**Introduction to gRPC on .NET**

[gRPC](https://grpc.io/) is a language agnostic, high-performance Remote Procedure Call (RPC) framework.

The main benefits of gRPC are:

* Modern, high-performance, lightweight RPC framework.
* Contract-first API development, using Protocol Buffers by default, allowing for language agnostic implementations.
* Tooling available for many languages to generate strongly-typed servers and clients.
* Supports client, server, and bi-directional streaming calls.
* Reduced network usage with Protobuf binary serialization.

These benefits make gRPC ideal for:

* Lightweight microservices where efficiency is critical.
* Polyglot systems where multiple languages are required for development.
* Point-to-point real-time services that need to handle streaming requests or responses.

## **gRPC vs REST**

REST has been one of the concrete pillars of the web programming recently, but the emergence of the gRPC has put some challenge before REST. The below tables explain the differences between them,

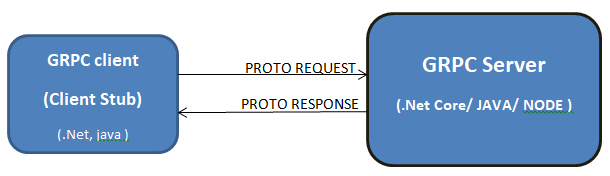
|  |  |  |
| --- | --- | --- |
| **Parameter** | **REST** | **gRPC** |
| **Serialization** | JSON/XML | protobuf |
| **Protocol** | HTTP/1.1 | HTTP/2.0 |
| **Browser Support** | Yes | No |
| **Data Exchange** | Resources and Verbs | Messages |
| **Request-Response Model** | Only supports Request Response as based on Http/1.1 | Supports All Types Of Streaming as based on Http/2 |
| **Payload Exchange Format** | Serialization like JSON/XML | Strong Typing |

**Core concepts, architecture and lifecycle**

# <https://grpc.io/docs/what-is-grpc/core-concepts/>

## **1. Overview of GRPC**

GRPC as a framework for high-performance RPC (Remote procedure calls) developed by Google which makes use of the features of Http2 like Bidirectional streaming and enhanced security. GRPC provides a way for a client application to call server application methods directly as if it is a local object. GRPC revolves around the idea of developing the service which will provide the method which will accept the Request and returns the values in the form of Response.



In the above diagram, we have a GRPC client and GRPC server in this.  Generally, the Server defines services that are defined in the ProtoBuff; it implements the methods and runs a GRPC Server to handle the Client calls. On the other hand at the client end, a proto stub provides the same method as the server and can be called a normal method call.

We can create the GRPC server in any language like Java or .net (.Net Core More specifically) and we can have a client which can be in any language like Python or Go what we need is just the proto file and the client stub will be generated automatically and will be able to call server methods.

## **2. Understanding ProtoBuff**

**Everything about Protocol Buffers**

By Default, GRPC uses Protocol buffers for structuring and serializing data let’s try to explore What, Why, and How we are going to use the Protobuff for the development of the GRPC services.

**What are Protocol Buffers?**

Protocol buffers are a flexible, efficient, and structured mechanism for serializing the data when compared to XML it is smaller, faster, and simpler, We structure how we want our data and then which is then converted specially generated code which can be read and write using data streams and by using a variety of language like C#, Java or Python.

**Why Protocol Buffers?**

Now time for why protocol buffers are needed despite having various things like XML or JSON. ProtoBuff provides many advantages over these traditional data exchange methods like below:

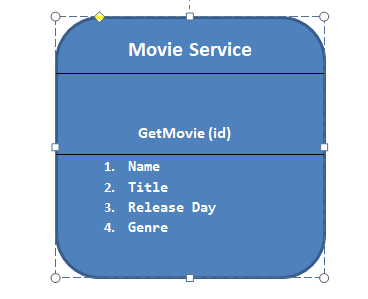
* Simpler
* 3 to 10 times smaller
* 20 to 100 times faster
* Less ambiguous
* Generates stub which are normal classes and easy to understand

When we compare the Protobuff payload and XML Payload we can see the significant changes and advantages of using protobuff over XML

**How to Use Protocol Buffers?**

How to use ProtoBuff can be divided into two major sections one being the Defining the Messages and another being defining the Service let us see step by step with the example in the below sections

Before going into details we are going to see these steps with the example of the Movie service which contains the data and Service to Get the Movie Item like below



Above section clears we have a Movie service that has one method which accepts the Id parameter and then it returns the 4 parameters depicted in the below sections.

As said above ProtoBuff have two sections one being the message and another is the service methods lets explore the first section first

### **1. Defining Message Types**

This is the first step in which we define the structure of the data that we want to serialize in Proto File, In this Data in Proto file, is structured as a message where each message is nothing but a collection of the Fields which are specified as the name-value pair. For our application, we have a message defined as follows

1. message Movie{
2. **string** Name=1;
3. **string** Title=2;
4. **string** ReleaseDay=3;
5. Genere genere=4;
6. }

Above we are defining the message Type Movie which contains the fields or name-value pairs.

**1.1 Specifying the Field Type**

The movie is the Message Type which contains the Field Types with the basic datatype like string along with that it also contains an Enumeration Type Defined there which is a genre which is nothing but the Enum holding all the genres of the Movie like below

1. **enum** Genere{
2. Action=0;
3. Romantic=1;
4. Comedy=2;
5. }

**1.2 Assigning Field Numbers**

In the Above Message Type you might have seen the numbers assigned to the fields where each field is assigned one unique number. These field numbers are used to identify fields in message binary format and should not be changed once the messages are designed and are in use.

## **1. Defining service**

Once we are done with the Message and data structure that will be transferred to and from in the network now it’s time to design an interface that will be called by clients to access these message types.

1. service MoviesService{
2. rpc GetMovie(queryparams) returns (Movie){};
3. }

Grpc is Mostly based on the idea of defining the service which will accept the parameter and sends back the data in for the response, By Default Grpc uses Protobuff IDL which is an Interface definition language to specify the service definition as well the message formats which will be sent over to the network.

**There are four ways we can define our Service methods**

**1. Unary RPC**

rpc GetMovie(queryparams) returns (Movie){};

It’s a typical client-server call where a request is sent to the server and the server respond with the Response

**2. Server Streaming RPC**

rpc GetMovie(queryparams) returns (stream Movie){};

Here client sends a request to the server and the server responds by sending streams of the data back client needs to make sure it reads the data till the server streams it back.

**3. Client streaming RPC**

rpc GetMovie(Stream Data) returns (Movie){};

In this client sends the stream of the data to the server and waits for the server to respond to it, In this sever reads all the data and returns the response accordingly, Grpc guarantees the ordering of the data which will be sent to the server

**4.Bidirectional Streaming**

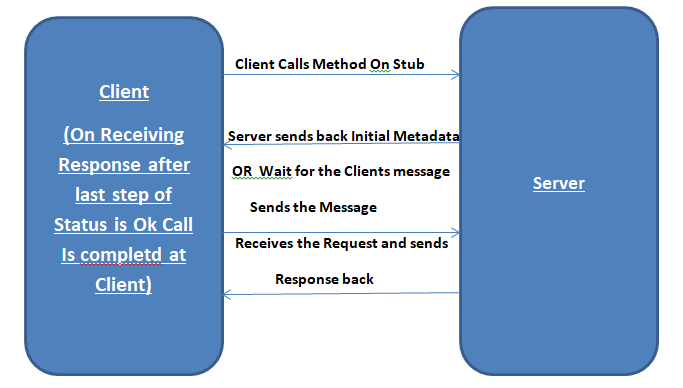
rpc GetMovie(Stream Data) returns (stream Movie){};

In this, both client and server sends the stream of the data to each other using Read-write stream

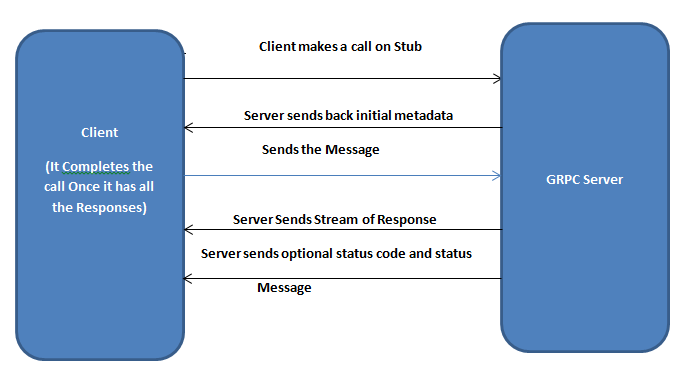
## **RPC Life Cycle**

Above we have seen there are 4 ways we can define the services using Grpc in this section let’s see how does GRPC client and Server communicate in all 4 of the Options

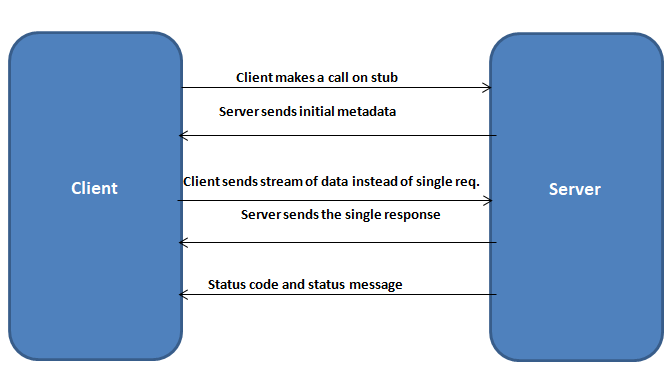
**Unary RPC**



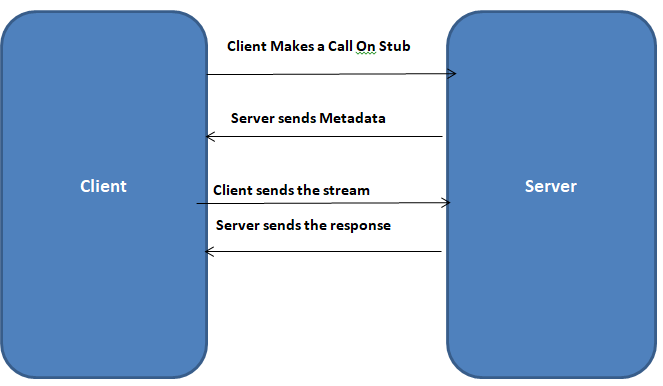
**Server streaming RPC**



**Client streaming RPC**



**Bidirectional streaming RPC**



**Note:**

In Bidirectional streaming the Clients and servers can write and read data in any order streams operate independently, in this case, the server can send the response after receiving all the data from the client otherwise it can send the response as soon as the request is received and then the client can send the again the request depending on the response from the server

**Getting started with gRPC services with C#**

# Create gRPC services and methods

# <https://docs.microsoft.com/en-us/aspnet/core/grpc/basics?view=aspnetcore-6.0>

# <https://docs.microsoft.com/en-us/aspnet/core/grpc/aspnetcore?view=aspnetcore-6.0&tabs=visual-studio>

# <https://www.c-sharpcorner.com/article/grpc-using-c-sharp-and-net-core-day-one/>

# Creating ProtoBuff Service Contract and Schema

# <https://docs.microsoft.com/en-us/aspnet/core/grpc/protobuf?view=aspnetcore-6.0>

# <https://developers.google.com/protocol-buffers/docs/csharptutorial>

# Call gRPC services

# <https://docs.microsoft.com/en-us/aspnet/core/grpc/client?view=aspnetcore-6.0>

# <https://www.c-sharpcorner.com/article/getting-started-with-grpc-client-and-server-using-asp-net-core/>

# Reflection

# <https://docs.microsoft.com/en-us/aspnet/core/grpc/test-tools?view=aspnetcore-6.0>

# Testing

# <https://medium.com/swlh/working-with-grpc-in-net-b8e56404674b>

# More References

# <https://medium.com/aspnetrun/using-grpc-in-microservices-for-building-a-high-performance-interservice-communication-with-net-5-11f3e5fa0e9d>

# <https://grpc.github.io/grpc/csharp/api/Grpc.Core.ServerCallContext.html>

# <https://medium.com/swlh/working-with-grpc-in-net-b8e56404674b>

# <https://medium.com/@niteshsinghal85/testing-grpc-with-grpcui-docker-and-swagger-9cfc9531512>

# <https://docs.microsoft.com/en-us/aspnet/core/grpc/test-tools?view=aspnetcore-6.0>

# <https://dev.to/425show/secure-grpc-service-with-net-core-and-azure-active-directory-4p6k>

# https://auth0.com/blog/implementing-microservices-grpc-dotnet-core-3/