Source >> What is RPC? gRPC Introduction(bytebytego.com)

What is gRPC? How it works?

>>
A gRPC is an open-source remote protocol framework done by Google in 2016. It’s the rewrite of the entire RPC infrastructure that they used for years.

RPC:

A local procedure call is a function call within a process to execute some code. A **remote procedure call** enables one machine to invoke code on another machine as if it is a local function call from a user's perspective. It's like asking someone to do something for you, but instead of being in the same room, you're asking over the internet or a network. It helps different computers talk to each other as if they are right next to each other.

gRPC is a popular implementation of RPC. To connect a large number of microservices running within and across data centers, gRPC is used.

Protocol Buffer

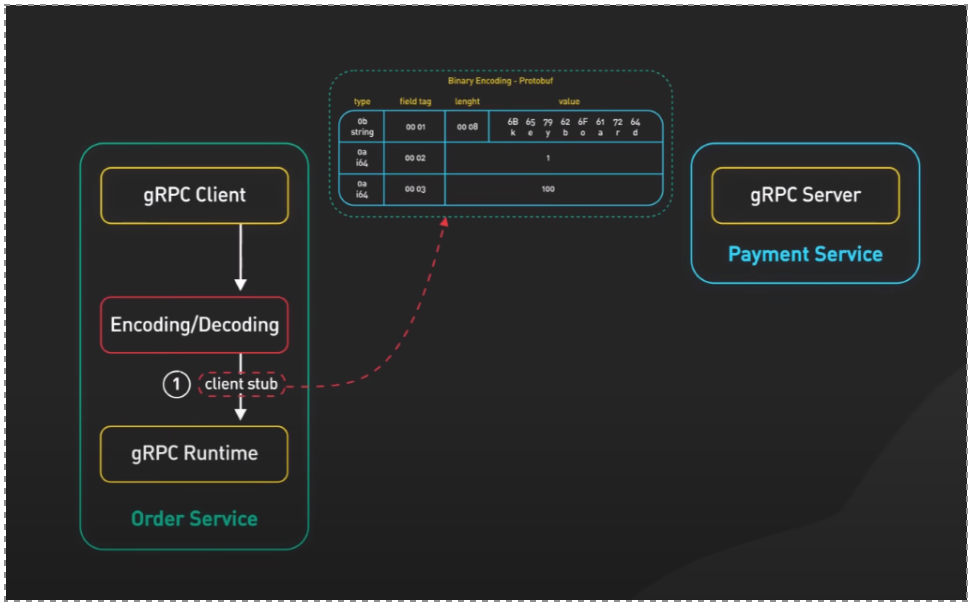
- The core of this ecosystem is the use of **Protocol Buffers** as its *data interchange format*.
- Protocol buffer is a language and platform-independent mechanism for encoding structured data. gRPC uses Protocol Buffers to encode and send data over the wire by default.
- gRPC supports encoding formats like JSON but Protocol Buffer has more advantages, that's why it was the default encoding format of choice for gRPC.
- Protocol Buffer supports strongly typed schema definitions. The structure of the data over the wire is defined in a proto file.
- Protocol Buffer provides broad tooling support to turn the schema(dto) defined into a proto file into data access classes for all popular programming languages.
- A gRPC service is also defined in a proto file by specifying the RPC methods parameters and the return types.

gRPC schema

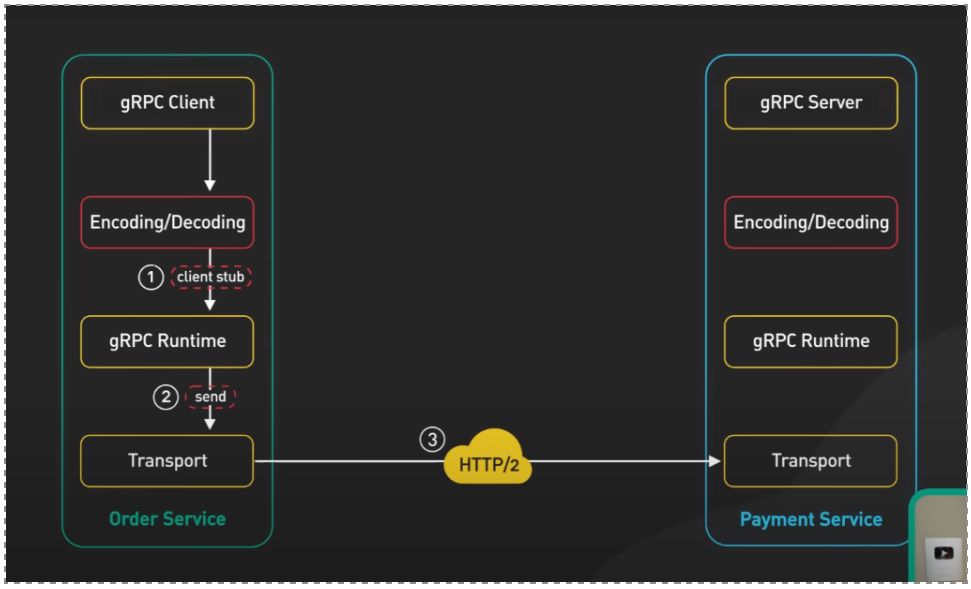
>>
Suppose there is an order service, a gRPC client, and a payment service, a gRPC server.

So, when a client hits a gRPC request to the server:

- It invokes the client code generated automatically by gRPC tooling at build time. This generated code is called a "**client stub**".
- gRPC encodes the data passed to the client stub into protocol buffers and sends it to the low-level **transportation** layer.

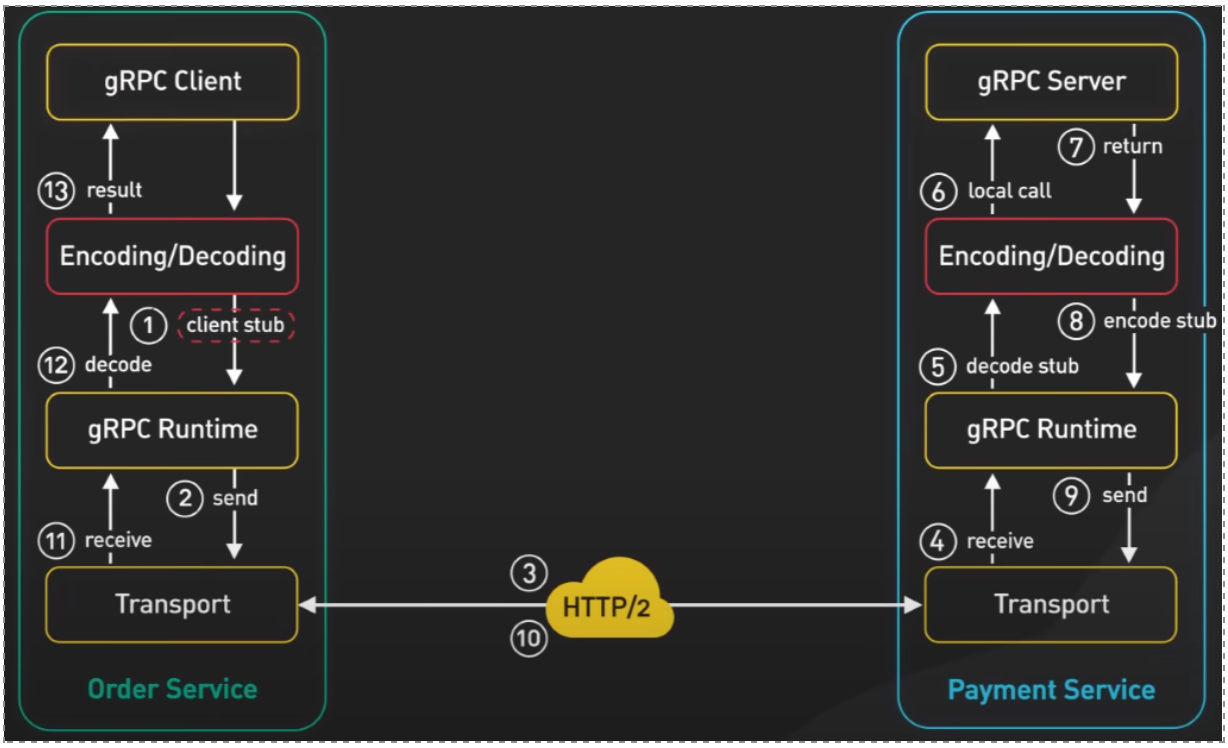


- gRPC sends the data over the network of HTTP/2 data frames.



- The Payment service receives the packets from the network, decodes them, and invokes the server application.
- The result returned from the server application gets encoded into Protocol Buffers and sent to the transport layer.

- The order service receives the packets, decodes them, and sends the result to the client application.

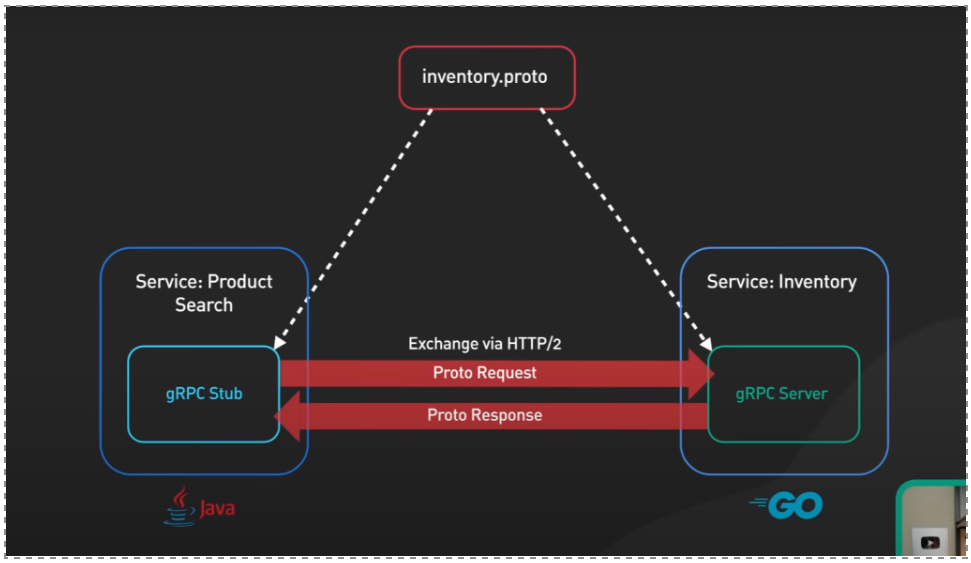


Because of binary encoding and network optimization, gRPC is called 5x faster than JSON. gRPC is very easy to implement.

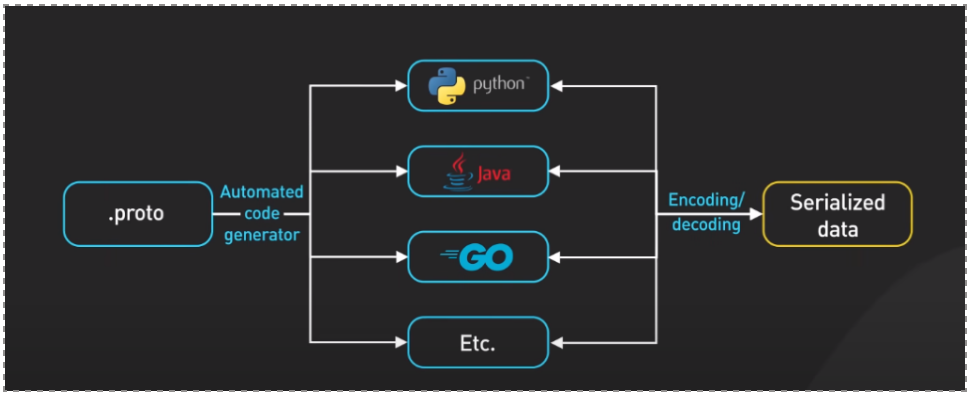
Why is so popular?

>>

- It has a thriving developer ecosystem. Every popular programming language has gRPC implementations.
- It makes it very easy to develop production-quality and **type-safe APIs** that scale very well.
- The same tooling is used to generate gRPC client and server code from the proto file.
- Developers use these generated classes (data access classes) in the client to make RPC calls, and in the server to fulfill RPC requests



- The client and the server can independently choose the programming languages and ecosystem best suited for their particular use cases. As most other RPC frameworks don't have this feature traditionally, gRPC has become so popular.



- Another reason for becoming so popular is because of its high performance out of the box. 2 factors contribute to this performance:

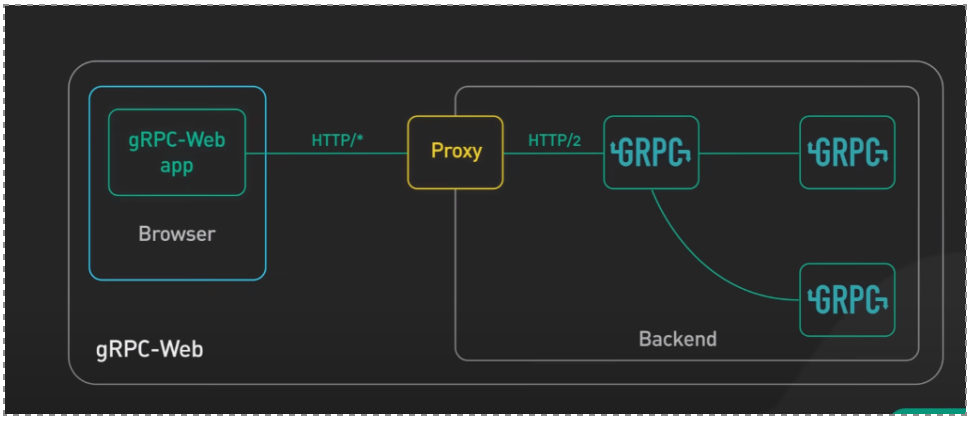
  a. First, the Protocol Buffer is a very efficient binary encoding format, much faster than JSON.

  b. gRPC is built on top of HTTP/2 to provide a high-performance foundation at scale. gRPC use HTTP2 streams. It allows multiple streams of messages over a single long-lived TCP connection. This allows the gRPC framework to handle many concurrent RPC calls.

Why has gRPC-Web not been so widely spread??

>>>

gRPC relies on lower-level access to HTTP/2 primitives. No browsers currently provide the level of control required over web requests to support a gRPC client. It is possible to make a gRPC call from a browser with the help of a **proxy**. This technique is called gRPC-Web. However, this feature set is not compatible with gRPC so this gRPC-web is not as popular as gRPC



Explanation of Type-Safe API

A type-safe API ensures that the data you send and receive is clearly defined and known in advance. It helps catch mistakes early in the development process by enforcing strict rules about the types of data that can be used. This can prevent errors and improve the reliability of your code.

gRPC with Spring boot

Problem Statement and Analysis:

Creating a project with currently 2 services and a proto module. There are an orderService and a paymentService. Now there can be the following scenarios:

Communication Direction:	1. OrderService initiates the payment Service 2. The PaymentService actively queries the OrderService for orders to process.

Asynchronous or Synchronous:	1. Does the customer need to wait for the payment to be completed before getting a response? Or,

	2. Is it acceptable for the payment to be processed asynchronously in the background?

Transaction Handling (how are we handling transactions between the two services?	1. If the payment fails, do we need to roll back the order placement? Or 2. is it acceptable to have the order placed even if the payment fails?

Communication Mechanism:	1. Are we considering direct gRPC communication between the services? or 2. are you planning to use another mechanism such as message queues (e.g., RabbitMQ, Apache Kafka) for asynchronous communication

As we are now considering a synchronous gRPC communication approach, here below could be the steps for solving this case. Note that there will be a GatewayService which will be dedicated to receiving the RESTful API requests it will forward these requests to the appropriate gRPC service using gRPC communications internally.

Steps:

1) **Create proto-module:**
   - Create a separate module(spring boot app) for the proto file Then define the service and message in the proto file. Then the necessary lines should be added to the pom.xml file:

```
<properties>
    <java.version>17</java.version>
    <protobuf.version>3.24.0</protobuf.version>
    <grpc.version>1.59.0</grpc.version>
</properties>
...
<dependency>
      <groupId>io.grpc</groupId>
      <artifactId>grpc-stub</artifactId>
      <version>1.59.0</version>
</dependency>
<dependency>
      <groupId>io.grpc</groupId>
      <artifactId>grpc-protobuf</artifactId>
```

```
      <version>1.59.0</version>
</dependency>

<dependency> <!-- necessary for Java 9+ -->
    <groupId>org.apache.tomcat</groupId>
    <artifactId>annotations-api</artifactId>
    <version>6.0.53</version>
    <scope>provided</scope>
</dependency>

...
<plugin>
      <groupId>org.xolstice.maven.plugins</groupId>
      <artifactId>protobuf-maven-plugin</artifactId>
      <version>0.6.1</version>
      <configuration>
<protocArtifact>com.google.protobuf:protoc:${protobuf.version}:exe:linux-x8
6_64</protocArtifact>
<pluginId>grpc-java</pluginId>
<pluginArtifact>io.grpc:protoc-gen-grpc-java:${grpc.version}:exe:linux-x86_
64</pluginArtifact>
      </configuration>
      <executions>
      <execution>
            <goals>
            <goal>compile</goal>
            <goal>compile-custom</goal>
            </goals>
      </execution>
      </executions>
</plugin>
...
```

2) Generate Java Code:
    - Using the Protobuf plugin to generate Java code from the proto file. We will have to use `mvn clean install` or use tools with IntelliJ given for the maven operations.
    - Add the generated code to the proto-module

3) Implement Order Service (Client):
    - Creating a spring boot project for the order service
    - Adding dependency of the proto module and the relevant dependencies:

```
<!-- using as a client -->
<dependency>
<groupId>net.devh</groupId>
<artifactId>grpc-client-spring-boot-starter</artifactId>
<version>2.15.0.RELEASE</version>
</dependency>
```

- Implement the gRPC client using the @GrpcClient Annotation in the @Service class. Mention the server name in the @GrpcClient("grpc-payment-service")
- For implementing unary RPC BlockingStub is enough. As we have imported the proto-module as a dependency, we will be able to use the stub, the generated class from the proto-module.

4) Implement Payment Service (Server):
    - another spring boot project with the same groupId. An app can be used by both the client and the server. Also, we have to add the dependency of the proto-module and the grpc-server library.

```
<dependency>
    <groupId>com.safatPay</groupId>
    <artifactId>proto-module</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>

<!-- using as a server -->
<dependency>
    <groupId>net.devh</groupId>
    <artifactId>grpc-server-spring-boot-starter</artifactId>
    <version>2.15.0.RELEASE</version>
</dependency>
```

5) Build and Run both projects:
    - The client and server should be able to communicate. Using bloomRPC (postman version of gRPC) we will be able to check the grpc-apis.