Engineer

gRPC

ett ramverk för att anropa fjärrprocedurer över HTTP/2

Vad det är

gRPC är ett ramverk för fjärrproceduranrop. Ett program anropar en metod på en server som körs på en annan maskin som om servern vore ett lokalt objekt, och gRPC för anropet över nätverket i ett eget protokoll ovanpå HTTP/2 och lämnar tillbaka svaret som ett returvärde.

Allt börjar med en tjänstedefinition: de metoder som kan anropas utifrån, med typen för varje begäran och varje svar. Som standard skrivs den i Protocol Buffers, Googles format med öppen källkod för att serialisera data oberoende av språk och plattform, i en fil vars namn slutar på `.proto`. Andra format kan kopplas in, men protocol buffers är det gRPC använder om inget annat anges. Kompilatorn `protoc` läser filen tillsammans med ett gRPC-plugin och genererar kod för båda halvorna. Klienten får en stub, ett lokalt objekt med samma metoder som tjänsten (i vissa språk kallas den helt enkelt klienten), och servern får ett gränssnitt att implementera. Eftersom båda halvorna kommer från samma fil kan en server skriven i Java svara klienter skrivna i Go, Python eller Ruby.

En metod har en av fyra former. En unär metod tar emot en begäran och returnerar ett svar, som ett vanligt funktionsanrop. En metod med serverströmning tar emot en begäran och skickar tillbaka en ström av svar, en metod med klientströmning gör tvärtom, och en metod med dubbelriktad strömning låter båda sidor skicka sina strömmar oberoende av varandra. Inom ett och samma anrop garanterar gRPC att meddelandena kommer fram i den ordning de skickades.

Under huven körs gRPC över HTTP/2. Varje anrop är en `POST` till en sökväg som består av tjänstens fullständigt kvalificerade namn och metodens namn, med innehållstypen `application/grpc`. Varje meddelande skickas bakom ett prefix på fem byte: en byte som anger om meddelandet är komprimerat och fyra som

anger dess längd. Anropets utfall är inte HTTP-statusen. Ett svar som gRPC själv skapar har status 200 även när anropet misslyckas, och det egentliga resultatet kommer i trailers, huvudfält som skickas efter meddelandekroppen: `grpc-status` med ett tal och eventuellt `grpc-message` med en beskrivning av felet.

Det gör gRPC till något helt annat än REST. Ett REST-API namnger resurser och arbetar med dem genom HTTP:s egna metoder och statuskoder, oftast med JSON-kroppar som vem som helst kan läsa. Ett gRPC-API är en lista med metoder med typade, binära meddelanden, och HTTP/2 är bara transporten under. Dokumentationen för Protocol Buffers sammanfattar själv sitt format så här: ”som JSON, fast mindre och snabbare”.

Använd begreppet i ett beslut

Använd ”Vad är gRPC?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Den del användarna aldrig ser: där data lagras, vilka reglerna är och hur andra system får kommunicera med den.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Utformade och drev 6 ETL/ELT-pipelines — 6 ETL-pipelines över tre databasmotorer

Utformade och drev 6 ETL/ELT-pipelines Hör av er →

Varför det spelar roll

Kontraktet kommer först och koden följer av det. I många HTTP-API:er är beskrivningen och koden som levererar API:et två separata saker som kan glida isär, och en klient skrivs ofta för hand utifrån ett dokument. I gRPC är `.proto`-filen det enda stället där en metod eller ett fält definieras, och alla klienter och servrar genereras från den. En ändring av ett meddelande når alla språk genom samma generator och inte genom att någon läser om ett dokument.

Kodningen är kompakt eftersom varje fält skickas som sitt nummer och sitt värde i stället för sitt namn. Priset är att ett meddelande inte kan tolkas fullt ut utan sin `.proto`-fil, eftersom protocol buffers inte beskriver sig själva, och att formatet förutsätter att ett helt meddelande ryms i minnet på en gång. Formatets egen dokumentation rekommenderar en annan lösning för data som är större än några megabyte.

Strömning är en del av designen och inte ett tillägg. Anrop körs som strömmar på HTTP/2-anslutningar, så en metod med serverströmning kan fortsätta skicka resultat genom samma maskineri som ett anrop som svarar en gång.

Tidsgränser följer med anropet. En klient sätter en tidsfrist, en deadline som gRPC kallar det, gRPC skickar det som återstår av den i huvudfältet `grpc-timeout`, och när tiden tar slut får klienten `DEADLINE_EXCEEDED` och serverns sida av anropet avbryts. När en server i sin tur anropar en annan server skickar vissa implementationer tidsfristen vidare: Java och Go gör det som standard, medan det måste slås på i C++. Eftersom tidsfristen passerar varje led som återstående tid och inte som ett klockslag, ger maskiner vars klockor går olika ändå upp vid samma tidpunkt.

Fel har ett gemensamt ordförråd. gRPC definierar 17 statuskoder, från 0 `OK` till 16 `UNAUTHENTICATED`, och de är en del av själva protokollet, så en Go-klient och en Java-server är överens om vad `NOT_FOUND` eller `UNAVAILABLE` betyder.

Allt detta passar trafik mellan tjänster som ett team eller ett företag kontrollerar, där båda ändar kan dela `.proto`-filerna och bygga från dem. Det passar sämre för webbläsaren. gRPC-projektets egna protokollanteckningar säger att webbläsare inte ger skript tillgång till ramarna i HTTP/2 och aldrig kommer att göra det, så kod i webbläsaren talar ett separat protokoll, gRPC-Web, och i den vanliga uppsättningen översätter en proxy det till vanlig gRPC.

Så ser det ut i praktiken

Här är en liten faktureringstjänst. En unär metod hämtar en faktura och en metod med serverströmning skickar en kunds fakturor en i taget:

```
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;
}
```

Siffrorna är inte standardvärden. De är fältnummer, och i den binära kodningen identifierar de varje fält i stället för dess namn. Nummer 1 till 15 tar en byte att koda, så de hör till de fält som oftast är satta. De två `reserved`-raderna håller reda på ett fält som en gång fanns, nummer 4 med namnet `discount_code`, så att varken numret eller namnet kan komma tillbaka som ett annat fält.

För Python genererar ett enda kommando `billing_pb2.py` med meddelandeklasserna och `billing_pb2_grpc.py` med klient- och serverklasserna:

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

Ett enskilt anrop till `GetInvoice` ser då ut så här som HTTP/2-ramar:

```
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 två sekunder, och `te = trailers` finns där för att upptäcka proxyservrar som inte är kompatibla med gRPC. Om fakturan inte finns börjar svaret ändå med `:status = 200`, och den sista ramen bär `grpc-status = 5`, som är `NOT_FOUND`, tillsammans med ett `grpc-message` som `invoice not found`. En server som misslyckas direkt kan till och med skicka en enda `HEADERS`-ram med statusen och inget meddelande alls.

Att generera båda sidor från ett kontrakt är inget unikt för gRPC. Ett projekt i webbplatsens portfölj genererade API-kontraktet utåt från databasen för ett API som beskrivs med OpenAPI. Kedjan börjar i databasen och går utåt: Go-typerna definierar API:et, Huma genererar OpenAPI-beskrivningen utifrån dem, en typad TypeScript-klient och mock-hanterare genereras från den beskrivningen, och en kontraktskontroll i CI stoppar bygget när det frontenden skickar inte längre stämmer med vad API:et förväntar sig. gRPC bygger in samma disciplin i sin verktygskedja, med `.proto`-filen som enda källa.

Var det går fel

Ett anrop utan tidsfrist kan vänta för evigt. gRPC sätter ingen tidsfrist som standard, så en klient som anropar en server som har hängt sig kan i praktiken vänta på ett svar för evigt. Ge varje anrop en realistisk tidsfrist. I Python är 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)
```

När tiden tar slut kastar anropet `grpc.RpcError`, och undantagets `code()` returnerar `grpc.StatusCode.DEADLINE_EXCEEDED`. På servern avbryter gRPC anropet med `CANCELLED` när tidsfristen har passerat, men det är tjänstens egen kod som måste stoppa det arbete som hanteraren har satt i gång, så en hanterare som arbetar länge måste själv kontrollera om anropet har avbrutits och sluta.

En timeout betyder inte att inget har hänt. Klient och server avgör var för sig om ett anrop lyckades, så ett anrop kan slutföras på servern och ändå misslyckas hos klienten med `DEADLINE_EXCEEDED`. Ändringar som gjordes före ett avbrott rullas inte heller tillbaka. En klient som bara försöker igen kan skapa samma faktura två gånger. Protokollet antar inte att ett anrop är idempotent om det inte uttryckligen är markerat så. `UNAVAILABLE` är den kod som oftast är tillfällig och värd att försöka igen med backoff, men dokumentationen av statuskoderna varnar för att det inte alltid är säkert att upprepa operationer som inte är idempotenta. Samma dokument överlåter åt koden som gör anropen att välja vilka koder som ska försökas igen.

Ett återanvänt fältnummer förstör data. Ett fältnummer kan inte ändras när meddelandetypen väl används. Om ett fält tas bort och dess nummer senare går till ett nytt fält, avkodar äldre klienter och lagrade meddelanden det nya fältet från byte som betydde något annat. Guiden för Protocol Buffers räknar upp vad som kan följa: parsningsfel, läckta personuppgifter och korrupta data. Numrera aldrig om ett fält, och reservera varje borttaget nummer och namn, som i exemplet ovan.

Varje fel ser ut som en framgång för HTTP. Eftersom gRPC svarar med HTTP-status 200 även när ett anrop misslyckas, räknar allt som bedömer trafik efter HTTP-statusen ensam, till exempel en åtkomstlogg eller en felfrekvens byggd på statuskoder, fel som lyckade anrop. Specifikationen ber servrar svara 415 på begäranden vars innehållstyp inte är `application/grpc`, just av det skälet, så att andra HTTP/2-klienter inte läser ett gRPC-fel som en framgång. Räkna på `grpc-status`, inte på `:status`.

Långa strömmar binder sin last till en server. Anrop körs som strömmar på en delad HTTP/2-anslutning, och när en ström väl har startat kan den inte flyttas till en annan server. En ström som lever i timmar behåller sin last där den började, hur många servrar som än läggs till senare. Varje anslutning har också en gräns för samtidiga strömmar, och när den är nådd väntar ytterligare anrop hos klienten tills en ström blir klar. Prestandaguiden för gRPC rekommenderar att strömma bara där det ger en väsentlig vinst i prestanda eller enkelhet, och en separat kanal för varje område med hög last.

Stora datamängder ryms inte i ett meddelande. gRPC tillämpar en maximal meddelandestorlek, och ett meddelande som är större än den inställda gränsen misslyckas med `RESOURCE_EXHAUSTED` på den sida som skickar eller tar emot det. Att höja gränsen flyttar bara problemet, eftersom protocol buffers förväntar sig hela

meddelandet i minnet, och med stora data kan flera serialiserade kopior av det ligga där samtidigt. Dela upp stora data i en ström av mindre meddelanden, eller flytta dem utanför gRPC.

Läs mer

- Core concepts, architecture and lifecycle — tjänstedefinitioner, de fyra typerna av metoder, stubbar, tidsfrister och avbrytande.
- gRPC over HTTP2 — protokollet på nätverksnivå: sökvägar, huvudfält, meddelandeprefixet och statusen i trailers.
- Status Codes — alla 17 koder, vad var och en betyder och vilka biblioteket självt aldrig skapar.
- Deadlines — varför det inte finns något standardvärde, hur tidsfrister når servern och hur de förs vidare mellan tjänster.
- Language Guide (proto 3) — fältnummer, reserverade fält och reglerna för att ändra en meddelandetyp på ett säkert sätt.



Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är gRPC?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

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

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/grpc/