Engineer

gRPC

Framework de llamadas a procedimientos remotos sobre HTTP/2

Qué es

gRPC es un framework de llamadas a procedimientos remotos. Un programa llama a un método de un servidor que se ejecuta en otra máquina como si el servidor fuera un objeto local; gRPC transporta la llamada por la red mediante un protocolo propio, sobre HTTP/2, y devuelve la respuesta como valor de retorno.

Todo empieza con una definición del servicio: una lista de operaciones remotas y los tipos de solicitud y respuesta que acepta cada una. De forma predeterminada, se escribe en Protocol Buffers, el formato de código abierto de Google para serializar datos con independencia del lenguaje y la plataforma, en un archivo cuyo nombre termina en `.proto`. Se pueden incorporar otros formatos, pero gRPC usa protocol buffers salvo que se le indique lo contrario. El compilador `protoc`, junto con un complemento de gRPC, lee el archivo y genera código para ambos extremos. El cliente recibe un stub, un objeto local con los mismos métodos que el servicio —algunos lenguajes lo llaman simplemente cliente—, y el servidor recibe una interfaz que debe implementar. Como ambos extremos proceden de un mismo archivo, un servidor escrito en Java puede responder a clientes escritos en Go, Python o Ruby.

Un método adopta una de cuatro formas. Un método unario recibe una solicitud y devuelve una respuesta, como una llamada a una función ordinaria. Un método de streaming del servidor recibe una solicitud y devuelve un flujo de respuestas; uno de streaming del cliente hace lo contrario, y uno de streaming bidireccional permite a ambos extremos enviar sus flujos de forma independiente. Dentro de una misma llamada, gRPC garantiza que los mensajes lleguen en el orden en que se enviaron.

En la red, gRPC funciona sobre HTTP/2. Cada llamada es un `POST` a una ruta formada por el nombre totalmente cualificado del servicio y el nombre del método, con el tipo de contenido `application/grpc`. Cada mensaje lleva delante un prefijo de cinco bytes: uno indica si está comprimido y cuatro expresan su longitud. El resultado de la llamada no es el estado HTTP. Una respuesta producida por gRPC lleva el estado 200 incluso si la llamada falla; el resultado real llega en los campos de cabecera finales, enviados después del cuerpo: `grpc-status`, con un número, y, opcionalmente, `grpc-message`, con una descripción del error.

Esto lo distingue de REST. Una API REST da nombre a los recursos y actúa sobre ellos mediante los métodos y códigos de estado del propio HTTP, normalmente con cuerpos JSON que cualquiera puede leer. Una API gRPC es una lista de métodos con mensajes binarios tipados, y HTTP/2 es solo el transporte subyacente. La documentación de Protocol Buffers compara el formato con JSON y destaca sus mensajes más pequeños y su procesamiento más rápido.

Utilice el concepto en una decisión

Aplique «¿Qué es gRPC?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. La parte que el usuario no ve: dónde se guardan los datos, qué reglas se aplican y cómo pueden comunicarse otros sistemas con ella.

Explore nuestros servicios Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. Diseño y operación de 6 flujos ETL/ELT — 6 flujos ETL en tres motores de bases de datos

Diseño y operación de 6 flujos ETL/ELT Póngase en contacto →

Por qué importa

Primero viene el contrato y de él se deriva el código. En muchas API HTTP, la descripción y el código que implementa la API son dos cosas separadas que pueden divergir, y a menudo el cliente se escribe a mano a partir de un documento. En gRPC, el archivo `.proto` es el único lugar donde se define un método o un campo, y de él se generan todos los clientes y servidores. Un cambio en un mensaje llega a cada lenguaje mediante el mismo generador, sin depender de que alguien relea un documento.

La codificación es compacta porque cada campo viaja como su número y su valor, en lugar de su nombre. El precio es que un mensaje no puede interpretarse por completo sin su archivo `.proto`, ya que protocol buffers no se describen a sí

mismos, y que el formato presupone que un mensaje completo cabe en memoria a la vez. Su propia documentación aconseja otra solución para datos que superen unos pocos megabytes.

El streaming forma parte del diseño, en lugar de ser un añadido. Las llamadas funcionan como flujos en conexiones HTTP/2, por lo que un método de streaming del servidor puede seguir enviando resultados mediante la misma infraestructura que una llamada que responde una sola vez.

Los límites de tiempo viajan con la llamada. El cliente fija un plazo, gRPC envía el tiempo restante en la cabecera `grpc-timeout` y, cuando se agota, el cliente recibe `DEADLINE_EXCEEDED` y se cancela el extremo de la llamada que corresponde al servidor. Cuando un servidor llama a otro, algunas implementaciones propagan el plazo: Java y Go lo hacen de forma predeterminada; en C++ hay que activarlo. Como el plazo cruza cada salto como tiempo restante y no como hora del reloj, las máquinas cuyos relojes discrepan siguen coincidiendo en cuándo desistir.

Los errores comparten un vocabulario. gRPC define 17 códigos de estado, desde 0 `OK` hasta 16 `UNAUTHENTICATED`, que forman parte del propio protocolo; por eso un cliente Go y un servidor Java coinciden en lo que significan `NOT_FOUND` o `UNAVAILABLE`.

Todo esto encaja con el tráfico entre servicios controlados por un mismo equipo o una misma empresa, donde ambos extremos pueden compartir los archivos `.proto` y compilar a partir de ellos. Encaja peor con el navegador. Las notas de protocolo del propio proyecto gRPC afirman que los navegadores no exponen a los scripts la estructura de tramas HTTP/2 y nunca lo harán; por eso el código del navegador habla un protocolo distinto, gRPC-Web, y, en la configuración habitual, un proxy lo traduce a gRPC nativo.

Cómo se aplica en la práctica

Este es un pequeño servicio de facturación. Un método unario obtiene una factura y otro de streaming del servidor envía las facturas de un cliente de una en una:

```
syntax = "proto3";

package billing.v1;
```

```
service InvoiceService {
  rpc GetInvoice(GetInvoiceRequest) returns (Invoice);
  rpc WatchInvoices(WatchInvoicesRequest) returns (stream Invoice);
}

message GetInvoiceRequest {
  string invoice_id = 1;
}

message WatchInvoicesRequest {
  string customer_id = 1;
}

message Invoice {
  reserved 4;
  reserved "discount_code";
  string invoice_id = 1;
  string customer_id = 2;
  int64 total_cents = 3;
  string currency = 5;
}
```

Los números no son valores predeterminados. Son números de campo y, en la red, identifican cada campo en lugar de su nombre. Los números del 1 al 15 se codifican en un byte, por lo que conviene reservarlos para los campos que se rellenan con más frecuencia. Las dos líneas `reserved` dejan constancia de un campo que existió, el número 4 llamado `discount_code`, para que ninguno de los dos pueda reutilizarse en otro campo.

Para Python, un solo comando genera `billing_pb2.py`, con las clases de los mensajes, y `billing_pb2_grpc.py`, con las clases del cliente y del servidor:

```
python -m grpc_tools.protoc -Iprotos \
  --python_out=. --grpc_python_out=. protos/billing.proto
```

Una llamada a `GetInvoice` tiene este aspecto como tramas HTTP/2:

```
HEADERS (flags = END_HEADERS)
:method = POST
:scheme = https
:path = /billing.v1.InvoiceService/GetInvoice
:authority = billing.example.com
te = trailers
content-type = application/grpc+proto
```

```
grpc-timeout = 2S

DATA (flags = END_STREAM)
<Length-Prefixed Message>

HEADERS (flags = END_HEADERS)
:status = 200
content-type = application/grpc+proto

DATA
<Length-Prefixed Message>

HEADERS (flags = END_STREAM, END_HEADERS)
grpc-status = 0
```

`2S` significa dos segundos, y `te = trailers` sirve para detectar proxies incompatibles con gRPC. Si la factura no existe, la respuesta sigue empezando con `:status = 200` y la última trama lleva `grpc-status = 5`, que es `NOT_FOUND`, junto con un `grpc-message` como `invoice not found`. Un servidor que falla de inmediato puede incluso enviar una única trama `HEADERS` con el estado y sin ningún mensaje.

Generar ambos extremos a partir de un contrato no es exclusivo de gRPC. Un proyecto del portafolio de este sitio generó el contrato de la API desde la base de datos hacia fuera para una API descrita con OpenAPI. La cadena empieza en la base de datos y avanza hacia el exterior: los tipos Go definen la API, Huma emite a partir de ellos la descripción OpenAPI, de esta se generan un cliente TypeScript tipado y manejadores simulados, y una comprobación de contrato en CI hace fallar la compilación cuando lo que envía el frontend deja de coincidir con lo que espera la API. gRPC incorpora esa misma disciplina a sus herramientas, con el archivo `.proto` como fuente única.

Dónde surgen los problemas

Una llamada sin plazo puede esperar indefinidamente. gRPC no fija ningún plazo de forma predeterminada, por lo que un cliente que llama a un servidor bloqueado puede esperar una respuesta, en la práctica, para siempre. Dé a cada llamada un plazo realista. En Python se usa el argumento `timeout`, en segundos:

```
import grpc

import billing_pb2
import billing_pb2_grpc
```

```
credentials = grpc.ssl_channel_credentials()
with grpc.secure_channel("billing.example.com:443", credentials) as channel:
    stub = billing_pb2_grpc.InvoiceServiceStub(channel)
    request = billing_pb2.GetInvoiceRequest(invoice_id="INV-1001")
    invoice = stub.GetInvoice(request, timeout=2.0)
```

Cuando se agota el tiempo, la llamada lanza `grpc.RpcError` y su `code()` devuelve `grpc.StatusCode.DEADLINE_EXCEEDED`. En el servidor, gRPC cancela la llamada con `CANCELLED` una vez vencido el plazo, pero detener el trabajo que haya iniciado el manejador corresponde al propio código del servicio: un manejador de larga duración tiene que comprobar la cancelación y detenerse por sí mismo.

Un timeout no significa que no haya ocurrido nada. Cliente y servidor deciden por separado si una llamada tuvo éxito, así que una llamada puede completarse en el servidor y aun así fallar en el cliente con `DEADLINE_EXCEEDED`. Cancelar una llamada no deshace el trabajo ya realizado. Un cliente que simplemente lo intenta otra vez puede crear dos veces la misma factura. El protocolo no presupone que una llamada sea idempotente salvo que esté marcada como tal. `UNAVAILABLE` es el código que con mayor probabilidad indica un problema temporal y justifica un reintento con espera progresiva, pero la documentación de los códigos de estado advierte que reintentar no siempre es seguro para operaciones que no son idempotentes. Ese mismo documento deja al código que hace las llamadas la decisión sobre qué códigos justifican un reintento.

Reutilizar un número de campo corrompe los datos. Un número de campo no puede cambiar una vez que se usa el tipo de mensaje. Si se elimina un campo y después su número se asigna a uno nuevo, se decodifica el campo nuevo a partir de bytes que antes significaban otra cosa, tanto con clientes antiguos como al leer mensajes almacenados. La guía de Protocol Buffers enumera las posibles consecuencias: errores de análisis, filtraciones de datos personales y datos corruptos. Nunca cambie la numeración de un campo y reserve todos los números y nombres eliminados, como en el ejemplo anterior.

Todo fallo parece un éxito para HTTP. Como gRPC responde con el estado HTTP 200 aunque falle una llamada, cualquier componente que evalúe el tráfico solo por el estado HTTP, como un registro de acceso o una tasa de errores basada en códigos de estado, cuenta los fallos como éxitos. Por ese mismo motivo, la especificación indica que los servidores respondan con 415 a las solicitudes cuyo tipo de contenido no sea `application/grpc`, para que otros clientes HTTP/2 no interpreten un error gRPC como un éxito. Cuente `grpc-status`, no `:status`.

Los flujos largos concentran su carga en un servidor. Las llamadas funcionan como flujos en una conexión HTTP/2 compartida y, una vez iniciado un flujo, no se puede trasladar a otro servidor. Un flujo que dura horas mantiene la carga donde empezó, por muchos servidores que se añadan después. Cada conexión también tiene un límite de flujos simultáneos; al alcanzarlo, las llamadas adicionales esperan en el cliente hasta que termine uno. La guía de rendimiento de gRPC aconseja usar streaming solo cuando aporte una mejora considerable de rendimiento o simplicidad, y un canal separado para cada área de carga elevada.

Las cargas útiles grandes no caben en un mensaje. gRPC impone un tamaño máximo de mensaje; uno mayor que el límite configurado falla con `RESOURCE_EXHAUSTED` en el extremo que lo envía o recibe. Aumentar el límite solo desplaza el problema, porque protocol buffers espera el mensaje entero en memoria y, con grandes volúmenes de datos, pueden acabar allí varias copias serializadas a la vez. Divida esos datos en un flujo de mensajes más pequeños o sáquelos de gRPC.

Lecturas adicionales

- Conceptos básicos, arquitectura y ciclo de vida — Definiciones de servicios, los cuatro tipos de método, stubs, plazos y cancelación.
- gRPC sobre HTTP2 — El protocolo en la red: rutas, cabeceras, prefijo del mensaje y estado en los campos finales.
- Códigos de estado — Los 17 códigos, el significado de cada uno y cuáles nunca produce la propia biblioteca.
- Plazos — Por qué no hay uno predeterminado, cómo llegan al servidor y cómo se propagan entre servicios.
- Guía del lenguaje (proto 3) — Números de campo, campos reservados y reglas para cambiar un tipo de mensaje de forma segura.

Glosario

Elija según sus restricciones reales

Describa el flujo de trabajo, las interfaces existentes y los usuarios que necesitan el cambio. Podemos definir una entrega comprobable y un traspaso que permita a su equipo mantener lo entregado.

Vea cuánto cuesta Póngase en contacto →

Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es gRPC?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

Explore nuestros servicios

welcome@engineer.company +45 50 10 61 84 Copenhague, Dinamarca

Llévese esta página

Tarjeta de contacto

Creado con esmero y cariño por D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.

Impreso desde https://engineer.company/es/glossary/grpc/