**Engineer**

# gRPC

et framework til at kalde fjernprocedurer over HTTP/2

## Hvad det er

gRPC er et framework til fjernprocedurekald. Et program kalder en metode på en server, der kører på en anden maskine, som om serveren var et lokalt objekt, og gRPC fører kaldet over netværket i sin egen protokol oven på HTTP/2 og leverer svaret tilbage som en returværdi.

Alt begynder med en tjenestedefinition: de metoder, der kan kaldes udefra, med typen på hver forespørgsel og hvert svar. Som standard skrives den i Protocol Buffers, Googles open source-format til at serialisere data uafhængigt af sprog og platform, i en fil, hvis navn ender på `.proto`. Andre formater er mulige, men protocol buffers er standardvalget. Compileren `protoc` læser filen sammen med et gRPC-plugin og genererer kode til begge halvdele. Klienten får en stub, et lokalt objekt med de samme metoder som tjenesten (i nogle sprog hedder den bare klienten), og serveren får en grænseflade, den skal implementere. Fordi begge halvdele kommer fra samme fil, kan en server skrevet i Java svare klienter skrevet i Go, Python eller Ruby.

En metode har en af fire former. En unær metode tager én forespørgsel og returnerer ét svar, ligesom et almindeligt funktionskald. En metode med serverstreaming tager én forespørgsel og sender en strøm af svar tilbage, en metode med klientstreaming gør det omvendte, og en metode med tovejsstreaming lader begge sider sende deres strømme uafhængigt af hinanden. Inden for ét kald garanterer gRPC, at beskederne når frem i den rækkefølge, de blev sendt i.

Under motorhjelmen kører gRPC over HTTP/2. Hvert kald er en `POST` til en sti, der består af tjenestens fuldt kvalificerede navn og metodens navn, med indholdstypen `application/grpc`. Hver besked sendes bag et præfiks på fem byte: én byte, der angiver, om beskeden er komprimeret, og fire, der angiver dens

længde. Kaldets udfald er ikke HTTP-statussen. Et svar, som gRPC selv danner, har status 200, også når kaldet fejler, og det egentlige resultat kommer i trailere, headerfelter, der sendes efter brødteksten: `grpc-status` med et tal og eventuelt `grpc-message` med en beskrivelse af fejlen.

Det gør gRPC til noget helt andet end REST. Et REST-API navngiver ressourcer og arbejder på dem med HTTP's egne metoder og statuskoder, som regel med brødtekster i JSON, som alle kan læse. Et gRPC-API er en liste af metoder med typede, binære beskeder, og HTTP/2 er bare transporten nedenunder. Dokumentationen til Protocol Buffers sammenfatter selv sit format som »ligesom JSON, bare mindre og hurtigere«.

Brug begrebet i en beslutning

Brug »Hvad er gRPC?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Den del, brugerne aldrig ser: hvor data ligger, hvilke regler der gælder, og hvordan andre systemer må tale med den.

**Se vores ydelser** Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har designet og drevet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tre databasemotorer

**Har designet og drevet 6 ETL/ELT-pipelines** Kontakt os →

## Hvorfor det betyder noget

Kontrakten kommer først, og koden følger af den. Med mange HTTP-API'er er beskrivelsen og den kode, der leverer API'et, to adskilte ting, der kan glide fra hinanden, og en klient bliver ofte skrevet i hånden ud fra et dokument. I gRPC er `.proto`-filen det ene sted, hvor en metode eller et felt defineres, og alle klienter og servere genereres ud fra den. En ændring af en besked når ud til alle sprog gennem den samme generator og ikke ved, at nogen genlæser et dokument.

Kodningen er kompakt, fordi hvert felt sendes som sit nummer og sin værdi i stedet for sit navn. Prisen er, at en besked ikke kan tolkes fuldt ud uden sin `.proto`-fil, da protocol buffers ikke beskriver sig selv, og at formatet går ud fra, at en hel besked kan ligge i hukommelsen på én gang. Formatets egen dokumentation anbefaler en anden løsning til data på mere end et par megabyte.

Streaming er en del af designet og ikke en tilføjelse. Kald kører som streams på HTTP/2-forbindelser, så en metode med serverstreaming kan blive ved med at sende resultater gennem det samme maskineri som et kald, der svarer én gang.

Tidsgrænser følger med kaldet. En klient sætter en deadline, gRPC sender det, der er tilbage af den, i headeren `grpc-timeout`, og når tiden løber ud, får klienten `DEADLINE_EXCEEDED`, og serverens side af kaldet bliver annulleret. Når en server selv kalder en anden server, sender nogle implementeringer deadlinen videre: Java og Go gør det som standard, mens det skal slås til i C++. Fordi deadlinen krydser hvert led som den resterende tid og ikke som et klokkeslæt, giver maskiner, hvis ure går forskelligt, alligevel op på samme tidspunkt.

Fejl har ét fælles ordforråd. gRPC definerer 17 statuskoder, fra 0 `OK` til 16 `UNAUTHENTICATED`, og de er en del af selve protokollen, så en Go-klient og en Java-server er enige om, hvad `NOT_FOUND` eller `UNAVAILABLE` betyder.

Alt dette passer til trafik mellem tjenester, som ét team eller én virksomhed styrer, hvor begge ender kan dele `.proto`-filerne og bygge ud fra dem. Det passer dårligere til browseren. gRPC-projektets egne protokolnoter siger, at browsere ikke giver scripts adgang til HTTP/2-framingen og aldrig vil gøre det, så kode i browseren taler en separat protokol, gRPC-Web, og i den typiske opsætning oversætter en proxy den til almindelig gRPC.

## Sådan ser det ud i praksis

Her er en lille faktureringstjeneste. En unær metode henter en faktura, og en metode med serverstreaming sender en kundes fakturaer én ad 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 standardværdier. Det er feltnumre, og i den binære kodning identificerer de hvert felt i stedet for dets navn. Numrene fra 1 til 15 fylder én byte, så de hører til de felter, der oftest er sat. De to `reserved` -linjer holder styr på et felt, der engang fandtes, nummer 4 med navnet `discount_code` , så hverken nummeret eller navnet kan vende tilbage som et andet felt.

Til Python genererer én kommando `billing_pb2.py` med beskedklasserne og `billing_pb2_grpc.py` med klient- og serverklasserne:

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

Et enkelt kald til `GetInvoice` ser så sådan ud som HTTP/2-frames:

```
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` betyder to sekunder, og `te = trailers` er der for at opdage proxyer, der ikke er kompatible med gRPC. Hvis fakturaen ikke findes, begynder svaret stadig med `:status = 200`, og den sidste frame bærer `grpc-status = 5`, som er `NOT_FOUND`, sammen med en `grpc-message` som `invoice not found`. En server, der fejler med det samme, kan endda sende en enkelt `HEADERS`-frame med statussen og slet ingen besked.

At generere begge sider ud fra én kontrakt er ikke noget særligt for gRPC. Et projekt i sitets portefølje genererede API-kontrakten udad fra databasen til et API beskrevet med OpenAPI. Kæden starter ved databasen og løber udad: Go-typerne definerer API'et, Huma genererer OpenAPI-beskrivelsen ud fra dem, en typet TypeScript-klient og mock-handlere genereres ud fra den beskrivelse, og et kontrakttjek i CI får buildet til at fejle, når det, frontenden sender, ikke længere stemmer med det, API'et forventer. gRPC bygger den samme disciplin ind i sin værktøjskæde med `.proto`-filen som eneste kilde.

## Hvor det går galt

**Et kald uden deadline kan vente for evigt.** gRPC sætter ingen deadline som standard, så en klient, der kalder en server, som er gået i stå, kan i praksis vente på et svar for evigt. Giv hvert kald en realistisk deadline. I Python er det argumentet `timeout`, angivet 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 løber ud, rejser kaldet `grpc.RpcError`, og dets `code()` returnerer `grpc.StatusCode.DEADLINE_EXCEEDED`. På serveren annullerer gRPC kaldet med `CANCELLED`, når deadlinen er overskredet, men det er tjenestens egen kode, der skal standse det arbejde, handleren har sat i gang, så en handler, der arbejder længe, selv må tjekke, om kaldet er annulleret, og stoppe.

**En timeout betyder ikke, at intet er sket.** Klient og server afgør hver for sig, om et kald lykkedes, så et kald kan gennemføres på serveren og alligevel fejle hos klienten med `DEADLINE_EXCEEDED`. Ændringer, der blev foretaget før en annullering, bliver heller ikke rullet tilbage. En klient, der bare prøver igen, kan oprette den samme faktura to gange. Protokollen antager ikke, at et kald er idempotent, medmindre det udtrykkeligt er angivet. `UNAVAILABLE` er den kode, der oftest er forbigående og værd at prøve igen med backoff, men dokumentationen af statuskoderne advarer om, at det ikke altid er sikkert at gentage operationer, der ikke er idempotente. Samme dokument overlader det til den kode, der foretager kaldene, at vælge, hvilke koder der skal prøves igen.

**Et genbrugt feltnummer ødelægger data.** Et feltnummer kan ikke ændres, når beskedtypen først er i brug. Hvis et felt slettes, og dets nummer senere går til et nyt felt, afkoder ældre klienter og gemte beskeder det nye felt ud fra bytes, der betød noget andet. Guiden til Protocol Buffers opregner, hvad der kan følge: parsefejl, lækkede persondata og ødelagte data. Omnummerer aldrig et felt, og reserver hvert slettet nummer og navn, som i eksemplet ovenfor.

**Enhver fejl ligner en succes for HTTP.** Fordi gRPC svarer med HTTP-status 200, også når et kald fejler, tæller alt, der bedømmer trafik ud fra HTTP-statussen alene, for eksempel en adgangslog eller en fejlrate bygget på statuskoder, fejl som succeser. Specifikationen beder servere svare 415 på forespørgsler, hvis indholdstype ikke er `application/grpc`, netop af den grund, så andre HTTP/2-klienter ikke læser en gRPC-fejl som en succes. Tæl på `grpc-status`, ikke på `:status`.

**Lange streams binder deres belastning til én server.** Kald kører som streams på en delt HTTP/2-forbindelse, og når en stream først er startet, kan den ikke flyttes til en anden server. En stream, der varer i timevis, holder sin belastning, hvor den begyndte, uanset hvor mange servere der kommer til senere. Hver forbindelse har også en grænse for samtidige streams, og når den er nået, venter yderligere kald hos klienten, til en stream bliver færdig. gRPC's guide om ydeevne anbefaler kun at streame, hvor det giver en væsentlig gevinst i ydeevne eller enkelhed, og at give hvert område med høj belastning sin egen kanal.

**Store mængder data passer ikke i én besked.** gRPC håndhæver en maksimal beskedstørrelse, og en besked, der er større end den indstillede grænse, fejler med `RESOURCE_EXHAUSTED` på den side, der sender eller modtager den. At hæve grænsen flytter kun problemet, fordi protocol buffers forventer hele beskeden i

hukommelsen, og med store data kan der ligge flere serialiserede kopier af den der på én gang. Del store data op i en stream af mindre beskeder, eller flyt dem uden for gRPC.

## Læs mere

- Core concepts, architecture and lifecycle — tjenestedefinitioner, de fire slags metoder, stubs, deadlines og annullering.
- gRPC over HTTP2 — protokollen på forbindelsen: stier, headere, beskedpræfikset og statussen i trailerne.
- Status Codes — alle 17 koder, hvad hver af dem betyder, og hvilke biblioteket selv aldrig danner.
- Deadlines — hvorfor der ikke er nogen standard, hvordan deadlines når frem til serveren, og hvordan de sendes videre mellem tjenester.
- Language Guide (proto 3) — feltnumre, reserverede felter og reglerne for at ændre en beskedtype sikkert.

Ordliste

Vælg ud fra jeres faktiske vilkår

Beskriv arbejdsgangen, de eksisterende grænseflader og de brugere, der har brug for ændringen. Vi kan definere en testbar leverance og en overdragelse, der gør teamet i stand til at vedligeholde løsningen.

**Se, hvad det koster** Kontakt os →

# Gør begreberne praktiske

Spørg, hvordan »Hvad er gRPC?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

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

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

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

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/grpc/