Engineer

gRPC

et rammeverk for å kalle fjernprosedyrer over HTTP/2

Hva det er

gRPC er et rammeverk for fjernprosedyrekall. Et program kaller en metode på en server som kjører på en annen maskin, som om serveren var et lokalt objekt, og gRPC frakter kallet over nettverket i en egen protokoll oppå HTTP/2 og leverer svaret tilbake som en returverdi.

Alt begynner med en tjenestedefinisjon: metodene som kan kalles utenfra, med typen til hver forespørsel og hvert svar. Som standard skrives den i Protocol Buffers, Googles format med åpen kildekode for å serialisere data uavhengig av språk og plattform, i en fil med et navn som slutter på `.proto`. Andre formater kan kobles inn, men protocol buffers er det gRPC bruker med mindre noe annet er angitt. Kompilatoren `protoc` leser filen sammen med en gRPC-plugin og genererer kode for begge halvdelene. Klienten får en stub, et lokalt objekt med de samme metodene som tjenesten (i noen språk heter den rett og slett klienten), og serveren får et grensesnitt den skal implementere. Fordi begge halvdelene kommer fra samme fil, kan en server skrevet i Java svare klienter skrevet i Go, Python eller Ruby.

En metode har én av fire former. En unær metode tar én forespørsel og returnerer ett svar, som et vanlig funksjonskall. En metode med serverstrømming tar én forespørsel og sender tilbake en strøm av svar, en metode med klientstrømming gjør det motsatte, og en metode med toveis strømming lar begge sider sende strømmene sine uavhengig av hverandre. Innenfor ett kall garanterer gRPC at meldingene kommer frem i den rekkefølgen de ble sendt.

Under panseret kjører gRPC over HTTP/2. Hvert kall er en `POST` til en sti som består av det fullt kvalifiserte navnet på tjenesten og navnet på metoden, med innholdstypen `application/grpc`. Hver melding sendes bak et prefiks på fem byte: én byte som sier om meldingen er komprimert, og fire som angir lengden.

Utfallet av kallet er ikke HTTP-statusen. Et svar som gRPC selv lager, har status 200 også når kallet feiler, og det egentlige resultatet kommer i trailere, headerfelt som sendes etter kroppen: `grpc-status` med et tall og eventuelt `grpc-message` med en beskrivelse av feilen.

Det gjør gRPC til noe helt annet enn REST. Et REST-API navngir ressurser og arbeider på dem med metodene og statuskodene i selve HTTP, som regel med kropper i JSON som alle kan lese. Et gRPC-API er en liste med metoder med typede, binære meldinger, og HTTP/2 er bare transporten under. Dokumentasjonen til Protocol Buffers kaller selv formatet sitt «som JSON, bare mindre og raskere».

Bruk begrepet i en beslutning

Bruk «Hva er gRPC?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Delen brukerne aldri ser: hvor dataene ligger, hvilke regler som gjelder, og hvordan andre systemer får lov til å snakke med den.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Designet og driftet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tvers av tre databasemotorer

Designet og driftet 6 ETL/ELT-pipelines Ta kontakt →

Hvorfor det er viktig

Kontrakten kommer først, og koden følger av den. I mange HTTP-API-er er beskrivelsen og koden som leverer API-et, to separate ting som kan gli fra hverandre, og en klient blir ofte skrevet for hånd ut fra et dokument. I gRPC er `.proto`-filen det ene stedet der en metode eller et felt defineres, og alle klienter og servere genereres fra den. En endring i en melding når ut til alle språk gjennom den samme generatoren og ikke ved at noen leser et dokument på nytt.

Kodingen er kompakt, fordi hvert felt sendes som nummeret og verdien sin i stedet for navnet sitt. Prisen er at en melding ikke kan tolkes fullt ut uten `.proto`-filen sin, siden protocol buffers ikke beskriver seg selv, og at formatet går ut fra at en hel melding får plass i minnet på én gang. Formatets egen dokumentasjon anbefaler en annen løsning for data på mer enn noen få megabyte.

Strømming er en del av designet og ikke et tillegg. Kall kjører som strømmer på HTTP/2-forbindelser, så en metode med serverstrømming kan fortsette å sende resultater gjennom det samme maskineriet som et kall som svarer én gang.

Tidsgrensene følger med kallet. En klient setter en tidsfrist, en deadline som gRPC kaller det, gRPC sender det som er igjen av den, i headeren `grpc-timeout`, og når tiden er ute, får klienten `DEADLINE_EXCEEDED`, og serverens side av kallet blir avbrutt. Når en server i sin tur kaller en annen server, sender noen implementasjoner tidsfristen videre: Java og Go gjør det som standard, mens det må slås på i C++. Fordi tidsfristen krysser hvert ledd som gjenværende tid og ikke som et klokkeslett, gir maskiner med klokker som går ulikt, likevel opp på samme tidspunkt.

Feil har ett felles vokabular. gRPC definerer 17 statuskoder, fra 0 `OK` til 16 `UNAUTHENTICATED`, og de er en del av selve protokollen, så en Go-klient og en Java-server er enige om hva `NOT_FOUND` eller `UNAVAILABLE` betyr.

Alt dette passer for trafikk mellom tjenester som ett team eller ett selskap kontrollerer, der begge ender kan dele `.proto`-filene og bygge fra dem. Det passer dårligere for nettleseren. gRPC-prosjektets egne protokollnotater sier at nettlesere ikke gir skript tilgang til rammene i HTTP/2 og aldri kommer til å gjøre det, så kode i nettleseren snakker en egen protokoll, gRPC-Web, og i det vanlige oppsettet oversetter en proxy den til vanlig gRPC.

Slik ser det ut i praksis

Her er en liten faktureringstjeneste. En unær metode henter en faktura, og en metode med serverstrømming sender en kundes fakturaer én om gangen:

```
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;
}
```

Tallene er ikke standardverdier. Det er feltnumre, og i den binære kodingen identifiserer de hvert felt i stedet for navnet. Numrene fra 1 til 15 tar én byte, så de hører til feltene som oftest er satt. De to `reserved`-linjene holder orden på et felt som fantes en gang, nummer 4 med navnet `discount_code`, slik at verken nummeret eller navnet kan komme tilbake som et annet felt.

For Python genererer én kommando `billing_pb2.py` med meldingsklassene og `billing_pb2_grpc.py` med klient- og serverklassene:

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

Ett enkelt kall til `GetInvoice` ser da slik ut som HTTP/2-rammer:

```
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` betyr to sekunder, og `te = trailers` er der for å oppdage proxyer som ikke er kompatible med gRPC. Hvis fakturaen ikke finnes, begynner svaret likevel med `:status = 200`, og den siste rammen bærer `grpc-status = 5`, som er `NOT_FOUND`, sammen med en `grpc-message` som `invoice not found`. En server som feiler med en gang, kan til og med sende én enkelt `HEADERS`-ramme med statusen og ingen melding i det hele tatt.

Å generere begge sider fra én kontrakt er ikke noe særegent for gRPC. Et prosjekt i nettstedets porteføljе genererte API-kontrakten fra databasen og utover for et API beskrevet med OpenAPI. Kjeden starter i databasen og går utover: Go-typene definerer API-et, Huma lager OpenAPI-beskrivelsen ut fra dem, en typet TypeScript-klient og mock-handlere genereres fra den beskrivelsen, og en kontraktskontroll i CI får bygget til å feile når det frontenden sender, ikke lenger stemmer med det API-et forventer. gRPC bygger den samme disiplinen inn i verktøykjeden sin, med `.proto`-filen som eneste kilde.

Hvor det går galt

Et kall uten tidsfrist kan vente for alltid. gRPC setter ingen tidsfrist som standard, så en klient som kaller en server som har hengt seg, kan i praksis vente på svar for alltid. Gi hvert kall en realistisk tidsfrist. I Python er det argumentet `timeout`, oppgitt i sekunder:

```
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)
```

Når tiden er ute, kaster kallet `grpc.RpcError`, og unntakets `code()` returnerer `grpc.StatusCode.DEADLINE_EXCEEDED`. På serveren avbryter gRPC kallet med `CANCELLED` når tidsfristen er passert, men det er tjenestens egen kode som må stoppe arbeidet handleren har satt i gang, så en handler som jobber lenge, selv må sjekke om kallet er avbrutt, og stoppe.

En timeout betyr ikke at ingenting har skjedd. Klient og server avgjør hver for seg om et kall lyktes, så et kall kan fullføres på serveren og likevel feile hos klienten med `DEADLINE_EXCEEDED`. Endringer som ble gjort før en avbrytelse, blir heller ikke rullet tilbake. En klient som bare prøver igjen, kan opprette den samme fakturaen to ganger. Protokollen antar ikke at et kall er idempotent, med mindre det uttrykkelig er merket slik. `UNAVAILABLE` er koden som oftest er forbigående og verdt å prøve igjen med backoff, men dokumentasjonen av statuskodene advarer om at det ikke alltid er trygt å gjenta operasjoner som ikke er idempotente. Det samme dokumentet overlater til koden som gjør kallene, å velge hvilke koder som skal prøves på nytt.

Et gjenbrukt feltnummer ødelegger data. Et feltnummer kan ikke endres når meldingstypen først er i bruk. Hvis et felt slettes og nummeret senere går til et nytt felt, dekoder eldre klienter og lagrede meldinger det nye feltet ut fra bytes som betydde noe annet. Veiledningen til Protocol Buffers lister opp hva som kan følge: parsefeil, lekkede personopplysninger og ødelagte data. Nummerer aldri om et felt, og reserver hvert slettede nummer og navn, som i eksemplet over.

Hver feil ser ut som en suksess for HTTP. Fordi gRPC svarer med HTTP-status 200 også når et kall feiler, teller alt som bedømmer trafikk ut fra HTTP-statusen alene, for eksempel en tilgangslogg eller en feilrate bygd på statuskoder, feil som suksesser. Spesifikasjonen ber servere svare 415 på forespørsler med en annen innholdstype enn `application/grpc`, nettopp av den grunn, slik at andre HTTP/2-klienter ikke leser en gRPC-feil som en suksess. Tell på `grpc-status`, ikke på `:status`.

Lange strømmer binder belastningen sin til én server. Kall kjører som strømmer på en delt HTTP/2-forbindelse, og når en strøm først har startet, kan den ikke flyttes til en annen server. En strøm som varer i timevis, holder belastningen der den begynte, uansett hvor mange servere som kommer til senere. Hver forbindelse har også en grense for samtidige strømmer, og når den er nådd, venter flere kall hos klienten til en strøm er ferdig. Ytelsesguiden til gRPC anbefaler å strømme bare der det gir en betydelig gevinst i ytelse eller enkelhet, og å gi hvert område med høy belastning sin egen kanal.

Store datamengder får ikke plass i én melding. gRPC håndhever en maksimal meldingsstørrelse, og en melding som er større enn den innstilte grensen, feiler med `RESOURCE_EXHAUSTED` på den siden som sender eller mottar den. Å heve grensen flytter bare problemet, fordi protocol buffers forventer hele meldingen

i minnet, og med store data kan flere serialiserte kopier av den ligge der samtidig. Del store data opp i en strøm av mindre meldinger, eller flytt dem ut av gRPC.

Les mer

- Core concepts, architecture and lifecycle — tjenestedefinisjoner, de fire typene metoder, stubber, tidsfrister og avbrytelse.
- gRPC over HTTP2 — protokollen på nettverksnivå: stier, headere, meldingsprefikset og statusen i trailerne.
- Status Codes — alle de 17 kodene, hva hver av dem betyr, og hvilke biblioteket selv aldri lager.
- Deadlines — hvorfor det ikke finnes noen standard, hvordan tidsfrister når frem til serveren, og hvordan de sendes videre mellom tjenester.
- Language Guide (proto 3) — feltnumre, reserverte felt og reglene for å endre en meldingstype trygt.

Ordliste

Velg ut fra deres faktiske rammer

Beskriv arbeidsflyten, eksisterende grensesnitt og brukerne som trenger endringen. Vi kan definere en testbar leveranse og en overlevering som gjør teamet i stand til å vedlikeholde løsningen.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er gRPC?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

welcome@engineer.company +45 50 10 61 84 København, Danmark

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/grpc/