Engineer

gRPC

Framework d'appels de procédures distantes sur HTTP/2

Ce que c'est

gRPC est un framework d'appels de procédures distantes. Un programme appelle une méthode sur un serveur exécuté sur une autre machine comme si ce serveur était un objet local. gRPC transporte l'appel sur le réseau dans son propre protocole, au-dessus d'HTTP/2, puis renvoie la réponse sous forme de valeur de retour.

Tout commence par une définition du service : la liste des opérations distantes et les types de requête et de réponse qu'accepte chacune. Dans la configuration standard, elle est écrite en Protocol Buffers, le format open source de Google qui sérialise les données indépendamment du langage et de la plateforme, dans un fichier dont le nom se termine par `.proto`. D'autres formats peuvent être intégrés, mais gRPC utilise Protocol Buffers sauf indication contraire. Le compilateur `protoc`, associé à un plugin gRPC, lit le fichier et génère le code des deux côtés. Le client reçoit un stub, objet local doté des mêmes méthodes que le service ; certains langages l'appellent simplement le client. Le serveur reçoit une interface à implémenter. Puisque les deux côtés proviennent d'un même fichier, un serveur écrit en Java peut répondre à des clients écrits en Go, Python ou Ruby.

Une méthode peut prendre quatre formes. Une méthode unaire reçoit une requête et renvoie une réponse, comme un appel de fonction ordinaire. Une méthode de streaming serveur reçoit une requête et renvoie un flux de réponses ; le streaming client fait l'inverse, et le streaming bidirectionnel permet aux deux côtés d'envoyer leurs flux indépendamment. Au sein d'un même appel, gRPC garantit que les messages arrivent dans l'ordre de leur envoi.

Sur le réseau, gRPC fonctionne sur HTTP/2. Chaque appel est un `POST` vers un chemin composé du nom pleinement qualifié du service et du nom de la méthode, avec le type de contenu `application/grpc`. Chaque message est précédé d'un préfixe de cinq octets : un indique si le message est compressé, les quatre autres donnent sa longueur. Le résultat de l'appel n'est pas le statut HTTP. Une réponse produite par gRPC porte le statut 200 même si l'appel échoue ; le résultat réel arrive dans les champs d'en-tête finaux, envoyés après le corps : `grpc-status`, avec un nombre, et éventuellement `grpc-message`, avec une description de l'erreur.

C'est une approche différente de REST. Une API REST nomme des ressources et agit sur elles au moyen des méthodes et codes de statut d'HTTP, généralement avec des corps JSON lisibles par tous. Une API gRPC est une liste de méthodes aux messages binaires typés, et HTTP/2 n'est que le transport sous-jacent. La documentation de Protocol Buffers compare le format à JSON en soulignant la taille inférieure des messages et la rapidité du traitement.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que gRPC ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. La partie invisible pour les utilisateurs : où résident les données, quelles règles s'appliquent et comment les autres systèmes peuvent y accéder.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 6 pipelines ETL/ELT conçus et exploités — 6 pipelines ETL sur trois moteurs de bases de données

6 pipelines ETL/ELT conçus et exploités Contactez-nous →

Pourquoi c'est important

Le contrat vient d'abord ; le code en découle. Dans de nombreuses API HTTP, la description et le code qui implémente l'API sont deux éléments distincts qui peuvent diverger, et le client est souvent écrit à la main à partir d'un document. Avec gRPC, le fichier `.proto` est l'unique endroit où une méthode ou un champ est défini, et tous les clients et serveurs sont générés à partir de ce fichier. Une modification d'un message atteint chaque langage par le même générateur, sans dépendre de la relecture d'un document.

L'encodage est compact, car chaque champ est transmis sous forme de numéro et de valeur plutôt que de nom. En contrepartie, il est impossible d'interpréter entièrement un message sans son fichier `.proto`, puisque le format Protocol

Buffers n'est pas autodescriptif, et le format suppose qu'un message entier tient simultanément en mémoire. Sa propre documentation conseille une autre solution pour des données dépassant quelques mégaoctets.

Le streaming fait partie de la conception, au lieu d'être ajouté après coup. Les appels sont des flux sur des connexions HTTP/2 : une méthode de streaming serveur peut donc continuer à envoyer des résultats avec les mêmes mécanismes qu'un appel qui ne répond qu'une fois.

Les limites de temps accompagnent l'appel. Le client fixe une échéance, gRPC transmet le temps restant dans l'en-tête `grpc-timeout` et, lorsque le temps est écoulé, le client reçoit `DEADLINE_EXCEEDED` tandis que le côté serveur de l'appel est annulé. Lorsqu'un serveur en appelle un autre, certaines implémentations propagent l'échéance : Java et Go le font sans activation supplémentaire, tandis qu'il faut l'activer en C++. Comme l'échéance franchit chaque étape sous forme de temps restant plutôt que d'heure absolue, des machines aux horloges désaccordées s'accordent tout de même sur le moment où abandonner.

Les erreurs partagent un vocabulaire. gRPC définit 17 codes de statut, de 0 `OK` à 16 `UNAUTHENTICATED`, qui font partie du protocole lui-même : un client Go et un serveur Java s'accordent donc sur le sens de `NOT_FOUND` ou de `UNAVAILABLE`.

Tout cela convient aux échanges entre services contrôlés par une même équipe ou une même entreprise, dont les deux extrémités peuvent partager les fichiers `.proto` et générer leur code à partir de ceux-ci. Cela convient moins au navigateur. Les notes de protocole du projet gRPC affirment que les navigateurs n'exposent pas les trames HTTP/2 aux scripts et ne le feront jamais. Le code du navigateur parle donc un protocole distinct, gRPC-Web, qu'un proxy traduit habituellement en gRPC natif.

À quoi cela ressemble en pratique

Voici un petit service de facturation. Une méthode unaire récupère une facture et une méthode de streaming serveur envoie les factures d'un client une par une :

```
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;
}
```

Les nombres ne sont pas des valeurs attribuées implicitement. Ce sont des numéros de champ qui identifient chaque champ sur le réseau à la place de son nom. Les numéros de 1 à 15 s'encodent sur un octet ; ils reviennent donc aux champs renseignés le plus souvent. Les deux lignes `reserved` gardent la trace d'un ancien champ, le numéro 4 nommé `discount_code`, afin que ni son numéro ni son nom ne puissent être réutilisés pour un autre champ.

Pour Python, une seule commande génère `billing_pb2.py`, avec les classes des messages, et `billing_pb2_grpc.py`, avec celles du client et du serveur :

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

Un appel à `GetInvoice` se présente alors sous la forme des trames HTTP/2 suivantes :

```
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` signifie deux secondes, et `te = trailers` sert à détecter les proxies incompatibles avec gRPC. Si la facture n'existe pas, la réponse commence tout de même par `:status = 200`, et la dernière trame porte `grpc-status = 5`, soit `NOT_FOUND`, ainsi qu'un `grpc-message` tel que `invoice not found`. Un serveur qui échoue immédiatement peut même envoyer une seule trame `HEADERS` contenant le statut, sans aucun message.

Générer les deux côtés à partir d'un contrat n'est pas propre à gRPC. Un projet du portfolio de ce site a généré le contrat de l'API depuis la base de données vers l'extérieur pour une API décrite avec OpenAPI. La chaîne part de la base de données : les types Go définissent l'API, Huma en produit la description OpenAPI, puis un client TypeScript typé et des gestionnaires simulés sont générés à partir de cette description. Un contrôle du contrat en CI fait échouer le processus de construction dès que ce qu'envoie le frontend ne correspond plus à ce qu'attend l'API. gRPC inscrit la même discipline dans ses outils, avec le fichier `.proto` comme source unique.

Où les problèmes surviennent

Un appel sans échéance peut attendre indéfiniment. gRPC ne fixe aucune échéance de lui-même : un client qui appelle un serveur bloqué peut donc attendre une réponse pratiquement sans fin. Donnez à chaque appel une échéance réaliste. En Python, il s'agit de l'argument `timeout`, exprimé en secondes :

```
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)
```

Lorsque le temps est écoulé, l'appel lève `grpc.RpcError`, dont la méthode `code()` renvoie `grpc.StatusCode.DEADLINE_EXCEEDED`. Sur le serveur, gRPC annule l'appel avec `CANCELLED` une fois l'échéance passée, mais arrêter le travail déjà lancé par le gestionnaire appartient au code du service. Un gestionnaire de longue durée doit donc vérifier l'annulation et s'arrêter lui-même.

Un délai dépassé ne signifie pas que rien ne s'est passé. Client et serveur décident chacun si un appel a réussi : il peut être achevé sur le serveur et pourtant échouer sur le client avec `DEADLINE_EXCEEDED`. Annuler un appel n'annule pas le travail déjà réalisé. Un client qui réessaie simplement peut créer deux fois la même facture. Le protocole ne suppose pas qu'un appel est idempotent sauf s'il est indiqué comme tel. `UNAVAILABLE` est le code qui a le plus de chances de signaler un problème temporaire justifiant une nouvelle tentative avec une temporisation entre les essais, mais la documentation des codes prévient qu'il n'est pas toujours sûr de réessayer une opération non idempotente. Elle laisse au code appelant le choix des codes sur lesquels réessayer.

Réutiliser un numéro de champ corrompt les données. Un numéro de champ ne peut plus changer une fois le type de message utilisé. Si un champ est supprimé et son numéro attribué ensuite à un nouveau champ, les anciens clients et messages stockés décodent ce nouveau champ à partir d'octets qui signifiaient autre chose. Le guide Protocol Buffers énumère les conséquences possibles : erreurs d'analyse, divulgation de données personnelles et corruption de données. Ne renumérotez jamais un champ et réservez chaque numéro et nom supprimé, comme dans l'exemple ci-dessus.

Chaque échec ressemble à un succès pour HTTP. Puisque gRPC répond avec le statut HTTP 200 même lorsqu'un appel échoue, tout ce qui juge le trafic d'après ce seul statut, comme un journal d'accès ou un taux d'erreur fondé sur les codes de statut, compte les échecs comme des succès. C'est précisément pour éviter

que d'autres clients HTTP/2 interprètent une erreur gRPC comme un succès que la spécification prévoit une réponse 415 aux requêtes dont le type de contenu n'est pas `application/grpc`. Comptez `grpc-status`, pas `:status`.

Les flux longs fixent leur charge sur un serveur. Les appels sont des flux sur une connexion HTTP/2 partagée, et un flux commencé ne peut plus être déplacé vers un autre serveur. Un flux qui dure des heures maintient sa charge là où il a commencé, quel que soit le nombre de serveurs ajoutés ensuite. Chaque connexion limite aussi le nombre de flux simultanés ; une fois cette limite atteinte, les appels supplémentaires attendent sur le client qu'un flux se termine. Le guide de performances de gRPC conseille le streaming uniquement lorsqu'il apporte un gain substantiel de performances ou de simplicité, et un canal distinct pour chaque zone de forte charge.

Les gros volumes ne tiennent pas dans un message. gRPC impose une taille maximale de message ; un message qui dépasse la limite configurée échoue avec `RESOURCE_EXHAUSTED` du côté qui l'envoie ou le reçoit. Augmenter cette limite ne fait que déplacer le problème : Protocol Buffers exige que le message entier tienne en mémoire, où plusieurs copies sérialisées de données volumineuses peuvent se retrouver simultanément. Découpez les données volumineuses en un flux de messages plus petits ou sortez-les de gRPC.

Pour aller plus loin

- Concepts fondamentaux, architecture et cycle de vie — Définitions de services, quatre formes de méthodes, stubs, échéances et annulation.
- gRPC sur HTTP2 — Protocole réseau : chemins, en-têtes, préfixe des messages et statut dans les champs finaux.
- Codes de statut — Les 17 codes, leur sens et ceux que la bibliothèque ne produit jamais elle-même.
- Échéances — Absence d'échéance attribuée automatiquement, transmission au serveur et propagation entre services.
- Guide du langage (proto 3) — Numéros et champs réservés, et règles pour modifier un type de message sans risque.



Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que gRPC ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

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

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par **D.**

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

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/grpc/