GRPC - BEYOND THE BASICS

gRPC is a modern, open-source, high performance Remote Procedure Call (RPC) framework

gRPC

- Officially supported from .NET Core 3.0
- Contract-based API development
- Designed for HTTP/2 and beyond

gRPC

- Microservices
- IoT devices
- Polyglot environments
- Streaming requirements

gRPC IN ACTION

- Contract definition - .proto file
- Code generated on server and client
  - Server – Abstract base class with virtual methods, messages as POCO
  - Client – Client with stub methods , messages as POCO
- Client apps require generated gRPC client to communicate with the server
- Grpc.Tools used for code generation using the *protoc* compiler

PROTOC COMPILER SUPPORT

- C#
- C++
- Dart
- Go
- Java
- Kotlin
- Node
- Objective-C
- PHP
- Python
- Ruby
- Third-Party Add-ons - https://bit.ly/protobuf-thirdparty

PROTOCOL BUFFERS

The name Protocol Buffer originates from a class of the same name used as a buffer

- Google's open-source mechanism to serialize structured data
- Language-neutral, platform-neutral, extensible to serialize data in both forward-compatible and backward-compatible way
- Interface Definition Language
- Message Exchange format

MESSAGES

- Requests and Responses
- Each message is a record of key-value pairs called fields
- Messages are transmitted in binary format
- Unique sequence number used as field identifier

```
message SendReadingRequest {
  int64 id=1;
  string deviceName=2;
  double temperature =3;
  google.protobuf.Timestamp updateTime = 4;
}
```

2^29-1 fields possible!

Protobuf Type	C# Type
int32	int
int64	long
uint32	uint
uint64	ulong
sint32	int
sint64	long
fixed32	uint
fixed64	ulong
sfixed32	int
sfixed64	long
double	Double
Float	float
bool	bool
string	string
bytes	Google.Protobuf.ByteString

```
message HelloReply {
  string message = 1;
  MessageType messageType = 2;
}

enum MessageType{
    TEXT = 0;
    EMAIL = 1;
    POST = 2;
}
```

```
new HelloReply
{
    Message = "Hello " + request.Name,
    MessageType = MessageType.Email
}
```

```
public enum MessageType {
  [pbr::OriginalName("TEXT")] Text = 0,
  [pbr::OriginalName("EMAIL")] Email = 1,
  [pbr::OriginalName("POST")] Post = 2,
}
```

```
message HelloReply {
  string message = 1;
  MessageType messageType = 2;
  NestedMessage nestedMessageField = 3;
}

message NestedMessage{
    string myField = 1;
}
```

```
new HelloReply
{
    Message = "Hello " + request.Name,
    MessageType = MessageType.Email,
    NestedMessageField = new NestedMessage()
    {
        MyField = "my value"
    }
}
```

Collections - Repeated

```
message HelloReply {
  string message = 1;
  MessageType messageType = 2;
  NestedMessage nestedMessageField = 3;
  repeated string myRepeatedString = 4;
}
```

Like arrays/lists in C#

Read-only field, cannot be set to null or another list

Any scalar or nested types can be repeated

```
var reply = new HelloReply
{
    Message = "Hello " + request.Name,
    MessageType = MessageType.Email,
    NestedMessageField = new NestedMessage()
    {
        MyField = "my value"
    }
};

List<string> myList = new () { "hello", "world" };

//repeated fields are read-only with no setter
reply.MyRepeatedString.AddRange(myList);
```

Collections - Map

```
message HelloReply {
  string message = 1;
  MessageType messageType = 2;
  NestedMessage nestedMessageField = 3;
  repeated string myRepeatedString = 4;
  map<int32,string> myMapField = 5;
}
```

Like dictionaries in C#

Read-only field, cannot be set to null or another dictionary

Key must be int/string type, value can be any scalar or nested type

```
reply.MyMapField.Add(1, "first value");
reply.MyMapField.Add(new Dictionary<int, string>
{
    {2, "second value" },
    {3, "third value" }
});
```

Datetime

- No built-in scalar in Protobuf to represent date and time
- Use the Google Well-Known Types

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


message SendReadingRequest {
  int64 id=1;
  string deviceName=2;
  double temperature =3;
  google.protobuf.Timestamp updateTime = 4;
}
```

```
new SendReadingRequest
{
    Id = 1,
    DeviceName = "Poorni's device",
    Temperature = 34.5,
    UpdateTime = DateTime.UtcNow.ToTimestamp()

}
```

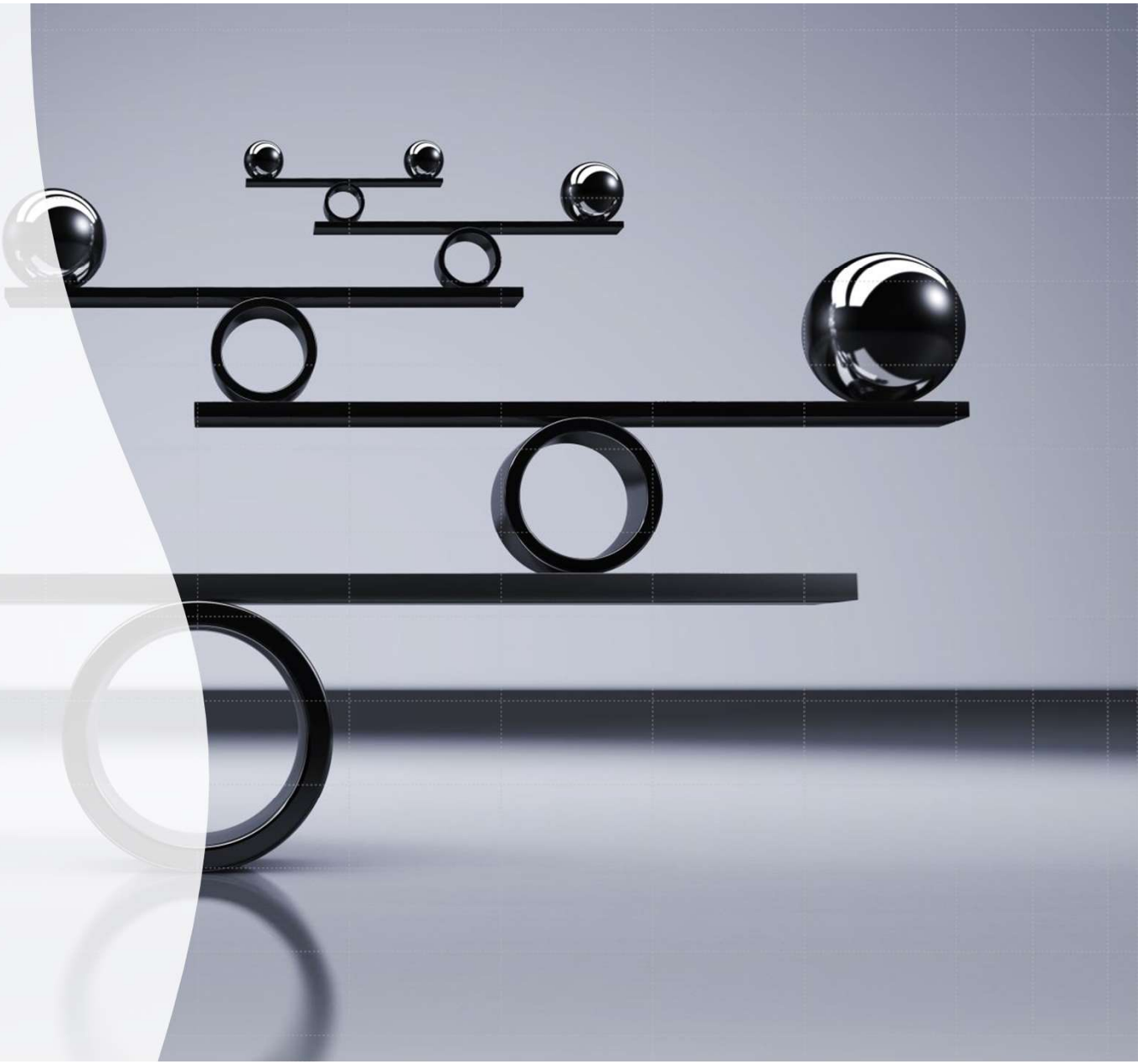
FIELD PRESENCE

- Fields are never null, assigned default values for the type
- Presence or absence of a field
- Proto3 uses no presence semantics, assumes that every field present in a message has value
- When serialized, fields with default value are not serialized
  - 0 for numeric types
  - Enum with 0 value
  - Empty string, empty list

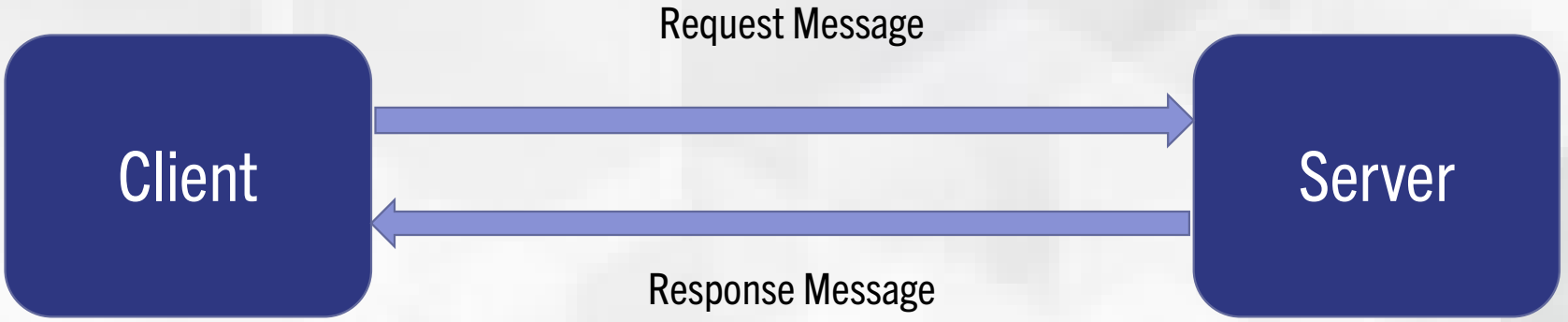
FIELD PRESENCE

- Default value for a field can mean
  - field was explicitly set to default value
  - field was cleared by setting it to default value
  - field was never set

GRPC MODES



UNARY




```
rpc SendReading(SendReadingRequest) returns(SendReadingResponse);
```

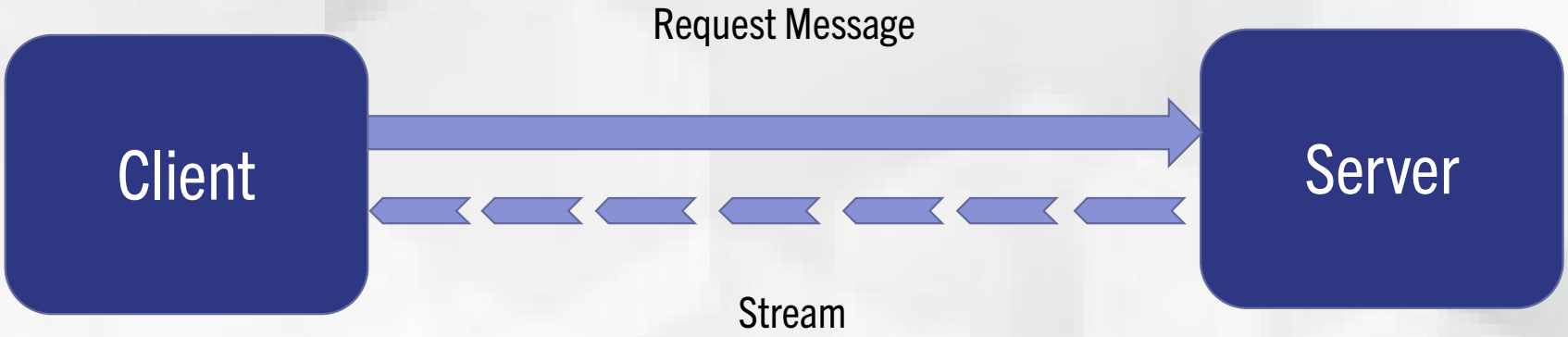
Send device reading in and get a response

```
rpc SendReading(SendReadingRequest) returns(SendReadingResponse);
```

- Call starts with client sending a message, ends with response returned

- .NET gRPC client implementation gives choice of blocking and non-blocking call

SERVER STREAMING



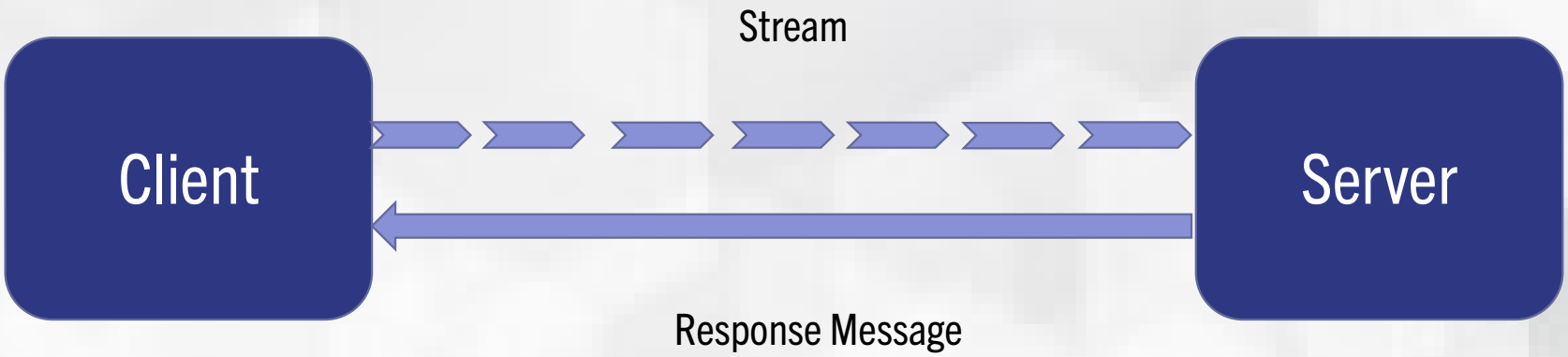

```
rpc GetReadings(GetReadingsRequest) returns (stream GetReadingsResponse);
```

Get a stream of readings for a given date

```
rpc SendReadings(stream SendReadingsRequest) returns (SendReadingsResponse);
```

- Client sends a single request object, triggering the stream to open
- Server streams multiple objects back to client
- Client cannot send additional data once server has started streaming
- Message placed on the stream immediately available to client
- Specify deadlines or cancellations to abort long running streams

CLIENT STREAMING



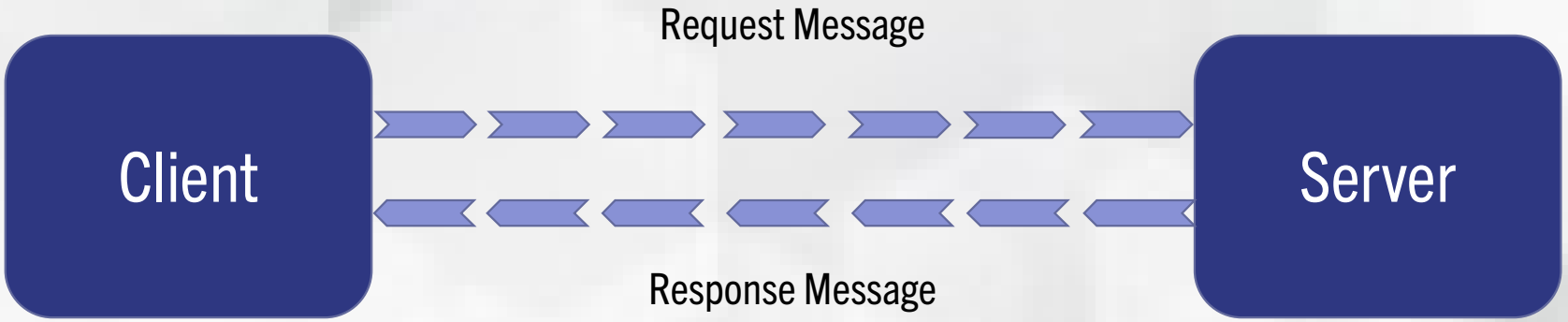

```
rpc SendReadings(stream SendReadingsRequest) returns (SendReadingsResponse);
```

Stream readings to server for processing

```
rpc GetReadings(GetReadingsRequest) returns (stream GetReadingsResponse);
```

- Streaming starts with client invoking the method
- Client places messages to the stream
- Server is notified by the client when streaming is finished
- Server processes the messages and sends the response, status and metadata back to the client
- Ideal when you want to send a large dataset as chunks

BI-DIRECTIONAL STREAMING




```
rpc ProcessReadings(stream ProcessReadingsRequest) returns (stream ProcessReadingsResponse);
```

Process a stream of readings

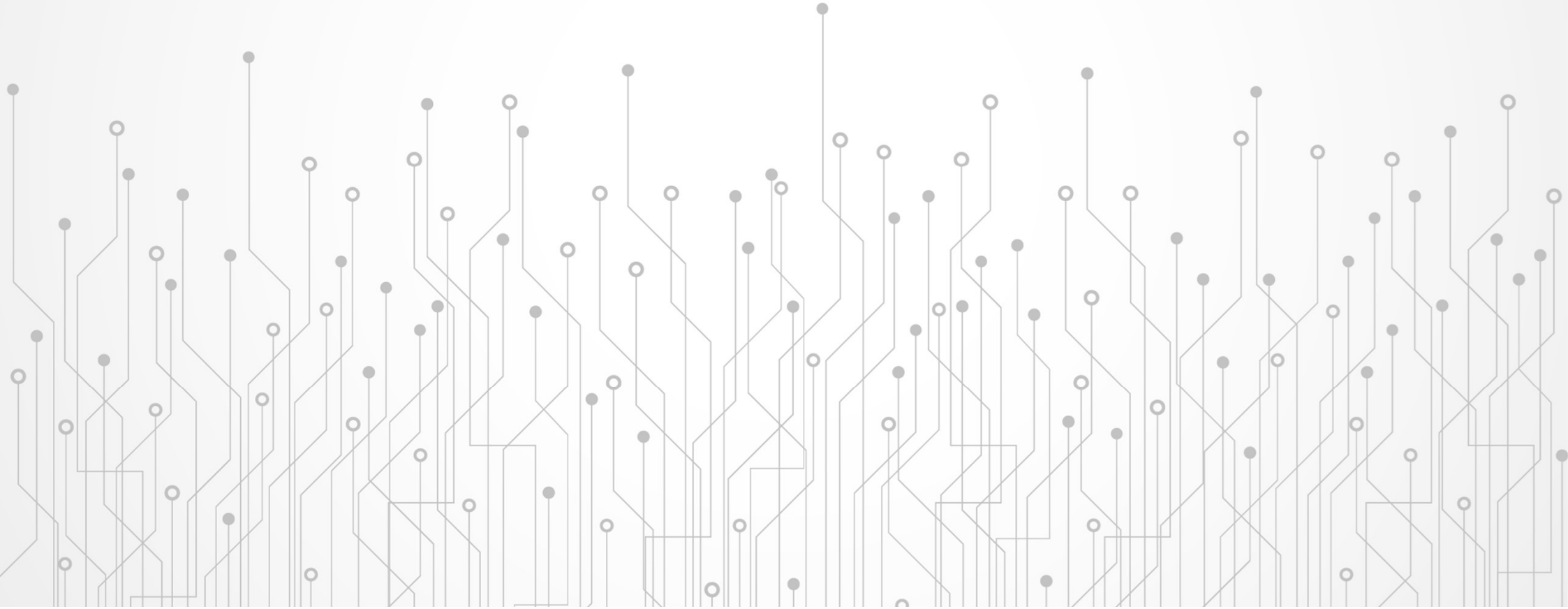
```
rpc ProcessReadings(stream ProcessReadingsRequest) returns (stream ProcessReadingsResponse);
```

- Streaming starts with client invoking the method
- Independent client and server streams
- Client and server can stream one after another or server can send a response for every client request
- Method is complete when streaming completes from server and status and metadata has been sent back


GRPC DEADLINE

- Specifies how long the client will wait for the call to complete
- Sets the upper limit on how long the call can run for
- Prevents services from running forever and exhausting resources
- Client throws a deadline exceeded exception which can be handled on the client
- On server, a cancellation token is raised that must be passed to any child calls as well

METADATA



- Request headers
- Response headers
- Response headers are sent alongside the response
- Response trailers
- Response trailers are served after the response is complete
- Response headers contain information about the call
- Response trailers are about the response!


GRPC
INTERCEPTORS

- Allows app to interact with incoming or outgoing gRPC calls
- Construct a pipeline for gRPC request
- Common use cases includes logging, validation, monitoring, authentication
- Client & Server Interceptors

MIDDLEWARE V/S INTERCEPTORS

- Middleware runs for all HTTP requests, before gRPC interceptors
- Interceptors act only on gRPC layer

Interceptor methods to override on the server

- UnaryServerHandler – for unary RPC
- ClientStreamingServerHandler – for client streaming RPC
- ServerStreamingServerHandler – for server streaming RPC
- DuplexStreamingServerHandler – for bi-directional RPC

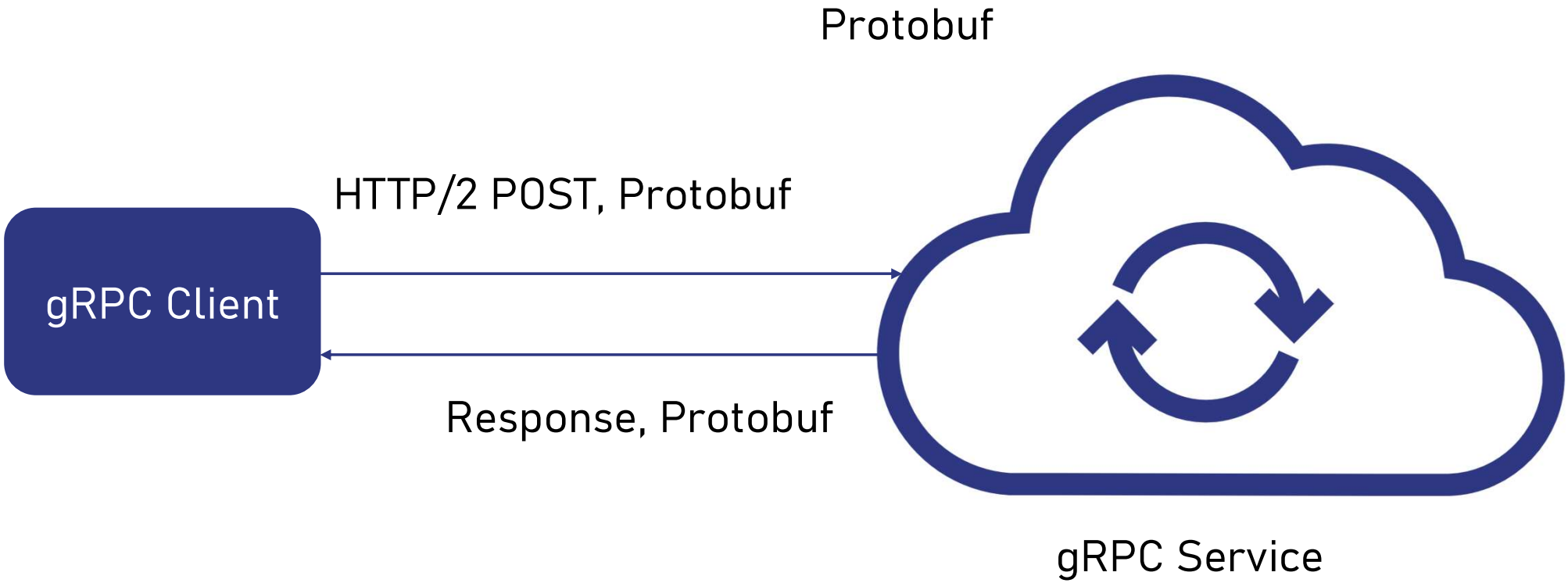
Interceptor methods to override on the client

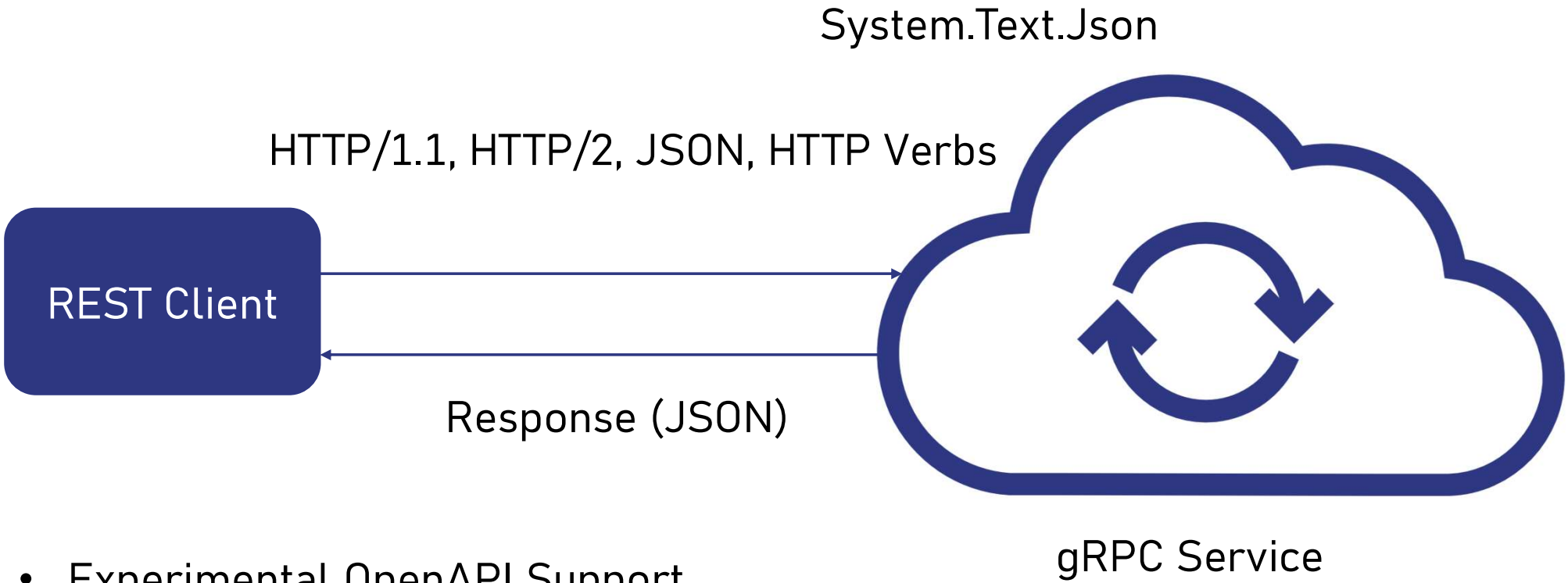
- BlockingUnaryCall – for blocking RPC
- AsyncUnaryCall – for unary RPC
- AsyncClientStreamingCall – for client streaming RPC
- AsyncServerStreamingCall – for server streaming RPC
- AsyncDuplexStreamingCall – for bi-directional RPC

GRPC JSON TRANSCODING

Action-oriented gRPC meets resource-oriented HTTP API

- Extension to ASP.NET Core, feature for mapping between a gRPC method and HTTP endpoints
- Build a single API service that supports both gRPC and HTTP APIs
- Clients can use the endpoint without a generated client
- gRPC service is still intact
- Previously called gRPC HTTP API (experimental), shipped with .NET 7


Protobuf
HTTP/2 POST, Protobuf
gRPC Client
Response, Protobuf
gRPC Service



- Experimental OpenAPI Support

- Wider reach to the API beyond gRPC Clients


HTTPRULE

- Defines the schema of gRPC-HTTP API mapping
- Typically specified as an annotation on the gRPC method using grpc.api.http
- Each mapping specifies a URL path template and an HTTP method(GET, POST, PUT, PATCH, DELETE)

HOSTING GRPC SERVICES

@poornimanayar

- Azure Container Apps
- Azure App Service (Linux Plans only, gRPC-Web on Windows plans)
- Azure Kubernetes Service
- Windows Server 2022

https://bit.ly/grpc-in-dotnet

@poornimanayar