



gRPC
is the new WCF



U Update
NOW 2020

What's gRPC

gRPC Stands for Remote Procedure Call

is a modern high-performance RPC framework

is based on HTTP/2, Protocol Buffers and other
modern standard-based technologies



What's gRPC

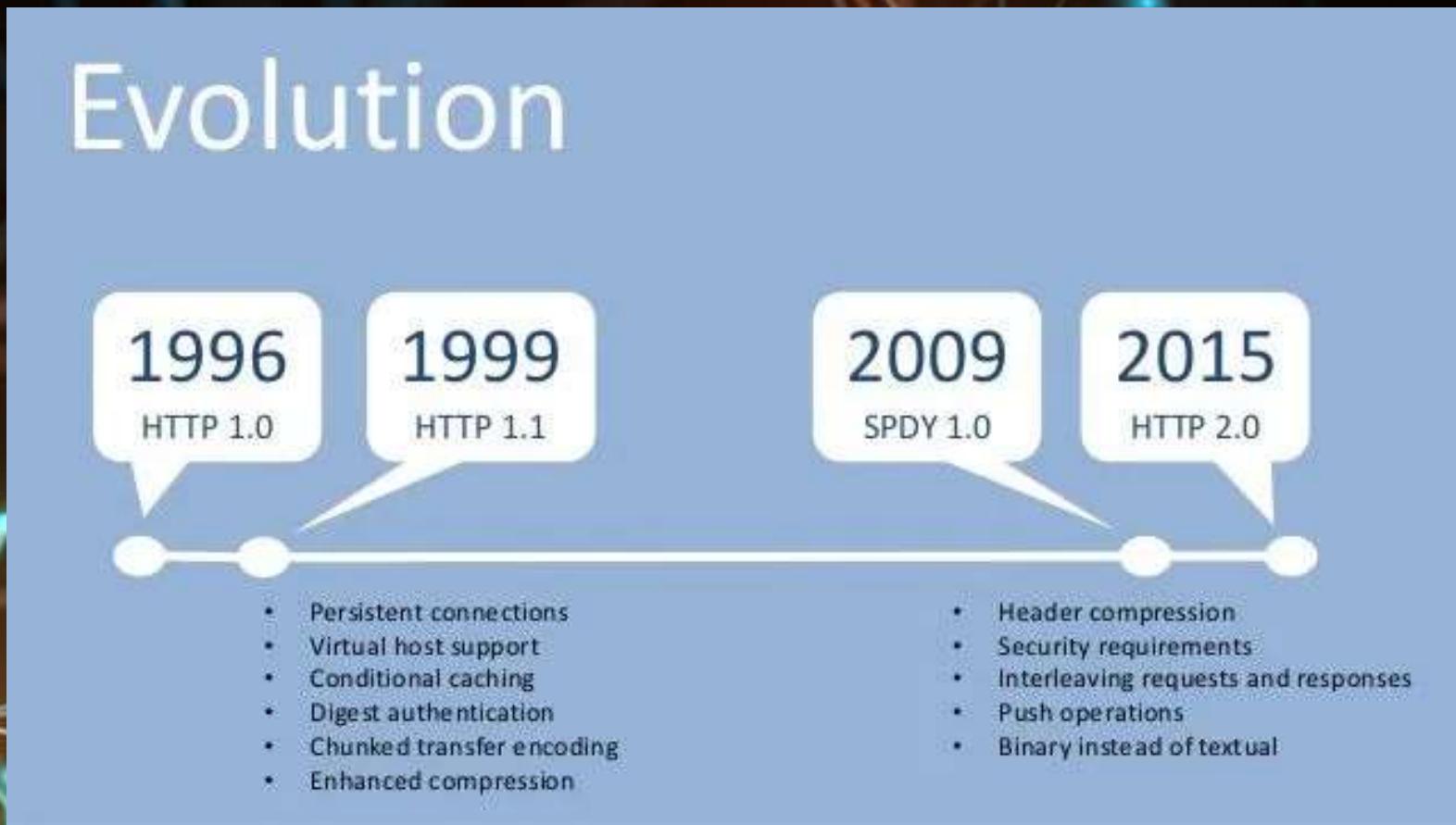
gRPC is Payload agnostic: it supports protobuf but other serialization formats like flatbuffers, json etc are easily configurable

The architecture is extensible, most of the features are provided by Filters: Auth, tracing, name resolution, LB, message size checks, RPC deadlines

Works with many transports, by default uses HTTP2 (other implementations: QUIC, Cronet, Inprocess and more)



History of HTTP



Source <https://frontnet.eu/>

HTTP 1.x: Limited parallelism

New TCP connection per HTTP connection

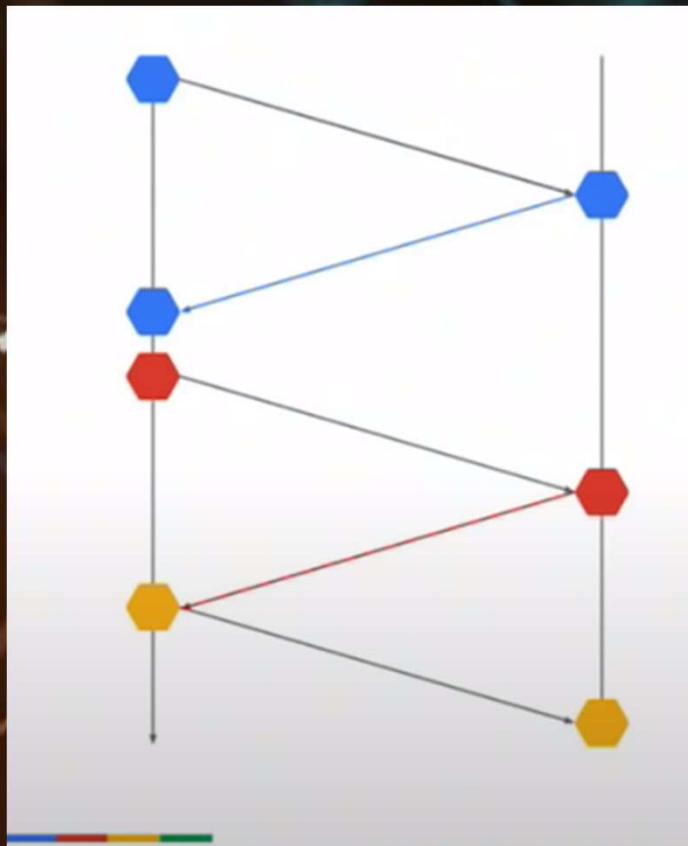
Number of parallel HTTP request
=
Number of TCP Connection

Initializing a connection is very expensive



HTTP 1.0: Head of line blocking

Without pipelining

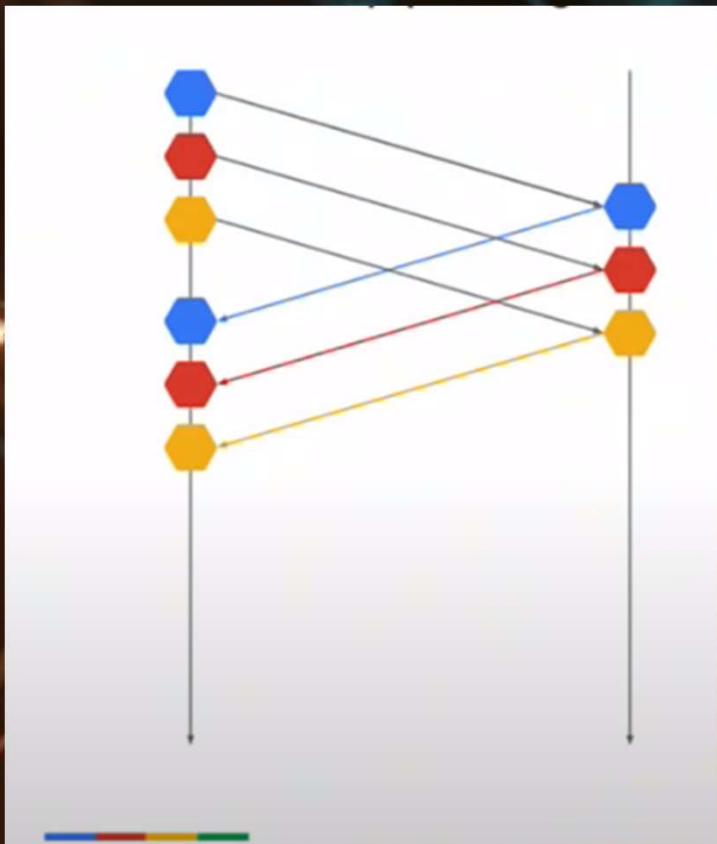


One request wait for the reply
Then a new request make a new reply



HTTP 1.1: Head of line blocking

With pipelining

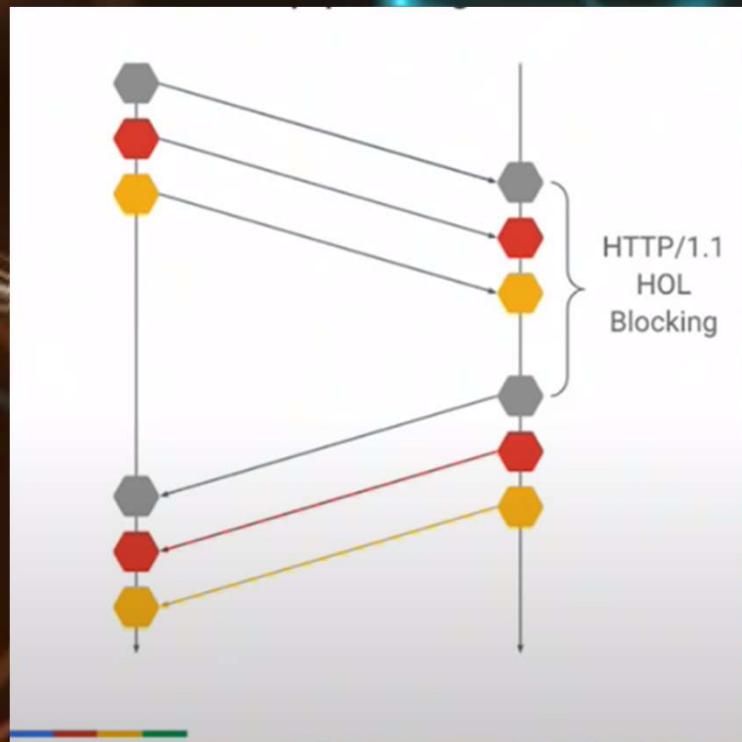


One request wait for the reply
Then a new request make a new reply



HTTP 1.1: Head of line blocking

With pipelining



One request wait for the reply
Then a new request make a new reply



HTTP 1.x: Protocol Overhead

HTTP Headers

Uncompressed plain text headers for each and every HTTP request



HTTP/2

Released in 2015. Extend (not replace) the semantics of HTTP/1.1

Improve end-user perceived latency

Address the “head of line blocking”

Not require multiple connection

Minimize protocol overhead

HTTP/2 Binary Framing

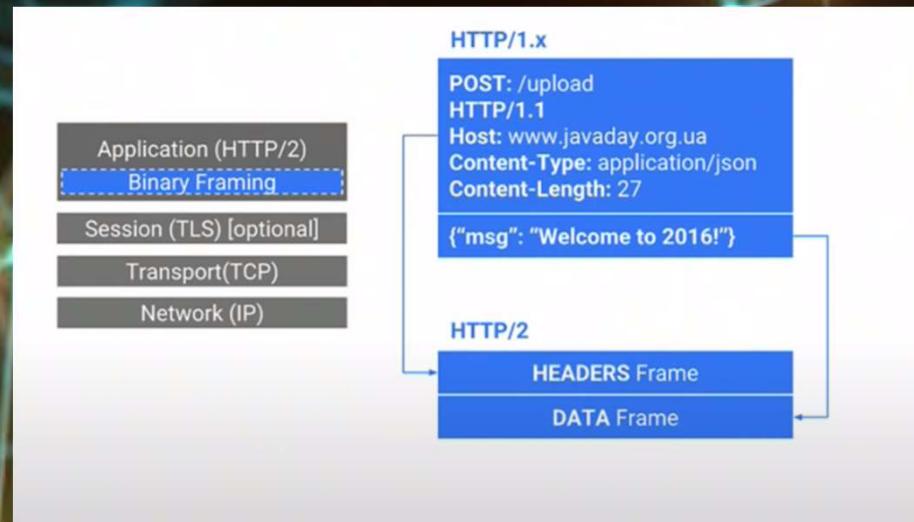
Tries to address all the previous version problems and extend it

Stream is a bidirectional flow of bytes within an established connection, which may carry one or more messages.

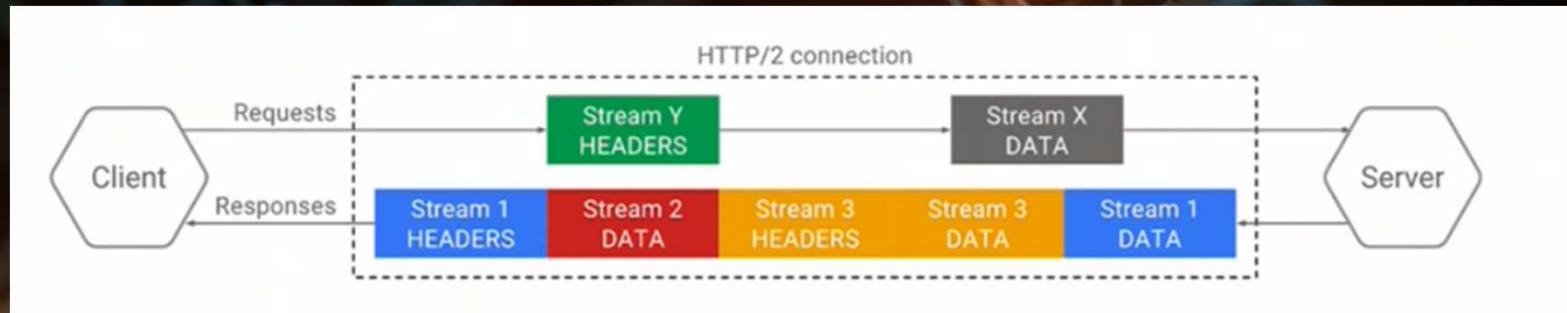
Message is a complete sequence of frames that map to a logical request or response message

Frame is the smallest unit of communication in HTTP/2, each containing a frame header, which at a minimum identifies the stream to which the frame belongs: HEADERS for metadata, DATA for payload, RST_STREAM, SETTINGS, PUSH_PROMISE, PING, GOAWAY, WINDOW_UPDATE, etc

HTTP/2



HTTP/2 Request/Response Multiplexing



Interleave multiple requests and responses in parallel without blocking on anyone

Use a **single TCP connection** to deliver multiple requests and responses in parallel

Enable flow-control, server push, etc

HTTP/2 Request/Response Multiplexing

Interleave multiple requests and responses in parallel without blocking on anyone

Use a **single TCP connection** to deliver multiple requests and responses in parallel

Enable flow-control, server push, etc

HTTP/2

<http://www.http2demo.io/>

HTTP/2 TECHNOLOGY DEMO

This test consists of 200 small images from CDN77.com so you
can see the difference clearly.

HTTP/1.1
0.89s

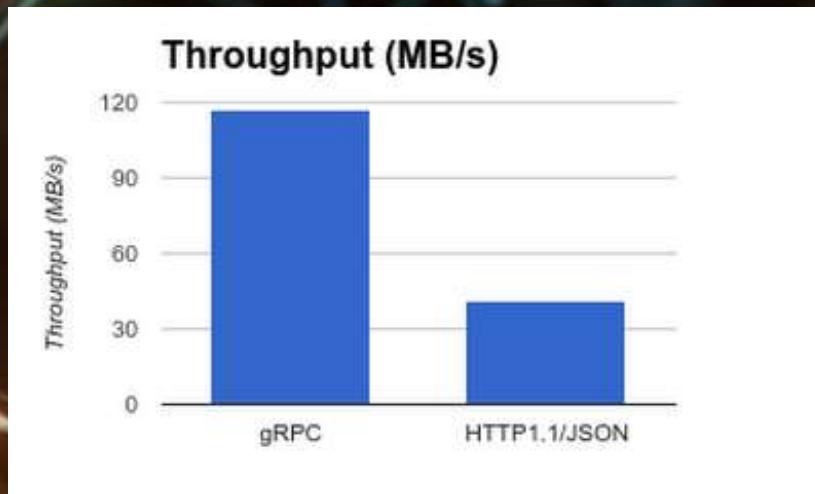


REFRESH

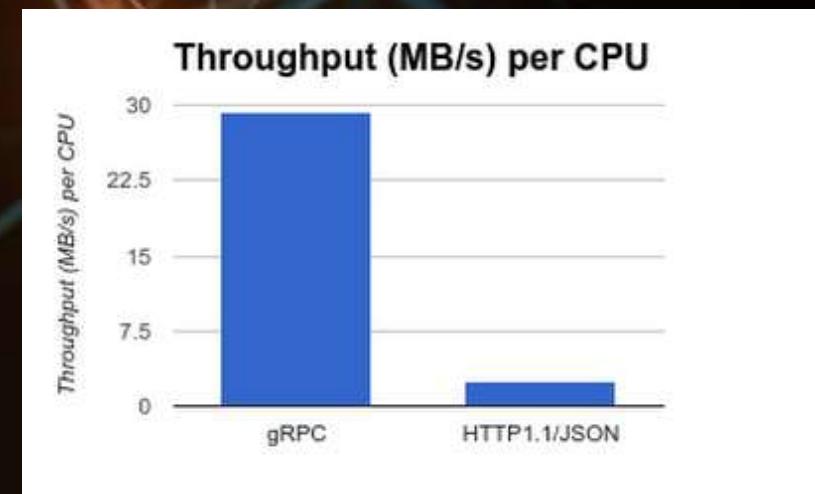
HTTP/2
0.16s



Protocol Buffers / HTTP2



3x increase throughput publishing 50KB messages at maximum throughput from a single n1-highcpu-16 Google Compute Engine Virtual Machine instance, using 9 gRPC channels.



11x difference per CPU
More impressive than the almost 3x increase in throughput, is that it took only 1/4 of the CPU resources

Work with many transports

- By default uses HTTP2 transport
- Other implementations: QUIC, Cronet

Designed for performance



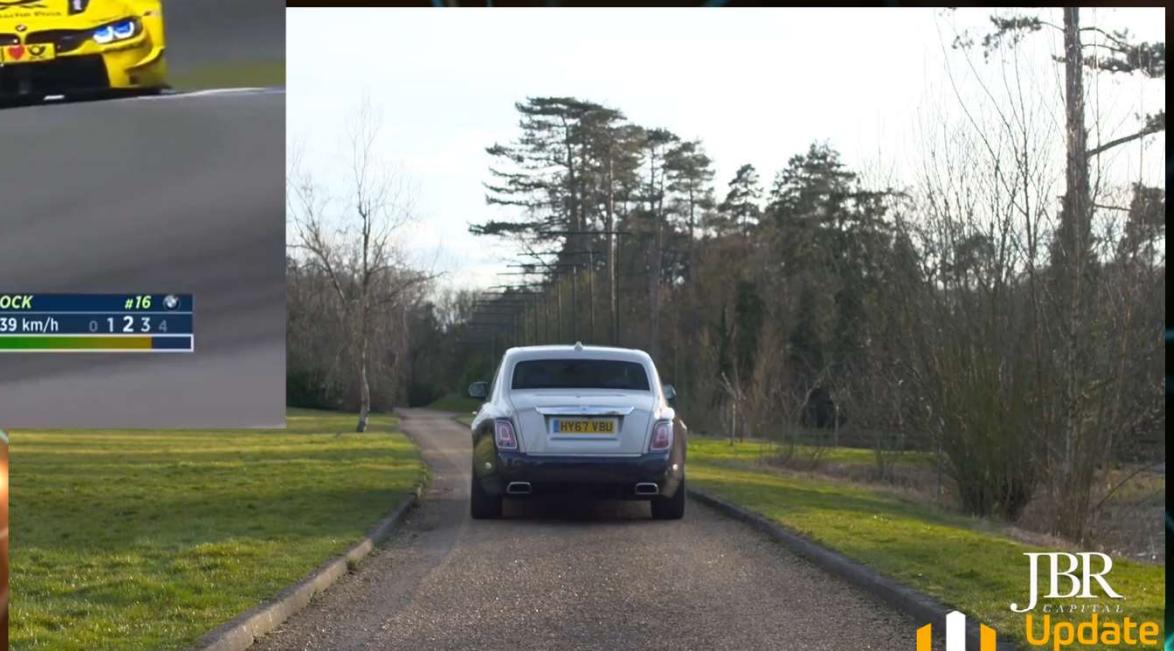
Performance vs Confort

Which software are you developing?



Source: Tuning Roadshow YouTube Channel

Source: various YouTube Videos



gRPC Implementations

Natively Implemented in three languages



gRPC-C-Core

gRPC C++, C#, Ruby, Node, Python, PHP and
Objective-C wrapped around gRPC-C-Core



gRPC Java



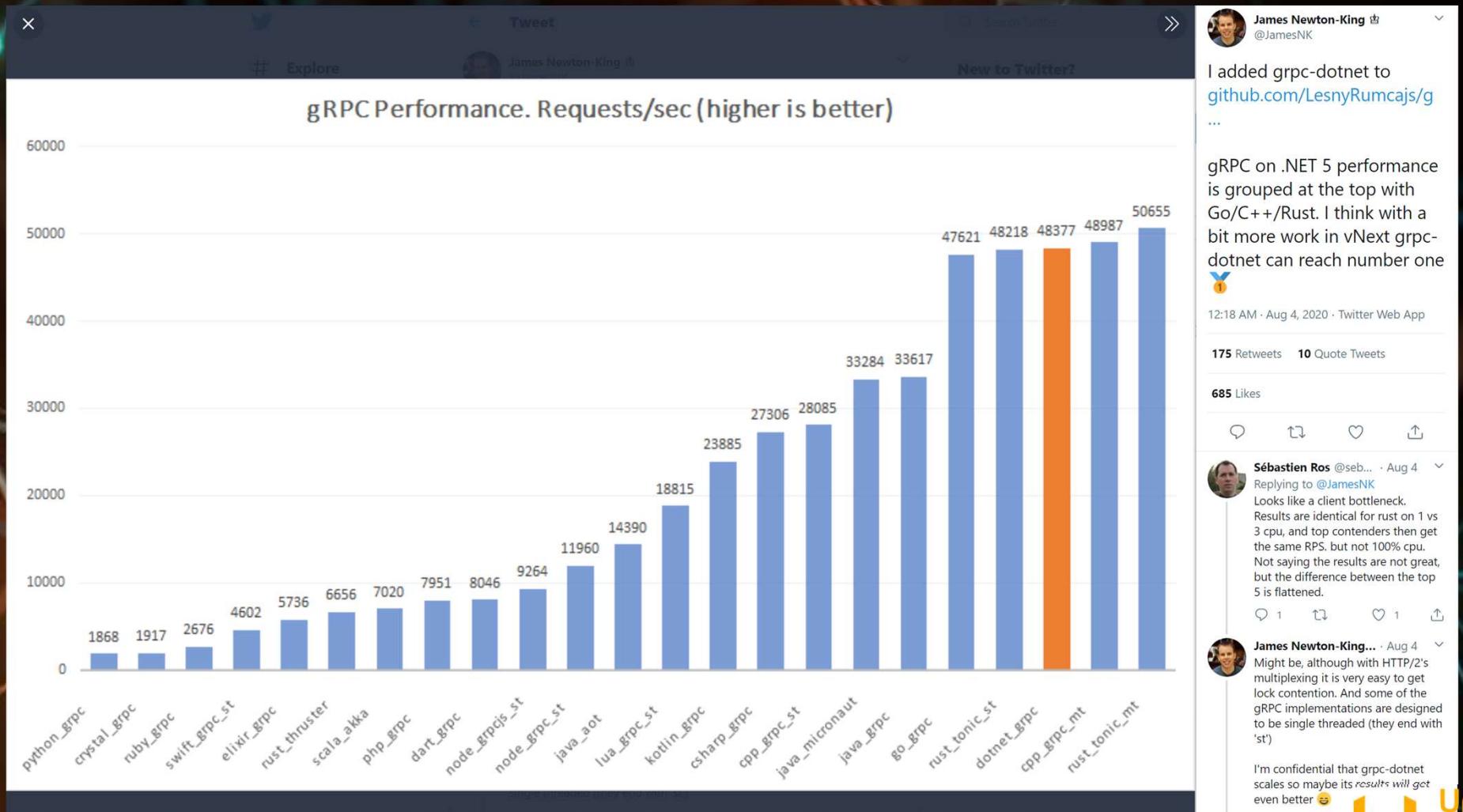
gRPC GO

gRPC Dart and Swift are coming soon

C++ vs Java...
will gRPC C++ vs gRPC Java be the same?



...and maybe .Net will beat them all 😊



Source <https://twitter.com/JamesNK/status/1290411721660592128/photo/1>

Protocol Buffers

Google's Lingua Franca for serializing data: RPC and storage

Binary data representation

Structure can be extended and maintain backward compatibility

Code generation for many languages

Strongly typed

Not required for gRPC

```
syntax = "proto3";

package greet;

// The greeting service definition.
service Greeter {
    // Sends a greeting
    rpc SayHello (HelloRequest) returns (HelloReply);
}

// The request message containing the user's name.
message HelloRequest {
    string name = 1;
}

// The response message containing the greetings
message HelloReply {
    string message = 1;
}
```

Protobuf Language guide

<https://developers.google.com/protocol-buffers/docs/proto3>

The screenshot shows the Protocol Buffers website with a dark background featuring glowing blue and orange network nodes. The top navigation bar includes 'Home', 'Guides' (which is the active tab), 'Reference', and 'Support'. A sidebar on the left lists 'Overview', 'Developer Guide' (with 'Language Guide (proto2)' and 'Language Guide (proto3)' listed, where 'proto3' is highlighted), 'Style Guide', 'Encoding', 'Techniques', 'Add-ons', 'Tutorials' (with 'Tutorials Overview', 'Basics: C++', 'Basics: C#', 'Basics: Dart', 'Basics: Go', 'Basics: Java', and 'Basics: Python'), and 'Related Guides' (with 'gRPC'). The main content area displays the 'Language Guide (proto3)' page, which includes the breadcrumb 'Home > Products > Protocol Buffers > Guides' and a list of topics: Defining A Message Type, Scalar Value Types, Default Values, Enumerations, Using Other Message Types, Nested Types, Updating A Message Type, Unknown Fields, Any, Oneof, Maps, Packages, Defining Services, JSON Mapping, Options, and Generating Your Classes.

Protocol Buffers

Home Guides Reference Support

Overview

Developer Guide

Language Guide (proto2)

Language Guide (proto3)

Style Guide

Encoding

Techniques

Add-ons

Tutorials

Tutorials Overview

Basics: C++

Basics: C#

Basics: Dart

Basics: Go

Basics: Java

Basics: Python

Related Guides

gRPC

Home > Products > Protocol Buffers > Guides

Language Guide (proto3)

- Defining A Message Type
- Scalar Value Types
- Default Values
- Enumerations
- Using Other Message Types
- Nested Types
- Updating A Message Type
- Unknown Fields
- Any
- Oneof
- Maps
- Packages
- Defining Services
- JSON Mapping
- Options
- Generating Your Classes

gRPC – Unary Type

Unary RPCs where the client sends a single request to the server and gets a single response back, just like a normal function call.

```
syntax = "proto3";

package sensorsystem;

// service definition
service SensorService {
    // unary
    rpc GetAvailableSensors (AvailableSensorsRequest) returns (AvailableSensorsResponse);
}

// The request message
message AvailableSensorsRequest {
    string username = 1;
    string message = 2;
}

// The response message
message AvailableSensorsResponse {
    string message = 1;
    string devices = 2;
}

message Device {
    string username = 1;
    string message = 2;
    bytes content = 3;
}
```

```
0 references | Gian Paolo Santopaolo, 8 hours ago | 1 author, 1 change
class Program
{
    0 references | Gian Paolo Santopaolo, 8 hours ago | 1 author, 1 change
    static async Task Main(string[] args)
    {
        var channel = GrpcChannel.ForAddress("https://localhost:5001");
        var client = new SensorService.SensorServiceClient(channel);

        var reply = await client.GetAvailableSensorsAsync(new Sensorsystem.AvailableSensorsRequest
            { Username = "username", Message = "message from client", });
        Console.WriteLine($"Server response: { reply.Message }");
        Console.WriteLine($"Devices from server: { reply.Devices }");

        Console.WriteLine("Press any key to exit...");
        Console.ReadKey();
    }
}
```

gRPC – Server Streaming Type

The client sends a request to the server and gets a stream to read a sequence of message back.

ResponseStream.MoveNext() reads messages streamed from the service.

The client reads from the stream until there are no more messages.

The server streaming call is complete when ResponseStream.MoveNext() returns false.

```
syntax = "proto3";

import "google/protobuf/empty.proto";

package sensorsystem;

// service definition
service SensorService {
    // unary
    rpc GetAvailableSensors (AvailableSensorsRequest) returns (AvailableSensorsResponse);

    // Server streaming
    rpc ReceiveTemperatureUpdates (TemperatureRequest) returns (stream TemperatureData);
}

message TemperatureRequest {
    int32 deviceid = 1;
}

message TemperatureData {
    string devicelocation = 1;
    int32 temperature = 2;
}
```

```
0 references | Gian Paolo Santopaolo, Less than 5 minutes ago | 1 author, 2 changes
class Program
{
    0 references | Gian Paolo Santopaolo, 5 hours ago | 1 author, 1 change
    static async Task Main(string[] args)
    {
        var channel = GrpcChannel.ForAddress("https://localhost:5001");
        var client = new SensorService.SensorServiceClient(channel);

        //Task unarySend = SendUnaryDataAasync(client);
        //Task clientStreaming = SendStremingDataAasync(client);
        Task serverStreaming = ReceiveStreamingDataAsync(client, 1);

        //await Task.WhenAll(unarySend, clientStreaming, serverStreaming);

        await Task.WhenAll(serverStreaming);

        Console.WriteLine("Press any key to exit...");
        Console.ReadKey();
    }

    //server streaming
    1 reference | Gian Paolo Santopaolo, Less than 5 minutes ago | 1 author, 2 changes
    private static async Task ReceiveStreamingDataAsync(SensorService.SensorServiceClient client, int deviceId)
    {
        var cts = new CancellationSource();
        cts.CancelAfter(TimeSpan.FromSeconds(5.0));

        using (var call = client.ReceiveTemperatureUpdates(new TemperatureRequest { Deviceid = deviceId }, cancellationToken: cts.Token))
        {
            try
            {
                await foreach (var message in call.ResponseStream.ReadAllAsync())
                {
                    Console.WriteLine($"Temperature for device located at: {message.Devicelocation} is {message.Temperature}");
                }
            }
            catch (RpcException ex) when (ex.StatusCode == StatusCode.Cancelled)
            {
                Console.WriteLine("Stream canceled.");
            }
        }
    }
}
```

gRPC – Client Streaming Type

The client send a sequence of messages to the server using a stream. Once the client has finished writing the messages it waits for the server to read them and return response.

```
syntax = "proto3";

import "google/protobuf/empty.proto";

package sensorsystem;

// service definition
service SensorService {
    // unary
    rpc GetAvailableSensors (AvailableSensorsRequest) returns (AvailableSensorsResponse);

    // Client streaming
    rpc SendSensorData (stream SensorData) returns (SensorDataResponse);
    rpc SendSensorDataNoResponse (stream SensorData) returns (google.protobuf.Empty);
}

// The request message
message AvailableSensorsRequest {
    string username = 1;
    string message = 2;
}

// The response message
message AvailableSensorsResponse {
    string message = 1;
    string devices = 2;
}

message SensorData {
    string data1 = 1;
    int32 data2 = 2;
}

message SensorDataResponse {
    string message = 1;
}

message Device {
    string username = 1;
    string message = 2;
    bytes content = 3;
}
```

0 references | Gian Paolo Santopaolo, 7 hours ago | 1 author, 1 change
class Program

```
{  
    0 references | Gian Paolo Santopaolo, 7 hours ago | 1 author, 1 change  
    static async Task Main(string[] args)  
    {  
        var channel = GrpcChannel.ForAddress("https://localhost:5001");  
        var client = new SensorService.SensorServiceClient(channel);  
  
        Task unarySend = SendUnaryDataAasync(client);  
        Task clientStreaming = SendStremingDataAasync(client);  
  
        await Task.WhenAll(unarySend, clientStreaming);  
  
        Console.WriteLine("Press any key to exit...");  
        Console.ReadKey();  
    }  
}  
1 reference | Gian Paolo Santopaolo, 7 hours ago | 1 author, 1 change  
private static async Task SendUnaryDataAasync(SensorService.SensorServiceClient client){...}  
1 reference | Gian Paolo Santopaolo, 7 hours ago | 1 author, 1 change  
private static async Task SendStremingDataAasync(SensorService.SensorServiceClient client)  
{  
    using var call = client.SendSensorData();  
    for (var i = 0; i < 100; i++)  
    {  
        await call.RequestStream.WriteAsync(new SensorData { Data1 = $"Message{i}", Data2 = {i} });  
        await Task.Delay(200);  
    }  
  
    await call.RequestStream.CompleteAsync();  
  
    var response = await call;  
    Console.WriteLine($"Message response: {response.Message}");  
}
```

gRPC – Bidi Type

Both sides send a sequence of messages using a read-write stream. The two streams operate independently. The order of messages in each stream is preserved

```
import "google/protobuf/empty.proto";

package sensorsystem;

// service definition
service SensorService {
    // unary
    rpc GetAvailableSensors (AvailableSensorsRequest) returns (AvailableSensorsResponse);

    // Client streaming
    rpc SendSensorData (stream SensorData) returns (SensorDataResponse);
    rpc SendSensorDataNoResponse (stream SensorData) returns (google.protobuf.Empty);

    // Server streaming
    rpc ReceiveTemperatureUpdates (TemperatureRequest) returns (stream TemperatureData);

    // BI-Directional streaming
    rpc StreamData (stream SensorData) returns (stream SensorData);
}

// The request message
message AvailableSensorsRequest {
    string username = 1;
    string message = 2;
}
```

0 references | Gian Paolo Santopaoolo, 11 hours ago | 1 author, 2 changes

```
static async Task Main(string[] args)
{
    var randomizer = new Random();
    var channel = GrpcChannel.ForAddress("https://localhost:5001");
    var client = new SensorService.SensorServiceClient(channel);

    Console.WriteLine("Please type the device ID:");
    var line = Console.ReadLine();

    int sensorId = 0;
    int.TryParse(line, out sensorId);

    using (var cli = client.StreamData())
    {
        _ = Task.Run(async () =>
        {
            while (await cli.ResponseStream.MoveNext(CancellationToken.None))
            {
                var response = cli.ResponseStream.Current;
                Console.WriteLine($"Receiving data from Sensor {response.SensorID} sent data1 {response.Data1}, data2 {response.Data2}");
            }
        });
    }

    Console.WriteLine("Press any key to send random data or Q to stop.");

    while ((line = Console.ReadLine()) != null)
    {
        if (line.ToLower() == "q")
        {
            break;
        }

        int data1 = randomizer.Next(1, 100), data2 = randomizer.Next(1, 100);
        await cli.RequestStream.WriteAsync(new SensorData { SensorID = sensorId, Data1 = data1.ToString(), Data2 = data2.ToString() });
        Console.WriteLine($"Sending data Sensor {sensorId} data1 {data1}, data2 {data2}");
    }
}
```



Let's do some code

Source <https://innerwill.org/>



gRPC – and HTTP2

gRPC not supported in App Service and IIS

<https://docs.microsoft.com/en-us/aspnet/core/grpc/aspnetcore?view=aspnetcore-3.0&tabs=visual-studio>

The screenshot shows a Microsoft Docs page titled "gRPC services with ASP.NET Core". The page is versioned for ASP.NET Core 3.0. It includes a sidebar with navigation links for Overview, Get started, Release notes, and Tutorials. The main content area features a warning box stating that ASP.NET Core gRPC is not currently supported on Azure App Service or IIS due to Http.Sys not supporting HTTP response trailing headers.

Microsoft | Docs Documentation Learn Q&A Code Samples

Search

ASP.NET Languages Workloads APIs Resources

Docs / .NET / ASP.NET Core / Remote Procedure Call apps / gRPC services with ASP.NET Core

Version

ASP.NET Core 3.0

Filter by title

ASP.NET Core documentation

What's new in ASP.NET Core docs

> Overview

> Get started

> Release notes

> Tutorials

gRPC services with ASP.NET Core

09/03/2019 • 4 minutes to read • 2 people +2

This document shows how to get started with gRPC services using ASP.NET Core.

⚠ Warning

ASP.NET Core gRPC is not currently supported on Azure App Service or IIS. The HTTP/2 implementation of Http.Sys does not support HTTP response trailing headers which gRPC relies on. For more information, see this [GitHub issue](#).



gRPC – and HTTP2

Initially only server to server

Today
Web gRPC (no bidi)

gRPC-Web for .NET now available

<https://devblogs.microsoft.com/aspnet/grpc-web-for-net-now-available/>

gRPC-Web for .NET now available



James

June 16th, 2020

gRPC-Web for .NET is now officially released. We [announced experimental support](#) in January and since then we've been making improvements based on feedback from early adopters.

With this release gRPC-Web graduates to a fully supported component of the [grpc-dotnet](#) project and is ready for production. Use gRPC in the browser with gRPC-Web and .NET today.

Getting started

Developers who are brand new to gRPC should check out [Create a gRPC client and server in ASP.NET Core](#). This tutorial walks through creating your first gRPC client and server using .NET.

If you already have a gRPC app then the [Use gRPC in browser apps](#) article shows how to add gRPC-Web to a .NET gRPC server.

What are gRPC and gRPC-Web

gRPC is a modern high-performance RPC (Remote Procedure Call) framework. gRPC is based on HTTP/2, Protocol Buffers and other modern standard-based technologies. gRPC is an open standard and is supported by many programming languages, including .NET.

It is currently impossible to implement the gRPC HTTP/2 spec in the browser because there are no browser APIs with enough fine-grained control over requests. [gRPC-Web](#) is a standardized protocol that solves this problem and makes gRPC usable in the browser. gRPC-Web brings many of gRPC's great features, like small binary messages and contract-first APIs, to modern browser apps.

New opportunities with gRPC-Web

gRPC-Web VS SignalR/messagepack

<https://github.com/grpc/grpc-dotnet/issues/812>

IS this really true?

Why is SignalR/messagepack 2 times faster than gRPC/protobuf ?

Closed ChristianRiedl opened this issue on Mar 6 · 5 comments

ChristianRiedl commented on Mar 6

I created an ASP.NET Core test app to compare gRPC/protobuf and SignalR/messagePack. I tested with a C# client and with a web Javascript client using new experimental grpc-web support. In both cases SignalR was approximately 2 times faster than gRPC/Web. My test simply replies an array of 100 structures. Each structure contains 5 mixed type properties and an array of 5 substructures containing 2 properties. Both are using web-sockets on a fast local network. I expected protobuf marshalling to be faster.

ChristianRiedl added the **question** label on Mar 6

JamesNK commented on Mar 6 · edited

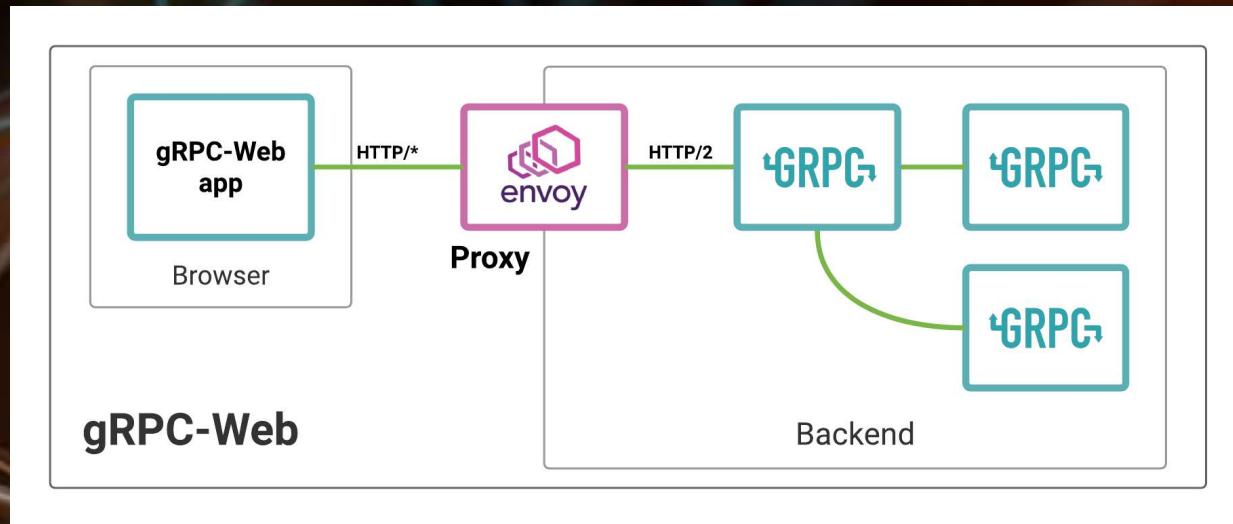
Two things here:

1. grpc-web adds some overhead to gRPC
2. HTTP/2 is a heavier protocol than SignalR over websockets. SignalR has custom framing protocol that is optimized for what it needs, while HTTP/2 is very generic. For example, HTTP/2 is designed to multiplex multiple calls over a single TCP socket, while websockets is optimized for one caller.

In 5.0 we're working to optimize HTTP/2 in ASP.NET Core, and optimize the protobuf serialization. Expect gRPC performance to improve but not reach SignalR's performance.

6

Using gRPC from a Web Client



- A handful of atypical headers — `x-grpc-web`, `grpc-status`, and `grpc-message` — are required for handling gRPC-Web client requests (the JavaScript library automatically handles those headers).
- The built-in `envoy.grpc_web` HTTP filter performs the “heavy lifting” for gRPC-Web proxying
- The `http2_protocol_options: {}` specifies that the `auth_service` takes HTTP/2 (in this case gRPC) connections.

Async Streaming and gRPC with async enumerables IAsyncEnumerable<T> and Span<T>

Generate and consume async streams using C# 8.0
and .NET Core 3.0

<https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/generate-asynchronous-stream>

Tutorial: Generate and consume async streams using C# 8.0 and .NET Core 3.0

02/10/2019 • 9 minutes to read • 5 comments

C# 8.0 introduces **async streams**, which model a streaming source of data. Data streams often retrieve or generate elements asynchronously. Async streams rely on new interfaces introduced in .NET Standard 2.1. These interfaces are supported in .NET Core 3.0 and later. They provide a natural programming model for asynchronous streaming data sources.

In this tutorial, you'll learn how to:

- ✓ Create a data source that generates a sequence of data elements asynchronously.
- ✓ Consume that data source asynchronously.
- ✓ Support cancellation and captured contexts for asynchronous streams.
- ✓ Recognize when the new interface and data source are preferred to earlier synchronous data sequences.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 compiler. The C# 8 compiler is available

Got questions ?

Sassion repo: <https://github.com/gsantopaolo/grpc-streaming>

gRPC not supported in App Service and IIS: <https://docs.microsoft.com/en-us/aspnet/core/grpc/aspnetcore?view=aspnetcore-3.0&tabs=visual-studio>

gRPC-Web for .NET now available: <https://devblogs.microsoft.com/aspnet/grpc-web-for-net-now-available/>

Async stream: <https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/generate-consume-asynchronous-stream>

Protobuf Language guide <https://developers.google.com/protocol-buffers/docs/proto3>





A large orange diagonal bar runs across the slide, partially obscuring the background image. The background image shows a satellite view of Earth at night, with city lights appearing as glowing yellow and white dots. A complex network of green lines connects these lights, symbolizing global communication and connectivity. The overall theme is technology and the interconnected world.

Thank You!

@gsantopaolo

