Protocol Buffers and gRPC

Luca Arrotta and Riccardo Presotto

EWLab – Università degli studi di Milano

Professor: Claudio Bettini

These slides are based on previous versions created by Letizia Berolaja, Sergio Mascetti, Dario Freni, Claudio Bettini and Gabriele Civitarese

Copyright

Some slides for this course are partly adapted from the ones distributed by the publisher of the reference book for this course (Distributed Systems: Principles and Paradigms, A. S. Tanenbaum, M. Van Steen, Prentice Hall, 2007).

Every Ware Lab

Outline

- Protocol Buffers
- The gRPC (Remote Procedure Call) Framework

Google's Protocol Buffers

- A mechanism proposed by Google to serialize structured data
- More compact and quicker than XML/JSON
- Data are represented in binary format (so it's more efficient)
- It's sufficient to define a data structure scheme:
  - Marshalling and unmarshalling code is automatically generated (based on the programming language used)
  - Then, it's possible to read/write these data from/to different streams
  - Highly interoperable and libraries for several languages exist

How to Use ProtocolBuffers

1. Define the format of the messages to be serialized throught a *.proto* file
   - It's necessary to define the intended fields with their type
   - *.proto* files must be shared among the programs that have to communicate
2. Build Gradle to run the Protocol Buffers compiler that automatically generates the marshalling and unmarshalling code
3. Then, it's possible to write code to send and receive messages

The Language

- Every message we want to model must be specified with the *message* keyword
  - Every *message* is a record that contains different fields
  - It can contain other *messages*
- Every field of a message must be defined with one of the following keywords:
  - **required**: the field is mandatory
  - **optional**: the field can be omitted
  - **repeated**: the field can be repeated 0 or more times (it's like a dynamic dimension array)

Every Ware Lab

Scalar Types

.proto Type	Notes	C++ Type	Java Type	Python Type[2]	Go Type
double		double	double	float	*float64
float		float	float	float	*float32
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int	*int32
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long[3]	*int64
uint32	Uses variable-length encoding.	uint32	int[1]	int/long[3]	*uint32
uint64	Uses variable-length encoding.	uint64	long[1]	int/long[3]	*uint64
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int	*int32
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long[3]	*int64

Scalar Types

.proto Type	Notes	C++ Type	Java Type	Python Type[2]	Go Type
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint32	int[1]	int/long[3]	*uint32
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	uint64	long[1]	int/long[3]	*uint64
sfixed32	Always four bytes.	int32	int	int	*int32
sfixed64	Always eight bytes.	int64	long	int/long[3]	*int64
bool		bool	boolean	bool	*bool
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	str/unicode[4]	*string
bytes	May contain any arbitrary sequence of bytes.	string	ByteString	str	[]byte

Tags

- Every field defined in a message has a **unique tag**
- Tags are used to identify the fields in the binary format
  - Used to match the fields during the marshalling and unmarshalling operations
  - Tags from 1 to 15 are represented through 1 byte that encodes both the tag and the field type (to be used for the most frequent fields)
  - Tags from 16 to 2047 require 2 bytes
- Tags help the back-compatibilty for the messages definition
  - Systems that didn’t update the messages definition can ignore the new fields
- Tags must not be changed once the message is being used within the system!

Protocol Buffers – An Example

```
message Actor {
    required string name = 1;
    required string surname = 2;

    enum Sex {
        MALE = 0;
        FEMALE = 1;
    }

    message Movie {
        required string title = 1;
        required int32 year = 2;
    }

    repeated Movie movie = 3;
    optional Sex sex = 4;
}
```

Every Ware Lab

Object Building in Java

```
Actor actor =
    Actor.newBuilder()
        .setName("Christian")
        .setSurname("Bale")
        .setSex(Actor.Sex.MALE)
        .addMovie(Actor.Movie.newBuilder()
            .setTitle("The Prestige")
            .setYear(2006))
        .addMovie(Actor.Movie.newBuilder()
            .setTitle("The Dark Knight")
            .setYear(2008))
        .build();
```

Every Ware Lab

Getters

- Besides setters, ProtocolBuffer automatically provides methods to access to every field. For example:
  - *actor.getName()* returns a string
  - *actor.getMovieList()* returns a list of Movie objects

Communication through Sockets

- As already explained, the automatically generated code handles the marshalling and unmarshalling operations for the data communication
- **Client** side: we can use the *writeTo()* method to wrap the socket output stream towards the server
  - *actor.writeTo(socket.getOutputStream());*
- **Server** side: we have to call the *parseFrom()* method to wrap the input stream
  - *actor.parseFrom(socket.getInputStream());*

Protocol Buffer References

- Protocol Buffer Basics: Java
- Protocol Buffer's Developer Guide
- Proto's Language Guide

Exercise – Students Stats with Protocol Buffers

- Build a server application *University* that receives data from a client about a *Student*
- The client sends through sockets to *University* data related to a specific *Student*:
  - Personal details: Name, Surname, Year of Birth, Place of Residence (multi-attributes field)
  - List of passed exams: for each exam, store Exam Name, Mark and Date of Verbalization
- The *University* receives the message from the socket and prints the student's stats
- The solution must be built by using Protocol Buffers
- Compare how many bytes are saved with Protocol Buffers instead of JSON!

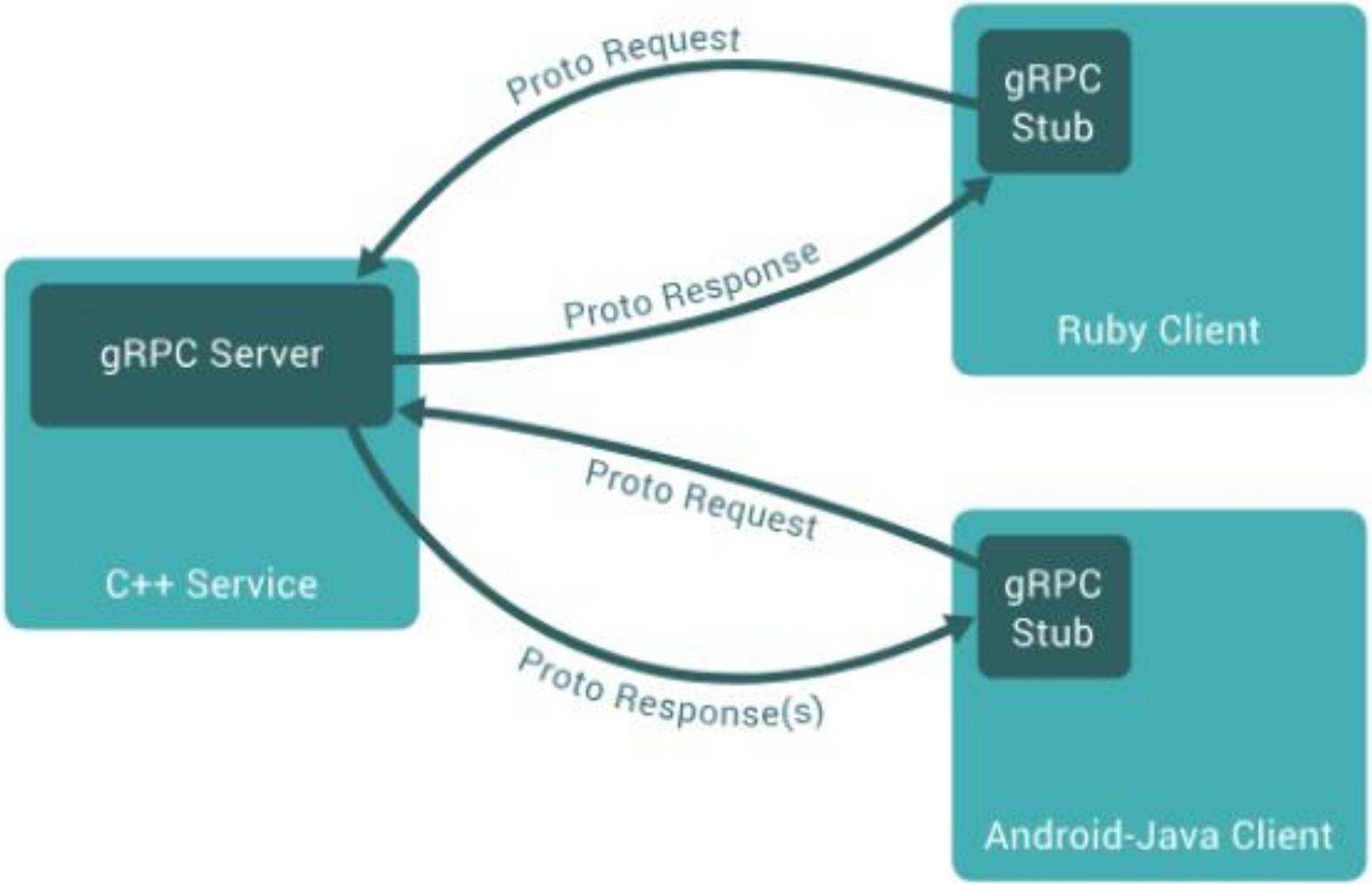
Remote Procedure Call (RPC)

- A paradigm for network services
- Instead of explicitly sending and receiving messages, the client invokes a remote service through a call to a local procedure
- The local procedure (*stub*) hides all the network communication details
- Internally, the RPC library performs the *marshalling*, transmitting to the server the parameters of the *stub* method
- The server performs the *unmarshalling*, invokes the method and sends the output back to the client
- The client is not aware of calling a network call while it calls the local procedure

The gRPC Framework

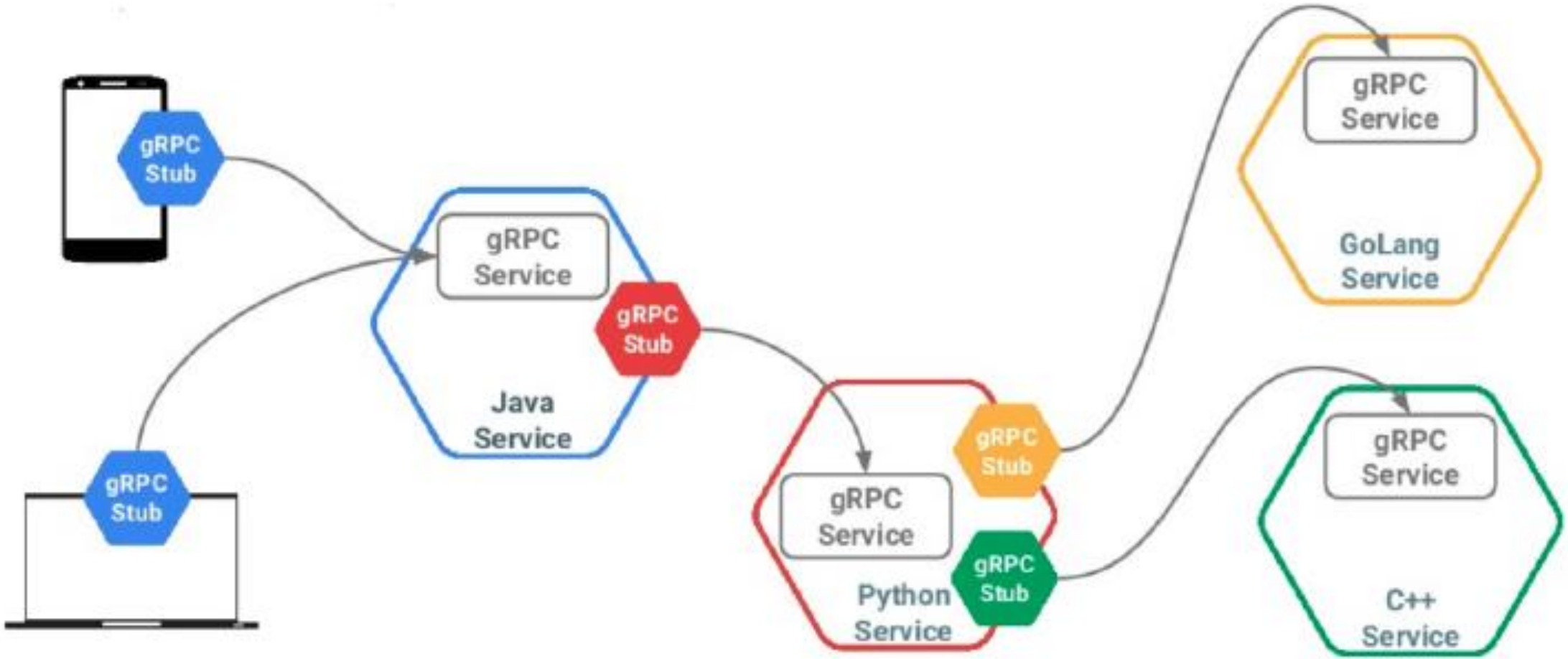
- RPC system developed and widely used by Google (and others, like Netflix)
- Services defined through **Protocol Buffers**
  - Method names, arguments, type of the returned value, …
- Server side it is necessary to implement the interface and run a gRPC server to handle the client calls
- Client side we just have to call the *stub* methods provided by the server

Architecture



Every Ware Lab

High Interoperability



Every Ware Lab

Pros

- Communication efficiency granted by Protocol Buffer since it uses data binary representation
- Communication channel based on *HTTP2*
  - It leverages *multiplexing*: different requests and responses can be asynchronously received with a single TCP connection
- Strongly typed
- More advanced than REST that is based on only on HTTP verbs

Definition Example

```
// The greeter service definition
service Greeter {
    // Sends a greeting
    rpc Greeting (HelloRequest) returns (HelloResponse) {}
}

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

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

Synchronous vs Asynchronous

- It’s possible to leverage RPC methods both in a synchronous and asynchronous ways (based on the application)
- Synchronous calls are blocking
  - The client call the stub method and it is blocked until it receives the response from the server
  - Approximation closer to the astraction of an RPC procedure
- Distributed systems are asynchronous, and gRPC allows the implementation of asynchronous stubs
  - The client calls the stub method without blocking the current thread
  - The server response will be received asynchronously

RPC Types

1. **Unary RPC**: the client sends a single request and receives a single response
2. **Server Streaming RPC**: the client sends a single request, the server returns a stream of responses. The client ends after all the responses have been received
3. **Client Streaming RPC**: the client sends a stream of requests, the server returns a single response (not necessarily once received all the requests of the stream)
4. **Bidirectional streaming RPC**: The streams are independent, client and server can write in any order.

RPC Types

1. **Unary RPC**:
   rpc Greeter (HelloRequest) returns (HelloResponse) {}

2. **Server Streaming RPC**:
   rpc LotsOfReplies (HelloRequest) returns (stream HelloResponse) {}

3. **Client Streaming RPC**:
   rpc LotsOfGreetings (stream HelloRequest) returns (HelloResponse) {}

4. **Bidirectional streaming RPC**:
   rpc BidiHello (stream HelloRequest) returns (stream HelloResponse) {}

Building Command

- Build Gradle to compile the service definition and generate the codes for the client and server stubs
- Server side, for each service defined in the proto file, it is necessary to implement its logic
- Client side, it is only required to specify the service address and call the stub methods

Greeter: Server Side

```
private class GreeterImpl extends GreeterImplBase {
    @Override
    public void greeting(HelloRequest req, StreamObserver<HelloResponse> responseObserver) {
        HelloResponse reply = HelloResponse.newBuilder()
        .setMessage(" Hello " + req.getName()).build();
        responseObserver.onNext(reply);
        responseObserver.onCompleted();
    }
}
```

Greeter: Server Side

```
Server server = ServerBuilder.forPort(8080)
    .addService(new GreetingServiceImpl())
    .build();

server.start();

server.awaitTermination();
```

- By default, the *ServerBuilder* is built with a pool of threads that handle the requests
  - Every gRPC call will be asynchronous!
  - We must handle the concurrency issues, like multi-threaded servers

Greeter: Client Side

```
// plaintext channel on the address which offers the GreetingService service
final ManagedChannel channel = ManagedChannelBuilder.forTarget("localhost:8080")
    .usePlaintext([redacted])
    .build();

// creating a blocking stub on the channel
GreetingServiceBlockingStub stub = GreetingServiceGrpc.newBlockingStub(channel);

// creating the HelloResponse object ( argument for RPC call )
HelloRequest request = HelloRequest.newBuilder().setName("Gino").build();

// calling the method . it returns an instance of HelloResponse
HelloResponse response = stub.greeting(request);

// printing the answer
System.out.println(response.getGreeting());

// closing the channel
channel.shutdown();
```

Lab

The *StreamObserver* class

- Class used to communicate asynchronously on a stream
- Used to receive asynchronous notifications from a stream of messages
- Used both by the client stub and the server services to communicate on a stream
- It provides three handler methods:
  - *onNext(V value)*: receive a value from the stream
  - *onError(Throwable t)*: receive an error occurred on the stream
  - *onCompleted()*: receive the communication that the stream correctly ended
- Who receives the stream has to implement these methods
- Who sends the stream has to call the *stub*s of these methods (that are RPC methods)

Greeting Stream Example – The .proto File

```
service Greeter {
    rpc StreamGreeting (HelloRequest) returns (stream HelloResponse) {}
}

message HelloRequest {
    string name = 1;
}

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

Every Ware Lab

Greeting Stream Example – Server Side

```
public void streamGreeting(HelloRequest request, StreamObserver<HelloResponse> responseObserver) {
    HelloResponse response = HelloResponse.newBuilder()
        .setGreeting("Hello, " + request.getName())
        .build();
    responseObserver.onNext(response);
    responseObserver.onNext(response);
    responseObserver.onNext(response);
    responseObserver.onCompleted();
}
```

Greeting Stream Example – Client Side

```
final ManagedChannel channel = ManagedChannelBuilder
    .forTarget("localhost:8080").usePlaintext([redacted]).build();

GreetingServiceStub stub = GreetingServiceGrpc.newStub(channel);

HelloRequest request = HelloRequest.newBuilder().setName("Gino").build();

stub.streamGreeting(request, new StreamObserver<HelloResponse>() {
    // this handler takes care of each item received on the stream
    public void onNext(HelloResponse helloResponse) {
        System.out.println(helloResponse.getGreeting());
    }
    // if there are errors, this method will be called
    public void onError(Throwable throwable) {
        // handle error
    }
    //when the stream is completed, just close the channel
    public void onCompleted() {
        channel.shutdownNow();
    }
});
```

Every Ware Lab

Exercise – gRPC Sums

- Build a service through gRPC
- The service must offer these RPC methods:
  - *SimpleSum*: given two integers, it returns their sum (try to build this method synchronously and asynchronously)
  - *RepeatedSum*: given two numbers *n* and *t*, the service returns a stream of *t* numbers. The first value of the stream is *n*, the second *n+n*, the third *n+n+n*, and so on, until the stream contains *t* values
  - *StreamSum*: the client sends on the stream couples of numbers to sum up, the server returns their sum

References

- Code Examples: https://ewserver.di.unimi.it/gitlab/luca.arrotta/lab3-examples

- Exercises Setup: https://ewserver.di.unimi.it/gitlab/riccardopresotto/setup_grpc

Every Ware Lab

Contact

- Contact the tutors via email for any clarification or meeting:

tutorsdp@di.unimi.it

Every Ware Lab