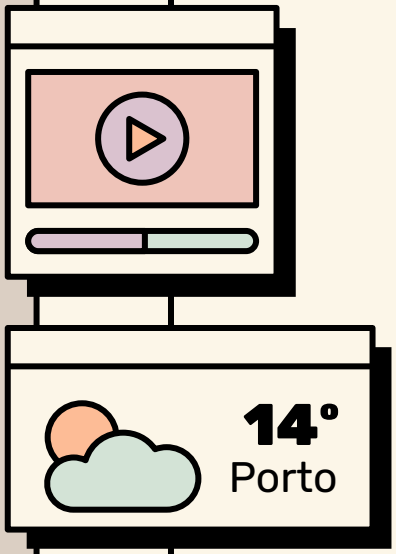
gRPC
Fundamentals



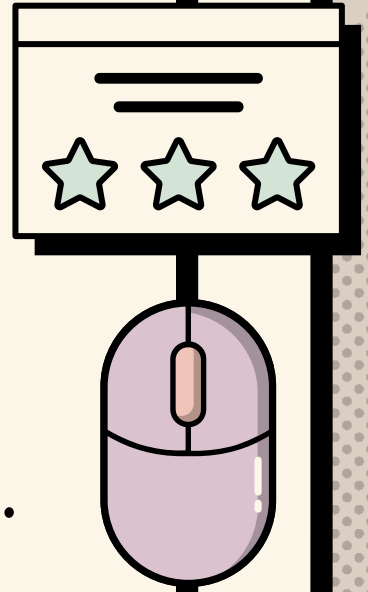


Table of contents

01

gRPC What?

Better to know the basics, huh?!

gRPC

From Wikipedia, the free encyclopedia

gRPC (acronym for **gRPC Remote Procedure Calls**[2]) is a cross-platform high-performance remote procedure call (RPC) framework. gRPC was initially created by Google, but is open source and is used in many organizations. Use cases range from microservices to the "last mile" of computing (mobile, web, and Internet of Things). gRPC uses HTTP/2 for transport, Protocol Buffers as the interface description language, and provides features such as authentication, bidirectional streaming and flow control, blocking or nonblocking bindings, and cancellation and timeouts. It generates cross-platform client and server bindings for many languages. Most common usage scenarios include connecting services in a microservices style architecture, or connecting mobile device clients to backend services.[3]

As of 2019, gRPC's use of HTTP/2 makes it impossible to implement a gRPC client in a browser, instead requiring a proxy.[4]

gRPC

Developer(s)	Google
Initial release	August 2016; 8 years ago
Stable release	1.70.1[1] / February 1, 2025; 23 days ago
Repository	github.com/grpc/grpc
Written in	Android Java, C#, C++, Dart, Go, Java, Kotlin/JVM, Node.js, Objective-C, PHP, Python, Ruby
Type	Remote procedure call framework
License	Apache License 2.0
Website	grpc.io

History [edit]

From about 2001, Google created a general-purpose RPC infrastructure called Stubby to connect the large number of microservices running within and across its data centers.[5] In March 2015, Google decided to build the next version of Stubby and make it open source. The result was gRPC.

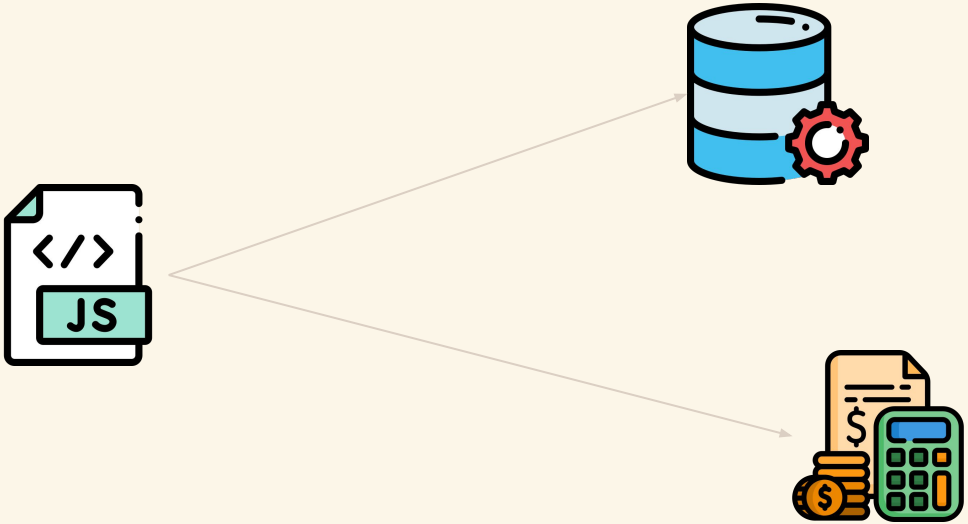
Authentication [edit]

gRPC supports the usage of Transport Layer Security (TLS) and token-based authentication. Connection to Google services must use TLS. There are two types of credentials: channel credentials and call credentials.

Write me a small and clear definition of RPC (not totally related to gRPC, tho)

Remote Procedure Call (RPC) is a protocol that allows a program to execute a procedure (function) on a remote server as if it were a local function. It abstracts network communication, enabling seamless interaction between distributed systems. RPC typically follows a **request-response model**, where the client sends a request, the server processes it, and returns a response.

Normal Application Flow

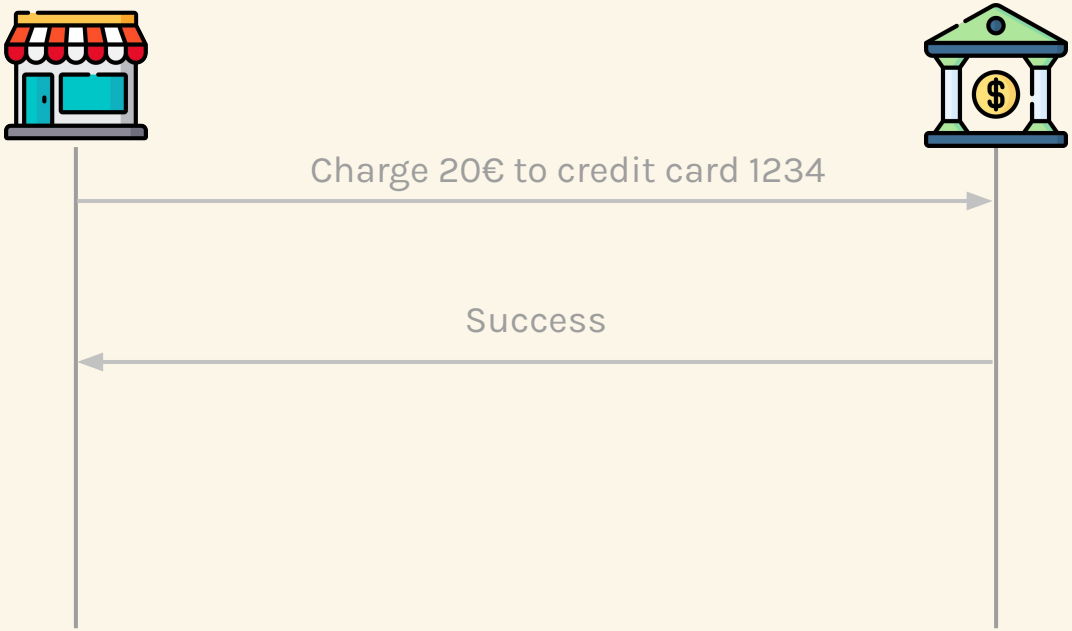


```
function main() {
    const result = add(1, 2);
    console.log(`The result is '${result}'`);
}

function add(a, b) {
    return a + b;
}

// Run the application
main();
```

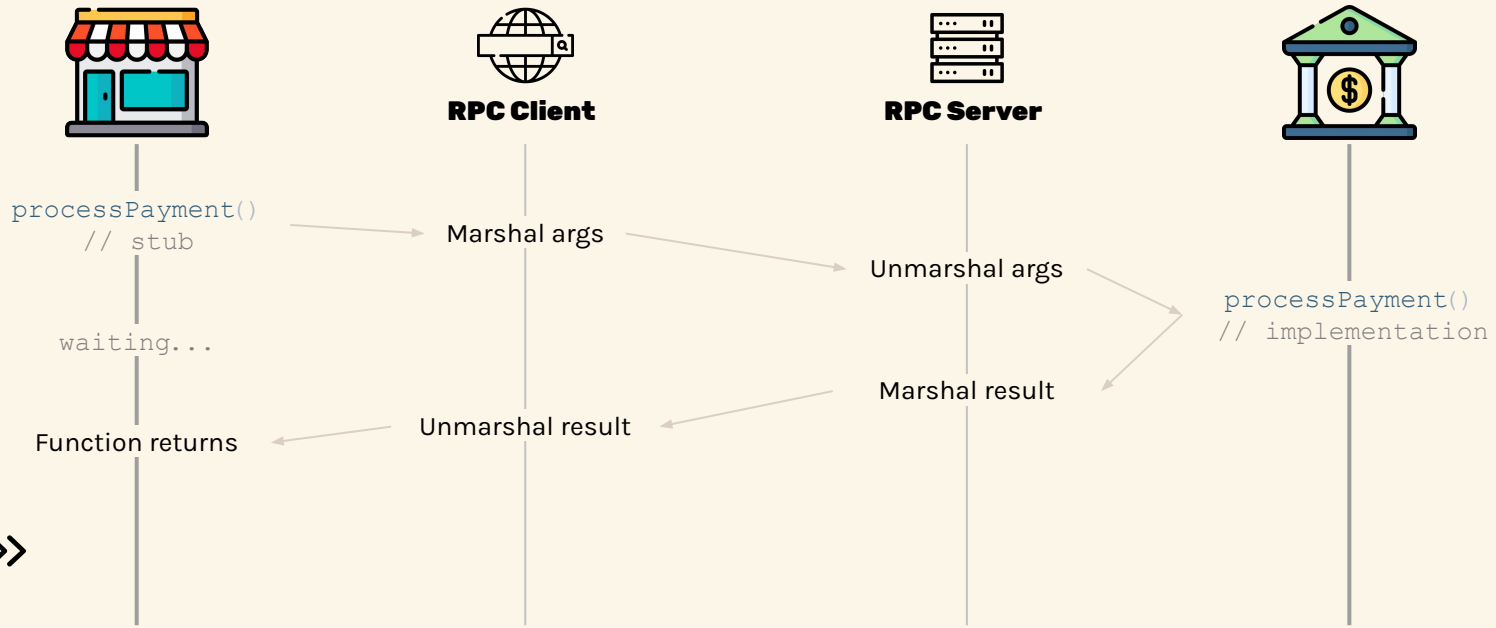
```
function main() {
    const result = add(1, 2);
    console.log(`The result is '${result}'`);
}

function add(a, b) {
    return a + b;
}

// Run the application
main();
```

Implementation of this function is on another node

RPC Flow Example



RPC Flow Example



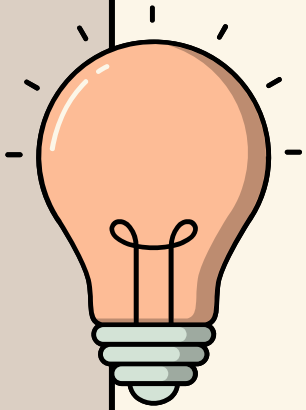
Ideally, RPC makes a call to a remote function look the same as a local function call.

Location Transparency

```
function main() {
    const result = add(1, 2);
    console.log(`The result is '${result}'`);
}

// Run the application
main(); // 3
```

RPC History

- SunRPC / ONC RPC (1980s, basis for NFS)
- COBRA: object-oriented middleware (1990s)
- Microsoft DCOM (1996)
- **SOAP/XML-RPC**: RPC using XML and HTTP (1998)
- JAVA RMI (1999)
- Thrift (Facebook, 2007)
- **Hermes** (Spotify, early 2010s)
- **REST** (1990s, widely adopted in the 2000s)
- **GraphQL** (2015)
- **gRPC** (Google, 2015)



Why gRPC?

02

Isn’t REST good enough?!

Why gRPC?

Protobuf

Efficient Serialization;
Language-neutral;
Schema Definition Language

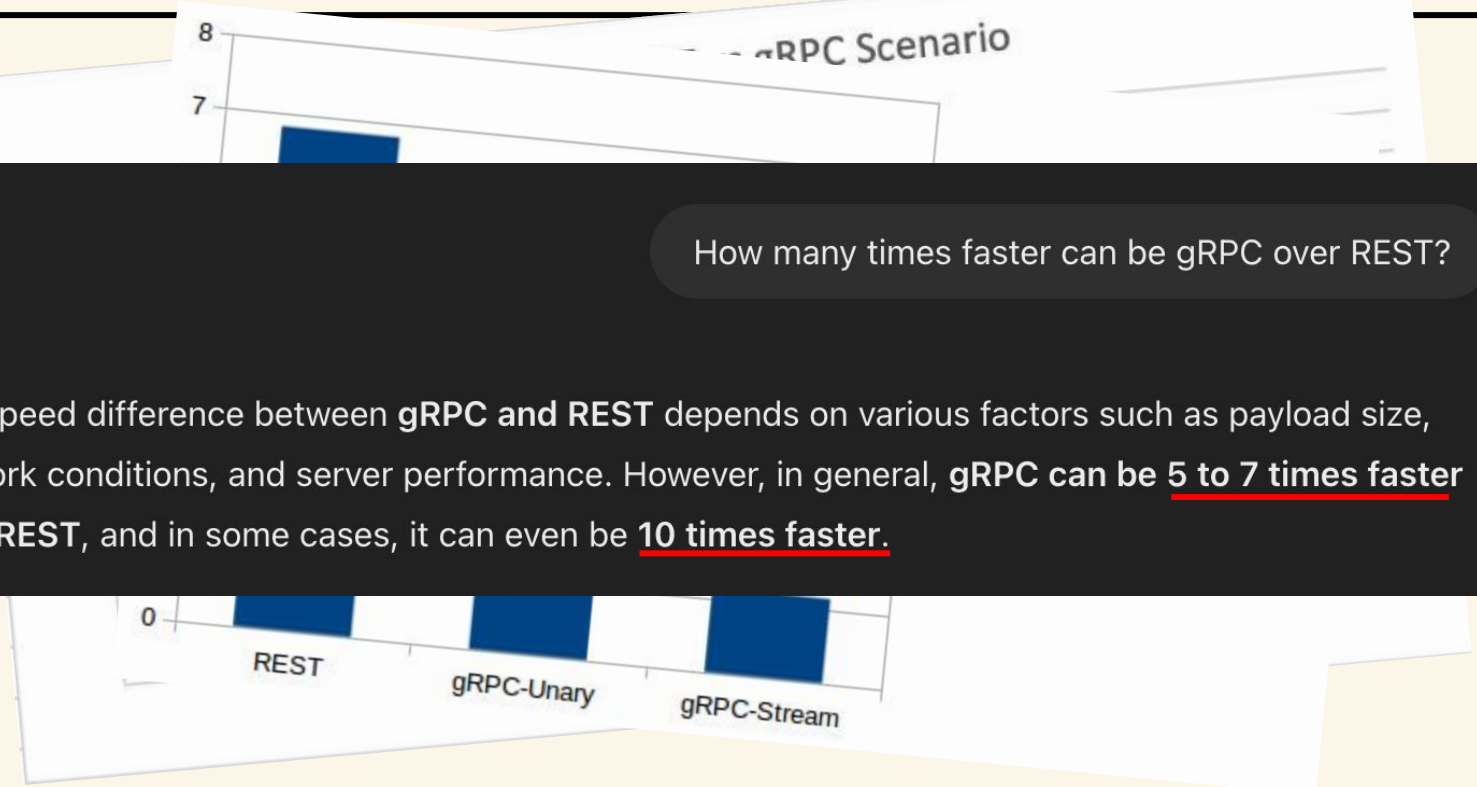
HTTP2

Multiplexing;
Bidirectional Streaming;
Server Push

Official support for multiple programming languages:
C++, Java, Python, Go, C#, Ruby, Node.js, Objective-C, PHP, Dart, Kotlin, Swift, Android, Java

Key Features

- Bi-directional streaming;
- Strongly-typed contracts;
- Support for multiple platforms and environments;
- Interceptors and middleware;
- Support for various authentication mechanisms;
- Code generation.



How many times faster can be gRPC over REST?

The speed difference between **gRPC and REST** depends on various factors such as payload size, network conditions, and server performance. However, in general, **gRPC can be 5 to 7 times faster than REST**, and in some cases, it can even be **10 times faster**.

'g' stands for something different every gRPC release:

- 1.0 'g' stands for 'gRPC'
- 1.1 'g' stands for 'good'
- 1.2 'g' stands for 'green'
- 1.3 'g' stands for 'gentle'
- 1.4 'g' stands for 'gregarious'
- 1.6 'g' stands for 'garcia'
- 1.7 'g' stands for 'gambit'
- 1.8 'g' stands for 'generous'
- 1.9 'g' stands for 'glossy'
- 1.10 'g' stands for 'glamorous'
- 1.11 'g' stands for 'gorgeous'
- 1.12 'g' stands for 'glorious'
- 1.13 'g' stands for 'gloriosa'
- 1.14 'g' stands for 'gladiolus'
- 1.15 'g' stands for 'glider'
- 1.16 'g' stands for 'gao'
- 1.17 'g' stands for 'gizmo'

Where to start?

.proto file

Syntax declaration

Namespace

Service definition,
with all possible RPCs

Messages definition


Show me the code, human!

04

Pros & Cons

Pros

- Compact
- Fast
- One Client Library
- Progress Feedback (eg, upload)
- H2/Protobuf
- Strongly Typed Contracts
- Built-in Authentication and Security

Cons

- Code Generation
- Error Handling
- Versioning
- Compression Overhead
- No native browser support

Thanks!

Does anyone have any questions?