gRPC en Producción: Balanceo Real sobre HTTP/2, Deadlines Propagados, Keepalive y Evolución de Protobuf sin Romper Clientes

gRPC in Production: Real Load Balancing over HTTP/2, Propagated Deadlines, Keepalive, and Evolving Protobuf Without Breaking Clients

---

Guía práctica de lo que separa un gRPC que funciona en el portátil de uno que aguanta producción: por qué un Service ClusterIP deja tres pods al 90% de CPU y nueve al 4% porque HTTP/2 reparte conexiones y no peticiones, los cuatro modelos de balanceo (capa 4, proxy L7, cliente con Service headless y round_robin, y xDS sin proxy) con el detalle del resolver DNS perezoso que hace invisible cada escalado hasta que configuras MaxConnectionAge en el servidor, deadlines frente a timeouts y cómo propagar el presupuesto restante reservando margen (con la trampa de context.Background() y el trabajo huérfano), qué reintentar de los 17 códigos de estado y por qué reintentar DEADLINE_EXCEEDED amplifica una sobrecarga, política de reintentos declarativa con retryThrottling como cortacircuitos, keepalive coherente entre cliente y servidor para no provocar GOAWAY con ENHANCE_YOUR_CALM, backpressure rea...

Autor / Author: Josue Puig
Tiempo de lectura / Reading time: ~50 min
Edicion / Edition: 11 de septiembre de 2026
Idiomas / Languages: Espanol + English
puigflows.com/recursos

El problema que no aparece hasta que escalas

La historia se repite en casi todos los equipos que migran a gRPC. Cambias un endpoint REST interno por una llamada gRPC, mides, y la latencia baja de 40 ms a 9 ms. Todo el mundo contento. Dos semanas después subes el servicio a doce réplicas, el tráfico se duplica, y abres el dashboard: tres pods al 90% de CPU y nueve al 4%. El autoescalado añade réplicas que no reciben nada. Reinicias pods y la carga salta a otros tres. Nadie ha desplegado un bug.

No es un bug. Es la consecuencia directa de que gRPC vive sobre HTTP/2, y HTTP/2 está diseñado para multiplexar muchas peticiones sobre una sola conexión TCP de vida larga. Un Service de Kubernetes de tipo ClusterIP balancea en capa 4: reparte conexiones, no peticiones. Tu cliente abre una conexión, esa conexión aterriza en un pod, y todas las peticiones de ese cliente durante las próximas horas van a ese pod.

La aritmética es brutal y conviene hacerla explícita. Si tienes 4 pods cliente y cada uno mantiene un único subcanal, como máximo 4 de tus 12 pods servidores reciben tráfico. Los otros 8 existen, pasan las probes, cuentan en la factura y no hacen nada. Con HTTP/1.1 esto no ocurría porque cada petición podía tomar una conexión distinta del pool y el reparto en capa 4 era una aproximación razonable. Con HTTP/2 esa aproximación se rompe.

Esta guía cubre las decisiones que separan un gRPC que funciona en el portátil de uno que aguanta producción: cómo balancear de verdad, cómo propagar deadlines para que un fallo no se convierta en una cascada, qué reintentar y qué no, keepalive sin provocar desconexiones, streaming con backpressure real, evolución del esquema sin romper clientes, y apagado limpio para que cada despliegue no genere un pico de errores.

Los cuatro modelos de balanceo

Hay exactamente cuatro formas de repartir tráfico gRPC, y elegir mal es el error más caro de esta lista.

1. Service ClusterIP en capa 4 (lo que casi todos tienen por defecto)

Funciona para HTTP/1.1 y está roto para gRPC por lo que acabamos de ver. Solo da un reparto aceptable si las conexiones se reciclan a menudo, lo que ocurre por accidente (despliegues, timeouts de NAT) y no por diseño. Si tu servicio gRPC está detrás de un ClusterIP, empieza por aquí.

2. Proxy de capa 7

Un proxy que entiende HTTP/2 (Envoy, nginx con grpc_pass, un sidecar de Istio o Linkerd, o un Gateway con appProtocol: kubernetes.io/h2c) termina la conexión del cliente y reparte cada petición entre los backends. Es correcto, te da mTLS y política de tráfico, y cuesta un salto de red extra más un componente que operar. Si ya tienes malla de servicios, esta discusión está resuelta.

3. Balanceo en el cliente con Service headless

La opción con mejor relación resultado/complejidad cuando no hay malla. Declaras el Service sin ClusterIP, el DNS devuelve todas las IPs de pod, y la librería gRPC abre un subcanal a cada una y reparte en round robin.

```
apiVersion: v1
kind: Service
metadata:
  name: orders-headless
spec:
  clusterIP: None          # esto es lo que hace que el DNS devuelva IPs de pod
  selector:
    app: orders
  ports:
    - name: grpc
      port: 50051
      targetPort: 50051
      appProtocol: kubernetes.io/h2c
```

En el cliente hay que pedir explícitamente el resolver DNS y la política round robin. El prefijo dns:/// no es decorativo: sin él, varias implementaciones se quedan con la primera dirección que devuelve la resolución.

```
// Go: cliente con balanceo en el cliente sobre un Service headless
conn, err := grpc.NewClient(
    "dns:///orders-headless.default.svc.cluster.local:50051",
    grpc.WithTransportCredentials(insecure.NewCredentials()),
    grpc.WithDefaultServiceConfig("{\"loadBalancingConfig\":[{\"round_robin\":{}}]}"),
)
if err != nil {
    return err
}
defer conn.Close()
client := pb.NewOrdersClient(conn)
```

```
# Python: mismo patrón con grpcio
channel = grpc.insecure_channel(
    "dns:///orders-headless.default.svc.cluster.local:50051",
    options=[
        ("grpc.lb_policy_name", "round_robin"),
        # no re-resolver más de una vez cada 10 s ante fallos en ráfaga
        ("grpc.dns_min_time_between_resolutions_ms", 10000),
    ],
)
stub = orders_pb2_grpc.OrdersStub(channel)
```

La trampa que arruina este montaje. El resolver DNS de gRPC no consulta el DNS en bucle: resuelve al conectar y vuelve a resolver cuando un subcanal falla, con un intervalo mínimo entre resoluciones y un refresco perezoso que en grpc-go ronda los 30 minutos. Escalas de 12 a 20 pods, los 8 nuevos pasan las probes y no reciben una sola petición hasta que alguna conexión existente se rompa. El arreglo no está en el cliente: está en el servidor.

```
// Go: forzar rotación de conexiones para que los clientes re-resuelvan
srv := grpc.NewServer(
    grpc.KeepaliveParams(keepalive.ServerParameters{
        // cada conexión vive 30 min y luego recibe GOAWAY (jitter +/-10%)
        MaxConnectionAge:      30 * time.Minute,
        // margen para que las peticiones en vuelo terminen
        MaxConnectionAgeGrace: 10 * time.Second,
```

```
        MaxConnectionIdle:     5 * time.Minute,
        Time:                  2 * time.Hour,
        Timeout:               20 * time.Second,
    }),
)
```

Con MaxConnectionAge el servidor envía un GOAWAY, el cliente cierra ordenadamente, re-resuelve el DNS y descubre los pods nuevos. El jitter del +/-10% que aplica gRPC existe precisamente para que 300 conexiones creadas en el mismo despliegue no caduquen todas en el mismo segundo. Es una línea de configuración y es la diferencia entre un autoescalado que sirve de algo y uno decorativo.

4. xDS sin proxy

El cliente gRPC habla el protocolo xDS con un plano de control (Istio, Traffic Director o un servidor xDS propio) y recibe la lista de endpoints y la política de balanceo sin sidecar en el camino. Da balanceo por petición, reparto consciente de la zona, detección de instancias degradadas y despliegues canary sin salto extra. El coste es operar un plano de control y depender del soporte de xDS de tu lenguaje, que es sólido en Go, Java y C-core y más limitado en el resto. Es la respuesta correcta a gran escala y una complejidad injustificada con cinco servicios.

Deadlines: el parámetro que casi nadie propaga

gRPC no tiene timeouts, tiene deadlines. La diferencia no es semántica. Un timeout es relativo ("falla si tardas más de 2 s") y cada salto de la cadena lo reinicia. Un deadline es un instante absoluto que viaja en la cabecera grpc-timeout y que todos los servicios de la cadena comparten. Esa distinción es lo que evita que una petición de usuario con un presupuesto de 2 segundos genere trabajo en cinco servicios durante 30 segundos.

El valor por defecto es el peor posible: sin deadline. Una llamada sin deadline puede quedarse colgada indefinidamente ocupando un hueco en tu pool de workers. Diez llamadas así y el servicio deja de aceptar tráfico sin que ninguna métrica de CPU se mueva.

El presupuesto y cómo repartirlo

La regla práctica: cada handler calcula cuánto presupuesto le queda, reserva un margen para serializar la respuesta y devolverla, y reparte el resto entre sus dependencias. Si al entrar ya no queda presupuesto, falla inmediatamente en lugar de llamar a tres servicios con 4 ms.

```
# Python: cálculo de presupuesto y propagación explícita
import grpc

RESERVA_S = 0.05          # margen para serializar y responder

def presupuesto(context: grpc.ServicerContext, por_defecto: float = 2.0) -> float:
    """Segundos utilizables que quedan del deadline entrante."""
    restante = context.time_remaining()   # None si el cliente no puso deadline
    if restante is None:
        return por_defecto
    return max(0.0, restante - RESERVA_S)


class OrdersService(orders_pb2_grpc.OrdersServicer):
    def GetOrder(self, request, context):
        budget = presupuesto(context)
        if budget <= 0.02:
```

```
            # fallar rápido es más útil que propagar una llamada condenada
            context.abort(
                grpc.StatusCode.DEADLINE_EXCEEDED,
                "sin presupuesto al entrar al handler",
            )

        # dos dependencias secuenciales: la mitad del presupuesto para cada una
        stock = self.inventory.CheckStock(
            inventory_pb2.StockRequest(sku=request.sku),
            timeout=budget * 0.5,
        )
        precio = self.pricing.Quote(
            pricing_pb2.QuoteRequest(sku=request.sku, qty=request.quantity),
            timeout=budget * 0.5,
        )
        return orders_pb2.Order(sku=request.sku, disponible=stock.available, total=precio.total)
```

En Python la propagación es manual: si te olvidas del argumento timeout=, la llamada saliente no hereda nada. En Go la propagación es casi automática porque el deadline entrante ya está instalado en el context.Context, y context.WithTimeout nunca extiende un deadline: se queda con el más cercano de los dos.

```
// Go: el ctx entrante ya trae el deadline; WithTimeout solo puede acortarlo
func (s *server) GetOrder(ctx context.Context, req *pb.GetOrderRequest) (*pb.Order, error) {
    if dl, ok := ctx.Deadline(); ok && time.Until(dl) < 50*time.Millisecond {
        return nil, status.Error(codes.DeadlineExceeded, "presupuesto agotado antes de empezar")
    }

    // 200 ms como techo propio, pero si al ctx le quedan 80 ms manda el ctx
    childCtx, cancel := context.WithTimeout(ctx, 200*time.Millisecond)
    defer cancel()

    stock, err := s.inventory.CheckStock(childCtx, &pb.StockRequest{Sku: req.Sku})
    if err != nil {
        return nil, err
    }
    return &pb.Order{Sku: req.Sku, Available: stock.Available}, nil
}
```

El error que convierte esto en nada. Usar context.Background() dentro de un handler porque "la llamada de fondo no debe cancelarse". Lo que consigues es trabajo huérfano: el cliente ya se fue, nadie va a leer el resultado, y tu servicio sigue consultando la base de datos. Bajo sobrecarga esto es exactamente lo contrario de lo que necesitas. Si de verdad hay trabajo que debe sobrevivir a la petición, va a una cola, no a una goroutine con contexto desvinculado.

En bucles largos, comprueba la cancelación en cada iteración. Un handler que procesa 5.000 filas y no mira ctx.Err() seguirá procesándolas cuando ya no importe:

```
for i, row := range rows {
    if err := ctx.Err(); err != nil {
        return nil, status.FromContextError(err).Err()   // Canceled o DeadlineExceeded
    }
    if err := s.process(ctx, row); err != nil {
        return nil, err
    }
    _ = i
}
```

Qué reintentar y qué no

gRPC tiene 17 códigos de estado y tratarlos todos igual es la forma más rápida de convertir una degradación en una caída. Los que importan para decidir un reintento:

- UNAVAILABLE: la conexión falló o el servidor no está aceptando. La petición probablemente no llegó a ejecutar lógica. Es el caso reintentable por excelencia.
- DEADLINE_EXCEEDED: no lo reintentes a ciegas. No sabes si el servidor terminó el trabajo y la respuesta se perdió, y además ya no te queda presupuesto. Reintentar aquí es añadir carga a un sistema que precisamente va lento.
- RESOURCE_EXHAUSTED: el servidor te está diciendo que pares. Reintentable solo con backoff generoso, y nunca de forma agresiva.
- INTERNAL y UNKNOWN: hay un bug. Reintentar da el mismo error con más coste.
- FAILED_PRECONDITION y INVALID_ARGUMENT: el estado o los datos están mal. Reintentar es inútil por definición.
- ABORTED: conflicto de concurrencia. Reintentable, pero la lógica de reintento pertenece a la aplicación, no al transporte.

gRPC implementa reintentos en el cliente vía service config, sin que tengas que escribir bucles a mano:

```
{
  "methodConfig": [
    {
      "name": [{ "service": "orders.v1.Orders", "method": "GetOrder" }],
      "timeout": "1.5s",
      "retryPolicy": {
        "maxAttempts": 4,
        "initialBackoff": "0.1s",
        "maxBackoff": "1s",
        "backoffMultiplier": 2,
        "retryableStatusCodes": ["UNAVAILABLE", "RESOURCE_EXHAUSTED"]
      }
    }
  ],
  "retryThrottling": { "maxTokens": 100, "tokenRatio": 0.1 }
}
```

Tres detalles que cambian el comportamiento y no son obvios:

- maxAttempts cuenta el intento original, y los clientes aplican por defecto un techo de 5 aunque pongas 20.
- gRPC ya añade jitter (del orden de +/-20%) al backoff. No necesitas implementarlo tú.
- Todos los intentos comparten el mismo deadline. Si el timeout del método es 1,5 s, cuatro intentos con backoff caben dentro de ese 1,5 s o simplemente no ocurren. Esto es lo correcto y es lo que impide que los reintentos multipliquen la latencia de cola.

retryThrottling es la pieza que más gente ignora y la que evita la tormenta de reintentos. Funciona como un token bucket por canal: cada fallo consume un token, cada éxito devuelve tokenRatio, y cuando el saldo baja de la mitad de maxTokens el cliente deja de reintentar hasta que el servidor se recupere. Es un cortacircuitos específico para reintentos, gratis y declarativo.

El hedging (enviar copias de la misma petición a varios backends y quedarse con la primera respuesta) es la otra herramienta disponible. Reduce la latencia p99 de forma espectacular y multiplica la carga:

úsalo solo en lecturas idempotentes y con holgura de capacidad demostrada. El deadline se aplica a toda la cadena de peticiones, así que no te salva de un sistema saturado.

Keepalive: el ajuste que provoca ENHANCE_YOUR_CALM

Una conexión HTTP/2 puede estar minutos sin tráfico. Entre tu cliente y tu servidor hay NAT, balanceadores y firewalls que cierran conexiones inactivas sin avisar a nadie: un NLB de AWS las corta a los 350 segundos, y muchos NAT domésticos y de nube lo hacen antes. El resultado es el clásico "la primera petición de la mañana siempre falla": la conexión está muerta hace horas y el cliente no lo sabe hasta que intenta escribir.

Keepalive resuelve eso enviando PINGs de HTTP/2 sobre la conexión:

```
// Go: keepalive en el cliente
conn, err := grpc.NewClient(target,
    grpc.WithTransportCredentials(creds),
    grpc.WithKeepaliveParams(keepalive.ClientParameters{
        Time:                30 * time.Second, // ping tras 30 s sin actividad
        Timeout:             10 * time.Second, // si no hay ACK en 10 s, conexión muerta
        PermitWithoutStream: true,             // también sin RPCs activas
    }),
)
```

```
# Python: mismas opciones por nombre de canal
channel = grpc.insecure_channel(target, options=[
    ("grpc.keepalive_time_ms", 30000),
    ("grpc.keepalive_timeout_ms", 10000),
    ("grpc.keepalive_permit_without_calls", 1),
    ("grpc.http2.max_pings_without_data", 0),   # 0 = sin límite
])
```

Y aquí está la trampa. El servidor tiene una política de defensa contra clientes abusivos. Si tu cliente hace ping cada 30 segundos y el servidor tiene MinTime en 5 minutos (el valor por defecto en grpc-go), el servidor considera el ping un abuso, envía GOAWAY con el código ENHANCE_YOUR_CALM y cierra la conexión. Desde la aplicación eso se ve como errores UNAVAILABLE intermitentes y sin patrón, y se diagnostica fatal porque nadie mira los frames de HTTP/2.

Keepalive se configura en los dos lados o no se configura:

```
srv := grpc.NewServer(
    grpc.KeepaliveEnforcementPolicy(keepalive.EnforcementPolicy{
        MinTime:             20 * time.Second, // por debajo del Time del cliente
        PermitWithoutStream: true,
    }),
    grpc.KeepaliveParams(keepalive.ServerParameters{
        Time:                  60 * time.Second,
        Timeout:               20 * time.Second,
        MaxConnectionAge:      30 * time.Minute,
        MaxConnectionAgeGrace: 10 * time.Second,
    }),
)
```

La regla es sencilla: MinTime del servidor estrictamente menor que Time del cliente, y Time del cliente por debajo del timeout de inactividad del elemento de red más restrictivo del camino.

Streaming: dónde vive el backpressure

HTTP/2 tiene control de flujo por stream y por conexión: el receptor anuncia una ventana y el emisor no puede escribir más allá de ella. Eso significa que gRPC ya te da backpressure gratis, siempre que no lo destruyas.

La forma de destruirlo es casi siempre la misma: leer del stream lo más rápido posible hacia un canal o una cola en memoria sin límite, porque "así no bloqueamos el stream". Con eso conviertes una presión que el protocolo manejaba en un crecimiento de heap que termina en OOMKilled. Si el consumidor es lento, lo correcto es que Send bloquee.

```
// Go: streaming de servidor que respeta cancelación y control de flujo
func (s *server) StreamEvents(req *pb.StreamRequest, stream pb.Events_StreamEventsServer) error
{
    ctx := stream.Context()   // se cancela si el cliente se va o expira el deadline

    for ev := range s.source.Subscribe(ctx, req.Topic) {
        // Send bloquea si la ventana de HTTP/2 está llena: eso es backpressure correcto
        if err := stream.Send(ev); err != nil {
            return err       // cliente desconectado o stream roto
        }
        if err := ctx.Err(); err != nil {
            return status.FromContextError(err).Err()
        }
    }
    return nil
}
```

Dos límites que conviene conocer antes de que te los enseñe producción:

- Tamaño máximo de mensaje. Por defecto son 4 MiB en recepción. Subirlo a 100 MiB porque "un cliente manda un lote grande" convierte tu servidor en un objetivo de agotamiento de memoria: cada petición concurrente reserva ese espacio. La respuesta correcta a un mensaje de 100 MiB es paginarlo o pasarlo por almacenamiento de objetos con una URL prefirmada, no subir el límite.
- Los streams no se rebalancean. Un stream de vida larga queda fijado al pod donde se estableció, para siempre. Si tienes 200 streams y despliegas, esos 200 se cortan. Un cliente de streaming en producción necesita reconexión con backoff y jitter, y un cursor para reanudar sin perder ni duplicar eventos. Si eso te suena a trabajo, es porque lo es: cuando el caso lo permite, unary con polling o con long-polling sale más barato de operar.

Evolución del esquema: el contrato es para siempre

La ventaja real de gRPC no es el ahorro de bytes: es que el contrato está escrito, versionado y genera código en todos los lenguajes. Esa ventaja se pierde el día que alguien edita un .proto sin entender las reglas de compatibilidad, porque Protobuf no valida nada en tiempo de ejecución: los campos se identifican por número, y un número reutilizado con otro tipo produce datos corruptos silenciosamente.

```
edition = "2024";

package orders.v1;

message Order {
  string id       = 1;
  string sku      = 2;
  int32  quantity = 3;
```

```
  // 4 y 5 se eliminaron: quedan quemados para siempre
  reserved 4, 5;
  reserved "legacy_price_cents", "coupon";

  string currency    = 6;
  int64  total_cents = 7;

  Status status = 8;
}

enum Status {
  // el valor 0 siempre es el "no especificado": es el default en el cable
  STATUS_UNSPECIFIED = 0;
  STATUS_PENDING     = 1;
  STATUS_PAID        = 2;
  STATUS_CANCELLED   = 3;
}
```

Las reglas, en orden de cuánto daño hacen al incumplirlas:

- Nunca reutilices un número de campo. Al borrar un campo, declara reserved con su número y con su nombre. Sin la reserva, alguien añadirá bool is_gift = 4 en seis meses y un servicio antiguo interpretará los bytes del viejo campo 4 como ese booleano.
- Nunca cambies el tipo de un campo. Pasar de int32 a string no es una migración, es corrupción. El cambio se hace añadiendo un campo nuevo y retirando el viejo en dos despliegues.
- Añadir campos es seguro en ambas direcciones: un lector antiguo ignora lo que no conoce y lo conserva en los campos desconocidos al reenviar el mensaje.
- Los enums necesitan un cero "unspecified". Sin él no puedes distinguir "no me lo enviaron" de "me enviaron el primer valor", y el día que añadas un valor nuevo los clientes antiguos lo recibirán como el cero.
- Presencia explícita. En proto3 clásico un int32 quantity = 3 con valor 0 es indistinguible de un campo ausente, y ese detalle genera bugs de facturación reales. La edición 2024 de Protobuf pone la presencia explícita como comportamiento por defecto, que es justo lo que la documentación recomendaba hacer a mano con optional en proto3.
- Los cambios incompatibles se hacen con un paquete nuevo. orders.v1 y orders.v2 conviven, el servidor implementa los dos, y los clientes migran a su ritmo. Renumerar campos para "limpiar" es la forma más eficiente de romper producción en un viernes.

Esto no se vigila con revisiones de código: se vigila en CI. buf compara tu esquema contra la rama principal y falla el pipeline ante un cambio incompatible.

```
# buf.yaml
version: v2
modules:
  - path: proto
lint:
  use:
    - STANDARD
breaking:
  use:
    - FILE
```

```
# en el pipeline de CI
buf lint
buf breaking --against "https://github.com/acme/apis.git#branch=main,subdir=proto"
buf generate
```

Y una regla de despliegue que no está en ninguna documentación de Protobuf pero causa incidentes todas las semanas: el lado que lee un campo nuevo se despliega antes que el lado que lo escribe. Para retirar un campo, el orden se invierte. Si despliegas el cliente que envía currency antes que el servidor que sabe leerlo, durante el rolling update la mitad de tus peticiones se procesan sin moneda.

Health checking y apagado limpio: el pico de errores de cada despliegue

Si tu servicio gRPC genera un pico de UNAVAILABLE en cada rollout, el problema casi nunca es gRPC: es que la salida de servicio y el cierre de conexiones están en el orden equivocado.

gRPC tiene un protocolo de salud estándar, grpc.health.v1.Health, con un método Check y otro Watch. Kubernetes lo soporta de forma nativa en las probes desde la versión 1.23 (alpha), 1.24 (beta) y estable desde 1.27, así que ya no hace falta ni exponer un endpoint HTTP paralelo ni meter el binario grpc-health-probe en la imagen.

```
from concurrent import futures
import grpc
from grpc_health.v1 import health, health_pb2, health_pb2_grpc

server = grpc.server(futures.ThreadPoolExecutor(max_workers=16))
orders_pb2_grpc.add_OrdersServicer_to_server(OrdersService(), server)

# servicio de salud: uno por servicio lógico, más el global ("")
health_servicer = health.HealthServicer()
health_pb2_grpc.add_HealthServicer_to_server(health_servicer, server)
health_servicer.set("", health_pb2.HealthCheckResponse.SERVING)
health_servicer.set("orders.v1.Orders", health_pb2.HealthCheckResponse.SERVING)

server.add_insecure_port("[::]:50051")
server.start()
```

```
readinessProbe:
  grpc:
    port: 50051
    service: orders.v1.Orders   # opcional; si se omite, se comprueba la salud global
  periodSeconds: 5
  failureThreshold: 2
livenessProbe:
  grpc:
    port: 50051
  periodSeconds: 10
  failureThreshold: 3
startupProbe:
  grpc:
    port: 50051
  periodSeconds: 2
  failureThreshold: 30          # hasta 60 s para arrancar sin que lo maten
```

Si el servidor solo habla TLS, la probe nativa no sirve tal cual: el kubelet abre la conexión en texto claro. En ese caso, o expones el servicio de salud sin TLS en un puerto interno, o vuelves a grpc-health-probe con sus opciones de TLS. Y una distinción que se confunde siempre: readiness decide si recibes tráfico y debe reflejar tus dependencias críticas; liveness decide si te matan y solo debe fallar si el proceso está irrecuperable. Un liveness que comprueba la base de datos convierte una caída de la base de datos en un CrashLoopBackOff de toda la flota.

La secuencia de apagado correcta

El orden importa porque la eliminación de un endpoint en Kubernetes es eventualmente consistente: entre que el pod falla la readiness y que el último cliente deja de enviarle peticiones pasan segundos. Si cierras el servidor en cuanto llega el SIGTERM, esas peticiones se estrellan.

1. Llega SIGTERM.
2. Marcas la salud como NOT_SERVING: la readiness empieza a fallar y kube-proxy retira el endpoint.
3. Esperas a que esa retirada se propague (un preStop con un sleep de 5-10 s, o la misma espera dentro del proceso). Este paso es el que falta en el 90% de los servicios que "fallan al desplegar".
4. Cierras con gracia: el servidor envía GOAWAY, deja terminar las RPCs en vuelo y no acepta nuevas.
5. Pasado un plazo duro, cierras a la fuerza lo que quede.

```
import signal, threading

parar = threading.Event()

def on_sigterm(signum, frame):
    # deja de anunciarse como disponible ANTES de cerrar nada
    health_servicer.enter_graceful_shutdown()
    parar.set()

signal.signal(signal.SIGTERM, on_sigterm)
parar.wait()

time.sleep(8)                 # margen para que se propague la retirada del endpoint
server.stop(grace=20).wait()  # GOAWAY + 20 s para las RPCs en vuelo
```

```
// Go: equivalente con GracefulStop y plazo duro
sig := make(chan os.Signal, 1)
signal.Notify(sig, syscall.SIGTERM, syscall.SIGINT)
<-sig

healthSrv.SetServingStatus("orders.v1.Orders", healthpb.HealthCheckResponse_NOT_SERVING)
time.Sleep(8 * time.Second)

done := make(chan struct{})
go func() { srv.GracefulStop(); close(done) }()
select {
case <-done:
case <-time.After(25 * time.Second):
    srv.Stop()   // algo no termina; cortamos
}
```

```
lifecycle:
  preStop:
    exec:
      command: ["/bin/sh", "-c", "sleep 8"]
terminationGracePeriodSeconds: 45   # > preStop + gracia del servidor + margen
```

Y comprueba la aritmética: terminationGracePeriodSeconds tiene que ser mayor que el preStop más la gracia del servidor. Si no, el kubelet manda SIGKILL en mitad de tu apagado ordenado y todo el trabajo anterior no sirve de nada.

Observabilidad: interceptores, no prints

gRPC tiene un punto de extensión limpio para métricas, trazas y logs: los interceptores (unary y stream) y, en Go, los StatsHandler. La instrumentación de OpenTelemetry se conecta en una línea por lado:

```
import "go.opentelemetry.io/contrib/instrumentation/google.golang.org/grpc/otelgrpc"

srv := grpc.NewServer(grpc.StatsHandler(otelgrpc.NewServerHandler()))

conn, err := grpc.NewClient(target,
    grpc.WithTransportCredentials(creds),
    grpc.WithStatsHandler(otelgrpc.NewClientHandler()),
)
```

```
# Python: instrumentación automática de cliente y servidor
from opentelemetry.instrumentation.grpc import GrpcInstrumentorServer, GrpcInstrumentorClient

GrpcInstrumentorServer().instrument()
GrpcInstrumentorClient().instrument()
```

Qué medir, en concreto: tasa, errores y duración por método y por código de estado. Un servicio con un 3% de UNAVAILABLE y otro con un 3% de INVALID_ARGUMENT tienen el mismo error rate y son dos incidentes completamente distintos: el primero es de infraestructura, el segundo es un cliente que acaba de desplegar algo roto. Agregar todo en una sola serie de "errores" borra justo la información que necesitas a las tres de la mañana.

Sobre cardinalidad: el nombre del método es una etiqueta acotada y segura. El identificador de la petición, el de usuario o el del tenant no lo son; esos van en las trazas y en los logs, nunca como etiqueta de una métrica.

Y dos herramientas de depuración que ahorran horas cuando el problema está en el resolver o en el balanceador:

```
# Go
export GRPC_GO_LOG_SEVERITY_LEVEL=info
export GRPC_GO_LOG_VERBOSITY_LEVEL=2

# C-core (Python, C++, Ruby): traza del resolver y del balanceo
export GRPC_VERBOSITY=debug
export GRPC_TRACE=http,round_robin,resolver,subchannel
```

Con eso ves literalmente cuántos subcanales tiene tu canal y a qué IPs apuntan, que es la única prueba definitiva de si tu balanceo funciona. El servicio channelz, cuando está habilitado, da lo mismo de forma consultable en caliente.

Seguridad del canal: TLS, mTLS y credenciales por llamada

gRPC separa dos cosas que en HTTP solemos mezclar: las credenciales de canal, que establecen quién es el servidor y cifran el transporte, y las credenciales de llamada, que dicen quién es el usuario en una petición concreta. Entenderlo bien evita el error más extendido en despliegues internos, que es dejar el canal en insecure "porque estamos dentro del clúster" y meter el token de servicio en un campo del mensaje.

El canal: de insecure a mTLS

Un canal inseguro no sólo no cifra: tampoco autentica al servidor, con lo que cualquiera que pueda envenenar el DNS interno recibe tu tráfico. Y hay un detalle práctico que sorprende: HTTP/2 sin TLS requiere que cliente y servidor acuerden usar prior knowledge, lo que hace que muchos proxies intermedios se comporten de forma distinta a como lo harán en producción con TLS. Es decir, además de inseguro, es un entorno de pruebas poco representativo.

```
// Servidor con mTLS: presentamos certificado y exigimos uno al cliente.
cert, err := tls.LoadX509KeyPair("server.crt", "server.key")
if err != nil {
    log.Fatalf("cargando par de claves: %v", err)
}

caPool := x509.NewCertPool()
caPEM, _ := os.ReadFile("ca.crt")
caPool.AppendCertsFromPEM(caPEM)

creds := credentials.NewTLS(&tls.Config{
    Certificates: []tls.Certificate{cert},
    ClientCAs:    caPool,
    // RequireAndVerifyClientCert es lo que convierte esto en mTLS de verdad.
    // Con RequireAnyClientCert aceptarias cualquier certificado presentado.
    ClientAuth:   tls.RequireAndVerifyClientCert,
    MinVersion:   tls.VersionTLS13,
})

srv := grpc.NewServer(grpc.Creds(creds))
```

Del lado del cliente, el error clásico es desactivar la verificación (InsecureSkipVerify: true) porque el certificado no coincide con el nombre por el que se conecta. Casi siempre el problema real es que el certificado se emitió para el nombre del Service y el cliente se conecta por IP de pod, o al revés. Se arregla con un SAN correcto, no apagando la verificación.

```
clientCert, _ := tls.LoadX509KeyPair("client.crt", "client.key")
creds := credentials.NewTLS(&tls.Config{
    Certificates: []tls.Certificate{clientCert},
    RootCAs:      caPool,
    // ServerName debe coincidir con un SAN del certificado del servidor.
    // Si te hace falta ponerlo a mano, es que tu DNS y tus certs no encajan.
    ServerName:   "pedidos.produccion.svc.cluster.local",
})

conn, err := grpc.NewClient("dns:///pedidos.produccion.svc.cluster.local:443",
    grpc.WithTransportCredentials(creds))
```

Rotación de certificados sin reiniciar

Con certificados de vida corta - y con SPIFFE o cert-manager la vida típica baja a horas - cargar el par de claves una vez al arrancar garantiza una caída programada. El tls.Config tiene dos callbacks pensados exactamente para esto: GetCertificate en el servidor y GetClientCertificate en el cliente. Se invocan en cada handshake, así que basta con que devuelvan lo que haya en disco en ese momento.

```
type rotador struct {
    mu   sync.RWMutex
    par  *tls.Certificate
}

func (r *rotador) recargar(certPath, keyPath string) error {
    par, err := tls.LoadX509KeyPair(certPath, keyPath)
    if err != nil {
        return err  // dejamos el anterior en pie: mejor viejo que ninguno
    }
    r.mu.Lock()
    r.par = &par
    r.mu.Unlock()
    return nil
}

func (r *rotador) get(*tls.ClientHelloInfo) (*tls.Certificate, error) {
    r.mu.RLock()
    defer r.mu.RUnlock()
    return r.par, nil
}

// tls.Config{ GetCertificate: rot.get }  -- y un ticker o un watcher de
// inotify que llame a recargar() cuando el volumen monte el cert nuevo.
```

Fíjate en el detalle de no invalidar el certificado antiguo si la recarga falla. Un fichero a medio escribir durante la rotación es un evento normal, no una emergencia; lo que no debe ocurrir es que ese evento normal tire el servicio.

Credenciales por llamada: el token va en metadata

El token de usuario no pertenece al canal, pertenece a la llamada. gRPC tiene un mecanismo específico, PerRPCCredentials, que lo inyecta en los metadatos de cada petición y - esto es importante - se niega a enviarlo por un canal sin TLS salvo que lo permitas explícitamente. Esa negativa es una funcionalidad, no un estorbo.

```
import grpc

class TokenAuth(grpc.AuthMetadataPlugin):
    def __init__(self, proveedor):
        self._proveedor = proveedor   # devuelve un token fresco

    def __call__(self, context, callback):
        # context.service_url y context.method_name estan disponibles
        # si necesitas tokens con audiencia por servicio.
        callback((("authorization", "Bearer " + self._proveedor()),), None)

canal_creds = grpc.ssl_channel_credentials(root_certificates=ca_pem)
llamada_creds = grpc.metadata_call_credentials(TokenAuth(obtener_token))
combinadas = grpc.composite_channel_credentials(canal_creds, llamada_creds)

canal = grpc.secure_channel("pedidos.interno:443", combinadas)
```

Dos avisos. Primero: las claves de metadata en gRPC se normalizan a minúsculas, y las que terminan en -bin se tratan como binarias y se codifican en base64 automáticamente. Si mandas bytes en una clave que no acaba en -bin, la librería lo rechaza. Segundo: authorization es una clave como cualquier otra para gRPC, no tiene tratamiento especial; si tu proxy la filtra o la reescribe, no habrá ningún error,

simplemente llegará vacía al servidor. Merece la pena tener un test de extremo a extremo que lo compruebe a través del proxy real.

El modelo de errores: códigos, detalles y qué significa cada uno

gRPC tiene dieciséis códigos de estado, y casi todo el mundo usa tres. El resultado es que el cliente no puede tomar decisiones automáticas: si todo es UNKNOWN, no hay política de reintentos posible, ni circuit breaker, ni alerta que distinga un fallo de infraestructura de una entrada inválida.

La distinción que más trabajo hace es entre los tres códigos que se parecen y no lo son:

- INVALID_ARGUMENT: la petición está mal formada y lo estará siempre, independientemente del estado del sistema. Nunca se reintenta.
- FAILED_PRECONDITION: la petición es válida, pero el sistema no está en un estado que permita atenderla. El cliente puede reintentar después de arreglar algo, no inmediatamente.
- UNAVAILABLE: el servicio no está disponible ahora mismo. Es el único de los tres que un cliente debería reintentar solo, y por eso es el que aparece en la política de reintentos.

Y el par que más confusión genera: ABORTED indica un conflicto de concurrencia (una transacción que perdió una carrera) y es reintentable en el nivel de la operación completa; ALREADY_EXISTS significa que la creación no puede repetirse, y si tu cliente lo trata como error cuando estaba reintentando una creación idempotente, acabas devolviendo un fallo al usuario por una operación que en realidad salió bien.

Detalles enriquecidos: el campo que hace útil el error

Un código y un mensaje de texto no bastan para que el cliente actúe. El modelo enriquecido - google.rpc.Status con un repeated google.protobuf.Any details - permite adjuntar mensajes tipados que el cliente puede inspeccionar sin parsear cadenas. Los tres que rentabilizan el esfuerzo el primer día son ErrorInfo (razón legible por máquina y dominio), BadRequest (violaciones por campo) y RetryInfo (cuánto esperar).

```
import (
    "google.golang.org/genproto/googleapis/rpc/errdetails"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
)

func (s *Servidor) CrearPedido(ctx context.Context, req *pb.CrearPedidoReq) (*pb.Pedido, error)
{
    if req.GetCantidad() <= 0 {
        st := status.New(codes.InvalidArgument, "la peticion tiene campos invalidos")
        st, err := st.WithDetails(
            &errdetails.BadRequest{
                FieldViolations: []*errdetails.BadRequest_FieldViolation{{
                    Field:       "cantidad",
                    Description: "debe ser mayor que cero",
                }},
            },
            &errdetails.ErrorInfo{
                Reason: "CANTIDAD_INVALIDA",          // estable, para maquinas
                Domain: "pedidos.puigflows.com",
                Metadata: map[string]string{"recibido": fmt.Sprint(req.GetCantidad())},
            },
```

```
        )
        if err != nil {
            // WithDetails falla si el status ya es OK; nunca deberia pasar aqui
            return nil, status.Error(codes.InvalidArgument, "peticion invalida")
        }
        return nil, st.Err()
    }
    // ...
}
```

Del lado del cliente, la lectura es un type switch sobre los detalles. Merece la pena encapsularlo una vez en la librería compartida en lugar de repetirlo en cada llamada:

```
st, ok := status.FromError(err)
if !ok {
    return err   // no venia de gRPC
}
for _, d := range st.Details() {
    switch info := d.(type) {
    case *errdetails.RetryInfo:
        // el servidor nos dice cuanto esperar: respetalo en lugar
        // de aplicar tu backoff generico
        time.Sleep(info.GetRetryDelay().AsDuration())
    case *errdetails.BadRequest:
        for _, v := range info.GetFieldViolations() {
            log.Printf("campo %s invalido: %s", v.GetField(), v.GetDescription())
        }
    case *errdetails.ErrorInfo:
        metrics.ErroresPorRazon.WithLabelValues(info.GetReason()).Inc()
    }
}
```

Ese ErrorInfo.Reason es la pieza que más rendimiento da a largo plazo: es una cadena estable, en mayúsculas, que puedes usar como etiqueta de métrica y como clave de documentación. Los mensajes de texto cambian con cada refactor y no sirven para agrupar; las razones, si las tratas como parte del contrato público, duran años.

Un último apunte sobre el tamaño: los detalles viajan en los trailers HTTP/2, que tienen un límite de cabeceras bastante ajustado (8 KiB es un valor común en proxies). Un DebugInfo con una traza de pila completa puede superarlo, y cuando eso pasa el proxy no trunca el detalle: corta la respuesta entera. Los stack traces van al log, no al trailer.

Ventanas, streams y tamaños: los límites que nadie mira hasta que duelen

Por debajo de gRPC hay una conexión HTTP/2 con dos mecanismos de control que operan a la vez y que casi nunca se configuran juntos: el número máximo de streams concurrentes y las ventanas de control de flujo. Cuando el rendimiento se estanca en un número raro - siempre 100 peticiones en vuelo, siempre 30 MB/s - suele ser uno de estos dos.

max_concurrent_streams: el límite de 100 que parece un misterio

La mayoría de servidores anuncian un máximo de cien streams concurrentes por conexión. Como un canal gRPC es una sola conexión HTTP/2, la llamada ciento uno no falla: se queda encolada en el cliente esperando a que se libere un hueco. Desde fuera eso se ve como latencia, no como saturación, y por eso cuesta tanto diagnosticarlo: los paneles del servidor están tranquilos mientras el cliente sufre.

La tentación es subir el límite. Casi siempre es mala idea: cientos de streams sobre una conexión generan contención entre las goroutines o hilos que escriben en el mismo socket, y cualquier pérdida de paquetes bloquea a todos los streams a la vez en la capa TCP. La solución que escala es la contraria, más conexiones: varios subcanales, o un pool de canales repartiendo la carga.

```
// Un pool de canales sencillo. Cada canal abre su propia conexion HTTP/2,
// asi que N canales dan N x max_concurrent_streams de paralelismo real.
type Pool struct {
    conns []*grpc.ClientConn
    next  atomic.Uint64
}

func NuevoPool(target string, n int, opts ...grpc.DialOption) (*Pool, error) {
    p := &Pool{conns: make([]*grpc.ClientConn, n)}
    for i := 0; i < n; i++ {
        c, err := grpc.NewClient(target, opts...)
        if err != nil {
            return nil, err
        }
        p.conns[i] = c
    }
    return p, nil
}

func (p *Pool) Get() *grpc.ClientConn {
    i := p.next.Add(1)
    return p.conns[i%uint64(len(p.conns))]
}
```

Antes de montar el pool conviene medir, porque el síntoma se confunde con muchos otros. En Go, grpc.WithStatsHandler te da los eventos de inicio y fin de cada RPC; la métrica que delata el problema es el tiempo entre "la aplicación pidió la llamada" y "el stream salió por el cable".

Ventanas de control de flujo y el producto ancho de banda por retardo

HTTP/2 tiene dos ventanas: una por stream y otra por conexión. El transporte de gRPC ajusta ambas de forma dinámica estimando el bandwidth-delay product (BDP), y en la mayoría de casos ese ajuste automático es mejor que lo que configurarías a mano. Donde falla es en enlaces con latencia alta y ancho de banda grande - replicación entre regiones, por ejemplo - , donde la ventana inicial tarda en crecer.

Si decides fijarlas, la regla que evita el error más común es esta: la ventana de conexión tiene que ser mayor que la de stream, idealmente varias veces mayor. Si pones 1 MiB por stream y 1 MiB de conexión, cien streams comparten ese único MiB y la ventana por stream no sirve de nada.

```
srv := grpc.NewServer(
    // Ventana de stream: cuanto puede haber en vuelo por llamada.
    grpc.InitialWindowSize(1<<20),        // 1 MiB
    // Ventana de conexion: 4x la de stream como punto de partida.
```

```
    grpc.InitialConnWindowSize(4<<20),    // 4 MiB
    grpc.MaxConcurrentStreams(250),
)
```

Un aviso importante: en cuanto fijas estos valores desactivas el sondeo de BDP. Estás cambiando un mecanismo adaptativo por una constante, y esa constante será la incorrecta en cuanto cambie la topología de red. Mide antes, mide después, y deja escrito en el repositorio por qué elegiste esos números.

El límite de mensaje de 4 MiB

Por defecto, el cliente y el servidor rechazan mensajes de más de cuatro mebibytes con RESOURCE_EXHAUSTED. Es un límite de cordura, no una recomendación de rendimiento, y el error que produce es claro. El problema real aparece cuando alguien lo sube a 100 MiB para "que quepa el informe": un mensaje grande se deserializa entero en memoria antes de que tu código lo vea, así que diez peticiones concurrentes son un gigabyte de heap y un OOMKilled.

La respuesta correcta casi siempre es convertir la operación en un server streaming que emita fragmentos, no subir el límite.

```
// En lugar de devolver un Informe gigante, emitimos paginas.
func (s *Servidor) ExportarInforme(req *pb.ExportarReq, stream
pb.Informes_ExportarInformeServer) error {
    const tam = 1000
    for offset := 0; ; offset += tam {
        filas, err := s.repo.Pagina(stream.Context(), offset, tam)
        if err != nil {
            return status.Errorf(codes.Internal, "leyendo pagina: %v", err)
        }
        if len(filas) == 0 {
            return nil
        }
        if err := stream.Send(&pb.LoteFilas{Filas: filas}); err != nil {
            return err   // el cliente se fue: no es un error nuestro
        }
    }
}
```

Sobre compresión: gzip por llamada está a un flag de distancia y reduce mucho el tráfico de mensajes con texto repetitivo, pero cuesta CPU en ambos extremos y no aporta nada en mensajes ya comprimidos (imágenes, audio, un bytes que lleva un zip). Actívala por método, no globalmente, y compruébalo con una medición real: en servicios con mensajes pequeños y muchas llamadas, gzip suele empeorar la latencia.

gRPC-Web: por qué el navegador no habla gRPC

Un navegador no puede hablar gRPC nativo, y no es por falta de ganas de los fabricantes: fetch y XMLHttpRequest no dan acceso a los frames HTTP/2 ni a los trailers, y gRPC usa ambos. De ahí gRPC-Web, que empaqueta el mismo protocolo sobre peticiones HTTP normales y mete los trailers al final del cuerpo de la respuesta.

La consecuencia práctica que hay que tener clara antes de diseñar la API: gRPC-Web soporta unario y

server streaming, pero no client streaming ni bidireccional. Si tu diseño depende de un stream bidireccional y el cliente es una web, necesitas otra cosa: WebSocket, SSE, o partir la operación en dos.

La traducción la hace un proxy. Con Envoy son unas pocas líneas:

```
http_filters:
  - name: envoy.filters.http.grpc_web
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.http.grpc_web.v3.GrpcWeb
  - name: envoy.filters.http.cors
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.http.cors.v3.Cors
  - name: envoy.filters.http.router
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.http.router.v3.Router

# CORS no es opcional aqui: el navegador necesita ver los trailers.
cors:
  allow_origin_string_match:
    - prefix: "https://app.puigflows.com"
  allow_methods: "POST, OPTIONS"
  allow_headers: "keep-alive,user-agent,content-type,x-grpc-web,grpc-timeout,authorization"
  expose_headers: "grpc-status,grpc-message"
  max_age: "1728000"
```

El expose_headers con grpc-status y grpc-message es el que más veces falta. Sin él, el navegador recibe la respuesta pero la librería no puede leer el estado, y toda la aplicación ve errores genéricos sin código. Es media hora de depuración que se ahorra leyendo esta línea.

Conviene mencionar la alternativa: el protocolo Connect implementa gRPC, gRPC-Web y un modo HTTP/JSON con la misma definición .proto y sin proxy intermedio, lo que para un backend que sirve a la vez a servicios internos y a una SPA elimina una pieza de infraestructura entera. No es la opción correcta siempre - si ya tienes Envoy por otros motivos, el filtro es trivial - pero sí merece una evaluación honesta antes de montar el proxy.

Cómo se testea un servicio gRPC

La tentación es levantar el servidor en un puerto y conectarse por localhost. Funciona, pero introduce puertos ocupados, tiempos de espera y flakiness en CI. La forma limpia en Go es bufconn: un listener en memoria que da un cliente y un servidor reales - con interceptores, códigos de estado y deadlines de verdad - sin tocar la red.

```
func nuevoClienteDeTest(t *testing.T, srvImpl pb.PedidosServer) pb.PedidosClient {
    t.Helper()
    lis := bufconn.Listen(1024 * 1024)

    s := grpc.NewServer(grpc.ChainUnaryInterceptor(InterceptorDeAuth, InterceptorDeMetricas))
    pb.RegisterPedidosServer(s, srvImpl)

    go func() {
        if err := s.Serve(lis); err != nil {
            t.Logf("servidor detenido: %v", err)
        }
    }()
    t.Cleanup(s.Stop)
```

```
    conn, err := grpc.NewClient("passthrough:///bufnet",
        grpc.WithContextDialer(func(ctx context.Context, _ string) (net.Conn, error) {
            return lis.DialContext(ctx)
        }),
        grpc.WithTransportCredentials(insecure.NewCredentials()),
    )
    if err != nil {
        t.Fatalf("conectando: %v", err)
    }
    t.Cleanup(func() { _ = conn.Close() })

    return pb.NewPedidosClient(conn)
}
```

Con eso, los tests que de verdad importan son los que comprueban el comportamiento bajo fallo, no el camino feliz: que un deadline vencido devuelve DEADLINE_EXCEEDED y no INTERNAL; que cancelar el contexto del cliente hace que el servidor abandone el trabajo; que un campo inválido devuelve INVALID_ARGUMENT con su BadRequest adjunto.

```
func TestDeadlineSePropaga(t *testing.T) {
    cli := nuevoClienteDeTest(t, &servidorLento{retardo: 200 * time.Millisecond})

    ctx, cancel := context.WithTimeout(context.Background(), 50*time.Millisecond)
    defer cancel()

    _, err := cli.CrearPedido(ctx, &pb.CrearPedidoReq{Cantidad: 1})
    if status.Code(err) != codes.DeadlineExceeded {
        t.Fatalf("esperaba DEADLINE_EXCEEDED, obtuve %v (%v)", status.Code(err), err)
    }
}
```

El test que protege el contrato

El error más caro de un servicio gRPC no se detecta con tests unitarios, porque no ocurre en tiempo de ejecución: ocurre cuando alguien cambia el .proto de forma incompatible y el fallo aparece semanas después en un cliente que nadie recordaba. Eso se detecta comparando el esquema nuevo contra el publicado, y es una tarea de CI, no de revisión humana.

```
# .github/workflows/proto.yml
name: proto
on: [pull_request]
jobs:
  breaking:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: bufbuild/buf-action@v1
        with:
          # compara contra la rama principal: detecta renombrados,
          # cambios de numero de campo y cambios de tipo
          breaking_against: "https://github.com/org/repo.git#branch=main"
```

Y para el rendimiento, una herramienta específica ahorra construir la tuya: ghz lanza carga contra un método concreto y devuelve percentiles reales, que es lo único que sirve para decidir si un cambio de

ventana o de pool ha mejorado algo.

```
ghz --insecure \
    --proto ./pedidos.proto \
    --call puigflows.pedidos.v1.Pedidos.CrearPedido \
    --concurrency 50 \
    --total 20000 \
    --data '{"cantidad": 3, "sku": "ABC-1"}' \
    localhost:50051
```

Un consejo sobre cómo leer su salida: mira la p99 y la cola por encima de ella, no la media. En gRPC, la media casi siempre está bien incluso cuando el servicio está roto, porque los problemas de este artículo - streams encolados, deadlines que no se propagan, un pod que recibe todo el tráfico - afectan a una fracción de las llamadas, no a todas. La media las esconde; los percentiles altos las enseñan.

Ocho errores que verás en producción

1. Service ClusterIP delante de un servicio gRPC. Síntoma: tres pods saturados y nueve ociosos, y un autoescalado que añade réplicas inútiles. Arreglo: headless + round_robin, proxy L7 o xDS.
2. Ninguna llamada tiene deadline. Síntoma: una dependencia lenta y tu pool de workers lleno de llamadas colgadas, con la CPU al 5%. Arreglo: deadline por método en el service config y presupuesto propagado en cada handler.
3. Reintentar DEADLINE_EXCEEDED. Síntoma: la latencia sube, los reintentos entran y el servicio se cae del todo. Arreglo: reintentar solo UNAVAILABLE (y RESOURCE_EXHAUSTED con cuidado), y activar retryThrottling.
4. Keepalive del cliente más agresivo que el MinTime del servidor. Síntoma: UNAVAILABLE intermitentes sin patrón. Arreglo: configurar los dos lados con MinTime del servidor por debajo del Time del cliente.
5. context.Background() dentro del handler. Síntoma: carga que no corresponde a ninguna petición viva y consultas que siguen corriendo cuando el cliente ya se fue. Arreglo: propagar el contexto entrante; lo que deba sobrevivir va a una cola.
6. Subir el tamaño máximo de mensaje en lugar de paginar. Síntoma: OOMKilled bajo concurrencia, aparentemente aleatorio. Arreglo: paginación, streaming por lotes o almacenamiento de objetos con URL prefirmada.
7. Reutilizar un número de campo en el .proto. Síntoma: datos corruptos solo entre versiones concretas, imposibles de reproducir en local. Arreglo: reserved siempre, y buf breaking en CI.
8. Ningún preStop ni salida de servicio antes de cerrar. Síntoma: pico de errores en cada despliegue, exactamente durante el rollout. Arreglo: NOT_SERVING, esperar la propagación, GracefulStop, plazo duro.

Un noveno que merece mención: dar por hecho que un stream de vida larga se rebalancea. No lo hace. Si tu diseño depende de repartir carga entre pods, los streams eternos trabajan contra ti.

Checklist de producción

- El balanceo está resuelto explícitamente (headless + round_robin, proxy L7 o xDS) y lo has verificado mirando la distribución de CPU o el número de subcanales, no asumiéndolo.
- El servidor tiene MaxConnectionAge con su Grace, para que el escalado horizontal se note en los clientes.
- Todo método tiene un deadline por defecto en el service config; ninguna llamada sale sin presupuesto.
- Cada handler calcula su presupuesto restante, reserva margen y falla rápido si no queda.
- La política de reintentos solo cubre códigos seguros y tiene retryThrottling activo.
- Keepalive configurado en cliente y servidor, coherente entre sí y por debajo del timeout de inactividad de la red.
- Límite de tamaño de mensaje conservador y una estrategia explícita para las cargas grandes.
- El servicio grpc.health.v1.Health está registrado y las probes de Kubernetes usan el campo grpc nativo.
- Readiness refleja dependencias; liveness no.
- Apagado en cuatro pasos: NOT_SERVING, espera de propagación, cierre con gracia, plazo duro. Con terminationGracePeriodSeconds mayor que la suma.
- Métricas por método y por código de estado, con trazas correlacionadas y sin etiquetas de alta cardinalidad.
- buf lint y buf breaking bloquean el merge; los cambios incompatibles van a un paquete nuevo.
- Los clientes de streaming reconectan con backoff y jitter y reanudan desde un cursor.
- TLS o mTLS entre servicios, con rotación de certificados que no dependa de reiniciar pods a mano.

Preguntas frecuentes

¿gRPC o REST para comunicación interna? gRPC gana cuando hay muchos servicios, varios lenguajes y el contrato tipado ahorra clases enteras de bugs; también cuando necesitas streaming bidireccional de verdad. REST con JSON sigue siendo mejor elección para APIs públicas, para cualquier cosa que un humano deba poder llamar con curl, y para equipos pequeños con un solo lenguaje, donde el coste de operar codegen y esquemas no se amortiza. El argumento del tamaño del payload rara vez es el decisivo.

¿Necesito una malla de servicios para usar gRPC? No. Un Service headless con round_robin y MaxConnectionAge resuelve el balanceo para la gran mayoría de los casos. La malla aporta mTLS automático, política de tráfico, reintentos y observabilidad uniformes; si ya la tienes, úsala, pero no la introduzcas solo para balancear gRPC.

¿Puedo llamar a gRPC desde el navegador? No directamente. El navegador no da control sobre los trailers de HTTP/2 que gRPC necesita. Las opciones son gRPC-Web o Connect con un proxy que traduzca (Envoy con el filtro correspondiente, o el propio servidor si tu framework lo soporta). Connect tiene la ventaja de hablar gRPC, gRPC-Web y su propio protocolo sobre HTTP/1.1 con el mismo esquema.

¿Cómo hago mTLS sin malla? Con credenciales de transporte y certificados montados como secretos, recargándolos sin reiniciar mediante un callback de credenciales. Es perfectamente viable; lo que cuesta es la rotación y la revocación, que es exactamente el problema que resuelve una malla o SPIFFE/SPIRE.

¿Cuánto ahorra Protobuf frente a JSON? Menos de lo que se suele decir en payloads pequeños, y bastante en CPU de serialización y en tamaño cuando hay muchos números y campos repetidos. El beneficio sostenido es otro: un esquema versionado, compatibilidad comprobada en CI y clientes generados en todos los lenguajes.

¿Los reintentos de gRPC sustituyen a un circuit breaker? No del todo. retryThrottling evita la tormenta de reintentos, que es la mitad del problema. La otra mitad - dejar de llamar a una dependencia caída y responder con un fallback - sigue siendo responsabilidad de la aplicación o de la malla.

Glosario

- Canal (channel): abstracción del cliente sobre una o varias conexiones a un destino lógico. Un canal puede tener muchos subcanales.
- Subcanal (subchannel): conexión a una dirección concreta. El balanceador reparte peticiones entre subcanales.
- Resolver: componente que traduce el destino (dns:///servicio:puerto) en una lista de direcciones.
- Service config: configuración por método (deadline, política de balanceo, reintentos) que el cliente aplica; puede venir del código o del resolver.
- Deadline: instante absoluto en el que la llamada debe haber terminado; viaja en la cabecera grpc-timeout y se propaga por la cadena.
- GOAWAY: frame de HTTP/2 con el que un extremo anuncia que va a cerrar la conexión y que no acepta streams nuevos.
- ENHANCE_YOUR_CALM: código de error de HTTP/2 que el servidor usa para decirle a un cliente que está haciendo demasiados pings.
- xDS: familia de APIs de descubrimiento de Envoy que gRPC habla directamente para recibir endpoints y políticas sin proxy en el camino.
- Hedging: enviar copias de la misma petición a varios backends y quedarse con la primera respuesta.
- Presencia explícita: capacidad de distinguir un campo ausente de un campo con el valor por defecto; comportamiento por defecto desde la edición 2024 de Protobuf.

English version

Complete guide - expanded edition, 11 September 2026

A practical guide to what separates gRPC that works on a laptop from gRPC that survives production: why a ClusterIP Service leaves three pods at 90% CPU and nine at 4% because HTTP/2 distributes connections rather than requests, the four load balancing models (layer 4, L7 proxy, client-side with a headless Service and round_robin, and proxyless xDS) including the lazy DNS resolver that makes every scale-up invisible until you set MaxConnectionAge on the server, deadlines versus timeouts and how to propagate the remaining budget while reserving a margin (with the context.Background() trap and orphaned work), which of the 17 status codes to retry and why retrying DEADLINE_EXCEEDED amplifies an...

The problem that stays hidden until you scale

The story repeats itself in almost every team that migrates to gRPC. You swap an internal REST endpoint for a gRPC call, you measure, and latency drops from 40 ms to 9 ms. Everyone is happy. Two weeks later you scale the service to twelve replicas, traffic doubles, and you open the dashboard: three pods at 90% CPU and nine at 4%. Autoscaling adds replicas that receive nothing. You restart pods and the load jumps to another three. Nobody shipped a bug.

It is not a bug. It is the direct consequence of gRPC living on top of HTTP/2, and HTTP/2 being designed to multiplex many requests over a single long-lived TCP connection. A Kubernetes ClusterIP Service balances at layer 4: it distributes connections, not requests. Your client opens one connection, that connection lands on one pod, and every request from that client for the next few hours goes to that pod.

The arithmetic is brutal and worth spelling out. With 4 client pods each holding a single subchannel, at most 4 of your 12 server pods get traffic. The other 8 exist, pass their probes, show up on the invoice and do nothing. Under HTTP/1.1 this did not happen, because each request could grab a different connection from the pool and layer-4 distribution was a reasonable approximation. Under HTTP/2 that approximation breaks.

This guide covers the decisions that separate gRPC that works on a laptop from gRPC that survives production: how to actually load balance, how to propagate deadlines so one slow dependency does not cascade, what to retry and what never to retry, keepalive without triggering disconnects, streaming with real backpressure, schema evolution that does not break clients, and a clean shutdown so every deploy stops producing an error spike.

The four load balancing models

There are exactly four ways to distribute gRPC traffic, and choosing wrong is the most expensive mistake on this list.

1. Layer-4 ClusterIP Service (what almost everyone has by default)

It works for HTTP/1.1 and it is broken for gRPC, for the reason above. It only gives acceptable distribution if connections are recycled often, which happens by accident (deploys, NAT timeouts) rather than by design. If your gRPC service sits behind a ClusterIP, start here.

2. A layer-7 proxy

A proxy that understands HTTP/2 (Envoy, nginx with grpc_pass, an Istio or Linkerd sidecar, or a Gateway with appProtocol: kubernetes.io/h2c) terminates the client connection and distributes every request across backends. It is correct, it gives you mTLS and traffic policy, and it costs an extra network hop plus one more component to operate. If you already run a service mesh, this discussion is settled.

3. Client-side load balancing with a headless Service

The best result-to-complexity ratio when there is no mesh. You declare the Service without a ClusterIP, DNS returns every pod IP, and the gRPC library opens a subchannel to each one and round-robins across them.

```
apiVersion: v1
kind: Service
metadata:
  name: orders-headless
spec:
  clusterIP: None          # this is what makes DNS return pod IPs
  selector:
    app: orders
  ports:
    - name: grpc
      port: 50051
      targetPort: 50051
      appProtocol: kubernetes.io/h2c
```

On the client you have to ask for the DNS resolver and the round robin policy explicitly. The dns:/// prefix is not decorative: without it, several implementations keep only the first address returned by resolution.

```
// Go: client-side load balancing over a headless Service
conn, err := grpc.NewClient(
    "dns:///orders-headless.default.svc.cluster.local:50051",
    grpc.WithTransportCredentials(insecure.NewCredentials()),
    grpc.WithDefaultServiceConfig("{\"loadBalancingConfig\":[{\"round_robin\":{}}]}"),
)
if err != nil {
    return err
}
defer conn.Close()
client := pb.NewOrdersClient(conn)
```

```
# Python: the same pattern with grpcio
channel = grpc.insecure_channel(
    "dns:///orders-headless.default.svc.cluster.local:50051",
    options=[
        ("grpc.lb_policy_name", "round_robin"),
        # do not re-resolve more than once every 10 s during failure bursts
        ("grpc.dns_min_time_between_resolutions_ms", 10000),
    ],
)
stub = orders_pb2_grpc.OrdersStub(channel)
```

The trap that ruins this setup. The gRPC DNS resolver does not poll DNS in a loop: it resolves on connect and re-resolves when a subchannel fails, with a minimum interval between resolutions and a lazy refresh that in grpc-go sits around 30 minutes. You scale from 12 to 20 pods, the 8 new ones pass their probes, and they do not receive a single request until an existing connection breaks. The fix is not on the client: it is on the server.

```
// Go: force connection rotation so clients re-resolve
srv := grpc.NewServer(
    grpc.KeepaliveParams(keepalive.ServerParameters{
        // each connection lives 30 min, then gets a GOAWAY (+/-10% jitter)
        MaxConnectionAge:      30 * time.Minute,
        // room for in-flight requests to finish
        MaxConnectionAgeGrace: 10 * time.Second,
        MaxConnectionIdle:     5 * time.Minute,
        Time:                  2 * time.Hour,
        Timeout:               20 * time.Second,
    }),
)
```

With MaxConnectionAge the server sends a GOAWAY, the client closes cleanly, re-resolves DNS and discovers the new pods. The +/-10% jitter gRPC applies exists precisely so that 300 connections created during the same deploy do not all expire in the same second. It is one line of configuration, and it is the difference between autoscaling that does something and autoscaling that is decorative.

4. Proxyless xDS

The gRPC client speaks the xDS protocol to a control plane (Istio, Traffic Director, or your own xDS server) and receives the endpoint list and load balancing policy with no sidecar in the path. You get per-request balancing, locality-aware routing, outlier detection and canary rollouts without an extra hop. The cost is operating a control plane and depending on your language's xDS support, which is solid in Go, Java and C-core and thinner elsewhere. It is the right answer at large scale and unjustified complexity with five services.

Deadlines: the parameter almost nobody propagates

gRPC does not have timeouts, it has deadlines. The difference is not cosmetic. A timeout is relative ("fail if you take more than 2 s") and every hop in the chain resets it. A deadline is an absolute instant that travels in the grpc-timeout header and that every service in the chain shares. That distinction is what stops a user request with a 2-second budget from generating work across five services for 30 seconds.

The default is the worst possible value: no deadline at all. A call with no deadline can hang indefinitely while occupying a slot in your worker pool. Ten of those and the service stops accepting traffic without a single CPU metric moving.

The budget and how to split it

The practical rule: every handler computes how much budget it has left, reserves a margin to serialize and return the response, and splits the rest across its dependencies. If there is no budget left on entry, fail immediately instead of calling three services with 4 ms.

```
# Python: budget computation and explicit propagation
import grpc

RESERVE_S = 0.05          # margin to serialize and respond

def budget(context: grpc.ServicerContext, default: float = 2.0) -> float:
    """Usable seconds left from the incoming deadline."""
    remaining = context.time_remaining()   # None if the client set no deadline
    if remaining is None:
        return default
```

```
    return max(0.0, remaining - RESERVE_S)


class OrdersService(orders_pb2_grpc.OrdersServicer):
    def GetOrder(self, request, context):
        left = budget(context)
        if left <= 0.02:
            # failing fast is more useful than propagating a doomed call
            context.abort(
                grpc.StatusCode.DEADLINE_EXCEEDED,
                "no budget left on handler entry",
            )

        # two sequential dependencies: half the budget for each
        stock = self.inventory.CheckStock(
            inventory_pb2.StockRequest(sku=request.sku),
            timeout=left * 0.5,
        )
        price = self.pricing.Quote(
            pricing_pb2.QuoteRequest(sku=request.sku, qty=request.quantity),
            timeout=left * 0.5,
        )
        return orders_pb2.Order(sku=request.sku, available=stock.available, total=price.total)
```

In Python propagation is manual: forget the timeout= argument and the outgoing call inherits nothing. In Go propagation is nearly automatic, because the incoming deadline is already installed on the context.Context, and context.WithTimeout can never extend a deadline: it keeps whichever of the two is sooner.

```
// Go: the incoming ctx already carries the deadline; WithTimeout can only shorten it
func (s *server) GetOrder(ctx context.Context, req *pb.GetOrderRequest) (*pb.Order, error) {
    if dl, ok := ctx.Deadline(); ok && time.Until(dl) < 50*time.Millisecond {
        return nil, status.Error(codes.DeadlineExceeded, "budget exhausted before starting")
    }

    // 200 ms as our own ceiling, but if the ctx has 80 ms left, the ctx wins
    childCtx, cancel := context.WithTimeout(ctx, 200*time.Millisecond)
    defer cancel()

    stock, err := s.inventory.CheckStock(childCtx, &pb.StockRequest{Sku: req.Sku})
    if err != nil {
        return nil, err
    }
    return &pb.Order{Sku: req.Sku, Available: stock.Available}, nil
}
```

The mistake that undoes all of this. Using context.Background() inside a handler because "the background call must not be cancelled". What you get is orphaned work: the client is gone, nobody will read the result, and your service keeps querying the database. Under overload that is the exact opposite of what you need. If there really is work that must outlive the request, it goes to a queue, not to a goroutine with a detached context.

In long loops, check cancellation on every iteration. A handler that processes 5,000 rows and never looks at ctx.Err() will keep processing them long after it stopped mattering:

```
for i, row := range rows {
    if err := ctx.Err(); err != nil {
        return nil, status.FromContextError(err).Err()   // Canceled or DeadlineExceeded
    }
    if err := s.process(ctx, row); err != nil {
        return nil, err
    }
```

```
    _ = i
}
```

What to retry and what not to

gRPC has 17 status codes, and treating them all the same is the fastest way to turn a degradation into an outage. The ones that matter for a retry decision:

- UNAVAILABLE: the connection failed or the server is not accepting. The request most likely never reached application logic. This is the retryable case par excellence.
- DEADLINE_EXCEEDED: do not retry blindly. You do not know whether the server finished the work and the response was lost, and you have no budget left anyway. Retrying here adds load to a system that is already slow.
- RESOURCE_EXHAUSTED: the server is telling you to back off. Retryable only with generous backoff, and never aggressively.
- INTERNAL and UNKNOWN: there is a bug. Retrying returns the same error at higher cost.
- FAILED_PRECONDITION and INVALID_ARGUMENT: the state or the data is wrong. Retrying is useless by definition.
- ABORTED: a concurrency conflict. Retryable, but that retry logic belongs to the application, not to the transport.

gRPC implements client-side retries through the service config, with no hand-written loops:

```
{
  "methodConfig": [
    {
      "name": [{ "service": "orders.v1.Orders", "method": "GetOrder" }],
      "timeout": "1.5s",
      "retryPolicy": {
        "maxAttempts": 4,
        "initialBackoff": "0.1s",
        "maxBackoff": "1s",
        "backoffMultiplier": 2,
        "retryableStatusCodes": ["UNAVAILABLE", "RESOURCE_EXHAUSTED"]
      }
    }
  ],
  "retryThrottling": { "maxTokens": 100, "tokenRatio": 0.1 }
}
```

Three details that change the behaviour and are not obvious:

- maxAttempts counts the original attempt, and clients apply a default ceiling of 5 even if you write 20.
- gRPC already adds jitter (on the order of +/-20%) to the backoff. You do not need to implement it yourself.
- All attempts share the same deadline. If the method timeout is 1.5 s, four attempts with backoff either fit inside that 1.5 s or simply never happen. This is correct, and it is what stops retries from multiplying your tail latency.

retryThrottling is the piece most people ignore and the one that prevents a retry storm. It works as a per-channel token bucket: each failure consumes a token, each success returns tokenRatio, and once the balance drops below half of maxTokens the client stops retrying until the server recovers. It is a

circuit breaker specifically for retries, free and declarative.

Hedging (sending copies of the same request to several backends and keeping the first response) is the other available tool. It cuts p99 latency dramatically and it multiplies load: use it only for idempotent reads and with demonstrated capacity headroom. The deadline applies to the whole hedged chain, so it will not save you from a saturated system.

Keepalive: the setting that triggers ENHANCE_YOUR_CALM

An HTTP/2 connection can sit idle for minutes. Between your client and your server there are NATs, load balancers and firewalls that close idle connections without telling anyone: an AWS NLB cuts them at 350 seconds, and plenty of cloud and home NATs do it sooner. The result is the classic "the first request of the morning always fails": the connection has been dead for hours and the client only finds out when it tries to write.

Keepalive solves that by sending HTTP/2 PINGs over the connection:

```
// Go: client keepalive
conn, err := grpc.NewClient(target,
    grpc.WithTransportCredentials(creds),
    grpc.WithKeepaliveParams(keepalive.ClientParameters{
        Time:                30 * time.Second, // ping after 30 s of inactivity
        Timeout:             10 * time.Second, // no ACK in 10 s, connection is dead
        PermitWithoutStream: true,             // ping even with no active RPCs
    }),
)
```

```
# Python: the same options as channel arguments
channel = grpc.insecure_channel(target, options=[
    ("grpc.keepalive_time_ms", 30000),
    ("grpc.keepalive_timeout_ms", 10000),
    ("grpc.keepalive_permit_without_calls", 1),
    ("grpc.http2.max_pings_without_data", 0),   # 0 = unlimited
])
```

And here is the trap. The server has an enforcement policy against abusive clients. If your client pings every 30 seconds and the server has MinTime at 5 minutes (the grpc-go default), the server treats the ping as abuse, sends a GOAWAY with the ENHANCE_YOUR_CALM code and closes the connection. From the application this looks like intermittent UNAVAILABLE errors with no pattern, and it gets diagnosed terribly because nobody inspects HTTP/2 frames.

Keepalive is configured on both sides or not at all:

```
srv := grpc.NewServer(
    grpc.KeepaliveEnforcementPolicy(keepalive.EnforcementPolicy{
        MinTime:             20 * time.Second, // below the client's Time
        PermitWithoutStream: true,
    }),
    grpc.KeepaliveParams(keepalive.ServerParameters{
        Time:                  60 * time.Second,
        Timeout:               20 * time.Second,
        MaxConnectionAge:      30 * time.Minute,
        MaxConnectionAgeGrace: 10 * time.Second,
    }),
)
```

The rule is simple: the server's MinTime strictly below the client's Time, and the client's Time below the

idle timeout of the most restrictive network device in the path.

Streaming: where backpressure actually lives

HTTP/2 has per-stream and per-connection flow control: the receiver advertises a window and the sender cannot write beyond it. That means gRPC already gives you backpressure for free, as long as you do not destroy it.

The way to destroy it is almost always the same: read from the stream as fast as possible into an unbounded in-memory channel or queue, because "that way we do not block the stream". What you achieve is converting pressure the protocol was handling into heap growth that ends in an OOMKill. If the consumer is slow, the correct outcome is for Send to block.

```
// Go: server streaming that respects cancellation and flow control
func (s *server) StreamEvents(req *pb.StreamRequest, stream pb.Events_StreamEventsServer) error
{
    ctx := stream.Context()   // cancelled if the client goes away or the deadline expires

    for ev := range s.source.Subscribe(ctx, req.Topic) {
        // Send blocks when the HTTP/2 window is full: that is correct backpressure
        if err := stream.Send(ev); err != nil {
            return err       // client disconnected or stream broken
        }
        if err := ctx.Err(); err != nil {
            return status.FromContextError(err).Err()
        }
    }
    return nil
}
```

Two limits worth knowing before production teaches them to you:

- Maximum message size. The default is 4 MiB on receive. Raising it to 100 MiB because "one client sends a big batch" turns your server into a memory-exhaustion target: every concurrent request reserves that space. The correct answer to a 100 MiB message is to paginate it or move it through object storage with a presigned URL, not to raise the limit.
- Streams do not rebalance. A long-lived stream is pinned to the pod where it was established, forever. If you have 200 streams and you deploy, those 200 break. A production streaming client needs reconnection with backoff and jitter, plus a cursor to resume without losing or duplicating events. If that sounds like work, it is: when the use case allows it, unary with polling or long-polling is cheaper to operate.

Schema evolution: the contract is forever

The real advantage of gRPC is not saving bytes: it is that the contract is written down, versioned and code-generated in every language. That advantage evaporates the day someone edits a .proto without understanding the compatibility rules, because Protobuf validates nothing at runtime: fields are identified by number, and a number reused with a different type produces silent data corruption.

```
edition = "2024";

package orders.v1;

message Order {
  string id       = 1;
```

```
  string sku      = 2;
  int32  quantity = 3;

  // 4 and 5 were removed: burned forever
  reserved 4, 5;
  reserved "legacy_price_cents", "coupon";

  string currency    = 6;
  int64  total_cents = 7;

  Status status = 8;
}

enum Status {
  // value 0 is always the "unspecified" one: it is the default on the wire
  STATUS_UNSPECIFIED = 0;
  STATUS_PENDING     = 1;
  STATUS_PAID        = 2;
  STATUS_CANCELLED   = 3;
}
```

The rules, ordered by how much damage breaking them does:

- Never reuse a field number. When you delete a field, declare reserved with its number and its name. Without the reservation, someone will add bool is_gift = 4 in six months and an older service will read the old field 4 bytes as that boolean.
- Never change a field's type. Going from int32 to string is not a migration, it is corruption. You make that change by adding a new field and retiring the old one across two deploys.
- Adding fields is safe in both directions: an older reader ignores what it does not know and preserves it in unknown fields when forwarding the message.
- Enums need a zero "unspecified" value. Without it you cannot tell "they did not send it" from "they sent the first value", and the day you add a new value older clients will receive it as zero.
- Explicit presence. In classic proto3 an int32 quantity = 3 holding 0 is indistinguishable from an absent field, and that detail causes real billing bugs. Protobuf edition 2024 makes explicit presence the default behaviour, which is exactly what the documentation recommended doing by hand with optional in proto3.
- Breaking changes go in a new package. orders.v1 and orders.v2 coexist, the server implements both, and clients migrate at their own pace. Renumbering fields to "clean things up" is the most efficient way to break production on a Friday.

None of this is enforced by code review: it is enforced in CI. buf compares your schema against the main branch and fails the pipeline on an incompatible change.

```
# buf.yaml
version: v2
modules:
  - path: proto
lint:
  use:
    - STANDARD
breaking:
  use:
    - FILE
```

```
# in the CI pipeline
buf lint
buf breaking --against "https://github.com/acme/apis.git#branch=main,subdir=proto"
buf generate
```

And one deployment rule that appears in no Protobuf documentation but causes incidents every week: the side that reads a new field ships before the side that writes it. To retire a field, the order reverses. If you deploy the client that sends currency before the server that knows how to read it, half your requests get processed without a currency during the rolling update.

Health checking and clean shutdown: the error spike on every deploy

If your gRPC service produces an UNAVAILABLE spike on every rollout, the problem is almost never gRPC: it is that taking the pod out of service and closing its connections happen in the wrong order.

gRPC has a standard health protocol, grpc.health.v1.Health, with a Check method and a Watch method. Kubernetes supports it natively in probes since 1.23 (alpha), 1.24 (beta) and stable since 1.27, so you no longer need a parallel HTTP endpoint or the grpc-health-probe binary baked into the image.

```
from concurrent import futures
import grpc
from grpc_health.v1 import health, health_pb2, health_pb2_grpc

server = grpc.server(futures.ThreadPoolExecutor(max_workers=16))
orders_pb2_grpc.add_OrdersServicer_to_server(OrdersService(), server)

# health service: one entry per logical service, plus the global one ("")
health_servicer = health.HealthServicer()
health_pb2_grpc.add_HealthServicer_to_server(health_servicer, server)
health_servicer.set("", health_pb2.HealthCheckResponse.SERVING)
health_servicer.set("orders.v1.Orders", health_pb2.HealthCheckResponse.SERVING)

server.add_insecure_port("[::]:50051")
server.start()
```

```
readinessProbe:
  grpc:
    port: 50051
    service: orders.v1.Orders   # optional; if omitted, overall health is checked
  periodSeconds: 5
  failureThreshold: 2
livenessProbe:
  grpc:
    port: 50051
  periodSeconds: 10
  failureThreshold: 3
startupProbe:
  grpc:
    port: 50051
  periodSeconds: 2
  failureThreshold: 30          # up to 60 s to boot without being killed
```

If the server only speaks TLS, the native probe will not work as-is: the kubelet opens the connection in plaintext. In that case, either expose the health service without TLS on an internal port, or fall back to grpc-health-probe with its TLS flags. And one distinction that always gets confused: readiness decides whether you receive traffic and should reflect your critical dependencies; liveness decides whether you get killed and should only fail when the process is unrecoverable. A liveness probe that checks the

database turns a database outage into a fleet-wide CrashLoopBackOff.

The correct shutdown sequence

Order matters because endpoint removal in Kubernetes is eventually consistent: seconds pass between a pod failing its readiness probe and the last client stopping sending it requests. If you close the server the moment SIGTERM arrives, those requests crash.

1. SIGTERM arrives.
2. You mark health as NOT_SERVING: readiness starts failing and kube-proxy removes the endpoint.
3. You wait for that removal to propagate (a preStop sleep of 5-10 s, or the same wait inside the process). This step is what is missing in 90% of services that "fail during deploys".
4. You shut down gracefully: the server sends GOAWAY, lets in-flight RPCs finish and accepts no new ones.
5. After a hard deadline, you force-close whatever is left.

```
import signal, threading, time

stop = threading.Event()

def on_sigterm(signum, frame):
    # stop advertising as available BEFORE closing anything
    health_servicer.enter_graceful_shutdown()
    stop.set()

signal.signal(signal.SIGTERM, on_sigterm)
stop.wait()

time.sleep(8)                 # room for endpoint removal to propagate
server.stop(grace=20).wait()  # GOAWAY + 20 s for in-flight RPCs
```

```
// Go: equivalent with GracefulStop and a hard deadline
sig := make(chan os.Signal, 1)
signal.Notify(sig, syscall.SIGTERM, syscall.SIGINT)
<-sig

healthSrv.SetServingStatus("orders.v1.Orders", healthpb.HealthCheckResponse_NOT_SERVING)
time.Sleep(8 * time.Second)

done := make(chan struct{})
go func() { srv.GracefulStop(); close(done) }()
select {
case <-done:
case <-time.After(25 * time.Second):
    srv.Stop()   // something is not finishing; cut it
}
```

```
lifecycle:
  preStop:
    exec:
      command: ["/bin/sh", "-c", "sleep 8"]
terminationGracePeriodSeconds: 45   # > preStop + server grace + margin
```

And check the arithmetic: terminationGracePeriodSeconds has to exceed the preStop plus the server grace. Otherwise the kubelet sends SIGKILL in the middle of your orderly shutdown and everything above was wasted effort.

Observability: interceptors, not prints

gRPC has a clean extension point for metrics, traces and logs: interceptors (unary and stream) and, in Go, StatsHandler. OpenTelemetry instrumentation plugs in with one line per side:

```
import "go.opentelemetry.io/contrib/instrumentation/google.golang.org/grpc/otelgrpc"

srv := grpc.NewServer(grpc.StatsHandler(otelgrpc.NewServerHandler()))

conn, err := grpc.NewClient(target,
    grpc.WithTransportCredentials(creds),
    grpc.WithStatsHandler(otelgrpc.NewClientHandler()),
)
```

```
# Python: automatic client and server instrumentation
from opentelemetry.instrumentation.grpc import GrpcInstrumentorServer, GrpcInstrumentorClient

GrpcInstrumentorServer().instrument()
GrpcInstrumentorClient().instrument()
```

What to measure, concretely: rate, errors and duration per method and per status code. A service with 3% UNAVAILABLE and one with 3% INVALID_ARGUMENT have the same error rate and are two completely different incidents: the first is infrastructure, the second is a client that just shipped something broken. Collapsing everything into a single "errors" series erases exactly the information you need at three in the morning.

On cardinality: the method name is a bounded, safe label. Request ID, user ID and tenant ID are not; those belong in traces and logs, never as a metric label.

And two debugging tools that save hours when the problem is in the resolver or the balancer:

```
# Go
export GRPC_GO_LOG_SEVERITY_LEVEL=info
export GRPC_GO_LOG_VERBOSITY_LEVEL=2

# C-core (Python, C++, Ruby): resolver and load balancing traces
export GRPC_VERBOSITY=debug
export GRPC_TRACE=http,round_robin,resolver,subchannel
```

With that you literally see how many subchannels your channel holds and which IPs they point at, which is the only definitive proof that your load balancing works. The channelz service, when enabled, gives you the same thing queryable at runtime.

Channel security: TLS, mTLS and per-call credentials

gRPC separates two things we usually blur together in HTTP: channel credentials, which establish who the server is and encrypt the transport, and call credentials, which say who the user is on one specific request. Getting that distinction right avoids the most widespread mistake in internal deployments, which is leaving the channel insecure "because we are inside the cluster" and stuffing the service token into a message field.

The channel: from insecure to mTLS

An insecure channel does not merely skip encryption: it also skips authenticating the server, so anyone who can poison internal DNS receives your traffic. And there is a practical detail that catches people out: HTTP/2 without TLS requires client and server to agree on prior knowledge, which makes many intermediate proxies behave differently from how they will behave in production with TLS. So on top of being insecure, it is an unrepresentative test environment.

```
// Server with mTLS: we present a certificate and demand one from the client.
cert, err := tls.LoadX509KeyPair("server.crt", "server.key")
if err != nil {
    log.Fatalf("loading key pair: %v", err)
}

caPool := x509.NewCertPool()
caPEM, _ := os.ReadFile("ca.crt")
caPool.AppendCertsFromPEM(caPEM)

creds := credentials.NewTLS(&tls.Config{
    Certificates: []tls.Certificate{cert},
    ClientCAs:    caPool,
    // RequireAndVerifyClientCert is what makes this real mTLS.
    // With RequireAnyClientCert you would accept any presented certificate.
    ClientAuth:   tls.RequireAndVerifyClientCert,
    MinVersion:   tls.VersionTLS13,
})

srv := grpc.NewServer(grpc.Creds(creds))
```

On the client side, the classic mistake is disabling verification (InsecureSkipVerify: true) because the certificate does not match the name being dialled. The real problem is almost always that the certificate was issued for the Service name and the client connects by pod IP, or the other way round. You fix that with a correct SAN, not by turning verification off.

```
clientCert, _ := tls.LoadX509KeyPair("client.crt", "client.key")
creds := credentials.NewTLS(&tls.Config{
    Certificates: []tls.Certificate{clientCert},
    RootCAs:      caPool,
    // ServerName must match a SAN on the server certificate.
    // If you need to set this by hand, your DNS and your certs do not line up.
    ServerName:   "orders.production.svc.cluster.local",
})

conn, err := grpc.NewClient("dns:///orders.production.svc.cluster.local:443",
    grpc.WithTransportCredentials(creds))
```

Rotating certificates without a restart

With short-lived certificates - and with SPIFFE or cert-manager typical lifetimes drop to hours - loading the key pair once at startup guarantees a scheduled outage. tls.Config has two callbacks designed for exactly this: GetCertificate on the server and GetClientCertificate on the client. They are invoked on every handshake, so all they need to do is return whatever is on disk at that moment.

```
type rotator struct {
    mu   sync.RWMutex
    pair *tls.Certificate
}

func (r *rotator) reload(certPath, keyPath string) error {
    pair, err := tls.LoadX509KeyPair(certPath, keyPath)
    if err != nil {
        return err  // keep the previous one: old beats none
    }
    r.mu.Lock()
    r.pair = &pair
    r.mu.Unlock()
    return nil
}

func (r *rotator) get(*tls.ClientHelloInfo) (*tls.Certificate, error) {
    r.mu.RLock()
    defer r.mu.RUnlock()
    return r.pair, nil
}

// tls.Config{ GetCertificate: rot.get }  -- plus a ticker or an inotify
// watcher that calls reload() when the volume mounts the new cert.
```

Note the detail of not invalidating the old certificate when the reload fails. A half-written file during rotation is a normal event, not an emergency; what must not happen is that a normal event takes the service down.

Per-call credentials: the token goes in metadata

The user's token does not belong to the channel, it belongs to the call. gRPC has a specific mechanism, PerRPCCredentials, which injects it into every request's metadata and - this matters - refuses to send it over a channel without TLS unless you explicitly allow it. That refusal is a feature, not an obstacle.

```
import grpc

class TokenAuth(grpc.AuthMetadataPlugin):
    def __init__(self, provider):
        self._provider = provider     # returns a fresh token

    def __call__(self, context, callback):
        # context.service_url and context.method_name are available
        # if you need per-service audience tokens.
        callback((("authorization", "Bearer " + self._provider()),), None)

channel_creds = grpc.ssl_channel_credentials(root_certificates=ca_pem)
call_creds = grpc.metadata_call_credentials(TokenAuth(get_token))
combined = grpc.composite_channel_credentials(channel_creds, call_creds)

channel = grpc.secure_channel("orders.internal:443", combined)
```

Two warnings. First: metadata keys in gRPC are normalised to lowercase, and keys ending in -bin are treated as binary and base64-encoded automatically. If you send bytes under a key that does not end in -bin, the library rejects it. Second: authorization is just another key to gRPC, it gets no special treatment; if your proxy strips or rewrites it there will be no error at all, it will simply arrive empty at the server. It is worth having an end-to-end test that checks this through the real proxy.

The error model: codes, details and what each one means

gRPC has sixteen status codes, and nearly everyone uses three. The result is that the client cannot make automatic decisions: if everything is UNKNOWN, no retry policy is possible, no circuit breaker, no alert that distinguishes an infrastructure failure from bad input.

The distinction that does the most work is between the three codes that look alike and are not:

- INVALID_ARGUMENT: the request is malformed and always will be, regardless of system state. Never retried.
- FAILED_PRECONDITION: the request is valid, but the system is not in a state that allows serving it. The client may retry after fixing something, not immediately.
- UNAVAILABLE: the service is not reachable right now. It is the only one of the three a client should retry on its own, which is why it is the one that appears in the retry policy.

And the pair that causes the most confusion: ABORTED signals a concurrency conflict (a transaction that lost a race) and is retryable at the level of the whole operation; ALREADY_EXISTS means the creation cannot be repeated, and if your client treats it as an error while retrying an idempotent creation, you end up returning a failure to the user for an operation that actually succeeded.

Rich details: the field that makes an error useful

A code and a text message are not enough for a client to act on. The rich model - google.rpc.Status with a repeated google.protobuf.Any details - lets you attach typed messages the client can inspect without parsing strings. The three that pay for themselves on day one are ErrorInfo (machine-readable reason and domain), BadRequest (per-field violations) and RetryInfo (how long to wait).

```
import (
    "google.golang.org/genproto/googleapis/rpc/errdetails"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
)

func (s *Server) CreateOrder(ctx context.Context, req *pb.CreateOrderReq) (*pb.Order, error) {
    if req.GetQuantity() <= 0 {
        st := status.New(codes.InvalidArgument, "the request has invalid fields")
        st, err := st.WithDetails(
            &errdetails.BadRequest{
                FieldViolations: []*errdetails.BadRequest_FieldViolation{{
                    Field:       "quantity",
                    Description: "must be greater than zero",
                }},
            },
            &errdetails.ErrorInfo{
                Reason: "INVALID_QUANTITY",           // stable, for machines
                Domain: "orders.puigflows.com",
                Metadata: map[string]string{"received": fmt.Sprint(req.GetQuantity())},
            },
        )
        if err != nil {
            // WithDetails fails if the status is already OK; should never happen here
            return nil, status.Error(codes.InvalidArgument, "invalid request")
        }
        return nil, st.Err()
    }
    // ...
}
```

On the client side, reading it is a type switch over the details. It is worth wrapping once in the shared library instead of repeating it at every call site:

```
st, ok := status.FromError(err)
if !ok {
    return err   // did not come from gRPC
}
for _, d := range st.Details() {
    switch info := d.(type) {
    case *errdetails.RetryInfo:
        // the server is telling us how long to wait: honour it instead
        // of applying your generic backoff
        time.Sleep(info.GetRetryDelay().AsDuration())
    case *errdetails.BadRequest:
        for _, v := range info.GetFieldViolations() {
            log.Printf("field %s invalid: %s", v.GetField(), v.GetDescription())
        }
    case *errdetails.ErrorInfo:
        metrics.ErrorsByReason.WithLabelValues(info.GetReason()).Inc()
    }
}
```

That ErrorInfo.Reason is the piece that pays off most over time: a stable, uppercase string you can use as a metric label and as a documentation key. Text messages change with every refactor and are useless for grouping; reasons, if you treat them as part of the public contract, last for years.

One final note on size: details travel in HTTP/2 trailers, which have a fairly tight header limit (8 KiB is a common proxy value). A DebugInfo carrying a full stack trace can exceed it, and when that happens the proxy does not truncate the detail: it drops the whole response. Stack traces go in the log, not in the trailer.

Windows, streams and sizes: the limits nobody looks at until they hurt

Underneath gRPC there is an HTTP/2 connection with two control mechanisms operating at once that are almost never configured together: the maximum number of concurrent streams, and the flow control windows. When throughput plateaus at an odd number - always 100 requests in flight, always 30 MB/s - it is usually one of these two.

max_concurrent_streams: the limit of 100 that looks like a mystery

Most servers advertise a maximum of one hundred concurrent streams per connection. Since a gRPC channel is a single HTTP/2 connection, call number one hundred and one does not fail: it queues on the client waiting for a slot. From the outside that looks like latency, not saturation, which is why it is so hard to diagnose: the server dashboards are calm while the client suffers.

The temptation is to raise the limit. It is almost always a bad idea: hundreds of streams over one connection create contention between the goroutines or threads writing to the same socket, and any packet loss blocks every stream at once at the TCP layer. The solution that scales is the opposite, more connections: several subchannels, or a channel pool spreading the load.

```
// A simple channel pool. Each channel opens its own HTTP/2 connection,
// so N channels give you N x max_concurrent_streams of real parallelism.
type Pool struct {
    conns []*grpc.ClientConn
    next  atomic.Uint64
}

func NewPool(target string, n int, opts ...grpc.DialOption) (*Pool, error) {
    p := &Pool{conns: make([]*grpc.ClientConn, n)}
    for i := 0; i < n; i++ {
        c, err := grpc.NewClient(target, opts...)
        if err != nil {
            return nil, err
        }
        p.conns[i] = c
    }
    return p, nil
}

func (p *Pool) Get() *grpc.ClientConn {
    i := p.next.Add(1)
    return p.conns[i%uint64(len(p.conns))]
}
```

Before building the pool it is worth measuring, because the symptom looks like many others. In Go, grpc.WithStatsHandler gives you begin and end events for every RPC; the metric that exposes the problem is the time between "the application asked for the call" and "the stream went out on the wire".

Flow control windows and the bandwidth-delay product

HTTP/2 has two windows: one per stream and one per connection. The gRPC transport tunes both dynamically by estimating the bandwidth-delay product (BDP), and in most cases that automatic tuning beats anything you would set by hand. Where it falls short is on high-latency, high-bandwidth links - cross-region replication, for example - where the initial window takes a while to grow.

If you decide to pin them, the rule that avoids the most common mistake is this: the connection window has to be larger than the stream window, ideally several times larger. If you set 1 MiB per stream and 1 MiB per connection, a hundred streams share that single MiB and the per-stream window is worthless.

```
srv := grpc.NewServer(
    // Stream window: how much can be in flight per call.
    grpc.InitialWindowSize(1<<20),        // 1 MiB
    // Connection window: 4x the stream window as a starting point.
    grpc.InitialConnWindowSize(4<<20),    // 4 MiB
    grpc.MaxConcurrentStreams(250),
)
```

An important warning: the moment you pin these values you turn off BDP probing. You are trading an adaptive mechanism for a constant, and that constant will be the wrong one as soon as the network topology changes. Measure before, measure after, and leave a note in the repository explaining why you picked those numbers.

The 4 MiB message limit

By default, client and server reject messages larger than four mebibytes with RESOURCE_EXHAUSTED. It is a sanity limit, not a performance recommendation, and the error it produces is clear. The real problem shows up when somebody raises it to 100 MiB "so the report fits": a large message is deserialised entirely into memory before your code ever sees it, so ten concurrent requests are a gigabyte of heap and an OOMKilled.

The right answer is nearly always to turn the operation into a server stream that emits chunks, not to raise the limit.

```
// Instead of returning one giant Report, we emit pages.
func (s *Server) ExportReport(req *pb.ExportReq, stream pb.Reports_ExportReportServer) error {
    const size = 1000
    for offset := 0; ; offset += size {
        rows, err := s.repo.Page(stream.Context(), offset, size)
        if err != nil {
            return status.Errorf(codes.Internal, "reading page: %v", err)
        }
        if len(rows) == 0 {
            return nil
        }
        if err := stream.Send(&pb.RowBatch{Rows: rows}); err != nil {
            return err   // the client left: not our error
        }
    }
}
```

On compression: per-call gzip is one flag away and cuts a lot of traffic for messages with repetitive text, but it costs CPU on both ends and buys nothing on already-compressed payloads (images, audio, a bytes field holding a zip). Enable it per method, not globally, and confirm it with a real measurement: on services with small messages and high call volume, gzip usually makes latency worse.

gRPC-Web: why the browser does not speak gRPC

A browser cannot speak native gRPC, and it is not for lack of vendor interest: fetch and XMLHttpRequest give no access to HTTP/2 frames or to trailers, and gRPC uses both. Hence gRPC-Web, which wraps the same protocol over ordinary HTTP requests and puts the trailers at the end of the response body.

The practical consequence to be clear about before designing the API: gRPC-Web supports unary and server streaming, but not client streaming or bidirectional. If your design depends on a bidirectional stream and the client is a web app, you need something else: WebSocket, SSE, or splitting the operation in two.

A proxy does the translation. With Envoy it is a few lines:

```
http_filters:
  - name: envoy.filters.http.grpc_web
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.http.grpc_web.v3.GrpcWeb
  - name: envoy.filters.http.cors
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.http.cors.v3.Cors
  - name: envoy.filters.http.router
```

```
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.http.router.v3.Router

# CORS is not optional here: the browser needs to see the trailers.
cors:
  allow_origin_string_match:
    - prefix: "https://app.puigflows.com"
  allow_methods: "POST, OPTIONS"
  allow_headers: "keep-alive,user-agent,content-type,x-grpc-web,grpc-timeout,authorization"
  expose_headers: "grpc-status,grpc-message"
  max_age: "1728000"
```

The expose_headers line with grpc-status and grpc-message is the one most often missing. Without it the browser receives the response but the library cannot read the status, and the whole application sees generic errors with no code. That is half an hour of debugging saved by reading one line.

The alternative deserves a mention: the Connect protocol implements gRPC, gRPC-Web and an HTTP/JSON mode from the same .proto definition with no intermediate proxy, which for a backend serving internal services and an SPA at the same time removes an entire piece of infrastructure. It is not always the right call - if you already run Envoy for other reasons, the filter is trivial - but it deserves an honest evaluation before you stand up the proxy.

How to test a gRPC service

The temptation is to bring the server up on a port and connect over localhost. It works, but it introduces port clashes, waits and flakiness in CI. The clean way in Go is bufconn: an in-memory listener that gives you a real client and a real server - with real interceptors, status codes and deadlines - without touching the network.

```
func newTestClient(t *testing.T, srvImpl pb.OrdersServer) pb.OrdersClient {
    t.Helper()
    lis := bufconn.Listen(1024 * 1024)

    s := grpc.NewServer(grpc.ChainUnaryInterceptor(AuthInterceptor, MetricsInterceptor))
    pb.RegisterOrdersServer(s, srvImpl)

    go func() {
        if err := s.Serve(lis); err != nil {
            t.Logf("server stopped: %v", err)
        }
    }()
    t.Cleanup(s.Stop)

    conn, err := grpc.NewClient("passthrough:///bufnet",
        grpc.WithContextDialer(func(ctx context.Context, _ string) (net.Conn, error) {
            return lis.DialContext(ctx)
        }),
        grpc.WithTransportCredentials(insecure.NewCredentials()),
    )
    if err != nil {
        t.Fatalf("dialing: %v", err)
    }
    t.Cleanup(func() { _ = conn.Close() })

    return pb.NewOrdersClient(conn)
}
```

With that in place, the tests that actually matter are the ones covering failure behaviour, not the happy path: that an expired deadline returns DEADLINE_EXCEEDED and not INTERNAL; that cancelling the client context makes the server abandon the work; that an invalid field returns INVALID_ARGUMENT with its BadRequest attached.

```
func TestDeadlinePropagates(t *testing.T) {
    cli := newTestClient(t, &slowServer{delay: 200 * time.Millisecond})

    ctx, cancel := context.WithTimeout(context.Background(), 50*time.Millisecond)
    defer cancel()

    _, err := cli.CreateOrder(ctx, &pb.CreateOrderReq{Quantity: 1})
    if status.Code(err) != codes.DeadlineExceeded {
        t.Fatalf("expected DEADLINE_EXCEEDED, got %v (%v)", status.Code(err), err)
    }
}
```

The test that protects the contract

The most expensive mistake in a gRPC service is not caught by unit tests, because it does not happen at runtime: it happens when somebody changes the .proto in an incompatible way and the failure surfaces weeks later in a client nobody remembered. You catch that by comparing the new schema against the published one, and it is a CI job, not a human review task.

```
# .github/workflows/proto.yml
name: proto
on: [pull_request]
jobs:
  breaking:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: bufbuild/buf-action@v1
        with:
          # compare against the main branch: catches renames,
          # field number changes and type changes
          breaking_against: "https://github.com/org/repo.git#branch=main"
```

And for performance, a purpose-built tool saves you writing your own: ghz drives load against a specific method and reports real percentiles, which is the only thing that lets you decide whether a window or pool change improved anything.

```
ghz --insecure \
    --proto ./orders.proto \
    --call puigflows.orders.v1.Orders.CreateOrder \
    --concurrency 50 \
    --total 20000 \
    --data '{"quantity": 3, "sku": "ABC-1"}' \
    localhost:50051
```

One piece of advice on reading its output: look at p99 and the tail above it, not the mean. In gRPC the mean is almost always fine even when the service is broken, because the problems in this article - queued streams, deadlines that do not propagate, one pod taking all the traffic - affect a fraction of

calls, not all of them. The mean hides them; the high percentiles show them.

Eight failures you will see in production

1. A ClusterIP Service in front of a gRPC service. Symptom: three saturated pods, nine idle, and autoscaling adding useless replicas. Fix: headless + round_robin, an L7 proxy, or xDS.
2. No call has a deadline. Symptom: one slow dependency and your worker pool full of hung calls, with CPU at 5%. Fix: a per-method deadline in the service config and a propagated budget in every handler.
3. Retrying DEADLINE_EXCEEDED. Symptom: latency rises, retries kick in, and the service goes down entirely. Fix: retry only UNAVAILABLE (and RESOURCE_EXHAUSTED carefully), and enable retryThrottling.
4. Client keepalive more aggressive than the server's MinTime. Symptom: intermittent UNAVAILABLE with no pattern. Fix: configure both sides, with the server's MinTime below the client's Time.
5. context.Background() inside a handler. Symptom: load that matches no live request, and queries still running after the client left. Fix: propagate the incoming context; anything that must outlive the request goes to a queue.
6. Raising the maximum message size instead of paginating. Symptom: OOMKills under concurrency, apparently at random. Fix: pagination, batched streaming, or object storage with a presigned URL.
7. Reusing a field number in the .proto. Symptom: data corruption only between specific versions, impossible to reproduce locally. Fix: always reserved, plus buf breaking in CI.
8. No preStop and no de-registration before closing. Symptom: an error spike on every deploy, precisely during the rollout. Fix: NOT_SERVING, wait for propagation, GracefulStop, hard deadline.

A ninth deserves a mention: assuming a long-lived stream rebalances. It does not. If your design depends on spreading load across pods, eternal streams work against you.

Production checklist

- Load balancing is solved explicitly (headless + round_robin, L7 proxy, or xDS) and you have verified it by looking at CPU distribution or subchannel count, not assumed it.
- The server sets MaxConnectionAge with its Grace, so horizontal scaling actually reaches clients.
- Every method has a default deadline in the service config; no call leaves without a budget.
- Every handler computes its remaining budget, reserves a margin and fails fast when there is none.
- The retry policy covers only safe status codes and has retryThrottling enabled.
- Keepalive configured on client and server, consistent with each other and below the network's idle timeout.
- A conservative maximum message size, plus an explicit strategy for large payloads.
- The grpc.health.v1.Health service is registered and Kubernetes probes use the native grpc field.
- Readiness reflects dependencies; liveness does not.
- Four-step shutdown: NOT_SERVING, propagation wait, graceful close, hard deadline. With terminationGracePeriodSeconds greater than the sum.
- Metrics per method and per status code, correlated traces, and no high-cardinality labels.

- buf lint and buf breaking block the merge; breaking changes go to a new package.
- Streaming clients reconnect with backoff and jitter and resume from a cursor.
- TLS or mTLS between services, with certificate rotation that does not depend on restarting pods by hand.

FAQ

gRPC or REST for internal communication? gRPC wins when there are many services, several languages, and the typed contract eliminates whole classes of bugs; also when you genuinely need bidirectional streaming. REST with JSON is still the better choice for public APIs, for anything a human should be able to call with curl, and for small single-language teams where the cost of operating codegen and schemas does not pay off. The payload-size argument is rarely the deciding one.

Do I need a service mesh to use gRPC? No. A headless Service with round_robin and MaxConnectionAge solves load balancing for the vast majority of cases. A mesh adds automatic mTLS, traffic policy, and uniform retries and observability; if you already run one, use it, but do not introduce it just to balance gRPC.

Can I call gRPC from the browser? Not directly. Browsers give no control over the HTTP/2 trailers gRPC needs. The options are gRPC-Web or Connect with a translating proxy (Envoy with the corresponding filter, or the server itself if your framework supports it). Connect has the advantage of speaking gRPC, gRPC-Web and its own HTTP/1.1 protocol from the same schema.

How do I do mTLS without a mesh? With transport credentials and certificates mounted as secrets, reloaded without a restart through a credentials callback. It is perfectly workable; the hard part is rotation and revocation, which is exactly the problem a mesh or SPIFFE/SPIRE solves.

How much does Protobuf actually save over JSON? Less than commonly claimed on small payloads, and a fair amount in serialization CPU and in size when there are many numbers and repeated fields. The lasting benefit is different: a versioned schema, compatibility checked in CI, and generated clients in every language.

Do gRPC retries replace a circuit breaker? Not entirely. retryThrottling prevents the retry storm, which is half the problem. The other half - stopping calls to a dead dependency and answering with a fallback - remains the responsibility of the application or the mesh.

Glossary

- Channel: the client-side abstraction over one or more connections to a logical target. One channel can hold many subchannels.
- Subchannel: a connection to a specific address. The load balancer distributes requests across subchannels.
- Resolver: the component that translates a target (dns:///service:port) into a list of addresses.
- Service config: per-method configuration (deadline, load balancing policy, retries) applied by the client; it can come from code or from the resolver.
- Deadline: the absolute instant by which the call must have finished; it travels in the grpc-timeout header and propagates down the chain.
- GOAWAY: the HTTP/2 frame with which one peer announces it is closing the connection and accepting no new streams.

- ENHANCE_YOUR_CALM: the HTTP/2 error code a server uses to tell a client it is sending too many pings.
- xDS: the family of Envoy discovery APIs that gRPC speaks directly to receive endpoints and policies with no proxy in the path.
- Hedging: sending copies of the same request to several backends and keeping the first response.
- Explicit presence: the ability to distinguish an absent field from a field holding the default value; the default behaviour since Protobuf edition 2024.