gRPC

Wael Kdouh - @waelkdouh

Senior Customer Engineer

v1.0

Module 1: Overview

Module Overview

Module 1: Overview

Section 1: gRPC

Lesson: Introduction

What is gRPC?

- Popular open source RPC framework
  - Largest RPC mindshare
  - Cloud Native Computing Foundation (CNCF) project

- Built with modern technologies
  - HTTP/2
  - Protobuf

- Designed for modern apps
  - High performance
  - Platform independent

What is gRPC?

- gRPC stands for gRPC **Remote Procedure Calls**
- High Performance, highly scalable, standards based, open source general purpose RPC Framework
- Binary data representation (compact)
- Requires you to be contract based (unlike Rest)
- Available across different ecosystems
- Secure by default – Requires you to use Http/2 which requires using TLS or SSL
- Uni and Bi-directional Client/Server Message Streaming

Performance – HTTP/2

	Http/2	Http/1.1
Transfer Protocol	Binary	Text
Headers	Compressed	Plain text
Multiplexing	Yes	No
Requests Per Connection	Multiple	1
Server Push	Yes	No
Release Year	2015	1997

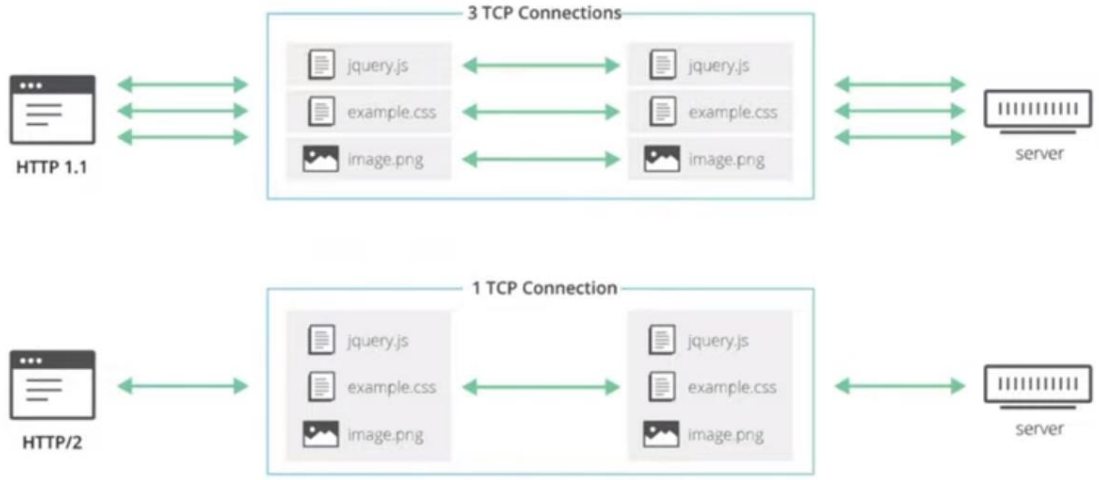


Image Source: https://blog.cloudfare.com

Demo: Http/2 vs Http/1.1

What Is a Contract?

- Contracts allows the generation of clients and servers that we know will be able to communicate with a predetermined data structure
  - Uses interface definition language called Protocol Buffers (ProtoBuf)
  - Language Agnostic
    - C#
    - Java
    - Obj-C
    - python
    - Ruby
    - GO
    - NodeJS

Evaluating Performance of gRPC vs. Rest

- ProtoBuf's goal is to be faster in Encoding and Decoding compared to Json

- ProtoBuf is especially faster in Decoding
  - The size of the messages is smaller as it uses a binary format
  - Inferring the types and the serialization becomes much faster due to its familiarity with the structure of the data

- ProtoBuf also leads to lower resource usage since it uses a binary format. Also Json is more flexible which makes more resource intensive

Protocol Buffers

- ProtoBuf
  - Interface Definition Language
  - Language-neutral
  - Platform-neutral
  - Extensible
  - Serializable

- ProtoBuf was not built for gRPC, but its an essential piece of gRPC

Protocol Buffers

- Protobuf Interface Definition Language (IDL) is a language neutral format for specifying the messages sent and received by gRPC services

- Protobuf messages are defined in .proto files

Protobuf Messages

- Messages are the main data transfer object in Protobuf. They are conceptually similar to .NET classes

```ProtoBuf
syntax = "proto3";

option csharp_namespace = "Contoso.Messages";

message Person {
    int32 id = 1;
    string first_name = 2;
    string last_name = 3;
}
```

- This message definition specifies three fields as name-value pairs. Like properties on .NET types, each field has a name and a type. The field type can be a Protobuf scalar value type, e.g. int32, or another message

Protobuf Messages

- In addition to a name, each field in the message definition has a unique number
- Field numbers are used to identify fields when the message is serialized to Protobuf
- **Serializing a small number is faster than serializing the entire field name**. Because field numbers identify a field it is important to take care when changing them

Protobuf Messages

- When an app is built the Protobuf tooling generates .NET types from .proto files. The Person message generates a .NET class

```C#
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

- For more information about Protobuf types follow this link

Protobuf Messages - Scalar Value Types

- Protobuf supports a range of native scalar value types. The following table lists them all with their equivalent C# type

Protobuf type	C# type
double	double
float	float
int32	int
int64	long
uint32	uint
uint64	ulong
sint32	int
sint64	long
fixed32	uint
fixed64	ulong
sfixed32	int
sfixed64	long
bool	bool
string	string
bytes	ByteString

Scalar **values always have a default value and can't be set to null**. This constraint includes string and ByteString which are C# classes. string defaults to an empty string value and ByteString defaults to an empty bytes value. Attempting to set them to null throws an error

Nullable wrapper types can be used to support null values

Protobuf Messages - Dates and Times

- The native scalar types don't provide for date and time values, equivalent to .NET's DateTimeOffset, DateTime, and TimeSpan

- These types can be specified by using some of Protobuf's Well-Known Types extensions. These extensions provide code generation and runtime support for complex field types across the supported platforms

Protobuf Messages - Dates and Times

- The following table shows the date and time types:

.NET type	Protobuf Well-Known Type
`DateTimeOffset`	`google.protobuf.Timestamp`
`DateTime`	`google.protobuf.Timestamp`
`TimeSpan`	`google.protobuf.Duration`

Protobuf Messages - Dates and Times

ProtoBuf

```
syntax = "proto3"

import "google/protobuf/duration.proto";
import "google/protobuf/timestamp.proto";

message Meeting {
    string subject = 1;
    google.protobuf.Timestamp start = 2;
    google.protobuf.Duration duration = 3;
}
```

C#

```
// Create Timestamp and Duration from .NET DateTimeOffset and TimeSpan.
var meeting = new Meeting
{
    Time = Timestamp.FromDateTimeOffset(meetingTime), // also FromDateTime()
    Duration = Duration.FromTimeSpan(meetingLength)
};

// Convert Timestamp and Duration to .NET DateTimeOffset and TimeSpan.
var time = meeting.Time.ToDateTimeOffset();
var duration = meeting.Duration?.ToTimeSpan();
```

The generated properties in the C# class aren't the .NET date and time types. The properties use the Timestamp and Duration classes in the Google.Protobuf.WellKnownTypes namespace. These classes provide methods for converting to and from DateTimeOffset, DateTime, and TimeSpan

Protobuf Messages - Collections

- Lists in Protobuf are specified by using the repeated prefix keyword on a field. The following example shows how to create a list

ProtoBuf

```
message Person {
    // ...
    repeated string roles = 8;
}
```

C#

```
public class Person
{
    // ...
    public RepeatedField<string> Roles { get; }
}
```

In the generated code, repeated fields are represented by the Google.Protobuf.Collections.RepeatedField<T> generic type.

Protobuf Messages – Other Types

- Other types can be found here

Utilizing gRPC In an ASP.Net Core Applicaiton

- Just another Middleware
- It can use other Middlewares like authentication, Logging, and Configuration

Module 1: Overview

Section 2: gRPC Services With C#

Lesson: Creating gRPC Service

Create new gRPC services

- gRPC services with C# introduced gRPC's contract-first approach to API development

- Services and messages are defined in .proto files. C# tooling then generates code from .proto files. For server-side assets, an abstract base type is generated for each service, along with classes for any messages

Create new gRPC services

- The following .proto file:
  - Defines a Greeter service
  - The Greeter service defines a SayHello call
  - SayHello sends a HelloRequest message and receives a HelloReply message

ProtoBuf

```
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

Create new gRPC services

- C# tooling generates the C# GreeterBase base type:

```
public abstract partial class GreeterBase
{
    public virtual Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
    {
        throw new RpcException(new Status(StatusCode.Unimplemented, ""));
    }
}

public class HelloRequest
{
    public string Name { get; set; }
}

public class HelloReply
{
    public string Message { get; set; }
}
```

Create new gRPC services

- By default the generated GreeterBase doesn't do anything. Its virtual SayHello method will return an UNIMPLEMENTED error to any clients that call it. For the service to be useful an app must create a concrete implementation of GreeterBase:

```C#
public class GreeterService : GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
    {
        return Task.FromResult(new HelloRequest { Message = $"Hello {request.Name}" });
    }
}
```

Create new gRPC services

- The service implementation is registered with the app. If the service is hosted by ASP.NET Core gRPC, it should be added to the routing pipeline with the MapGrpcService method.

```C#
app.UseEndpoints(endpoints =>
{
    endpoints.MapGrpcService<GreeterService>();
});
```

Module 1: Overview

Section 2: gRPC Services With C#

Lesson: Implementing gRPC Methods

Implementing gRPC Methods

- A gRPC service can have different types of methods. How messages are sent and received by a service depends on the type of method defined. The gRPC method types are:
  - Unary
  - Server streaming
  - Client streaming
  - Bi-directional streaming

Implementing gRPC Methods

- Streaming calls are specified with the stream keyword in the .proto file. stream can be placed on a call's request message, response message, or both

ProtoBuf

```
syntax = "proto3";

service ExampleService {
  // Unary
  rpc UnaryCall (ExampleRequest) returns (ExampleResponse);

  // Server streaming
  rpc StreamingFromServer (ExampleRequest) returns (stream ExampleResponse);

  // Client streaming
  rpc StreamingFromClient (stream ExampleRequest) returns (ExampleResponse);

  // Bi-directional streaming
  rpc StreamingBothWays (stream ExampleRequest) returns (stream ExampleResponse);
}
```

- Each call type has a different method signature. Overriding generated methods from the abstract base service type in a concrete implementation ensures the correct arguments and return type are used

Implementing gRPC Methods - Unary Method

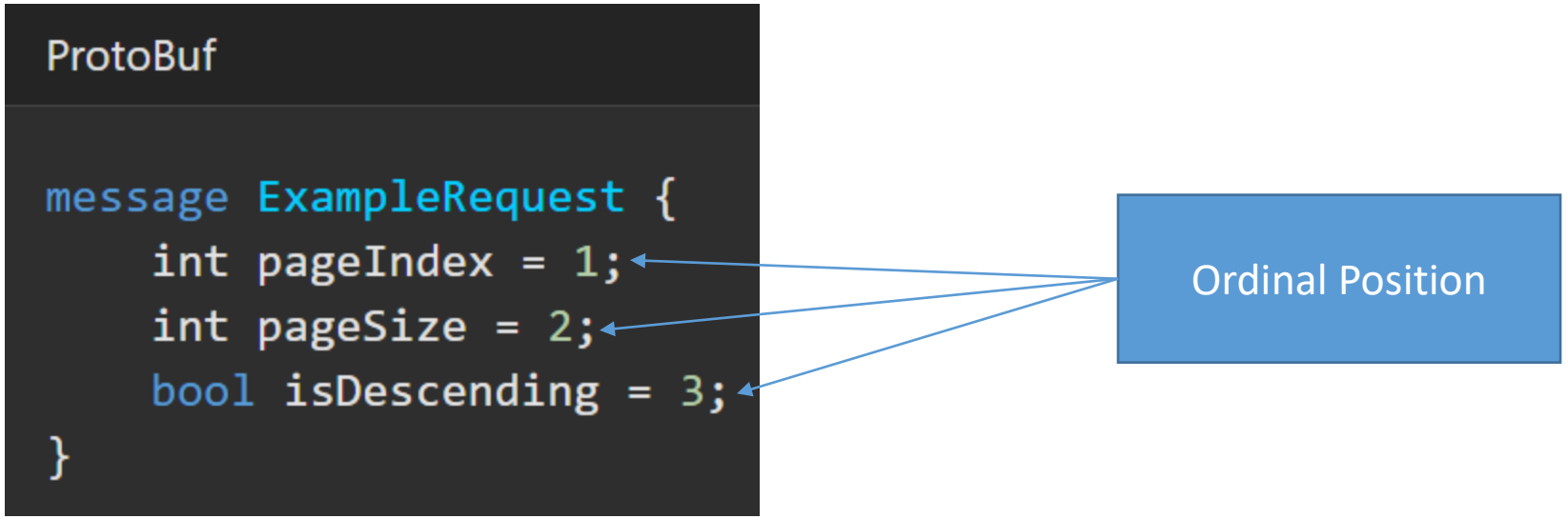
- A unary method gets the request message as a parameter, and returns the response. A unary call is complete when the response is returned

```
public override Task<ExampleResponse> UnaryCall(ExampleRequest request,
    ServerCallContext context)
{
    var response = new ExampleResponse();
    return Task.FromResult(response);
}
```

Implementing gRPC Methods - Unary Method

- Unary calls are the most similar to actions on web API controllers. One important difference gRPC methods have from actions is gRPC methods are not able to bind parts of a request to different method arguments

- gRPC methods always have one message argument for the incoming request data. Multiple values can still be sent to a gRPC service by making them fields on the request message



Demo: gRPC – Unary Method

Implementing gRPC Methods - Server Streaming Method

- A server streaming method gets the request message as a parameter. Because multiple messages can be streamed back to the caller, responseStream.WriteAsync is used to send response messages. A server streaming call is complete when the method returns

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext context)
{
    for (var i = 0; i < 5; i++)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1));
    }
}
```

The ServerCallContext provides access to data about the current call

We use the IServerStreamWriter to write messages to the stream

Implementing gRPC Methods - Server Streaming Method

- The client has no way to send additional messages or data once the server streaming method has started

- Some streaming methods are designed to run forever. For continuous streaming methods, a client can cancel the call when it's no longer needed. When cancellation happens the client sends a signal to the server and the ServerCallContext.CancellationToken is raised. The CancellationToken token should be used on the server with async methods so that:
  - Any asynchronous work is canceled together with the streaming call
  - The method exits quickly

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext context)
{
    while (!context.CancellationToken.IsCancellationRequested)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1), context.CancellationToken);
    }
}
```

Implementing gRPC Methods - Server Streaming Method

- gRPC is designed so that it can guarantee that the client receives the messages in the same order as they are sent from the server

Where Would Server Streaming Rpc Be Useful?

- Streaming is useful in any scenario where the server needs to send multiple items in response to a client call

- One option that would be typical in a traditional **REST approach** would be to include an array of items in the serialized model. In this case, **all items would need to be generated/loaded before the request can be returned**. One reason, you may do this is to reduce the number of requests required to gather the data, instead, bundling them into a single response

- With server streaming in gRPC, the **server sends messages as soon as it has them available**. If each message requires some load or generation time, the server can reduce the time to first message by streaming data when the first message can be produced. We can even produce messages in parallel. The client can begin to act on the message immediately, upon receiving it on the stream

Where Would Server Streaming Rpc Be Useful?

- When streaming, each item sent as a message will usually be some distinct object that can be handled in isolation. This way, the client can work with each message as they are received. That doesn't always have to be the case, but it's where the best performance gains will be seen

- When streaming, a single gRPC channel can be used which avoids repeated connections being made to the server. This helps to reduce the load on the clients and servers

Demo: gRPC – Server Streaming Method

Implementing gRPC Methods - Client Streaming Method

- A client streaming method starts without the method receiving a message. The requestStream parameter is used to read messages from the client. A client streaming call is complete when a response message is returned:

```C#
public override async Task<ExampleResponse> StreamingFromClient(
    IAsyncStreamReader<ExampleRequest> requestStream, ServerCallContext context)
{
    while (await requestStream.MoveNext())
    {
        var message = requestStream.Current;
        // ...
    }
    return new ExampleResponse();
}
```

Implementing gRPC Methods - Client Streaming Method

- When using C# 8 or later, the await foreach syntax can be used to read messages. The IAsyncStreamReader<T>.ReadAllAsync() extension method reads all messages from the request stream:

```C#
public override async Task<ExampleResponse> StreamingFromClient(
    IAsyncStreamReader<ExampleRequest> requestStream, ServerCallContext context)
{
    await foreach (var message in requestStream.ReadAllAsync())
    {
        // ...
    }
    return new ExampleResponse();
}
```

Implementing gRPC Methods - Bi-directional Streaming Method

- A bi-directional streaming method starts without the method receiving a message
- The requestStream parameter is used to read messages from the client
- The method can choose to send messages with responseStream.WriteAsync
- A bi-directional streaming call is complete when the method returns

```C#
public override async Task StreamingBothWays(IAsyncStreamReader<ExampleRequest> requestStream,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext context)
{
    await foreach (var message in requestStream.ReadAllAsync())
    {
        await responseStream.WriteAsync(new ExampleResponse());
    }
}
```

Sends a response for each request. This is a basic usage of bi-directional streaming

Implementing gRPC Methods - Bi-directional Streaming Method

- It is possible to support more complex scenarios, such as reading requests and sending responses simultaneously:

```C#
public override async Task StreamingBothWays(IAsyncStreamReader<ExampleRequest> requestStream,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext context)
{
    // Read requests in a background task.
    var readTask = Task.Run(async () =>
    {
        await foreach (var message in requestStream.ReadAllAsync())
        {
            // Process request.
        }
    });

    // Send responses until the client signals that it is complete.
    while (!readTask.IsCompleted)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1), context.CancellationToken);
    }
}
```

Implementing gRPC Methods - Bi-directional Streaming Method

- In a bi-directional streaming method, the client and service can send messages to each other at any time. The best implementation of a bi-directional method varies depending upon requirements

Access gRPC Request Headers

- A request message is not the only way for a client to send data to a gRPC service. Header values are available in a service using ServerCallContext.RequestHeaders

C#

```
public override Task<ExampleResponse> UnaryCall(ExampleRequest request, ServerCallContext context)
{
    var userAgent = context.RequestHeaders.GetValue("user-agent");
    // ...

    return Task.FromResult(new ExampleResponse());
}
```

Module 1: Overview

Section 2: gRPC Services With C#

Lesson: Versioning gRPC services

Versioning gRPC Services

- New features added to an app can require gRPC services provided to clients to change, sometimes in unexpected and breaking ways. When gRPC services change:
  - Consideration should be given on how changes impact clients
  - A versioning strategy to support changes should be implemented

Backwards Compatibility

- **The gRPC protocol is designed to support services that change over time.** Generally, additions to gRPC services and methods are non-breaking. Non-breaking changes allow existing clients to continue working without changes

- Changing or deleting gRPC services are breaking changes. When gRPC services have breaking changes, clients using that service have to be updated and redeployed

- Making non-breaking changes to a service has a number of benefits:
  - Existing clients continue to run
  - Avoids work involved with notifying clients of breaking changes, and updating them
  - Only one version of the service needs to be documented and maintained

Non-breaking changes

- These changes are **non-breaking at a gRPC protocol level and .NET binary level**
  - Adding a new service
  - Adding a new method to a service
  - Adding a field to a request message - Fields added to a request message are deserialized with the default value on the server when not set. To be a non-breaking change, the service must succeed when the new field isn't set by older clients
  - Adding a field to a response message - Fields added to a response message are deserialized into the message's unknown fields collection on the client
  - Adding a value to an enum - Enums are serialized as a numeric value. New enum values are deserialized on the client to the enum value without an enum name. To be a non-breaking change, older clients must run correctly when receiving the new enum value

Binary Breaking Changes

- The following changes are non-breaking at a gRPC protocol level, but the client needs to be updated if it upgrades to the latest .proto contract or client .NET assembly. Binary compatibility is important if you plan to publish a gRPC library to NuGet
  - Removing a field - Values from a removed field are deserialized to a message's unknown fields. This isn't a gRPC protocol breaking change, but the client needs to be updated if it upgrades to the latest contract. It's important that a removed field number isn't accidentally reused in the future. To ensure this doesn't happen, specify deleted field numbers and names on the message using Protobuf's reserved keyword
  - Renaming a message - Message names aren't typically sent on the network, so this isn't a gRPC protocol breaking change. The client will need to be updated if it upgrades to the latest contract. One situation where message names are sent on the network is with Any fields, when the message name is used to identify the message type
  - Changing csharp_namespace - Changing csharp_namespace will change the namespace of generated .NET types. This isn't a gRPC protocol breaking change, but the client needs to be updated if it upgrades to the latest contract

Protocol Breaking Changes

- The following items are protocol and binary breaking changes:
  - Renaming a field - With Protobuf content, the field names are only used in generated code. The field number is used to identify fields on the network. Renaming a field isn't a protocol breaking change for Protobuf. However, if a server is using JSON content then renaming a field is a breaking change
  - Changing a field data type - Changing a field's data type to an incompatible type will cause errors when deserializing the message. Even if the new data type is compatible, it's likely the client needs to be updated to support the new type if it upgrades to the latest contract
  - Changing a field number - With Protobuf payloads, the field number is used to identify fields on the network
  - Renaming a package, service or method - gRPC uses the package name, service name, and method name to build the URL. The client gets an UNIMPLEMENTED status from the server
  - Removing a service or method - The client gets an UNIMPLEMENTED status from the server when calling the removed method

Behavior Breaking Changes

- When making non-breaking changes, you must also consider whether older clients can continue working with the new service behavior. For example, adding a new field to a request message:
  - Isn't a protocol breaking change
  - Returning an error status on the server if the new field isn't set makes it a breaking change for old clients

- Behavior compatibility is determined by your app-specific code

Version Number Services

- Services should strive to remain backwards compatible with old clients. Eventually changes to your app may require breaking changes

- **Breaking old clients and forcing them to be updated along with your service isn't a good user experience**. A way to **maintain backwards compatibility while making breaking changes is to publish multiple versions of a service**

Version Number Services

- gRPC supports an optional package specifier, which functions much like a .NET namespace. In fact, the package will be used as the .NET namespace for generated .NET types if option csharp_namespace is not set in the .proto file. The package can be used to specify a version number for your service and its messages:

```ProtoBuf
syntax = "proto3";

package greet.v1;

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

Version Number Services

- The package name is combined with the service name to identify a service address. A service address allows multiple versions of a service to be hosted side-by-side:
  - greet.v1.Greeter
  - greet.v2.Greeter

- Implementations of the versioned service are registered in Startup.cs:

```C#
app.UseEndpoints(endpoints =>
{
    // Implements greet.v1.Greeter
    endpoints.MapGrpcService<GreeterServiceV1>();

    // Implements greet.v2.Greeter
    endpoints.MapGrpcService<GreeterServiceV2>();
});
```

Version Number Services

- Including a version number in the package name gives you the opportunity to publish a v2 version of your service with breaking changes, while continuing to support older clients who call the v1 version. Once clients have updated to use the v2 service, you can choose to remove the old version. When planning to publish multiple versions of a service:
  - Avoid breaking changes if reasonable
  - Don't update the version number unless making breaking changes
  - Do update the version number when you make breaking changes

Version Number Services

- Publishing multiple versions of a service duplicates it. To reduce duplication, consider moving business logic from the service implementations to a centralized location that can be reused by the old and new implementations:

```C#
using Greet.V1;
using Grpc.Core;
using System.Threading.Tasks;

namespace Services
{
    public class GreeterServiceV1 : Greeter.GreeterBase
    {
        private readonly IGreeter _greeter;
        public GreeterServiceV1(IGreeter greeter)
        {
            _greeter = greeter;
        }

        public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
        {
            return Task.FromResult(new HelloReply
            {
                Message = _greeter.GetHelloMessage(request.Name)
            });
        }
    }
}
```

- Services and messages generated with different package names are different .NET types. Moving business logic to a centralized location requires mapping messages to common types

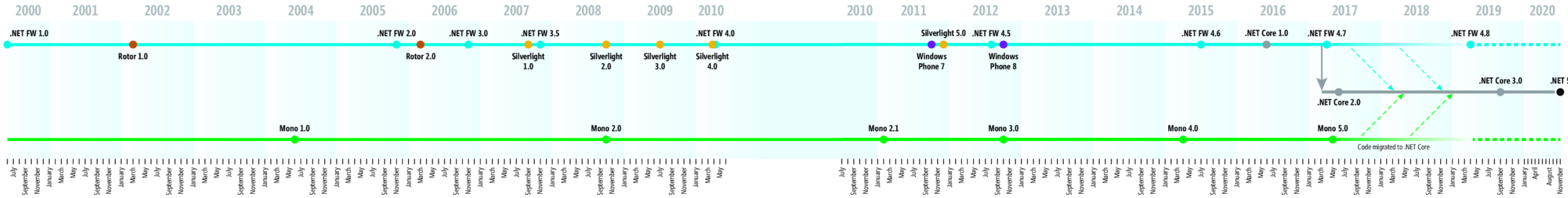
Demo: gRPC – Versioning gRPC Services

Module 1: Overview

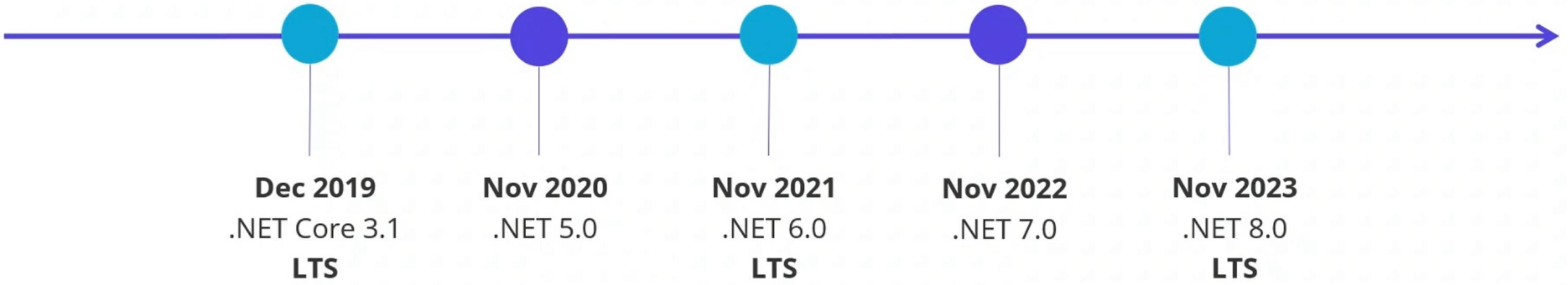
Section 3: gRPC Evolution

Lesson: .Net 5 Enhancements

.NET Reunified: Microsoft's Plans for .NET 5



.NET Schedule

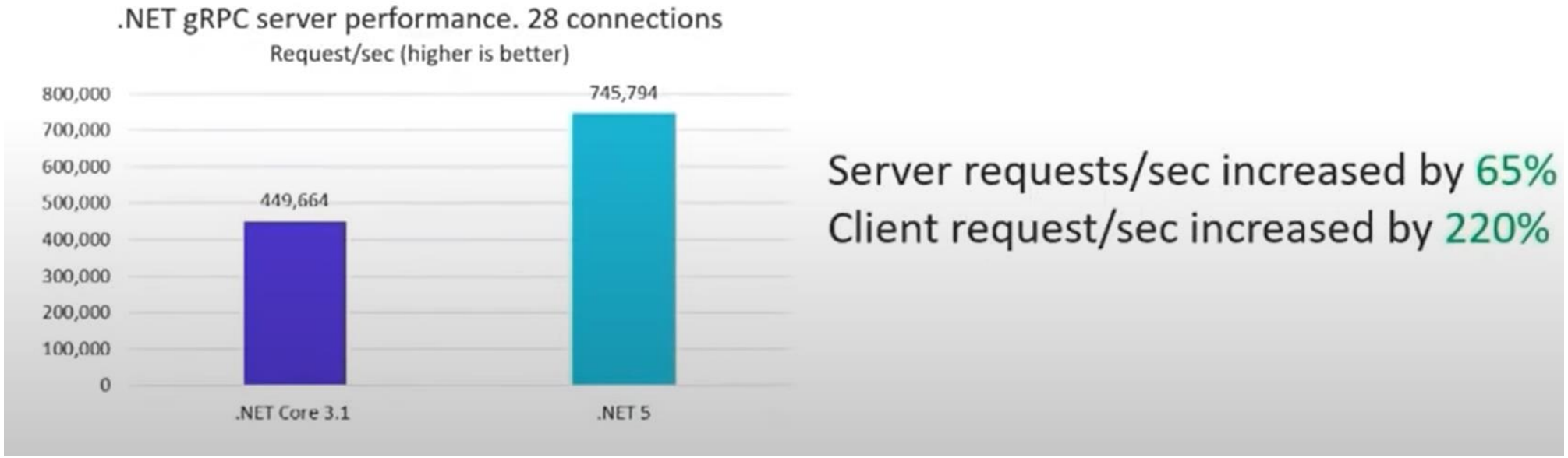


- .Net 5 release in November 2020
- Major releases every year
- LTS for even numbered releases
- Predictable schedule, minor releases as needed

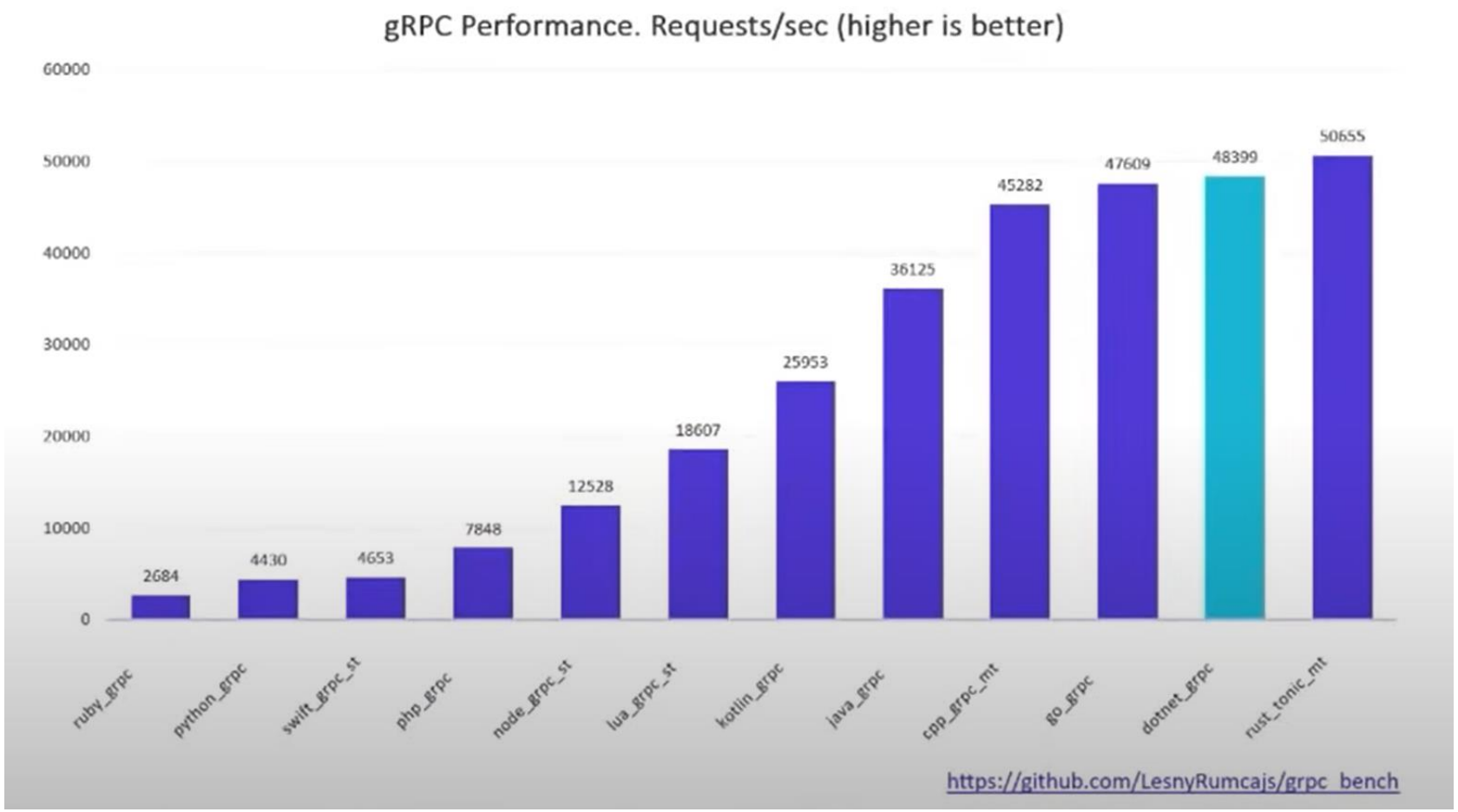
Performance - .NET 5 improvements

- HTTP/2 server allocations reduced by 92%

- HPack response compression



Performance - Implementations



Module Summary

- In this module, you learned about:
  - Introduction to gRPC
  - Implementing gRPC Methods
  - Versioning gRPC services
  - .Net 5 Enhancements

Lab 1: Overview


Microsoft