backend de locks, esa es una buena pregunta para la lista: ¿me da un número que crece? Si la respuesta es no, tendrás que fabricarlo tú, y fabricarlo bien es más trabajo del que parece.

Trabajos programados: el cron que corre dos veces y el CronJob que no corre ninguna

La demanda de locks distribuidos que más veces llega a una revisión de diseño no viene de un sistema exótico: viene de un cron. Alguien puso una entrada de crontab en tres máquinas para tener alta disponibilidad y descubrió que el informe mensual se envía tres veces. La petición que llega al equipo es "necesitamos un lock distribuido". La respuesta casi siempre es mejor que eso.

En Kubernetes, el CronJob parece resolverlo con concurrencyPolicy: Forbid, y conviene leer bien qué promete. Forbid significa que el controlador no crea un Job nuevo si el anterior aún no ha terminado. No significa que no pueda haber dos pods ejecutando tu código: un Job con reintentos puede lanzar un pod nuevo, y un nodo particionado puede tener un pod viejo todavía vivo mientras el plano de control lo da por perdido. Es una política de planificación, no una exclusión mutua.

Hay además una trampa operativa que muerde a mucha gente. Si el controlador acumula más de cien horarios perdidos y startingDeadlineSeconds no está puesto, deja de planificar ese CronJob por completo y escribe un evento que nadie mira. Un CronJob que corre cada minuto llega a cien horarios perdidos en una hora y cuarenta minutos: el tiempo de una actualización de cluster, o de haber dejado el CronJob suspendido un rato. El síntoma es un trabajo que simplemente deja de existir. Poner startingDeadlineSeconds evita ese estado y, de paso, expresa algo útil: cuánto retraso tiene sentido para este trabajo antes de que sea mejor saltárselo.

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: cierre-diario
spec:
  schedule: "0 2 * * *"
  timeZone: "Europe/Madrid"        # el horario es local; el DST se respeta
  concurrencyPolicy: Forbid
  startingDeadlineSeconds: 300      # sin esto, 100 horarios perdidos lo paran
  successfulJobsHistoryLimit: 3
```

```
  failedJobsHistoryLimit: 5
  jobTemplate:
    spec:
      backoffLimit: 2
      activeDeadlineSeconds: 3600   # cota superior real de la duracion
      template:
        spec:
          restartPolicy: Never
          terminationGracePeriodSeconds: 60
          containers:
            - name: job
              image: registry.interno/cierre:1.4.2
```

Y ahora la parte que de verdad resuelve el problema, que no es el YAML. Un trabajo programado tiene una propiedad que un lock genérico no puede aprovechar: tiene una identidad. No es "el cierre", es "el cierre del 18 de septiembre de 2026". En cuanto esa identidad es explícita, la exclusión mutua se convierte en una restricción de unicidad, que es el primero de los cinco diseños sin lock que vimos antes.

```
-- El registro de ejecuciones es, a la vez, el lock y la idempotencia.
CREATE TABLE job_runs (
    job_name      text        NOT NULL,
    scheduled_for date        NOT NULL,   -- dia de negocio, no timestamp
    started_at    timestamptz NOT NULL DEFAULT now(),
    finished_at   timestamptz,
    PRIMARY KEY (job_name, scheduled_for)
);
```

```
import sys, psycopg

with psycopg.connect(DSN) as conn:
    with conn.transaction():
        row = conn.execute(
            'INSERT INTO job_runs (job_name, scheduled_for) VALUES (%s, %s) '
            'ON CONFLICT DO NOTHING RETURNING started_at',
            ('cierre_diario', dia_de_negocio),
        ).fetchone()
        if row is None:
            print('ya ejecutado para', dia_de_negocio, '- saliendo')
            sys.exit(0)

        ejecutar_cierre(dia_de_negocio)

        conn.execute(
            'UPDATE job_runs SET finished_at = now() '
            'WHERE job_name = %s AND scheduled_for = %s',
            ('cierre_diario', dia_de_negocio),
        )
```

Merece la pena detenerse en lo que hace PostgreSQL aquí, porque es más bonito de lo que parece. Si dos pods arrancan a la vez, el primero inserta la fila y mantiene la transacción abierta mientras trabaja. El segundo intenta insertar, choca con el índice único de una fila aún no confirmada y se queda bloqueado esperando: no obtiene un error, espera. Cuando el primero confirma, el ON CONFLICT DO NOTHING del segundo ve la fila y devuelve cero filas, y el proceso sale limpiamente. Si el primero falla y revierte, el segundo consigue insertar y hace el trabajo. Sin lock explícito, sin TTL, sin fencing y sin una sola línea sobre relojes: la exclusión mutua y el reintento correcto salen del mismo índice.

El coste de este patrón es que la transacción dura lo que dure el trabajo, lo cual es aceptable para un cierre de treinta segundos y es un problema serio para uno de dos horas -una transacción larga bloquea la limpieza de tuplas muertas en toda la base de datos-. Para trabajos largos, la variante es

confirmar la fila de inmediato con un estado running y una marca de latido, y que el segundo proceso decida si adoptar el trabajo a partir de la antigüedad de ese latido. En ese momento has vuelto a construir un lease, esta vez sabiendo exactamente por qué lo necesitas.

Granularidad y coste: lo que de verdad paga un lock

Un lock es un punto de serialización, y los puntos de serialización tienen una aritmética implacable. Si la sección crítica dura doscientos milisegundos y veinte trabajadores compiten por la misma clave, el techo del sistema es cinco operaciones por segundo. Añadir trabajadores no sube ese número: sube la cola. Es la ley de Amdahl aplicada a algo que nadie mide porque no parece código costoso.

A eso se suma el coste fijo. Una adquisición y una liberación son dos idas y vueltas al almacén del lock como mínimo, más el tráfico de renovación si hay watchdog. Con un Redis a un milisegundo, una sección crítica de dos milisegundos pasa a costar cuatro. Si tu sección crítica es más corta que la ida y vuelta al servidor de locks, estás pagando más por coordinar que por trabajar, y la solución no es un lock más rápido: es empujar el trabajo dentro del almacén -un único UPDATE condicional, un INCR, una restricción de unicidad- y quedarte sin sección crítica.

Las tres métricas que hacen visible todo esto son fáciles de instrumentar y raras de encontrar en un dashboard:

- Tiempo de espera (desde que pides el lock hasta que lo tienes). Si su p99 crece mientras el p50 no se mueve, tienes contención en la cola, no un sistema lento.
- Tiempo de retención (desde que lo tienes hasta que lo sueltas). Es el numerador del techo de rendimiento. Reducirlo a la mitad duplica la capacidad; ningún otro cambio tiene esa propiedad.
- Ratio de contención (adquisiciones que esperaron / adquisiciones totales). Por debajo del 5% el lock es prácticamente gratis. Por encima del 50% el lock es tu arquitectura, y conviene que sea una decisión consciente.

Cuando el ratio de contención se dispara, la palanca correcta casi nunca es un backend más rápido: es partir la clave. Un lock global informes se convierte en informes:{tenant_id} y la contención se divide entre el número de inquilinos activos. Un lock de inventario global se convierte en uno por SKU. El límite de esta técnica es el punto en el que la clave ya no corresponde a una unidad de aislamiento real del negocio; pasado ese punto, partir más solo mueve la corrupción de sitio.

Hay un último modo de fallo que merece nombre propio porque aparece de golpe: el convoy. Si el tiempo de retención se acerca al intervalo medio entre peticiones, la cola deja de vaciarse y crece sin límite. Los trabajadores se quedan esperando, los pools de conexiones se agotan, las peticiones agotan su timeout aguas arriba y el sistema entero se cae por un lock que "solo tarda doscientos milisegundos". La defensa es un tiempo máximo de espera acotado y una respuesta honesta cuando se agota: ahora no, en vez de esperar indefinidamente. Es la misma idea del load shedding, aplicada a la coordinación.

Cómo demostrar que tu lock está roto antes de que lo demuestre producción

La mayoría de los tests de locks que se escriben comprueban lo que no hace falta comprobar: que dos hilos en la misma máquina no entran a la vez. Eso funciona siempre y no dice nada sobre los fallos reales, que necesitan un proceso congelado, una red partida o un failover. Vale la pena escribir tres tests, y son más fáciles de lo que parece.

El primero y más importante no prueba el lock: prueba el fencing. Y la aserción que buscas no es "solo uno escribió", que es indemostrable en una sola ejecución, sino "el almacén rechazó al escritor caducado".

```
def test_el_escritor_caducado_es_rechazado(redis, pg):
    a = acquire(redis, 'cuenta:42', ttl=2.0)
    assert a is not None

    time.sleep(2.5)                # A se congela; su lease caduca

    b = acquire(redis, 'cuenta:42', ttl=30.0)
    assert b is not None
    assert b.fence > a.fence       # el token crece: sin esto nada funciona

    assert escribir_con_fence(pg, 'cuenta:42', b.fence, saldo=100) is True
    # A despierta y escribe con su token viejo. Debe rebotar.
    assert escribir_con_fence(pg, 'cuenta:42', a.fence, saldo=999) is False
    assert leer_saldo(pg, 'cuenta:42') == 100
```

```
-- escribir_con_fence: la unica linea que garantiza la correccion
UPDATE cuentas
   SET saldo = $3, fence_token = $2
 WHERE id = $1
   AND fence_token < $2;
-- rowcount == 0  =>  llegamos tarde: alguien mas nuevo ya escribio
```

El segundo test congela un proceso de verdad. SIGSTOP es la forma más barata de simular una pausa de recolección de basura o un throttling de cgroups: el proceso deja de ejecutar instrucciones pero conserva sus conexiones y su estado, que es exactamente lo que hace peligrosa a una pausa.

```
import os, signal, subprocess, sys, time

def test_proceso_congelado_no_corrompe(pg):
    worker = subprocess.Popen([sys.executable, 'worker.py', '--ttl', '3'])
    time.sleep(1)                     # deja que adquiera y empiece

    os.kill(worker.pid, signal.SIGSTOP)   # congelado: el watchdog tambien
    time.sleep(4)                         # el lease caduca mientras duerme

    segundo = subprocess.Popen([sys.executable, 'worker.py', '--ttl', '30'])
    time.sleep(2)                         # el segundo adquiere y escribe

    os.kill(worker.pid, signal.SIGCONT)   # despierta creyendose duenyo
    worker.wait(timeout=30)

    # Lo que comprobamos no es que no escribiera: es que no CORROMPIO.
    assert invariantes_de_negocio_se_cumplen(pg)
    assert contar_escrituras_rechazadas(pg) >= 1   # el fence hizo su trabajo
    segundo.terminate()
```

El tercero corta la red entre el trabajador y el almacén de locks sin tocar el resto. Con Testcontainers y Toxiproxy delante de Redis, añadir un toxic de tipo timeout deja las conexiones colgadas igual que una partición real, que es distinto de un connection refused: en una partición no recibes un error, no recibes nada. Ese es el caso que descubre si tu código trata "no sé si sigo teniendo el lock" como "sigo teniendo el lock", que es el error por omisión más común de todos.

Estos tres tests corren en segundos y caben en CI. Su valor no está en la primera ejecución -normalmente pasan- sino en la vigésima, cuando alguien simplifica la liberación a un DEL porque "el script de Lua era innecesario" y el test lo para antes de que llegue a producción. Un lock distribuido es código que casi nunca falla y, cuando falla, lo hace en silencio y con dinero de por medio. Merece una

red de seguridad que grite.

Preguntas frecuentes

¿Puedo usar simplemente un lock en memoria si sólo tengo una réplica? Sí, y deberías: un threading.Lock es correcto, rápido y no tiene ventana. El problema es que "sólo una réplica" deja de ser cierto sin avisar, normalmente durante un despliegue con RollingUpdate. Si la corrección depende de que haya una sola réplica, escríbelo en el manifiesto con Recreate y en un comentario.

¿ZooKeeper sigue teniendo sentido en 2026? Tiene sentido si ya lo operas. Sus znodes efímeros y secuenciales dan leases y tokens monótonos con una semántica muy limpia, pero si partes de cero y estás en Kubernetes, los Leases nativos o etcd te dan lo mismo sin un sistema más que mantener.

¿El fencing token tiene que ser un entero? Tiene que estar totalmente ordenado y ser monótono. Un entero de una secuencia, la revisión de etcd o leaseTransitions lo cumplen. Un UUID v4 no. Un timestamp tampoco: el reloj es exactamente lo que no queremos que decida.

¿Qué TTL pongo si el trabajo tarda un tiempo muy variable? No pongas un TTL enorme; pon uno moderado y renuévalo. Un TTL de dos horas significa que un worker muerto bloquea el sistema dos horas. Un TTL de treinta segundos con heartbeat te da lo mejor de ambos: recuperación rápida ante caídas y duración ilimitada mientras el proceso esté vivo.

¿Y si el almacenamiento no soporta escrituras condicionales? Entonces no puedes hacer fencing ahí, y debes asumirlo explícitamente. La mayoría sí puede: PostgreSQL con un WHERE, S3 con If-Match sobre el ETag, DynamoDB con expresiones condicionales, Redis con Lua. Si de verdad no puede, mueve la decisión a un sistema que sí pueda y deja el otro como derivado.

¿Los locks distribuidos sirven para transacciones entre servicios? No. Ese problema es el de la consistencia distribuida y se resuelve con sagas, outbox transaccional e idempotencia, no manteniendo un lock abierto mientras esperas a otro servicio. Un lock que abarca una llamada de red es una forma elegante de construir un interbloqueo distribuido.

Glosario

- Lease. Un lock con caducidad concedido por un servicio: el titular manda mientras lo renueve, y lo pierde automáticamente si deja de hacerlo.
- Fencing token. Entero estrictamente creciente entregado en cada concesión del lock, que el almacenamiento usa para rechazar escrituras de titulares caducados.
- Advisory lock. Lock de PostgreSQL sobre un entero arbitrario elegido por la aplicación, sin relación con ninguna fila. Existe en variante de sesión y de transacción.
- SKIP LOCKED. Modificador de FOR UPDATE que omite las filas ya bloqueadas por otra transacción en vez de esperarlas. La base de las colas en PostgreSQL.
- Redlock. Algoritmo de lock distribuido sobre varias instancias independientes de Redis mediante mayoría. Válido para eficiencia, discutido para corrección.
- Split brain. Situación en la que dos nodos creen simultáneamente ser el líder y ambos escriben.
- Stop-the-world. Pausa en la que el recolector de basura detiene todos los hilos de la aplicación. La causa más común de la ventana de dos dueños.
- Quórum. Mayoría de nodos necesaria para tomar una decisión en un sistema replicado; la base de etcd, ZooKeeper y Raft en general.

Para cerrar

La idea que merece la pena llevarse es incómoda y libera bastante: un lock distribuido no garantiza exclusión mutua, sólo la hace probable. Si tu sistema sobrevive a que dos procesos hagan el mismo trabajo, ya has terminado y una instancia de Redis te sobra. Si no sobrevive, ningún lock te va a salvar por sí solo, y la garantía tiene que vivir donde viven los datos: en una condición de escritura que el almacenamiento evalúa de forma atómica. Todo lo demás -el número de réplicas de Redis, la sofisticación del algoritmo, la elegancia del decorador- es ruido alrededor de esa única decisión.

English version

A team I worked with ran a nightly billing job across three replicas. Only one was supposed to execute it, so they put a lock in Redis: SET with NX, a thirty-second TTL, and a DEL at the end. It worked for fourteen months. On night four hundred and twenty-three, a new customer multiplied the batch volume by six, the process chewed through its memory, the garbage collector paused for 1.4 seconds, the TTL expired, another replica grabbed the lock, and at 03:11 the system issued 812 duplicate invoices.

The lock code was correct. That is the uncomfortable point of this article: most distributed mutual-exclusion bugs are not in the lock code, they are in the assumption that a TTL-based lock guarantees mutual exclusion. It does not. It never has. What follows is the full map: where the failure window lives, when you can ignore it, how it is actually closed when you cannot, and -the chapter that saves the most money- how to design the system so you never need the lock.

Before picking a tool: efficiency or correctness?

Martin Kleppmann drew this distinction in 2016 and it remains the single most useful question to ask before writing a line of code.

You lock for efficiency when a duplicate run only wastes work. Two replicas regenerate the same thumbnail, two workers recompute the same aggregate, two nodes refresh the same cache. The end state is identical; you just burned extra CPU. A probabilistic lock is perfectly reasonable here: one Redis instance, a TTL, and if a double run slips through once in a thousand, nothing breaks.

You lock for correctness when a duplicate run corrupts data: charging twice, decrementing the same inventory twice, issuing two invoices with the same number, writing two incompatible versions of the same file. Here "almost always exclusive" is worth nothing. A lock that fails once every ten thousand runs, running every five minutes, gives you an incident a month.

> *The practical rule: if you can write in a runbook "if this runs twice, just retry", it is efficiency. If you have to write "if this runs twice, open a ticket with finance", it is correctness, and you need fencing.*

Why every TTL lock has a two-owner window

The chain of reasoning is short and there is no way around it. A TTL lock expires by time. For expiring by time to be safe, the process holding it must guarantee that it finishes -or at least stops writing- before the TTL runs out. No process can guarantee that, because no process controls when the CPU is taken away from it.

This is the exact sequence of the invoice incident:

```
t=0.00  Worker A: SET lock:billing <token-a> NX PX 30000  -> OK
t=0.10  Worker A: starts processing the batch
t=12.40 Worker A: stop-the-world GC pause (1.4 s)
t=13.80 Worker A: resumes... but the pod sat on a node with CPU
                  throttling; the kernel does not hand the CPU back
t=30.00 Redis:    the TTL expires, the key vanishes
t=30.05 Worker B: SET lock:billing <token-b> NX PX 30000  -> OK
t=30.10 Worker B: starts processing THE SAME batch
```

```
t=31.20 Worker A: runs again, still believes it is the owner,
                  and writes to the database
        ^^^ two concurrent writers
```

The causes of the pause are more varied than people assume, and almost none of them show up in tests:

- Stop-the-world GC pauses. The JVM with large heaps, Go with a pathological cycle, Python collecting cycles across millions of objects.
- cgroup throttling. The most frequent cause in Kubernetes and the least suspected: if the container exceeds its CPU quota, the kernel freezes it until the next 100 ms period. With a badly chosen limit you accumulate seconds of freezing without a single application metric reflecting it.
- Page faults and swap. The process is "running" while it waits on disk.
- VM live migration or hypervisor suspension.
- Network partition. The process keeps running perfectly well; what it cannot do is talk to the lock service. From the outside that is indistinguishable from a crash, which is exactly why the TTL exists in the first place.
- A blocking call with no timeout. A PUT to S3 that takes forty seconds because nobody configured the client timeout.

You cannot eliminate the window. You can make it improbable -generous TTL, heartbeat, timeouts everywhere- but improbable is not impossible, and at sufficient scale improbable happens on Thursdays. The only real way out is for the storage layer to reject writes from an expired owner.

Fencing tokens: the one defence that closes the window

The idea fits in three sentences. Every time the lock service grants the lock, it returns a strictly increasing integer: the fencing token. The holder includes that token in every write. Storage remembers the highest token it has seen for that resource and rejects any write carrying a lower or equal one.

With that in place, worker A comes back from its GC pause holding token 41, tries to write, and the database -which has already seen B's token 42- rejects it. A does not need to know it lost the lock. Storage knows on its behalf.

```
Worker A acquires the lock  -> token 41
Worker A freezes
Worker B acquires the lock  -> token 42
Worker B writes (42)        -> accepted, fence_token becomes 42
Worker A writes (41)        -> 41 does not beat 42  -> REJECTED
```

In PostgreSQL the mechanism is one column and one WHERE clause:

```
ALTER TABLE accounts ADD COLUMN fence_token bigint NOT NULL DEFAULT 0;

-- The guarded write. If it returns 0 rows, you are a stale leader.
UPDATE accounts
   SET balance_cents = balance_cents - $2,
       fence_token   = $3
 WHERE id = $1
   AND fence_token < $3;
```

Notice there is no lock at all in that statement. The token turns mutual exclusion into a data check, which

is something the database already does atomically. That is the whole trick.

The usual objection is that this forces you to touch every write. True, and that is why almost nobody implements it. It is also why almost everybody has the bug. If the guarded work writes to five tables, you need the token in all five; if it writes to S3, you need it in the conditional-write precondition; if it calls a third-party API, you need that API to accept an idempotency key derived from the token.

Level 1 - Redis: a decent lock, done properly

For efficiency locking, a single Redis instance is the right answer 90% of the time. What matters is not making the two classic mistakes: acquiring without a TTL (if the process dies, the lock is stuck forever) and releasing with a bare DEL (you delete someone else's lock once yours has already expired).

The release has to be an atomic compare-and-delete, and in Redis that means Lua:

```
-- release.lua: delete the key only if it is still mine
if redis.call('GET', KEYS[1]) == ARGV[1] then
  return redis.call('DEL', KEYS[1])
end
return 0
```

Renewal is the same shape: extend the TTL only if you are still the owner.

```
-- extend.lua: refresh the TTL only if it is still mine
if redis.call('GET', KEYS[1]) == ARGV[1] then
  return redis.call('PEXPIRE', KEYS[1], ARGV[2])
end
return 0
```

The full client in Python, with a heartbeat on a separate thread and an event that tells the work if we lose the lock:

```
import contextlib
import logging
import secrets
import threading
import redis

log = logging.getLogger(__name__)
r = redis.Redis(host="redis", socket_timeout=2, socket_connect_timeout=2)

_RELEASE = r.register_script(open("release.lua").read())
_EXTEND  = r.register_script(open("extend.lua").read())


@contextlib.contextmanager
def redis_lock(key: str, ttl_ms: int = 30_000):
    """An EFFICIENCY lock. Do not use it where a double run corrupts data.

    Yields a threading.Event: if it fires, we have lost the lock and the
    work must abort as soon as it can check.
    """
    token = secrets.token_hex(16).encode()
    if not r.set(key, token, nx=True, px=ttl_ms):
        raise TimeoutError(f"lock is held: {key}")

    lost = threading.Event()
    stop = threading.Event()
```

```
    def heartbeat() -> None:
        # Renew at one third of the TTL: three attempts before expiry.
        interval = ttl_ms / 3000
        while not stop.wait(interval):
            try:
                if not _EXTEND(keys=[key], args=[token, ttl_ms]):
                    log.error("lock %s lost: another process holds it", key)
                    lost.set()
                    return
            except redis.RedisError:
                log.warning("renewal of %s failed; will retry", key)

    t = threading.Thread(target=heartbeat, daemon=True, name=f"hb:{key}")
    t.start()
    try:
        yield lost
    finally:
        stop.set()
        t.join(timeout=1)
        with contextlib.suppress(redis.RedisError):
            _RELEASE(keys=[key], args=[token])
```

Usage:

```
with redis_lock("lock:rebuild-search-index", ttl_ms=60_000) as lost:
    for batch in batches():
        if lost.is_set():
            raise RuntimeError("lock lost; aborting to avoid duplicate work")
        process(batch)
```

Three details that are usually missing. The client socket_timeout: without it, a renewal can hang for longer than the TTL itself. The heartbeat interval at one third of the TTL, which gives you three chances to renew before losing it instead of the single chance you get renewing at half. And the lost event: renewing is pointless if the work never checks whether it is still the owner.

Redlock, and what it actually buys you

Redlock is the algorithm Redis proposes for locks across N independent instances (usually five): the client tries to acquire on all of them, considers itself the owner if it gets a majority within a time budget, and subtracts the elapsed time from the lock's remaining validity.

Kleppmann's critique, still valid a decade later, is that Redlock looks fit for correctness -it has quorum, it has majorities, it smells like consensus- and it is not. The argument in three steps:

1. Lock validity depends on the clocks of the five instances advancing at comparable rates. A clock jump on any one of them (NTP correcting abruptly, an operator adjusting the time, a virtualised clock drifting) can make it expire earlier than the client computes.
2. Subtracting elapsed time assumes the client can measure how long it took. If the client pauses between "I acquired it" and "I compute validity", the computation is already invalid.
3. And above all: none of those problems go away with more replicas, because the dangerous window sits between the holder and storage, not between the holder and the lock service.

The practical conclusion is not "Redlock is garbage", it is duller than that: if you lock for efficiency, five Redis instances buy you nothing that one does not, and they cost five times more to operate. If you lock for correctness, Redlock is not enough either, and what you need is a fencing token. Redis can hand you that token -an INCR on a dedicated counter- but at that point the guarantee is coming from your storage, not from Redis.

```
-- acquire_fenced.lua
-- KEYS[1] = lock key, KEYS[2] = fencing counter
-- ARGV[1] = owner identity, ARGV[2] = TTL in ms
-- Returns the fencing token, or nil if the lock is held.
if redis.call('SET', KEYS[1], ARGV[1], 'NX', 'PX', ARGV[2]) then
  return redis.call('INCR', KEYS[2])
end
return nil
```

One durability note that almost nobody mentions: with asynchronous replication, a Redis failover can lose the very SET that granted the lock, and the new primary hands it to another client. The INCR counter suffers exactly the same fate, which breaks token monotonicity. If you are serious about fencing, the counter must live in a store with consensus -etcd, or a PostgreSQL sequence- and not on the replica that was just promoted.

Level 2 - PostgreSQL: advisory locks and SKIP LOCKED

If you already run PostgreSQL -and you do- you probably do not need Redis for this. Advisory locks are locks over arbitrary integers you define: they never touch the heap, generate no dead tuples, create no MultiXact entries, and clean themselves up.

There are two families, and picking the wrong one is mistake number one:

- Transaction level (pg_try_advisory_xact_lock): released automatically on COMMIT or ROLLBACK. This is what you want 95% of the time.
- Session level (pg_try_advisory_lock): lives until you explicitly release it or the connection closes. If the worker dies without a clean shutdown, the lock lingers until the server notices the dead connection, which with default TCP keepalive settings can be more than two hours.

> *The trap that eats the most afternoons: session-level advisory locks are incompatible with PgBouncer in transaction pooling mode. In that mode the next statement may travel over a different server connection, so you acquire the lock on one connection and try to release it from another. The lock stays put and the unlock returns a warning nobody reads. If you use PgBouncer in transaction mode -and almost everyone does- only transaction-level locks are safe.*

The lock key is a bigint. Many people use hashtext(), an internal function with no stability guarantee across major versions. Compute it in the application and you get a reproducible, auditable value:

```
import hashlib

def lock_key(name: str) -> int:
    """Deterministic, stable bigint for a PostgreSQL advisory lock."""
    digest = hashlib.blake2b(name.encode(), digest_size=8).digest()
    return int.from_bytes(digest, "big", signed=True)
```

And the full pattern with SQLAlchemy. Note that everything happens inside a single transaction: the lock protects exactly the work that commits with it.

```
from contextlib import contextmanager
from sqlalchemy import text
from sqlalchemy.orm import Session


@contextmanager
```

```
def advisory_xact_lock(session: Session, name: str):
    """Try the lock. If someone else holds it, raise instead of waiting.

    The lock is released when the transaction ends: there is no code path,
    not even a kill -9, that can leave it stuck.
    """
    key = lock_key(name)
    acquired = session.execute(
        text("SELECT pg_try_advisory_xact_lock(:k)"), {"k": key}
    ).scalar_one()
    if not acquired:
        raise TimeoutError(f"another process holds {name}")
    yield


with Session(engine) as s, s.begin():
    with advisory_xact_lock(s, "nightly-billing"):
        generate_invoices(s)   # commits on exit; the lock falls with it
```

If you would rather wait than give up, use the blocking variant pg_advisory_xact_lock, but always with a lock_timeout set first, or you will turn contention into an indefinite hang:

```
SET LOCAL lock_timeout = '5s';
SELECT pg_advisory_xact_lock(-4488901523771214477);
```

When what you want is a queue, not a lock

A very common design mistake is using a global advisory lock so that "only one worker processes the queue". That serialises the work and wastes your replicas. If the work is per item rather than global, what you need is FOR UPDATE SKIP LOCKED, which lets N workers claim different items without colliding or waiting:

```
WITH next AS (
  SELECT id
    FROM jobs
   WHERE status = 'pending'
     AND run_after <= now()
   ORDER BY priority DESC, run_after
   LIMIT 10
   FOR UPDATE SKIP LOCKED
)
UPDATE jobs j
   SET status     = 'running',
       started_at = now(),
       attempts   = j.attempts + 1
  FROM next n
 WHERE j.id = n.id
RETURNING j.id, j.payload;
```

Two warnings. First: the actual work goes outside that transaction, not inside; if you put it inside, a thirty-second task holds a transaction open for thirty seconds, and long transactions block autovacuum and bloat the table. Second: you need a reaper that returns rows stuck in 'running' back to 'pending', because a worker that dies after the UPDATE leaves the job orphaned. That reaper is, quite literally, your TTL.

And the fencing token, if you need one, comes free from a sequence:

```
CREATE SEQUENCE fence_billing AS bigint;

-- At job startup, right after acquiring the lock:
SELECT nextval('fence_billing');   -- monotonic, durable, backed by consensus
```

Level 3 - etcd and Kubernetes Leases

When what you need is not a one-off lock but a leader -a process that is in charge while it lives and steps aside when it dies- the right mechanism is a lease over a consensus store. In Kubernetes you already have one: Lease objects in the coordination.k8s.io group are the same primitive the kube-controller-manager and the scheduler use to elect their own leaders.

The reference implementation in Go, with client-go:

```
package main

import (
    "context"
    "os"
    "os/signal"
    "syscall"
    "time"

    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes"
    "k8s.io/client-go/rest"
    "k8s.io/client-go/tools/leaderelection"
    "k8s.io/client-go/tools/leaderelection/resourcelock"
    "k8s.io/klog/v2"
)

func main() {
    id := os.Getenv("POD_NAME") // valueFrom: fieldRef: metadata.name
    cfg, err := rest.InClusterConfig()
    if err != nil {
        klog.Fatal(err)
    }
    client := kubernetes.NewForConfigOrDie(cfg)

    lock := &resourcelock.LeaseLock{
        LeaseMeta: metav1.ObjectMeta{
            Name:      "billing-scheduler",
            Namespace: os.Getenv("POD_NAMESPACE"),
        },
        Client:     client.CoordinationV1(),
        LockConfig: resourcelock.ResourceLockConfig{Identity: id},
    }

    ctx, cancel := signal.NotifyContext(
        context.Background(), syscall.SIGTERM, syscall.SIGINT)
    defer cancel()

    leaderelection.RunOrDie(ctx, leaderelection.LeaderElectionConfig{
        Lock: lock,
        // Invariant: RenewDeadline < LeaseDuration, and RetryPeriod
        // around RenewDeadline/3. Break it and client-go starts stealing
        // leadership back and forth between perfectly healthy replicas.
        LeaseDuration:   15 * time.Second,
        RenewDeadline:   10 * time.Second,
        RetryPeriod:     2 * time.Second,
        ReleaseOnCancel: true, // hand the lease back on SIGTERM
        Callbacks: leaderelection.LeaderCallbacks{
            OnStartedLeading: func(ctx context.Context) {
                runScheduler(ctx) // must honour ctx.Done()
            },
```

```
            OnStoppedLeading: func() {
                // We lost the lease. We do not know whether our work is
                // still in flight, so the only safe move is to die and let
                // the kubelet restart us as a follower.
                klog.Error("lease lost; terminating process")
                os.Exit(1)
            },
            OnNewLeader: func(identity string) {
                if identity != id {
                    klog.Infof("new leader: %s", identity)
                }
            },
        },
    })
}
```

The minimum permissions. Without create the first election fails in a clean namespace, and that is the most common day-one failure:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: leader-election
  namespace: billing
rules:
  - apiGroups: ["coordination.k8s.io"]
    resources: ["leases"]
    verbs: ["get", "list", "watch", "create", "update"]
```

And here is the detail almost nobody knows: client-go leader election does not fence. Its own documentation says so. If the leader freezes, lets the lease expire and then resumes, it will still believe it is in charge. OnStoppedLeading only fires when the process gets to run and notices the loss; while frozen, it notices nothing. That is why the os.Exit(1) above is not laziness: it is the cheapest way to guarantee that a degraded leader stops writing.

If you need real fencing, the Lease object itself gives you the token: the spec.leaseTransitions field is an integer that client-go increments only when the lease changes holder, not on every renewal. That is exactly fencing-token semantics.

```
# The token: increments on every leadership change, not on renewals.
kubectl get lease billing-scheduler -n billing \
  -o jsonpath='{.spec.leaseTransitions}'
```

Talking to etcd directly, the equivalent is the creation revision of the mutex key, which is globally monotonic:

```
sess, err := concurrency.NewSession(cli, concurrency.WithTTL(10))
if err != nil {
    return err
}
defer sess.Close() // revokes the lease: the successor takes over in ms

mu := concurrency.NewMutex(sess, "/locks/reindex")
if err := mu.TryLock(ctx); err != nil {
    if errors.Is(err, concurrency.ErrLocked) {
        return nil // someone else has it; nothing to do this cycle
    }
    return err
}
defer mu.Unlock(context.WithoutCancel(ctx)) // unlock even if ctx is dead
```

```
fence := mu.Header().Revision // cluster-wide monotonic: use it as the token
```

End-to-end fencing: a complete example

Let us put the pieces together into something you can actually copy. The scenario: a job that adjusts account balances, run by several replicas, where a double run is an accounting problem. The lock provides coordination; the database provides the guarantee.

```
import logging
import secrets
import redis
from sqlalchemy import text
from sqlalchemy.orm import Session

log = logging.getLogger(__name__)
r = redis.Redis(host="redis", socket_timeout=2)

_ACQUIRE = r.register_script(open("acquire_fenced.lua").read())


class StaleLeader(Exception):
    """Our fencing token is no longer the highest: someone replaced us."""


def acquire_fenced(key: bytes, ttl_ms: int) -> tuple[bytes, int] | None:
    owner = secrets.token_hex(16).encode()
    token = _ACQUIRE(keys=[key, key + b":fence"], args=[owner, ttl_ms])
    return (owner, int(token)) if token else None


def adjust_balance(s: Session, account_id: int, delta: int, fence: int) -> None:
    """Guarded write. Rejected if a later leader has already written."""
    rows = s.execute(
        text("""
            UPDATE accounts
               SET balance_cents = balance_cents + :delta,
                   fence_token   = :fence
             WHERE id = :id
               AND fence_token < :fence
        """),
        {"delta": delta, "fence": fence, "id": account_id},
    ).rowcount

    if rows == 0:
        # Either the account does not exist, or we were overtaken. The
        # difference matters: the first is a data bug, the second is
        # the system working as designed.
        current = s.execute(
            text("SELECT fence_token FROM accounts WHERE id = :id"),
            {"id": account_id},
        ).scalar_one_or_none()
        if current is None:
            raise LookupError(f"account {account_id} does not exist")
        raise StaleLeader(f"our token {fence} does not beat {current}")


def run_adjustments(engine, adjustments: list[tuple[int, int]]) -> None:
    granted = acquire_fenced(b"lock:adjustments", 30_000)
    if granted is None:
        log.info("another replica holds the lock; doing nothing")
        return

    _owner, fence = granted
    log.info("lock acquired with fencing token %d", fence)
```

```
    try:
        with Session(engine) as s, s.begin():
            for account_id, delta in adjustments:
                adjust_balance(s, account_id, delta, fence)
    except StaleLeader:
        # Not an error worth paging for: the system did its job. The
        # transaction rolls back and the live leader finishes the batch.
        log.warning("stale leader detected; aborting without writing")
```

What makes this design elegant is that there is no "am I still the leader?" check anywhere in the business logic. There is no check-then-write race, because the check is the write. That is the test to apply to any fencing implementation you come across: if somewhere there is an if i_am_leader(): write(), a GC pause fits between those two calls.

A note on per-resource versus global monotonic tokens. In the example the counter is global to the lock while the fence_token column is per account. That works because a globally monotonic token is also monotonic for each individual account. The side effect is that an account untouched for months keeps an old token, which is harmless. What you cannot do is use a different counter per lock and share the column across several locks: tokens from two counters are not comparable.

The best distributed lock is the one you never write

This is the chapter that should be read first. In practice, more than half the distributed locks in a typical system can be removed by changing the design, and each one you remove is an entire class of incidents that disappears.

1. Uniqueness constraints instead of mutual exclusion

If the point of the lock is "do not create this twice", tell the database and let it enforce that. A unique index is perfect mutual exclusion: no TTL, no clocks, no window.

```
CREATE TABLE payments (
  id              bigserial PRIMARY KEY,
  idempotency_key text NOT NULL,
  order_id        bigint NOT NULL,
  amount_cents    bigint NOT NULL,
  created_at      timestamptz NOT NULL DEFAULT now()
);

CREATE UNIQUE INDEX payments_idem ON payments (idempotency_key);

-- The second attempt does not fail: it creates nothing and returns zero
-- rows, at which point you read the original payment and return it as is.
INSERT INTO payments (idempotency_key, order_id, amount_cents)
VALUES ($1, $2, $3)
ON CONFLICT (idempotency_key) DO NOTHING
RETURNING id;
```

2. Optimistic concurrency instead of pessimistic

Read the version, compute, and write conditioned on the version not having changed. If it changed, retry with fresh data. It costs the occasional retry and saves you the entire locking infrastructure.

```
UPDATE inventory
   SET units   = units - $2,
       version = version + 1
 WHERE sku = $1
   AND version = $3
   AND units >= $2;
-- 0 rows: either someone wrote before you, or there is not enough stock.
-- Re-read, decide, and retry with a bounded attempt count.
```

3. Let the database serialise

Many "I need a lock" situations are really "I need this operation to be atomic", and a single statement already is. UPDATE balances SET amount = amount - 10 WHERE id = 7 needs no external lock: PostgreSQL locks the row for the duration of the statement and other transactions wait. You only need an external lock when something the database cannot see happens between the read and the write.

4. Partition by key instead of coordinating

If messages for the same aggregate always land on the same Kafka partition, and each partition is consumed by exactly one group member, you already have per-aggregate mutual exclusion with no lock at all. It is the same principle as sharding by user hash: coordination replaced by routing.

5. A single writer by design

A Deployment with replicas: 1 and strategy: Recreate, or a single-replica StatefulSet, is the cheapest way to have exactly one writer. It is not fault tolerant -during the restart nobody is running- but for a nightly job that can wait two minutes, that is an operational advantage rather than a defect. Watch out for RollingUpdate: it leaves a window where the old and new pods coexist, so Recreate here is a deliberate decision, not the default.

Eight mistakes that show up in every code review

1. Releasing with DEL instead of compare-and-delete. The most widespread and the quietest bug: your lock expired, someone else took it, and you delete theirs on the way out. From that moment on, three processes are convinced they own it. Always Lua, always comparing the token.
2. Acquiring without a TTL "because we always release in the finally block". The finally block does not run when the kernel OOM-kills the process, when the node powers off, or when someone runs kill -9. A lock with no expiry is an outage waiting for the worst possible moment.
3. Renewing the lock without ever checking whether you lost it. The heartbeat tells you when you stopped being the owner, but if the work never consults that fact, the information is worthless. Propagate the loss as a cancellation, not as a log line.
4. Using session-level advisory locks behind PgBouncer in transaction mode. You acquire on one server connection and release -you think- on another. The lock stays put until the connection dies. Always use the _xact variant.
5. Picking the TTL by intuition. The TTL should come from the measured p99 runtime, with headroom, and it must be revisited as the job grows. The invoice incident at the top of this article was exactly this: a TTL calibrated for a volume that no longer existed.
6. Violating the leader-election invariants. With RenewDeadline greater than or equal to LeaseDuration, or an oversized RetryPeriod, client-go starts a leadership dance between perfectly healthy replicas. The healthy ratio is 15/10/2 seconds, and you do not touch it without measuring.

7. Treating OnStoppedLeading as a warning log. Losing the lease means someone else is working right now. If your callback logs a message and carries on, you have two leaders by design. Cancel the context, wait for the work to stop, and if you cannot prove it stopped, terminate the process.
8. Locking around a third-party call. A lock cannot protect an effect that happens outside your system: if the payment gateway received the request, it already charged, even if you lose the lock a millisecond later. The tool there is the provider's idempotency key, not the lock.

Observability: what to measure so you find out before the incident

A badly instrumented distributed lock fails silently for months. These four metrics catch the problem before a customer does.

```
from prometheus_client import Counter, Histogram

LOCK_WAIT = Histogram(
    "lock_acquire_seconds", "Time spent waiting to acquire the lock",
    ["name"], buckets=(0.001, 0.01, 0.1, 1, 5, 30),
)
LOCK_HELD = Histogram(
    "lock_held_seconds", "How long the lock was held",
    ["name"], buckets=(0.1, 1, 5, 15, 30, 60, 300),
)
LOCK_LOST = Counter(
    "lock_lost_total", "Times the lock was lost mid-work",
    ["name"],
)
FENCE_REJECTS = Counter(
    "fence_token_rejected_total", "Writes rejected due to a stale token",
    ["resource"],
)
```

The metric that really matters is the ratio between lock_held_seconds and the TTL. If the p99 hold time approaches the TTL, you do not have a potential problem: you have a scheduled incident. The matching alert, for a 30-second TTL:

```
groups:
  - name: distributed-locks
    rules:
      - alert: LockHeldCloseToTTL
        expr: |
          histogram_quantile(
            0.99,
            sum by (name, le) (rate(lock_held_seconds_bucket[30m]))
          ) > 0.6 * 30
        for: 15m
        labels:
          severity: warning
        annotations:
          summary: "Lock is held close to its TTL"
          description: >-
            p99 hold time exceeds 60% of the TTL. Raise the TTL or shrink
            the guarded work before two owners show up.

      - alert: FencedWritesRejected
        expr: increase(fence_token_rejected_total[10m]) > 0
        labels:
          severity: critical
        annotations:
          summary: "A stale leader tried to write"
          description: >-
            Fencing did its job, but this confirms the two-owner window is
```

```
            real in this system. Investigate the pause.
```

And a logging detail that saves hours: log the fencing token on every line the guarded job emits. When a suspicious write turns up, the token immediately tells you which leader instance produced it and whether that leader was the current one.

Production checklist

- It is written down somewhere whether this lock is for efficiency or for correctness. If it is for correctness, there is a fencing token and the storage layer checks it.
- The TTL comes from the measured p99 of the guarded work, times at least three, and an alert fires when hold time approaches the TTL.
- Release is an atomic compare-and-delete, never an unconditional delete.
- The heartbeat renews at one third of the TTL and its failure reaches the work as a cancellation.
- The lock-service client has connect and socket timeouts smaller than the TTL.
- In PostgreSQL you use transaction-level advisory locks, not session-level, and blocking variants are preceded by SET LOCAL lock_timeout.
- Queues use FOR UPDATE SKIP LOCKED with the work outside the transaction, and a reaper rescues orphaned items.
- Leader election satisfies RetryPeriod < RenewDeadline < LeaseDuration, and OnStoppedLeading genuinely stops the work.
- A test exists that simulates losing the lock mid-work (by freezing the process or deleting the key by hand) and asserts there is no double write.
- Someone has asked, in writing, whether this lock could be replaced by a unique index, a version column, or partitioning by key.

Clocks: why a TTL measured against the wall clock is a broken promise

So far we have talked about the lock as if there were a shared clock between your process and the server holding the key. There is not. When you ask for a lock with a thirty-second TTL, the server starts counting thirty seconds on its clock, at the instant it processes the command. You start counting on your clock, at the instant the reply reaches you. Between those two instants there is a network round trip that might be two milliseconds or two seconds, and for the whole life of the lease those two clocks advance at slightly different rates.

The practical consequence is that the real safety margin of your lock is not the TTL: it is the TTL minus the round trip of the acquisition, minus the accumulated drift between the two clocks, minus however long it takes you to notice. Subtracting the full round trip rather than half of it is not gratuitous pessimism: you do not know whether the latency was symmetric, and the bad case is always the one where you assume you have more time than you do.

Then there is the question of which clock you use. If you measure elapsed time with the wall clock - time.time() in Python, System.currentTimeMillis() in Java, Date.now() in Node - you are measuring with a clock that can jump. NTP can step backwards when drift is large; an operator can correct the time by hand; a virtual machine restored from a snapshot wakes up with the snapshot time; a leap smear

spreads one second across twenty-four hours, and for that day your clock runs at 99.9988% speed. A backwards jump extends the lease you believe you hold. That is precisely the failure you are trying to avoid.

The monotonic clock does not jump. time.monotonic() on Linux is backed by CLOCK_MONOTONIC, which is unaffected by NTP adjustments or manual time changes. It has one limitation worth knowing: CLOCK_MONOTONIC does not count time the system spends suspended. If your workload runs on a virtual machine the hypervisor can pause, a lease measured with CLOCK_MONOTONIC under-counts on wake-up and you will believe you still have time left. CLOCK_BOOTTIME does count suspension, and in Python you reach it through time.clock_gettime(time.CLOCK_BOOTTIME). For a lock, that is the right choice anywhere suspension is possible.

```
import time, uuid
from dataclasses import dataclass

def _now() -> float:
    # CLOCK_BOOTTIME also counts suspended time; CLOCK_MONOTONIC does not.
    try:
        return time.clock_gettime(time.CLOCK_BOOTTIME)
    except (AttributeError, OSError):
        return time.monotonic()

@dataclass(frozen=True)
class Lease:
    key: str
    token: str          # owner identity, for CAS release and renewal
    fence: int          # monotonic token that travels to the data store
    ttl: float          # what the server promised to keep the key for
    granted_at: float   # monotonic clock read JUST AFTER the reply
    rtt: float          # measured round trip of the acquisition
    safety_margin: float = 1.0

    @property
    def deadline(self) -> float:
        # The server started counting before the reply arrived: subtract the
        # whole RTT, not half of it. We do not know it was symmetric.
        return self.granted_at + self.ttl - self.rtt - self.safety_margin

    def remaining(self) -> float:
        return self.deadline - _now()

    def good_for(self, seconds: float) -> bool:
        'True if there is enough headroom for an operation that long.'
        return self.remaining() > seconds


def acquire(redis, key: str, ttl: float = 30.0) -> Lease | None:
    token = uuid.uuid4().hex
    t0 = _now()
    ok = redis.set(key, token, nx=True, px=int(ttl * 1000))
    t1 = _now()
    if not ok:
        return None
    fence = redis.incr('fence:' + key)   # monotonic per key
    return Lease(key=key, token=token, fence=fence, ttl=ttl,
                 granted_at=t1, rtt=t1 - t0)
```

With that in hand a concept appears that almost nobody names and that decides whether the code is correct: the point of no return. Before every effectful operation - writing to the database, sending the email, charging the card - it is not enough to ask whether the lease is still alive. You have to ask whether there is enough headroom left to finish that operation. A lease with two hundred milliseconds remaining is an expired lease for a write that takes half a second.

```
PER_ITEM = 0.8   # measured p99 of processing one item, in seconds

for item in batch:
    if not lease.good_for(PER_ITEM):
        # We do not try to renew here: renewing does not give ownership back
        # if we already lost it. We stop and let someone else take the rest.
        log.warning('lease exhausted, %d items left', len(batch) - batch.index(item))
        break
    process(item, fence=lease.fence)
```

> *A rule that saves incidents: the TTL is not the time you have, it is the time the server promised you. The time you have is the TTL minus the RTT, minus the safety margin, minus however long the thing you are about to do takes. If that number is not comfortably positive, do not start.*

Renewing the lease does not close the window: watchdogs and conditional release

The natural reaction when the work outlives the TTL is to lengthen the TTL. The second natural reaction, which is better, is to renew it periodically from a background thread: the watchdog. Redisson does this by default with a lockWatchdogTimeout of thirty seconds renewed every ten; plenty of home-grown implementations do the same. It is worth being precise about what that buys and what it does not.

What it buys: a healthy process will not lose a lock simply because the work ran longer than expected. That is real and useful, and it avoids the anti-pattern of setting a one-hour TTL "just in case", which is the fastest way to turn a crash into a one-hour outage.

What it does not buy: safety. A paused process has a paused watchdog. While it is frozen nobody renews, the TTL expires, someone else acquires the lock and, when the first process wakes up, its watchdog does the first thing on its list: renew. If the renewal is a bare PEXPIRE, you have just extended somebody else lock. It is the same mistake as releasing with DEL without checking the token, and it has the same fix: compare and act atomically.

```
-- renew.lua: extend the TTL only if we are still the owner
-- KEYS[1] = lock key, ARGV[1] = our token, ARGV[2] = TTL in ms
if redis.call('GET', KEYS[1]) == ARGV[1] then
  return redis.call('PEXPIRE', KEYS[1], ARGV[2])
else
  return 0
end
```

The second important detail is what to do when the renewal returns zero. The answer you see in most code is "retry", and it is the wrong one: a zero means the key is no longer ours, and no amount of retrying brings it back. The correct move is to signal the loss and make sure the worker sees it before its next effectful operation.

```
import threading

class Watchdog(threading.Thread):
    'Renews the lease and reports its loss. It decides nothing on its own.'

    def __init__(self, redis, lease, renew_script):
        super().__init__(daemon=True)
        self.redis, self.lease, self.renew = redis, lease, renew_script
        self.lost = threading.Event()     # set if we lose ownership
```

```
        self.stop = threading.Event()     # set by the worker when done
        self.deadline = lease.deadline    # the only field the worker reads

    def run(self):
        interval = self.lease.ttl / 3.0   # three attempts before expiry
        while not self.stop.wait(interval):
            t0 = _now()
            try:
                ok = self.renew(keys=[self.lease.key],
                                args=[self.lease.token, int(self.lease.ttl * 1000)])
            except Exception:
                # A network failure does not prove we lost the lock, but it
                # does not prove the opposite: we do not move the deadline.
                continue
            if not ok:
                self.lost.set()            # confirmed loss: no retries
                return
            t1 = _now()
            self.deadline = t1 + self.lease.ttl - (t1 - t0) - self.lease.safety_margin
```

```
wd = Watchdog(redis, lease, renew_script); wd.start()
try:
    for item in batch:
        if wd.lost.is_set():
            raise LockLost(lease.key)
        if wd.deadline - _now() < PER_ITEM:
            break
        process(item, fence=lease.fence)
finally:
    wd.stop.set()
    release(redis, lease)     # CAS on the token, never a bare DEL
```

Notice what this design does not claim: it does not guarantee mutual exclusion. The watchdog lowers the probability that two processes overlap and, more usefully, lowers the time a lock is held longer than needed. The guarantee stays exactly where it was: in the fence that travels with every write and in the WHERE fence_token < $1 in the data store. If someone proposes dropping fencing because "we renew the lease now", they have just mistaken an optimisation for a proof.

The arithmetic of leader election in Kubernetes

The client-go leader election is probably the most widely running distributed lock in the industry, because every controller and every operator ships with one. It has three parameters and almost nobody touches them, which is fine until the day your controller changes leader every minute and nobody knows why.

- LeaseDuration (default 15 s): how long a non-leader candidate waits, counting from the last renewal it observed, before trying to take over.
- RenewDeadline (default 10 s): how long the leader has to successfully renew its Lease before giving up and calling OnStoppedLeading.
- RetryPeriod (default 2 s): how often acquisition or renewal is retried.

The invariant the library enforces is LeaseDuration > RenewDeadline > RetryPeriod * 1.2, and the defaults satisfy it with room to spare. What is interesting is what the gap between the first two means. At 15 and 10 seconds, the leader gives up five seconds before any other candidate dares to take over. Those five seconds are the entire safety margin of the system.

And that margin depends completely on the leader noticing it has failed and stopping. If the process is frozen for twenty seconds by a garbage collection pause or by cgroup throttling, it notices nothing at ten

seconds: it notices at twenty, by which point a successor has been working for five. It is exactly the two-owner window from the start of this article, wearing Kubernetes names.

From which comes the operational rule that matters most and is broken most often: OnStoppedLeading is not a place to "clean up and carry on". It is a place to exit the process. Any code that tries to gracefully shut down reconciliation goroutines is running, by definition, after the lease was lost, and every millisecond it takes is a millisecond of overlap.

```
lock := &resourcelock.LeaseLock{
    LeaseMeta:  metav1.ObjectMeta{Name: "my-controller", Namespace: ns},
    Client:     clientset.CoordinationV1(),
    LockConfig: resourcelock.ResourceLockConfig{Identity: podName},
}

leaderelection.RunOrDie(ctx, leaderelection.LeaderElectionConfig{
    Lock:            lock,
    ReleaseOnCancel: true,             // release on shutdown: handover in ~1s
    LeaseDuration:   15 * time.Second,
    RenewDeadline:   10 * time.Second,
    RetryPeriod:     2 * time.Second,
    Callbacks: leaderelection.LeaderCallbacks{
        OnStartedLeading: func(ctx context.Context) {
            fence := leaseTransitions(ctx, clientset, ns, "my-controller")
            reconcileLoop(ctx, fence)
        },
        OnStoppedLeading: func() {
            // We are no longer leader. Do not clean up: exit. K8s restarts us.
            klog.Error("lease lost; terminating the process")
            os.Exit(1)
        },
        OnNewLeader: func(id string) {
            if id != podName { klog.Infof("new leader: %s", id) }
        },
    },
})
```

That leaves fencing. client-go does not do it for you: it tells you whether you are the leader, it does not hand you a token the data store can reject. But the Lease object itself carries one for free. The spec.leaseTransitions field is a counter the algorithm increments every time the holder changes, and not when the same holder renews. That is: monotonic, stable for as long as you remain leader, and different for your successor. That is a fencing token in every meaningful sense.

```
func leaseTransitions(ctx context.Context, cs kubernetes.Interface, ns, name string) int32 {
    l, err := cs.CoordinationV1().Leases(ns).Get(ctx, name, metav1.GetOptions{})
    if err != nil || l.Spec.LeaseTransitions == nil {
        // No token, no writes. We would rather not start than start unfenced.
        klog.Fatalf("could not read leaseTransitions: %v", err)
    }
    return *l.Spec.LeaseTransitions
}
```

With that integer travelling all the way to the UPDATE ... WHERE fence_token < $1 we saw earlier, the controller moves from "I trust the lease to work" to "if I am wrong, the database stops me". For diagnosis, the metric you want is the leader transition rate: if leaseTransitions climbs several times an hour, the problem is not the lock but the apiserver, etcd latency or the pod CPU limits, and raising RetryPeriod and LeaseDuration is the right stopgap while you find the cause.

A Redis failover, second by second: losing a lock with nobody at fault

Every failure above needs something to go wrong: a pause, a skewed clock, a slow deploy. This one does not. It happens with two correct clients, perfect clocks, no expired TTL and a Redis cluster behaving exactly as designed.

- t = 0.000 s - Client A runs SET lock:account-42 tokenA NX PX 30000 against the primary. The primary writes the key in memory and replies OK.
- t = 0.001 s - A receives the OK and starts working. Redis replication is asynchronous: propagation to replicas is queued, not awaited.
- t = 0.010 s - The primary dies. Hardware fault, OOM kill, a cable. The key never reached a single replica.
- t = 5.2 s - Sentinel (or the cluster itself) declares the failure and promotes a replica. The promoted replica never saw lock:account-42.
- t = 5.3 s - Client B runs the same SET ... NX. As far as the new primary knows, the key does not exist. It replies OK.
- t = 5.3 s - A and B both believe they own the same lock, and will keep believing it for the next twenty-five seconds.

Nobody did anything wrong. It is the replication model. Which is why it pays to be precise about what the usual remedies actually fix.

WAIT numreplicas timeout blocks until a given number of replicas acknowledge the preceding writes. It narrows the window, and narrowing it has value. It does not close it, for three reasons: a replica acknowledgement does not mean durability (with appendfsync everysec a replica can acknowledge and lose it on restart); WAIT cannot force the failover to pick one of the replicas that acknowledged; and WAIT adds a round trip to every acquisition, which is exactly the cost you were trying to avoid by not using a consensus system in the first place.

The min-replicas-to-write and min-replicas-max-lag settings make the primary reject writes when it does not have enough up-to-date replicas. They also narrow the window and, better, they turn a partition into a visible error rather than a silent loss, which is a real operability improvement. They are still not a guarantee.

The honest conclusion is the one you already knew, now with an example that rests on no debatable assumption: if the lock store replicates asynchronously, mutual exclusion is not a property of the lock. It is a property of the fencing. Redis is a perfectly reasonable choice for coordination - it is fast, operationally simple and almost everyone already has one - as long as the correctness argument is written in the WHERE clause of your writes and not in the README of your Redis client.

Two more backends that work: DynamoDB and Consul sessions

Redis, PostgreSQL and etcd are not the only options, and sometimes they are not the ones you have. Two alternatives deserve a place on the map because they solve the problem with different primitives.

DynamoDB offers exactly what is needed: atomic conditional writes, strongly consistent reads and atomic counters. A lock is an item with an owner, an expiry and a fencing counter incremented in the very operation that acquires it. The reference library from AWS, the DynamoDB Lock Client, has run on this model inside Amazon for years.

```
import time, uuid, boto3
from botocore.exceptions import ClientError

def acquire(table, key: str, owner: str, ttl_s: int = 30):
    'Returns the fencing token on success, None if the lock is held.'
    now_ms = int(time.time() * 1000)
    try:
        resp = table.update_item(
            Key={'lock_key': key},
            UpdateExpression=(
                'SET #own = :owner, lease_expiry = :exp, rvn = :rvn ADD fence :one'
            ),
            # Acquire if absent, if it is already us, or if it expired.
            ConditionExpression=(
                'attribute_not_exists(lock_key) OR #own = :owner '
                'OR lease_expiry < :now'
            ),
            ExpressionAttributeNames={'#own': 'owner'},
            ExpressionAttributeValues={
                ':owner': owner,
                ':exp': now_ms + ttl_s * 1000,
                ':rvn': uuid.uuid4().hex,   # changes on every heartbeat
                ':one': 1,
                ':now': now_ms,
            },
            ReturnValues='UPDATED_NEW',
        )
        return int(resp['Attributes']['fence'])
    except ClientError as e:
        if e.response['Error']['Code'] == 'ConditionalCheckFailedException':
            return None
        raise
```

There is one detail in that code worth staring at, because it is the clock problem from the previous section in disguise: lease_expiry < :now is evaluated on the server, but :now is supplied by the client from its own wall clock. A client whose clock runs fast steals locks that have not expired. The DynamoDB Lock Client sidesteps this entirely with an elegant trick: instead of comparing timestamps, it reads the item, notes the rvn (the identifier that changes on every heartbeat), waits locally for a full lease duration measured on its own monotonic clock, and reads again. If the rvn did not change, the previous owner has gone a whole lease without a sign of life and the lock is considered abandoned. No clock is ever compared against another.

Consul contributes a piece none of the others has out of the box: lock-delay. When a session is invalidated - by TTL expiry or by a failing health check - Consul refuses to grant the key to anyone for a configurable interval, fifteen seconds by default. It is a safety margin built into the server: precisely the window in which the previous owner might still believe it is the owner. It is not a proof of correctness, but it is the only popular implementation that acknowledges the problem in its design rather than leaving it to you.

```
# Session with a TTL, automatic key deletion on invalidation, and lock-delay
SESSION=$(curl -s -X PUT http://consul:8500/v1/session/create -d '{
  "Name": "daily-close", "TTL": "30s",
  "Behavior": "delete", "LockDelay": "15s"
}' | jq -r .ID)

# Acquisition: returns true or false
curl -s -X PUT "http://consul:8500/v1/kv/locks/close?acquire=$SESSION" -d 'worker-3'

# TTL renewal (this is the watchdog)
curl -s -X PUT "http://consul:8500/v1/session/renew/$SESSION"

# The key ModifyIndex is cluster-wide monotonic: use it as the fence
curl -s "http://consul:8500/v1/kv/locks/close" | jq '.[0].ModifyIndex'
```

Consul ModifyIndex and Kubernetes leaseTransitions are the same gift: a monotonic integer the server already maintains that you can use as a fencing token without writing an extra line. When you evaluate a lock backend, that is a good question for the list: does it give me a number that grows? If the answer is no, you will have to manufacture one, and manufacturing one properly is more work than it looks.

Scheduled jobs: the cron that runs twice and the CronJob that never runs at all

The request for a distributed lock that most often reaches a design review does not come from an exotic system: it comes from a cron. Someone put a crontab entry on three machines for high availability and discovered the monthly report goes out three times. The ticket says "we need a distributed lock". The answer is almost always better than that.

In Kubernetes, CronJob appears to solve it with concurrencyPolicy: Forbid, and it is worth reading carefully what that promises. Forbid means the controller will not create a new Job while the previous one has not finished. It does not mean two pods cannot be running your code: a Job with retries can start a fresh pod, and a partitioned node can have an old pod still alive while the control plane has written it off. It is a scheduling policy, not mutual exclusion.

There is also an operational trap that bites a lot of people. If the controller accumulates more than a hundred missed schedules and startingDeadlineSeconds is unset, it stops scheduling that CronJob entirely and writes an event nobody reads. A CronJob that runs every minute reaches a hundred missed schedules in one hour and forty minutes: the length of a cluster upgrade, or of leaving the CronJob suspended for a while. The symptom is a job that simply stops existing. Setting startingDeadlineSeconds avoids that state and, along the way, expresses something useful: how late this job can be before skipping it is the better outcome.

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: daily-close
spec:
  schedule: "0 2 * * *"
  timeZone: "Europe/Madrid"        # schedule is local; DST is honoured
  concurrencyPolicy: Forbid
  startingDeadlineSeconds: 300      # without this, 100 misses stop scheduling
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 5
  jobTemplate:
    spec:
      backoffLimit: 2
      activeDeadlineSeconds: 3600   # a real upper bound on duration
      template:
        spec:
          restartPolicy: Never
          terminationGracePeriodSeconds: 60
          containers:
            - name: job
              image: registry.internal/close:1.4.2
```

And now the part that actually solves the problem, which is not the YAML. A scheduled job has a property a generic lock cannot exploit: it has an identity. It is not "the close", it is "the close for 18 September 2026". As soon as that identity is explicit, mutual exclusion turns into a uniqueness constraint - the first of the five lock-free designs we saw earlier.

```
-- The run registry is the lock and the idempotency key at the same time.
CREATE TABLE job_runs (
    job_name      text        NOT NULL,
    scheduled_for date        NOT NULL,   -- business day, not a timestamp
    started_at    timestamptz NOT NULL DEFAULT now(),
    finished_at   timestamptz,
    PRIMARY KEY (job_name, scheduled_for)
);
```

```
import sys, psycopg

with psycopg.connect(DSN) as conn:
    with conn.transaction():
        row = conn.execute(
            'INSERT INTO job_runs (job_name, scheduled_for) VALUES (%s, %s) '
            'ON CONFLICT DO NOTHING RETURNING started_at',
            ('daily_close', business_day),
        ).fetchone()
        if row is None:
            print('already ran for', business_day, '- exiting')
            sys.exit(0)

        run_close(business_day)

        conn.execute(
            'UPDATE job_runs SET finished_at = now() '
            'WHERE job_name = %s AND scheduled_for = %s',
            ('daily_close', business_day),
        )
```

It is worth pausing on what PostgreSQL is doing here, because it is prettier than it looks. If two pods start at once, the first inserts the row and keeps the transaction open while it works. The second tries to insert, collides with the unique index on a row that is not yet committed, and blocks waiting: it does not get an error, it waits. When the first commits, the second ON CONFLICT DO NOTHING sees the row, returns zero rows, and the process exits cleanly. If the first fails and rolls back, the second gets to insert and does the work. No explicit lock, no TTL, no fencing and not one line about clocks: mutual exclusion and correct retry fall out of the same index.

The cost of this pattern is that the transaction lasts as long as the work, which is fine for a thirty-second close and a serious problem for a two-hour one - a long transaction blocks dead tuple cleanup across the whole database. For long jobs, the variant is to commit the row immediately with a running status and a heartbeat column, and let the second process decide whether to adopt the job based on the age of that heartbeat. At that moment you have rebuilt a lease, this time knowing exactly why you need one.

Granularity and cost: what a lock actually charges you

A lock is a serialisation point, and serialisation points have unforgiving arithmetic. If the critical section takes two hundred milliseconds and twenty workers contend for the same key, the ceiling of the system is five operations per second. Adding workers does not raise that number: it raises the queue. It is Amdahl law applied to something nobody measures, because it does not look like expensive code.

On top of that sits the fixed cost. An acquire and a release are at least two round trips to the lock store, plus renewal traffic if there is a watchdog. Against a one-millisecond Redis, a two-millisecond critical section now costs four. If your critical section is shorter than the round trip to the lock server, you are paying more to coordinate than to work, and the fix is not a faster lock: it is pushing the work into the store - a single conditional UPDATE, an INCR, a uniqueness constraint - and ending up with no critical section at all.

The three metrics that make all of this visible are easy to instrument and rare to find on a dashboard:

- Wait time (from asking for the lock to holding it). If its p99 climbs while the p50 stays flat, you have queue contention, not a slow system.
- Hold time (from holding it to releasing it). It is the numerator of your throughput ceiling. Halving it doubles capacity; no other change has that property.
- Contention ratio (acquisitions that had to wait / total acquisitions). Below 5% the lock is essentially free. Above 50% the lock is your architecture, and that should be a deliberate decision.

When the contention ratio climbs, the right lever is almost never a faster backend: it is splitting the key. A global reports lock becomes reports:{tenant_id} and contention divides by the number of active tenants. A global inventory lock becomes one per SKU. The limit of this technique is the point where the key no longer corresponds to a real isolation unit in the business; past that point, splitting further only moves the corruption somewhere else.

There is one last failure mode that deserves its own name because it arrives all at once: the convoy. If hold time approaches the mean interval between requests, the queue stops draining and grows without bound. Workers sit waiting, connection pools drain, upstream requests time out and the whole system falls over because of a lock that "only takes two hundred milliseconds". The defence is a bounded maximum wait and an honest answer when it runs out: not now, rather than waiting forever. It is the same idea as load shedding, applied to coordination.

How to prove your lock is broken before production proves it

Most lock tests people write check the thing that does not need checking: that two threads on one machine do not enter at the same time. That always works and says nothing about the real failures, which need a frozen process, a partitioned network or a failover. Three tests are worth writing, and they are easier than they sound.

The first and most important one does not test the lock: it tests the fencing. And the assertion you want is not "only one writer got through", which is unprovable in a single run, but "the store rejected the stale writer".

```
def test_stale_writer_is_rejected(redis, pg):
    a = acquire(redis, 'account:42', ttl=2.0)
    assert a is not None

    time.sleep(2.5)                 # A freezes; its lease expires

    b = acquire(redis, 'account:42', ttl=30.0)
    assert b is not None
    assert b.fence > a.fence        # the token grows: nothing works without it

    assert write_with_fence(pg, 'account:42', b.fence, balance=100) is True
    # A wakes up and writes with its old token. It must bounce.
    assert write_with_fence(pg, 'account:42', a.fence, balance=999) is False
    assert read_balance(pg, 'account:42') == 100
```

```
-- write_with_fence: the one line that guarantees correctness
UPDATE accounts
   SET balance = $3, fence_token = $2
 WHERE id = $1
   AND fence_token < $2;
-- rowcount == 0  =>  we are late: someone newer already wrote
```

The second test freezes a real process. SIGSTOP is the cheapest way to simulate a garbage collection pause or cgroup throttling: the process stops executing instructions but keeps its connections and its state, which is exactly what makes a pause dangerous.

```
import os, signal, subprocess, sys, time

def test_frozen_process_does_not_corrupt(pg):
    worker = subprocess.Popen([sys.executable, 'worker.py', '--ttl', '3'])
    time.sleep(1)                      # let it acquire and start

    os.kill(worker.pid, signal.SIGSTOP)   # frozen: the watchdog too
    time.sleep(4)                         # the lease expires while it sleeps

    second = subprocess.Popen([sys.executable, 'worker.py', '--ttl', '30'])
    time.sleep(2)                         # the second one acquires and writes

    os.kill(worker.pid, signal.SIGCONT)   # wakes up believing it is the owner
    worker.wait(timeout=30)

    # We assert not that it did not write, but that it did not CORRUPT.
    assert business_invariants_hold(pg)
    assert count_rejected_writes(pg) >= 1   # the fence did its job
    second.terminate()
```

The third one cuts the network between the worker and the lock store without touching anything else. With Testcontainers and Toxiproxy in front of Redis, adding a timeout toxic leaves connections hanging exactly like a real partition, which is different from a connection refused: in a partition you do not get an error, you get nothing. That is the case that reveals whether your code treats "I do not know if I still hold the lock" as "I still hold the lock", which is the most common default mistake of all.

These three tests run in seconds and fit in CI. Their value is not in the first run - they usually pass - but in the twentieth, when someone simplifies the release path to a DEL because "the Lua script was unnecessary" and the test stops it before it reaches production. A distributed lock is code that almost never fails and, when it does, fails silently with money on the line. It deserves a safety net that shouts.

FAQ

Can I just use an in-memory lock if I only run one replica? Yes, and you should: a threading.Lock is correct, fast, and has no window. The problem is that "only one replica" stops being true without warning, usually during a RollingUpdate deploy. If correctness depends on there being a single replica, write it into the manifest with Recreate and into a comment.

Does ZooKeeper still make sense in 2026? It does if you already operate it. Its ephemeral sequential znodes give you leases and monotonic tokens with very clean semantics, but if you are starting fresh and you are on Kubernetes, native Leases or etcd give you the same thing without another system to maintain.

Does the fencing token have to be an integer? It has to be totally ordered and monotonic. A sequence integer, an etcd revision, or leaseTransitions all qualify. A UUID v4 does not. Neither does a timestamp: the clock is precisely what we do not want making this decision.

What TTL do I pick if the job takes a wildly variable amount of time? Do not pick a huge TTL; pick a moderate one and renew it. A two-hour TTL means a dead worker blocks the system for two hours. A thirty-second TTL with a heartbeat gives you both: fast recovery from crashes and unlimited duration while the process is alive.

What if my storage does not support conditional writes? Then you cannot fence there, and you should say so explicitly. Most storage can: PostgreSQL with a WHERE clause, S3 with If-Match on the ETag, DynamoDB with condition expressions, Redis with Lua. If it truly cannot, move the decision into a system that can and treat the other one as derived data.

Are distributed locks the answer for transactions across services? No. That is the distributed consistency problem, and it is solved with sagas, a transactional outbox and idempotency, not by holding a lock open while you wait on another service. A lock that spans a network call is an elegant way to build a distributed deadlock.

Glossary

- Lease. A lock with an expiry granted by a service: the holder is in charge while it keeps renewing, and loses it automatically when it stops.
- Fencing token. A strictly increasing integer handed out on each lock grant, which storage uses to reject writes from expired holders.
- Advisory lock. A PostgreSQL lock over an arbitrary integer chosen by the application, unrelated to any row. Comes in session and transaction flavours.
- SKIP LOCKED. A FOR UPDATE modifier that skips rows already locked by another transaction instead of waiting on them. The basis of PostgreSQL queues.
- Redlock. A distributed lock algorithm over several independent Redis instances using majorities. Fine for efficiency, contested for correctness.
- Split brain. The situation where two nodes simultaneously believe they are the leader and both write.
- Stop-the-world. A pause in which the garbage collector halts every application thread. The most common cause of the two-owner window.
- Quorum. The majority of nodes required to make a decision in a replicated system; the basis of etcd, ZooKeeper and Raft in general.

Closing thought

The idea worth taking away is uncomfortable and rather liberating: a distributed lock does not guarantee mutual exclusion, it only makes it likely. If your system survives two processes doing the same work, you are done and one Redis instance is more than enough. If it does not survive, no lock will save you on its own, and the guarantee has to live where the data lives: in a write condition that storage evaluates atomically. Everything else -the number of Redis replicas, the sophistication of the algorithm, the elegance of the decorator- is noise around that one decision.