Engineer

gRPC

a framework for calling remote procedures over HTTP/2

What it is

gRPC is a framework for remote procedure calls. A program calls a method on a server that runs on another machine as though the server were a local object, and gRPC carries the call across the network in a protocol of its own, on top of HTTP/2, and brings the answer back as a return value.

Everything starts with a service definition: a list of remote operations and the request and response types each one accepts. By default it is written in Protocol Buffers, Google's open source format for serialising data independently of language and platform, in a file whose name ends in `.proto`. Other formats can be plugged in, but protocol buffers are what gRPC uses unless told otherwise. The compiler `protoc`, together with a gRPC plugin, reads the file and generates code for both halves. The client gets a stub, a local object with the same methods as the service (some languages simply call it the client), and the server gets an interface to implement. Since both halves come from one file, a server written in Java can answer clients written in Go, Python or Ruby.

A method has one of four shapes. A unary method takes one request and returns one response, like an ordinary function call. A server streaming method takes one request and sends back a stream of responses, a client streaming method does the reverse, and a bidirectional streaming method lets both sides send their streams independently. Within a single call, gRPC guarantees that messages arrive in the order they were sent.

On the wire, gRPC runs over HTTP/2. Every call is a `POST` to a path made of the fully qualified service name and the method name, with the content type `application/grpc`. Each message travels behind a five-byte prefix: one byte that says whether the message is compressed, and four that give its length. The outcome of the call is not the HTTP status. A response that gRPC produces

carries status 200 even when the call fails, and the real result comes in trailers, header fields sent after the body: `grpc-status` with a number and, optionally, `grpc-message` with a description of the error.

That makes it a different animal from REST. A REST API names resources and acts on them with HTTP's own methods and status codes, usually with JSON bodies that anyone can read. A gRPC API is a list of methods with typed, binary messages, and HTTP/2 is only the transport underneath. The Protocol Buffers documentation compares the format with JSON while emphasising its smaller messages and faster processing.

Use the concept in a decision

Put "What is gRPC?" to work in a real decision. We can help relate the concept to the system you are building or buying. The part users never see: where the data lives, what the rules are, and how other systems are allowed to talk to it.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

The contract comes first, and the code follows from it. With many HTTP APIs, the description and the code that serves it are two separate things that can drift apart, and a client is often written by hand from a document. In gRPC, the `.proto` file is the one place where a method or a field is defined, and every client and server is generated from it. A change to a message reaches every language through the same generator, not through someone rereading a document.

The encoding is compact, because each field travels as its number and its value rather than its name. The price is that a message cannot be fully interpreted without its `.proto` file, since protocol buffers do not describe themselves, and that the format assumes a whole message fits in memory at once. Its own documentation advises a different solution for data beyond a few megabytes.

Streaming is part of the design rather than an add-on. Calls run as streams on HTTP/2 connections, so a server streaming method can keep sending results through the same machinery as a call that answers once.

Time limits travel with the call. A client sets a deadline, gRPC sends what remains of it in the `grpc-timeout` header, and when the time runs out, the client gets `DEADLINE_EXCEEDED` and the server's side of the call is cancelled. When a server calls another server in turn, some implementations pass the deadline on: Java and Go do it by default, and C++ needs it switched on. Because the deadline crosses each hop as the time remaining rather than as a clock time, machines whose clocks disagree still agree on when to give up.

Errors share one vocabulary. gRPC defines 17 status codes, from 0 `OK` to 16 `UNAUTHENTICATED`, and they are part of the protocol itself, so a Go client and a Java server agree on what `NOT_FOUND` or `UNAVAILABLE` means.

All of this suits traffic between services that one team or one company controls, where both ends can share the `.proto` files and build from them. It suits the browser less well. The gRPC project's own protocol notes say that browsers do not expose HTTP/2 framing to scripts and never will, so browser code speaks a separate protocol, gRPC-Web, and in the usual setup a proxy translates it into native gRPC.

What it looks like in practice

Here is a small billing service. One unary method fetches an invoice, and one server streaming method sends a customer's invoices one at a time:

```
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;
}
```

The numbers are not default values. They are field numbers, and on the wire they identify each field instead of its name. Numbers from 1 to 15 take one byte to encode, so they belong to the fields that are set most often. The two `reserved` lines record a field that once existed, number 4 called `discount_code`, so that neither can come back for a different field.

For Python, one command generates `billing_pb2.py` with the message classes and `billing_pb2_grpc.py` with the client and server classes:

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

One call to `GetInvoice` then looks like this as 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` means two seconds, and `te = trailers` is there to detect proxies that are incompatible with gRPC. If the invoice does not exist, the answer still begins with `:status = 200`, and the last frame carries `grpc-status = 5`, which is `NOT_FOUND`, along with a `grpc-message` such as `invoice not found`. A server that fails at once may even send a single `HEADERS` frame with the status in it and no message at all.

Generating both sides from one contract is not unique to gRPC. One project in this site's portfolio generated the API contract from the database out for an API described with OpenAPI. The chain starts at the database and runs outward: Go types define the API, Huma emits the OpenAPI description from them, a typed TypeScript client and mock handlers are generated from that description, and a contract check in CI fails the build when what the frontend sends no longer matches what the API expects. gRPC builds the same discipline into its toolchain, with the `.proto` file as the single source.

Where it goes wrong

A call without a deadline can wait forever. gRPC sets no deadline by default, so a client calling a server that has hung can wait for an answer effectively forever. Give every call a realistic deadline. In Python, it is the `timeout` argument, in seconds:

```
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)
```

When the time runs out, the call raises `grpc.RpcError`, and its `code()` returns `grpc.StatusCode.DEADLINE_EXCEEDED`. On the server, gRPC cancels the call with `CANCELLED` once the deadline has passed, but stopping any work the handler has started is up to the service's own code, so a long-running handler has to check for the cancellation and stop by itself.

A timeout does not mean nothing happened. Client and server each decide for themselves whether a call succeeded, so a call can complete on the server and still fail on the client with `DEADLINE_EXCEEDED`. Cancelling a call does not undo work that has already happened. A client that simply tries again can create the same invoice twice. The protocol does not assume that a call is idempotent unless it is marked as such. `UNAVAILABLE` is the code most likely to be temporary and worth retrying with a backoff, but the status code documentation warns that retrying is not always safe for operations that are not idempotent. The same document leaves the choice of codes to retry on to the code that makes the calls.

A reused field number corrupts data. A field number cannot change once the message type is in use. If a field is deleted and its number later goes to a new field, older clients and stored messages decode the new field from bytes that meant something else. The Protocol Buffers guide lists what can follow: parse errors, leaked personal data and corrupted data. Never renumber a field, and reserve every deleted number and name, as in the example above.

Every failure looks like a success to HTTP. Because gRPC answers with HTTP status 200 even when a call fails, anything that judges traffic by the HTTP status alone, such as an access log or an error rate built on status codes, counts failures as successes. The specification has servers answer 415 to requests whose content type is not `application/grpc` for that very reason, so that other HTTP/2 clients do not read a gRPC error as a success. Count `grpc-status`, not `:status`.

Long streams pin their load to one server. Calls run as streams on a shared HTTP/2 connection, and once a stream has started, it cannot be moved to another server. A stream that lives for hours keeps its load where it began, however many servers are added later. Each connection also has a limit on concurrent streams, and when it is reached, further calls wait in the client until a stream finishes. The gRPC performance guide advises streaming only where it brings a substantial gain in performance or simplicity, and a separate channel for each area of high load.

Big payloads do not fit in one message. gRPC enforces a maximum message size, and a message larger than the configured limit fails with `RESOURCE_EXHAUSTED` on whichever side sends or receives it. Raising the limit only moves the problem, since protocol buffers expect the whole message in memory, and with large data several serialised copies can end up there at once. Split large data into a stream of smaller messages, or move it outside gRPC.

Further reading

- Core concepts, architecture and lifecycle — service definitions, the four kinds of method, stubs, deadlines and cancellation.
- gRPC over HTTP2 — the wire protocol: paths, headers, the message prefix and the status in trailers.
- Status Codes — all 17 codes, what each means, and which ones the library itself never produces.
- Deadlines — why there is no default, how deadlines reach the server, and how they propagate between services.
- Language Guide (proto 3) — field numbers, reserved fields and the rules for changing a message type safely.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is gRPC?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

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

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/grpc/