



↑GRPC↓

1.0 [2016]

1.42 [2022]



gRPC

- Mecanismo de RPC moderno, com estrutura semelhante aos RPC tradicionais, mas desenvolvido com **foco em serviços cloud**
 - E não o paradigma cliente/servidor mais “básico”
- Baseado no mecanismo RPC usado internamente pela **Google** para implementar os seus serviços cloud
 - Resultado de mais de uma década de experiência da empresa a usar RPC para construir serviços altamente escaláveis
- Atualmente **open-source**
 - Também por isso é atualmente o RPC mais popular



gRPC = Google RPC?

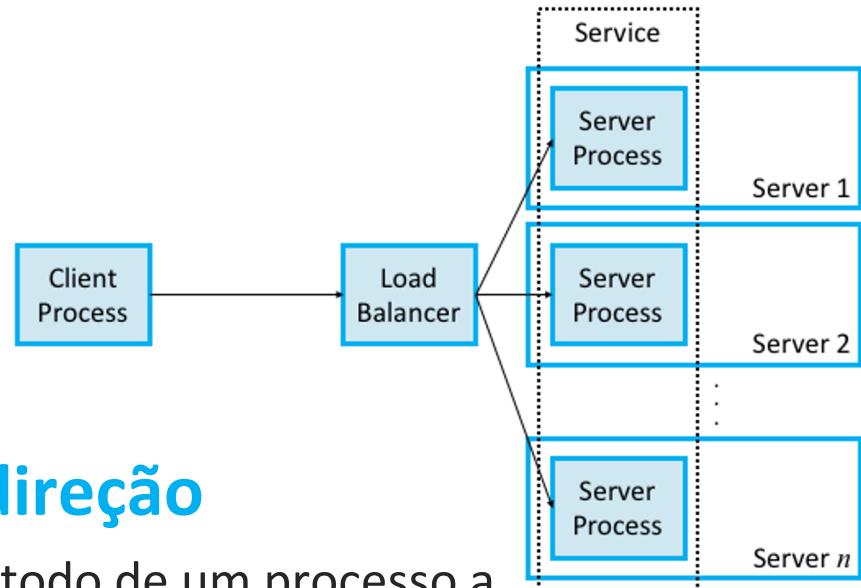
- Não!
- O significado do **g** varia como número de versão...
 - 1.0 'g' stands for 'gRPC' (acrônimo recursivo)
 - 1.1 'g' stands for 'good'
 - 1.2 'g' stands for 'green'
 - 1.3 'g' stands for 'gentle'
 - ...
 - 1.42 'g' stands for 'great'





gRPC: RPC da era cloud

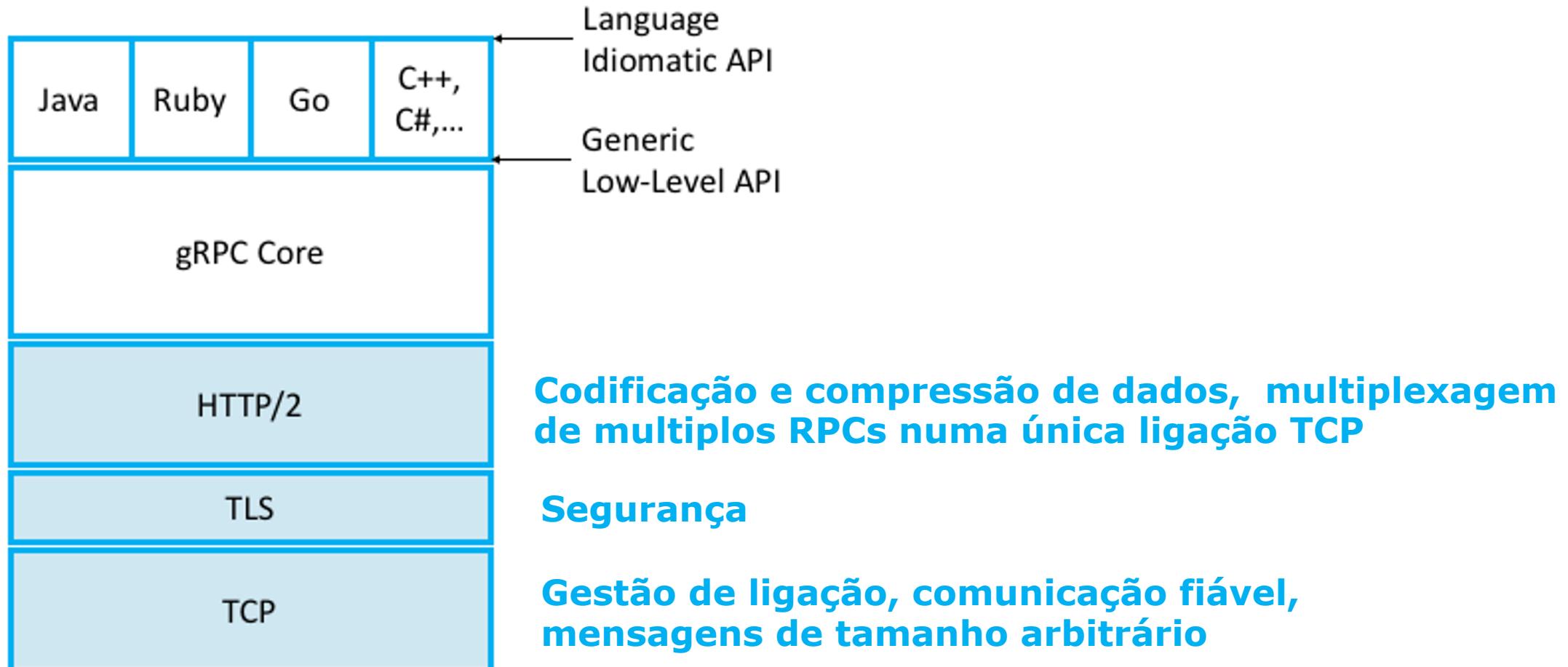
- Paradigma centrado em **serviços de cloud**
 - vs paradigma cliente-servidor
- Diferença essencial: um nível extra de **indireção**
 - Nos sistemas cliente-servidor, o cliente invoca um método de um processo a correr numa máquina
 - Em serviços cloud, o cliente invoca um método de um **serviço** tipicamente implementado num número elevado de processos a correr em máquinas distintas
- Na realidade, os serviços lançam **containers**
 - Um *container* é essencialmente um processo encapsulado num ambiente isolado que inclui todas as *packages* de software necessárias para o processo correr





gRPC stands on the shoulders of giants

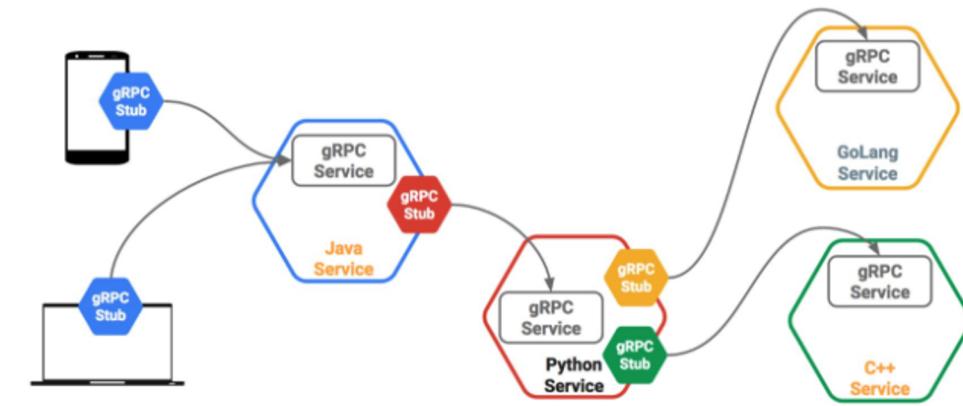
- O gRPC faz “outsourcing” de vários problemas para outros protocolos





Cenários de utilização do gRPC

- Comunicação para sistemas distribuídos com requisitos de:
 - baixa **latência** e elevada **escalabilidade**
- **Multi-linguagem**, multi-plataforma
- Exemplo de referência:
 - Clientes móveis a comunicar com servidores na nuvem
- Protocolo **eficiente**, **preciso** e **independente da linguagem** de programação





Linguagens de programação suportadas

- C++, Java, Python, Go
 - Implementação **completa**, incluindo reflexão
- Ruby, C#, JavaScript, Dart
 - Sem reflexão (i.e., não se consegue descobrir tipos de dados em tempo de execução)
- Android Java, Objective-C, PHP
 - Apenas cliente



Sistema gRPC

- Usa uma **IDL** para definir os tipos de dados e as operações
- Disponibiliza **ferramenta de geração de código** a partir do IDL
 - Trata da conversão de dados
 - Gestão da invocação remota
- Permite ter chamadas remotas **síncronas e assíncronas**
 - As chamadas síncronas esperam pela resposta
 - Nas chamadas assíncronas a resposta é verificada mais tarde



Estrutura do RPC

Linguagem de descrição de interfaces remotas

Stubs para adaptar dados de cada procedimento

Biblioteca de *run-time* para o suporte genérico

Gestor de nomes para localizar servidores



Estrutura do RPC

Linguagem de descrição de interfaces remotas

Stubs para adaptar dados de cada procedimento

Biblioteca de *run-time* para o suporte genérico

Gestor de nomes para localizar servidores



gRPC IDL: *Protocol Buffers*

- Os *protocol buffers* (protobuf) são uma **IDL** para descrever mensagens e operações
 - Permite estender/modificar os esquemas, mantendo compatibilidade com versões anteriores
 - O protobuf é um formato canónico
 - A apresentação de dados tem uma estrutura explícita
- O compilador **protoc** converte a IDL em código para uma grande variedade de linguagens de programação
- As camadas mais baixas do gRPC não dependem da IDL
 - Em teoria é possível usar alternativas ao protobuf



Protobuf passo a passo

```
message Person {  
    string name = 1;  
    int32 id = 2;  
    string email = 3;  
}
```

```
Person person =  
Person.newBuilder()  
.setId(12521)  
.setName("Victor Smith")  
.setEmail("vs@example.com")  
.build();  
output = new  
socket.getOutputStream();  
person.writeTo(output);
```

12 07 74 65 73 74 69 6e 67 ...

Definir o esquema
de dados

Gerar código de acesso
para uma linguagem específica
do cliente ou servidor
(ex. Java)

Dados são
serializados/deserializados para
formato binário



Vantagens protobuf (vs. XML e JSON)

- Formato **eficiente**
 - Mais eficiente que JSON e que XML
- Boa integração em várias linguagens



Variáveis e etiquetas (*tags*) protobuf

- Todas as variáveis são **fortemente tipificadas**
- São seguidas por um **número de etiqueta sequencial** (*tag*)
 - Define a **ordem** de serialização dos campos
 - É necessário saber as etiquetas para conseguir interpretar a mensagem.
Esta ordem **não pode mudar** depois de definida
- Protobuf usa **codificação explícita** devido às *tags*
 - Fundamental para eficiência
 - A *tag* identifica o campo mas não o tipo
 - Por isso, o conhecimento da definição do proto é necessária para reconstruir a estrutura de dados original



Ficheiro .proto

- Um ficheiro .proto **define os objetos e campos desejados**, bem como as **operações** que usam essas mensagens
- Cada campo tem que ter uma **etiqueta numérica única**



Exemplo: HelloWorld.proto

```
message HelloRequest {  
    string name = 1;  
    repeated string hobbies = 2;  
}  
  
message HelloResponse {  
    string greeting = 1;  
}  
  
service HelloWorldService {  
    rpc greeting(HelloRequest) returns (HelloResponse);  
}
```

Variáveis fortemente tipificadas

Variáveis de uma mensagem têm tags diferentes

Lista

Definição de serviço. Um serviço pode conter múltiplas operações RPC.

Operações têm apenas um argumento/mensagem de entrada e um único resultado



Operação gRPC

- A definição de uma operação remota em *Protocol Buffers* permite apenas **um único argumento** e **um único resultado**
 - É preciso definir o tipo de dados do pedido e o tipo de dados da resposta
 - Para passar múltiplos argumentos para uma operação, é necessário que eles estejam agrupados num mesmo tipo de mensagem
 - O mesmo se aplica ao retorno de uma operação

```
service HelloWorldService {  
    rpc greeting(HelloRequest) returns (HelloResponse);  
}
```



Protobuf: tipos de dados simples

Protobuf	C++	Java	Python	Go
double	double	double	float	*float64
int32	int32	int	int	*int32
int64	int64	long	long	*uint64
bool	bool	boolean	bool	*bool
string	string	String	unicode	*string
bytes	string	com.google.protobuf.ByteString	str	[]byte
...				



Protobuf: outros tipos de dados (1/2)

- Tipos aninhados (*nested types*)
 - Permite definir mensagens dentro de mensagens
- Mapas
 - Mapas associativos
- *Oneof*
 - Permite ter vários campos opcionais sendo que apenas um pode ser definido
 - Semelhante a uma *union* da linguagem C

```
message SearchResponse {  
    message Result {  
        string url = 1;  
        string title = 2;  
        repeated string snippets = 3;  
    }  
    repeated Result results = 1;  
}
```

```
map<string, Project> projects = 3;
```

```
message SampleMessage {  
    oneof test_oneof {  
        string name = 4;  
        SubMessage sub_message = 9;  
    }  
}
```



Protobuf: outros tipos de dados (2/2)

- Enums

- Alternativa a enviar *strings* repetidas pela rede com códigos de resultado, mensagens de erro
- O valor serializado contém apenas a etiqueta do campo

- Services (RPC)

- Definição que permite ao compilador de *protocol buffers* gerar a interface do serviço e *stubs* (cliente e servidor) adequados à linguagem escolhida

```
message SearchRequest {  
    string query = 1;  
    int32 page_number = 2;  
    int32 result_per_page = 3;  
    enum Corpus {  
        UNIVERSAL = 0;  
        WEB = 1;  
        IMAGES = 2;  
        LOCAL = 3;  
        NEWS = 4;  
        PRODUCTS = 5;  
        VIDEO = 6;  
    }  
    Corpus corpus = 4;  
}
```

```
service HelloWorldService {  
    rpc greeting(HelloRequest) returns (HelloResponse);  
}
```



gRPC bibliotecas de tipos adicionais

- "google/protobuf" fornece mais tipos de dados
- Exemplo: uma representação de marcas temporais:
 - `import "google/protobuf/timestamp.proto";`
- Mais informação
 - <https://developers.google.com/protocol-buffers/docs/proto3>



protoc

- O compilador protoc **gera código** a partir da IDL
- Este compilador pode ser chamado a partir de ferramentas como o Maven
 - mvn generate-sources
- A partir do exemplo anterior, foram gerados dois ficheiros .java
 - **HelloWorld.java** – tipos de dados
 - Métodos *getter*, *setter*, etc.
 - **HelloWorldServiceGrpc.java** – definição da operação remota



Exemplo: HelloWorld.java (gerado pelo compilador)

```
// Generated by the protocol buffer compiler. DO NOT EDIT!

public final class Hello {
// ...

    public interface HelloRequestOrBuilder extends
com.google.protobuf.MessageOrBuilder {

        java.lang.String getName();

        java.util.List<java.lang.String> getHobbiesList();
```

```
message HelloRequest {
    string name = 1;
    repeated string hobbies = 2;
}
```



Exemplo: HelloWorld.java (continuação)

```
// ...
```

```
public interface HelloResponseOrBuilder extends  
com.google.protobuf.MessageOrBuilder {
```

```
java.lang.String getGreeting();
```

```
}
```

```
}
```

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



Protobuf: codificação

- Objetivo principal:
 - Bom **desempenho** através de elevada **eficiência** na serialização
- Diferentes estratégias
 - Quando cada campo está codificado em binário é necessário **concatenar na ordem definida** pelas etiquetas do ficheiro .proto
 - Para representar os campos existem **diferentes estratégias para delimitar** o início de um novo campo
 - Se o receptor não conhece uma *tag*, pode **ignorar** e seguir para a próxima, permitindo que o .proto evolua



Protobuf: estratégias de codificação

Strategy	Name	Types
0	Varint (variable size integers)	Int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	Non-varint 64-bit (fixed size integers)	Fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
5	Non-varint 32-bit	Fixed32, sfixed32, float



Varint encoding

- Varint - *variable size integer*
- Comprimem-se os inteiros **no menor espaço possível**, em vez de enviar um número fixo de bytes
 - Exemplo: o número inteiro (int32) **227**
 - a) $227 \Rightarrow \textcolor{blue}{1110\ 0011}$
 - b) Partir em grupos de 7 bits: $\textcolor{blue}{1110\ 0011} \Rightarrow 000\ 000\textcolor{red}{1}\ \textcolor{blue}{110\ 0011}$
 - c) Adicionar bit 0 em cada byte a indicar fim da mensagem:
 $0000\ 000\textcolor{red}{1}\ \textcolor{blue}{1110\ 0011}$
 - Quando o valor está codificado, é necessário prefixar com:
 - Número de etiqueta (1 neste caso)
 - Estratégia de codificação adotada (0 neste caso: bastam 3 bits)
 - Resultado: **227** armazenado usando 3 bytes, em vez de 4.

```
message Test {  
    int32 a = 1;  
}
```



Outros tipos

- *Non-Varint encoding*
 - *Fixed size integer*
 - *Cada variável ocupa espaço de acordo com o tipo*
 - *O valor é guardado em little-endian byte order*
- *Length-delimited encoding*
 - A estratégia de codificação do tipo é idêntica à anterior, incluindo o **tipo** seguido do **comprimento da cadeia de caracteres**, que estará codificada com ASCII ou UTF-8



Length-delimited encoding

- A estratégia de codificação do tipo é idêntica à anterior, incluindo o **tipo** seguido do **comprimento da cadeia de caracteres**, que estará codificada com ASCII ou UTF-8



Estrutura do RPC

Linguagem de descrição de interfaces remotas

Stubs para adaptar dados de cada procedimento

Biblioteca de *run-time* para o suporte genérico

Gestor de nomes para localizar servidores



HelloServiceGrpc.java (generated client-side code)

```
// Generated by the protocol buffer compiler. DO NOT EDIT!
```

```
public final class HelloServiceGrpc {
```

O “canal” é a abstração usada para nos ligarmos ao *endpoint* de um serviço

```
    public static HelloServiceStub newStub(io.grpc.Channel channel){
```

```
        return new HelloServiceStub(channel);
```

```
}
```

O cliente é que decide se quer um stub bloqueante ou um stub assíncrono.

```
    public static HelloServiceBlockingStub newBlockingStub(
```

```
        io.grpc.Channel channel) {
```

```
        return new HelloServiceBlockingStub(channel);
```

```
}
```

```
service HelloWorldService {  
    rpc greeting(HelloRequest) returns (HelloResponse);  
}
```



HelloClient.java (código do cliente)

```
final ManagedChannel channel =  
ManagedChannelBuilder.forTarget(target).usePlaintext().build();
```

Canal não-seguro

```
HelloServiceGrpc>HelloServiceBlockingStub stub =  
HelloServiceGrpc.newBlockingStub(channel);
```

Cliente opta por stub bloqueante.

```
Hello>HelloRequest request =  
Hello>HelloRequest.newBuilder().setName("friend").build();
```

Pedido enviado através do stub.

```
Hello>HelloResponse response = stub.greeting(request);
```

```
System.out.println(response);  
channel.shutdownNow();
```

O HelloResponse gera automaticamente
(e muito convenientemente) um método
toString().



HelloServer.java (código do servidor)

```
public class HelloServer {  
    public static void main(String[] args) throws Exception {  
        (...)  
        final int port = Integer.parseInt(args[0]);  
        final BindableService impl = new HelloWorldServiceImpl();  
  
        Server server = ServerBuilder.forPort(port).addService(impl).build();  
  
        server.start();  
        System.out.println("Server started");  
  
        // Do not exit the main thread. Wait until server is terminated.  
        server.awaitTermination();  
    }  
}
```

Instanciando o objeto servidor.

Inicia servidor.

Cria novo servidor à escuta no porto definido.



HelloServiceGrpc.java (generated server-side code)

```
// Generated by the protocol buffer compiler. DO NOT EDIT!
```

```
public final class HelloServiceGrpc {
```

```
// ...
```

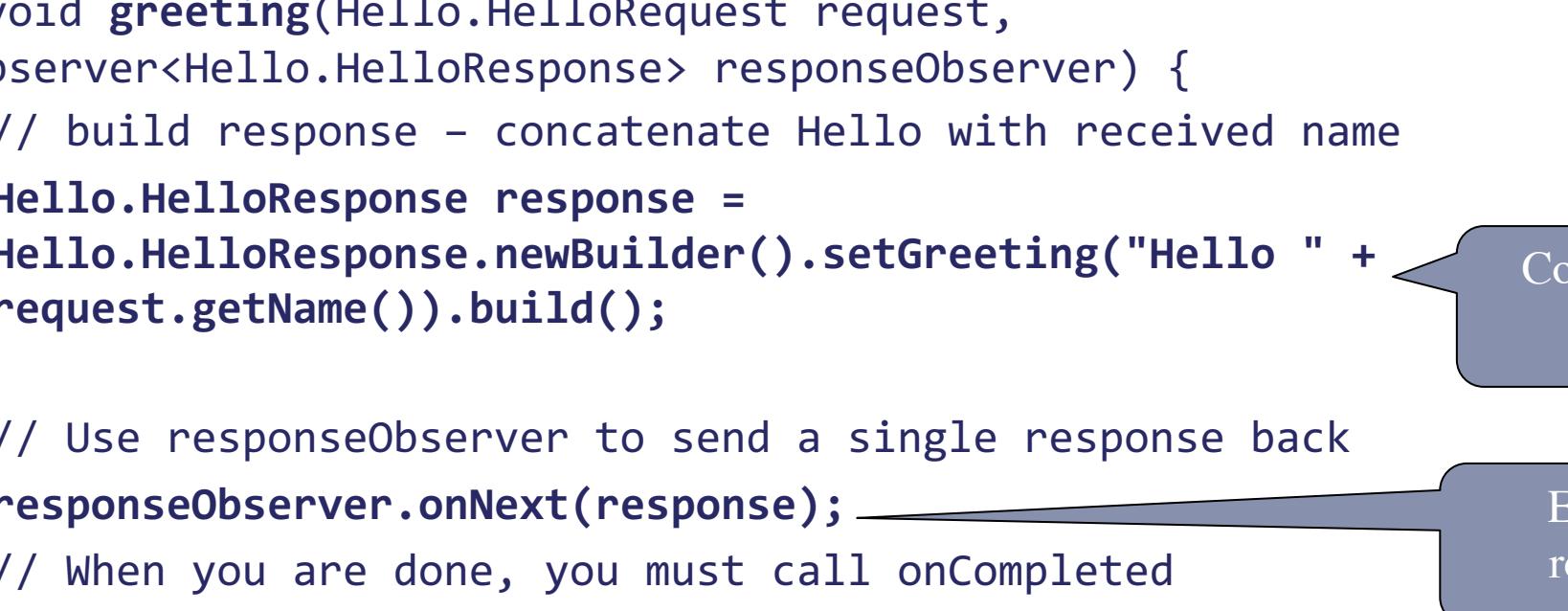
```
public static abstract class HelloServiceImplBase implements  
io.grpc.BindableService {  
public void greeting(Hello>HelloRequest request,  
io.grpc.stub.StreamObserver<Hello>HelloResponse> responseObserver) {  
asyncUnimplementedUnaryCall(getGreetingMethod(), responseObserver);  
}
```

Necessário implementar esta classe abstrata.



HelloWorldServiceImpl.java (código do servidor)

```
public class HelloServiceImpl extends HelloServiceGrpc.HelloServiceImplBase {  
    @Override  
    public void greeting(Hello>HelloRequest request,  
        StreamObserver<Hello>HelloResponse> responseObserver) {  
        // build response - concatenate Hello with received name  
        Hello>HelloResponse response =  
            Hello>HelloResponse.newBuilder().setGreeting("Hello " +  
                request.getName()).build();  
  
        // Use responseObserver to send a single response back  
        responseObserver.onNext(response);  
        // When you are done, you must call onCompleted  
        responseObserver.onCompleted();  
    }  
}
```



Construção da resposta.

Envio da resposta.



Estrutura do RPC

Linguagem de descrição de interfaces remotas

Stubs para adaptar dados de cada procedimento

Biblioteca de *run-time* para o suporte genérico

Gestor de nomes para localizar servidores



gRPC *run-time*

- Cabe à biblioteca de *run-time* a gestão do **canal** para realizar a chamada remota
- Um canal
 - É uma ligação virtual que liga o cliente ao servidor
 - Pode corresponder a um ou mais *sockets* físicos ao longo do tempo
- O transporte de dados em gRPC é feito exclusivamente com **HTTP/2**



Canais gRPC

- Tipos
 - **Unary RPC**: cliente envia 1 pedido, servidor envia 1 resposta
 - Client streaming RPC: Cliente envia múltiplas mensagens, servidor envia 1 resposta
 - Server streaming RPC: Cliente envia 1 pedido, servidor envia múltiplas mensagens
 - Bidi streaming RPC: Cliente e servidor envia múltiplas mensagens
- Streaming permite que um pedido ou resposta possam ser constituídos por múltiplas mensagens de tamanhos arbitrariamente grandes
 - e.g., upload ou download de grandes quantidades de informação



gRPC códigos de resultado

- Como os RPC envolvem a rede, as **falhas** vão acontecer
 - Os clientes devem ser escritos para esperar e tratar estas falhas
- Uma operação remota pode devolver um código de resultado (**erro**) em alternativa ao tipo de mensagem definido na IDL
 - 0 – OK
 - 2 – unknown (default error code)
 - 4 – deadline exceed
 - 6 – already exists
 - 8 – resource exhausted
 - 10 – aborted
 - 12 – unimplemented
 - 14 – unavailable (temporary)
 - 16 – unauthenticated
 - 1 – cancelled
 - 3 – invalid argument
 - 5 – not found
 - 7 – permission denied
 - 9 – failed precondition
 - 11 – out of range
 - 13 – internal
 - 15 – data loss



Estrutura do RPC

Linguagem de descrição de interfaces remotas

Stubs para adaptar dados de cada procedimento

Biblioteca de *run-time* para o suporte genérico

Gestor de nomes para localizar servidores



Resolução de nomes em gRPC

- O gRPC suporta DNS como serviço de nomes por omissão
- A biblioteca cliente fornece mecanismo de *plug-ins* que permite a resolução de nomes em diferentes sistemas
- A definição de um canal gRPC usa a sintaxe de URI [RFC 3986]
- Esquema comum: **dns:[//authority/]host[:port]**
 - *host* é o servidor a resolver via DNS
 - *port* é o porto. Se não for especificado, assume-se 443 (seguro) ou 80
 - *authority* indica o servidor DNS a usar
- Os *resolvers* contactam a autoridade de resolução e devolvem o par <endereço IP, porto>
 - juntamente com um booleano que indica se se trata do servidor de destino ou de um平衡ador de carga



Bibliografia recomendada

- **Principais**

- K. Indrasiri & D. Kuruppu, “gRPC Up & Running”
- Site oficial: <https://grpc.io/docs/>

- **Opcionais**

- <http://dist-prog-book.com/chapter/1/gRPC.html#grpc>
- Secção 5.3 do livro *Peterson and Davie, “Computer Networks, A Systems Approach”*
- <https://book.systemsapproach.org/e2e/rpc.html#rpc-implementations-sunrpc-dce-grpc>
- Caps. 1-4 do livro *Humphries et al., “Practical gRPC”*
- [Coulouris2012], Secção 21.4.1

