COMPUTAÇÃO NA NUVEM
ISEL – LEIRT / LEIC

Evolução dos mecanismos para chamadas remotas (Parte 2)

RPC - *Remote Procedure Call* - O caso *Google RPC (gRPC)*

José Simão jsimao@cc.isel.ipl.pt ; jose.simao@isel.pt

Luís Assunção lass@isel.ipl.pt ; luis.assuncao@isel.pt

Algumas limitações da abordagem Java RMI

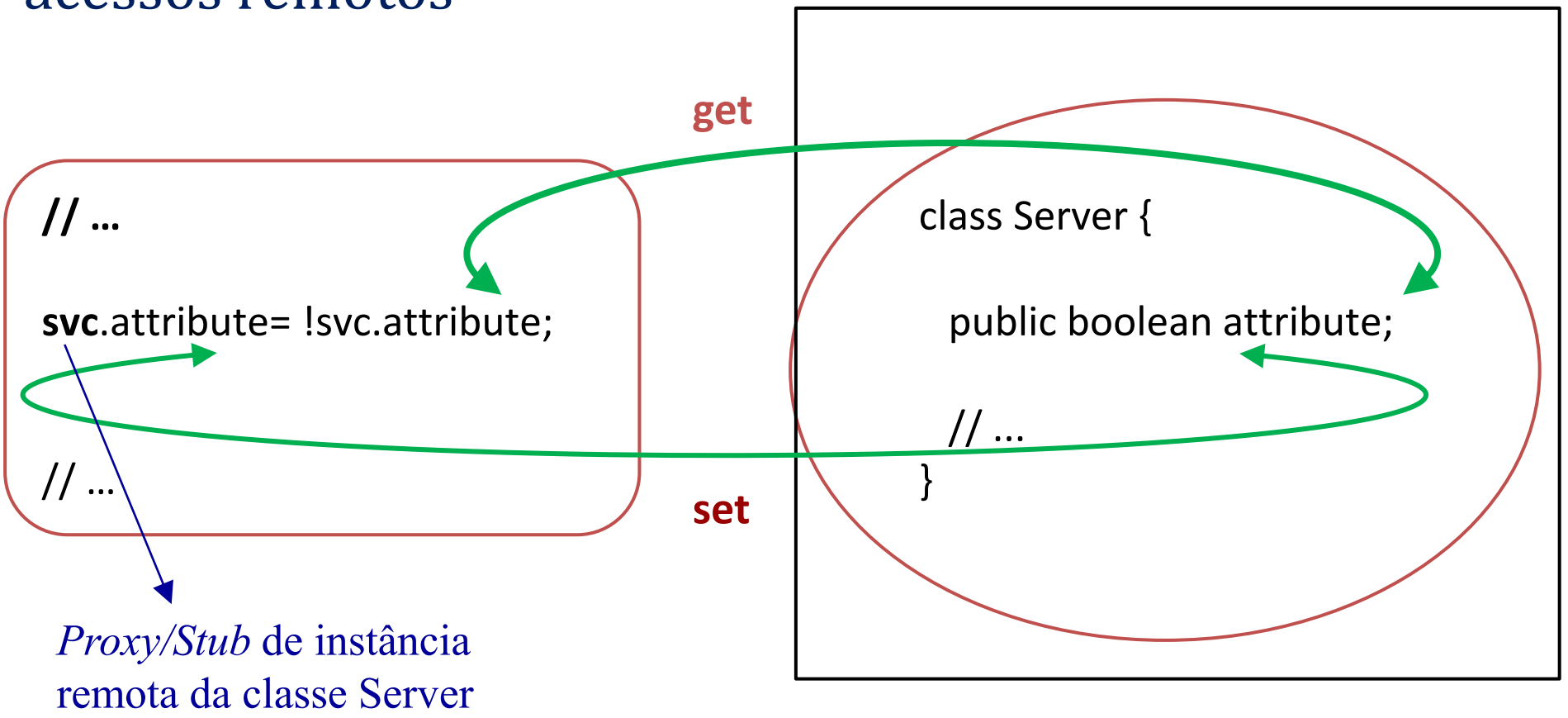
- Dependente da plataforma Java
- Protocolo de serialização proprietário e difícil de usar entre diferentes linguagens
  - Não existem compiladores para gerar *stubs/skeletons* para outras linguagens
- Ainda muito dependente do protocolo TCP/IP, nomeadamente de endereços IP e portos, nomeadamente:
  - Os *stubs* encapsulam a localização física (IP, porto) dos objetos remotos;
  - O uso de *callbacks* implica novas ligações TCP no sentido **servidor → cliente** que pode ser impossível, dada a estrutura da rede não passar firewalls, (e.g. resolução e tradução de endereços IPs (NAT));
  - Mesmo nas chamadas **cliente → servidor** podem esbarrar em regras de *firewall* por usar portos/protocolos não disponíveis ou menos escrutináveis

Algumas limitações da abordagem Java RMI

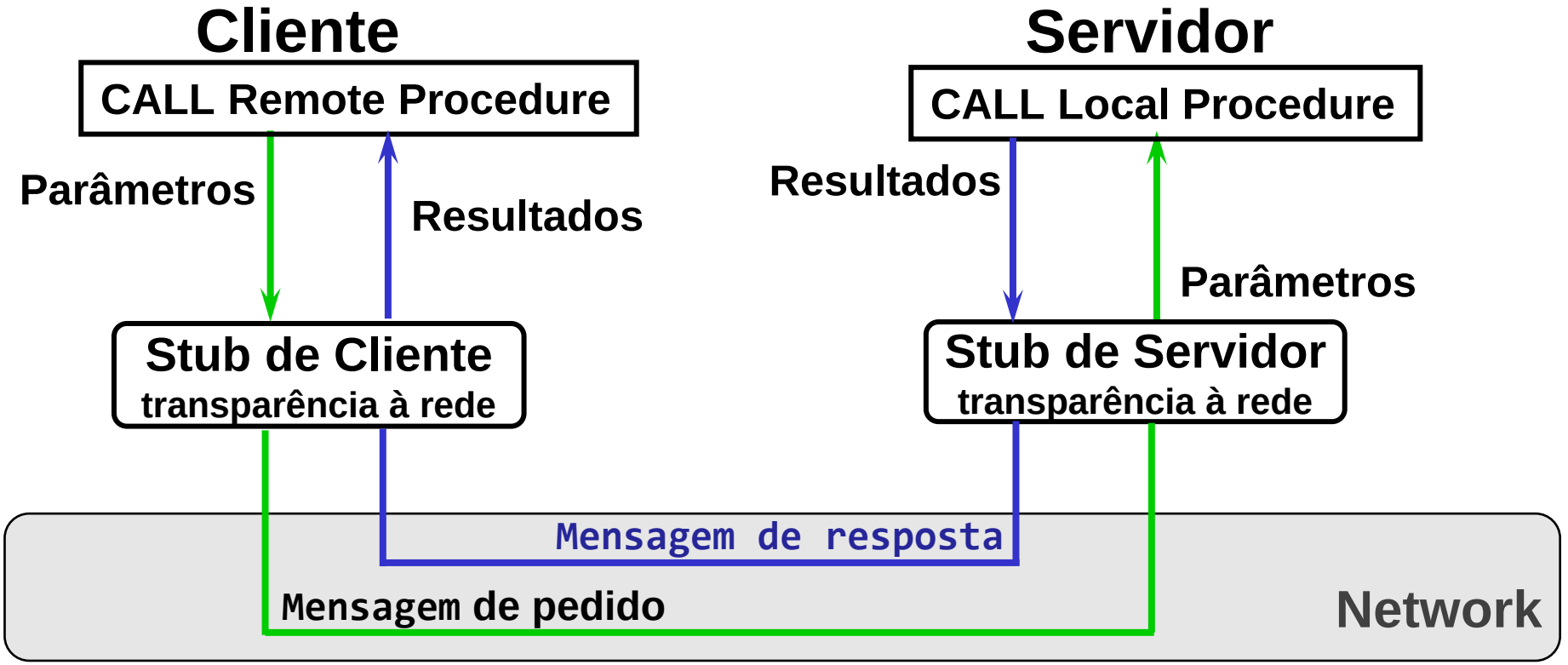
- Sem suporte intrínseco para operações assíncronas
  - Servidor e cliente podem definir abstrações para transformar chamadas síncronas em assíncronas mas tal implica repetir a abordagem em todas as aplicações
- A transparência à localização dos objetos RMI pode ser mal usada, com efeitos negativos no desempenho
  - Fazer chamadas remotas com semântica de chamadas locais induz o programador a não considerar a rede e assim não ter em conta algumas das 8 falácias da computação distribuída

Um caso inocente

Uma simples linha de código pode envolver vários acessos remotos



RPC - *Remote Procedure Call* (década 80)



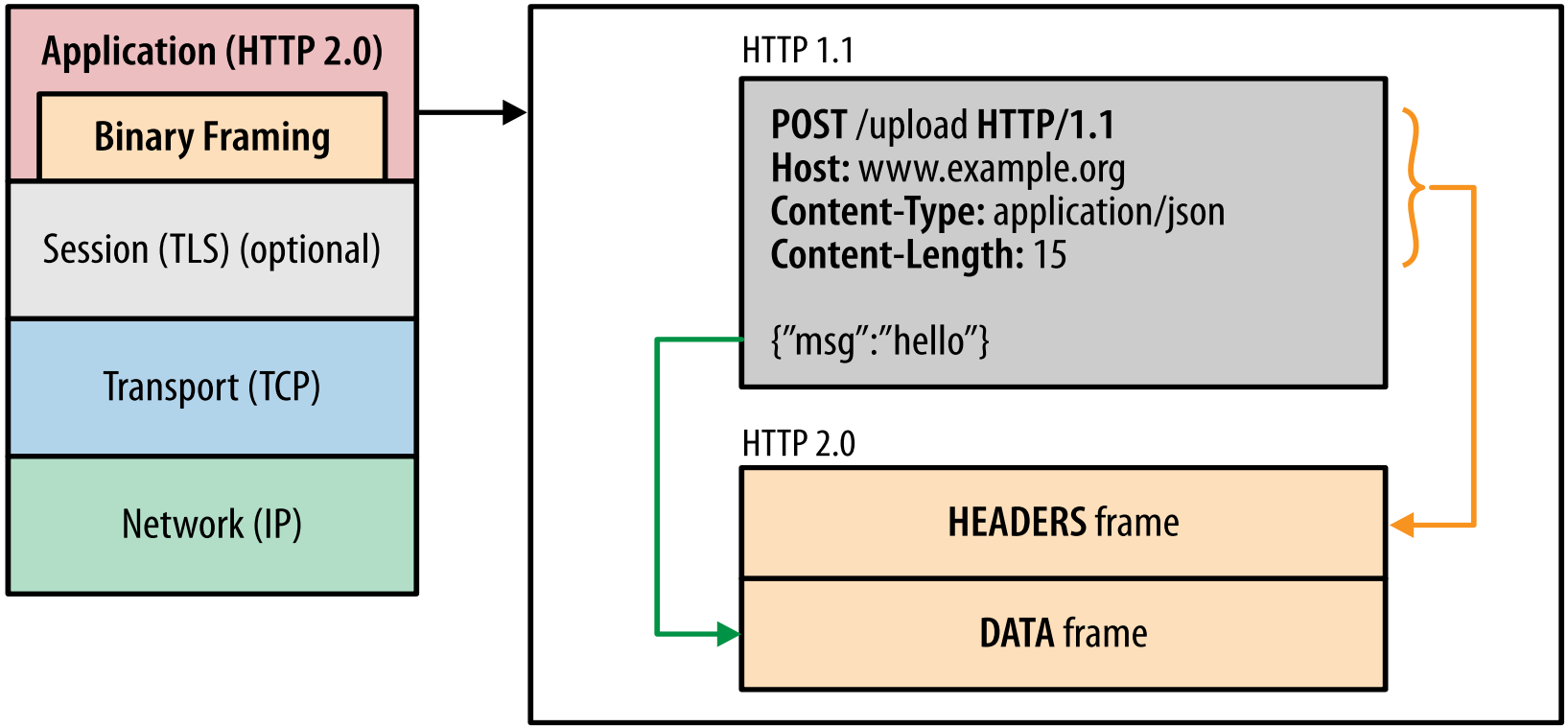
- O Cliente e o Servidor comunicam através de *Stubs*;
- Os dois *Stubs* comunicam, através da rede, usando mensagens;
- Suporte eficaz para o modelo cliente/servidor, mas com limitações para *callbacks*, invocações assíncronas e *streaming*.

Principais aspetos do *middleware* Google RPC (gRPC)

- Utilização do protocolo HTTP/2 na camada de transporte
- Uma linguagem (*protobuf*) para definição de contratos entre clientes e serviços
- O compilador do contrato gera classes *Stub* para múltiplas linguagens de programação (Java, C#, etc.)
- Diferentes modelos de interações cliente/servidor com 4 tipos de chamada:
  1. **Unária**: Semelhante à chamada de um método local. O cliente envia um pedido e recebe uma resposta;
  2. ***Streaming* do servidor**: O cliente envia um pedido e obtém uma sequência de respostas num *stream*;
  3. ***Streaming* do cliente**: O cliente envia uma sequência de pedidos num *stream* e espera por uma única resposta do servidor;
  4. ***Streaming* do cliente e do servidor**: O cliente envia uma sequência de pedidos num *stream* e o servidor responde com uma sequência de respostas noutro *stream*. O número de pedidos e de respostas pode ser diferente e arbitrário

HTTP/2

- Em HTTP/2 as tramas têm uma codificação binária eficiente, com possibilidade de compressão de *headers* e uma menor latência

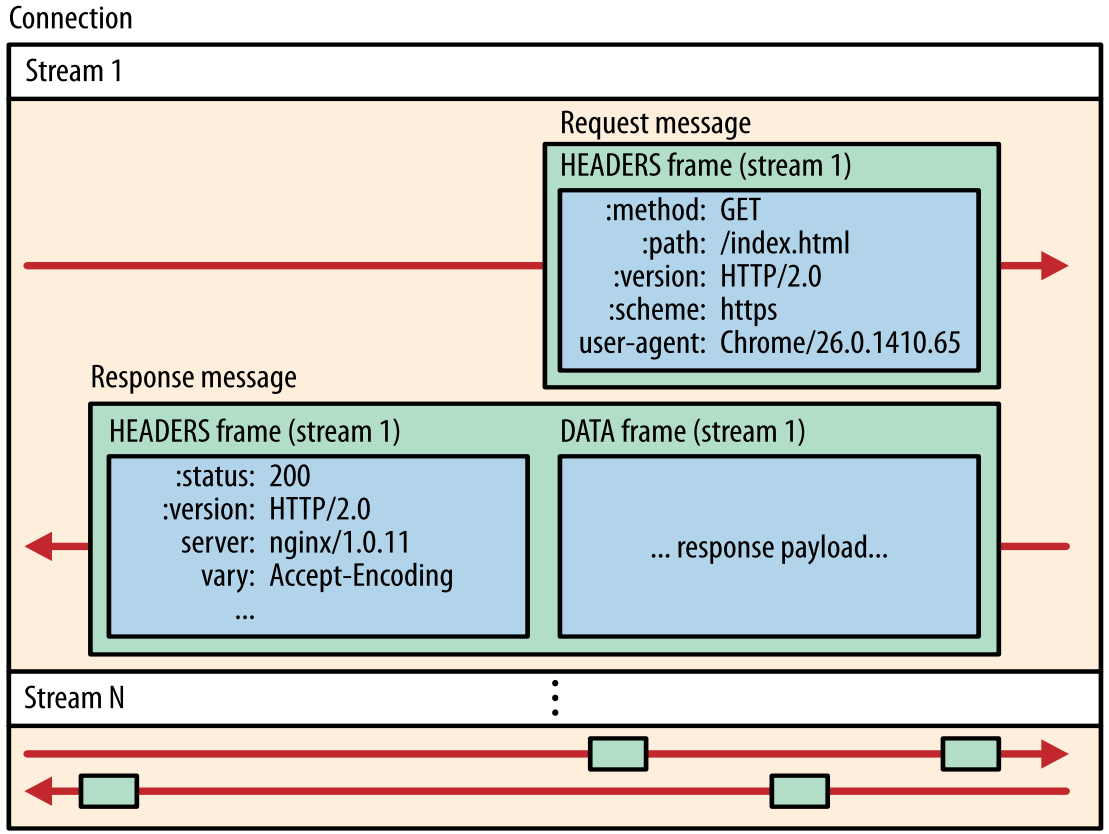


https://developers.google.com/web/fundamentals/performance/http2/

HTTP/2

- O HTTP 1.1 obriga a uma ligação TCP, por recurso, levando a um uso pouco eficiente do *stack* TCP/IP

- O HTTP/2 permite, na mesma ligação TCP, a **multiplexagem concorrente** de vários pedidos HTTP, intercalando pedidos e respostas em *streams*



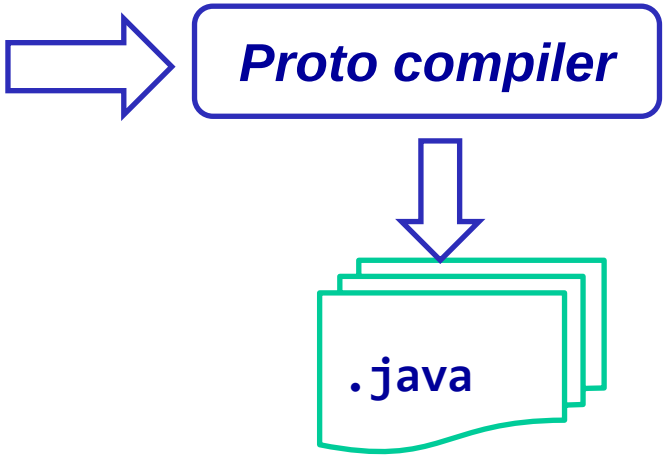
gRPC: Linguagem e compilador

- O contrato do serviço é definido de forma independente da linguagem de programação, na linguagem ***protocol buffers***
- Exemplo de serviço com uma operação *add*, parâmetro do tipo *Request* e retorno do tipo *Reply*

Calc.proto

```
service CalcService {
  // adds two numbers
  rpc add(Request) returns (Reply);
}
message Request {
  double op1 = 1; double op2 = 2;
}
message Reply {
  double res = 1;
}
```

Existem *plugins* do compilador *proto* para diferentes linguagens



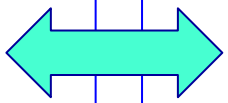
***stubs* e classes de serialização**

https://developers.google.com/protocol-buffers/docs/proto

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3 [default = 10];
}
```

```
message SearchResponse {
  repeated Result result = 1;
}

message Result {
  required string url = 1;
  optional string title = 2;
}
```



```
message SearchResponse {
  message Result {
    required string url = 1;
    optional string title = 2;
  }
  repeated Result result = 1;
}
```

```
message Project { /*. . .*/ }
```

```
message AllProjets {
   map<string, Project> projects = 1;
}
```

```
message Advice { /*. . .*/ }
```

```
message Msg  {
     string msgID = 1;
     oneof MsgOptions {
          string pid = 2;
          double tmp = 3;
          Advice adv = 4;
     }
}
```

Protocol buffers: linguagem e serialização

- Alternativas para o processo de serialização
  - *Serializable*, Ad-hoc, XML, JSON
- Protocol buffers
  - Linguagem definida no contexto da Google para língua franca na comunicação entre serviços internos e persistência

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

```
Person john = Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .build();
output = new FileOutputStream(args[0]);
john.writeTo(output);
```

- Num contrato gRPC, uma operação que recebe o conteúdo de uma sequência de bytes pode usar o tipo ***bytes*** da linguagem *protobuf*, que nos *stubs* gRPC é do tipo ***ByteString*** que pode ser iniciado a partir de um *array* de bytes como:

```
ByteString byteSeq=ByteString.copyFrom(new byte[500]);
```

O processo de desenvolvimento

```
service CalcService {
  // service operations
  rpc add(Request) returns (Reply);
}
message Request {
    double op1 = 1; double op2 = 2;
}
message Reply {
    double res = 1;
}
```

Contrato
calc.proto

ProtoBuf Compiler

Client

- Classes com *stubs* para chamada do serviço;
- Classes de serialização das mensagens

gRPC STUB

Duplex

Channel

HTTP/2

TCP/IP

Streams

gRPC STUB

Server

- Classes base para implementação do serviço;
- Classes de serialização das mensagens

```
Channel = ManagedChannelBuilder
            .forAddress(serverIP, serverPort)
// stub bloqueante
stubBlock = newBlockingStub(channel)
// stub não bloqueante
stubNoBlock = newStub(channel)
// stub não bloqueante com Futures
stubFuture = newFutureStub(channel)
```

CalcServiceGrpc.CalcServiceImplBase

CalcServer

Tipos de chamadas – 4 casos

1. **Unária**: semelhante à chamada de um método local

   rpc oper(Request) returns (Reply)

2. ***Streaming* do servidor**: O cliente envia um pedido e obtém uma sequência de respostas. É garantida a ordenação das várias respostas para cada chamada.

   rpc oper(Request) returns (**stream** Reply)

3. ***Streaming* do cliente**: O cliente envia uma sequência de mensagens de pedido num *stream* e espera por uma única mensagem de resposta do servidor. É garantida a ordenação das várias mensagens de pedido.

   rpc oper(**stream** Request) returns (Reply)

4. ***Streaming* do cliente e do servidor**: O cliente envia uma sequência de mensagens de pedido num *stream* e o servidor responde com uma sequência de mensagens de resposta noutro *stream*. Em cada *stream* é garantida a ordem das mensagens. O número de mensagens de pedido e de resposta pode ser diferente

   rpc oper(**stream** Request) returns (**stream** Reply)

Interface StreamObserver

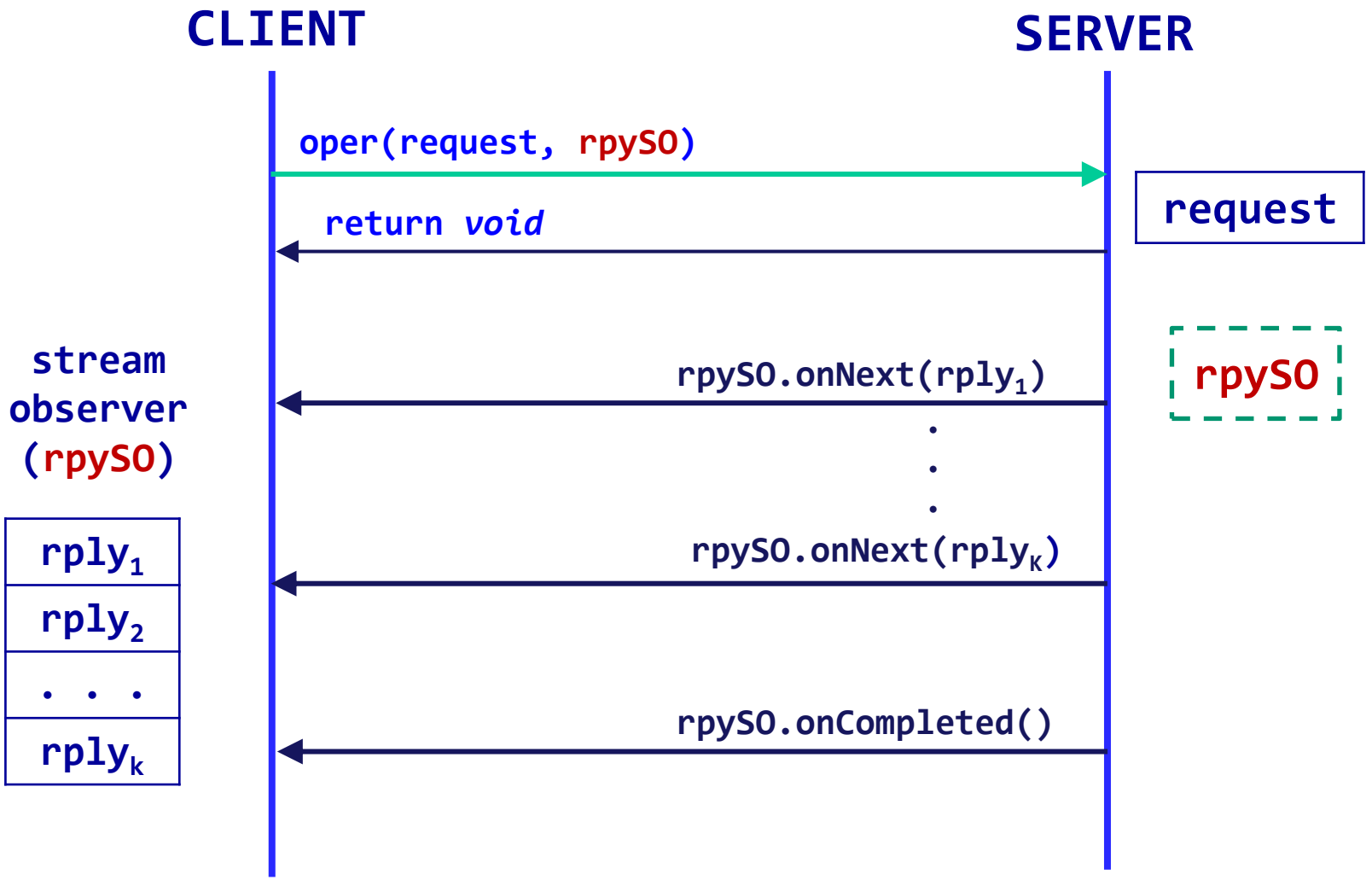
- Nos casos que envolvem *stream* de cliente e servidor utilizam-se objetos que implementam o padrão *observer* e que têm de implementar a seguinte interface:

```
public interface StreamObserver<Type> {
    void onNext(Type msg);
    void onError(Throwable msg);
    void onCompleted();
}
```

- ***OnNext***: envio de mais uma mensagem no *stream*
- ***OnError***: envio de mensagem de erro
- ***OnCompleted***: fecho do *stream*, isto é, o emissor não enviará mais mensagens, permitindo ao recetor concluir a receção de mensagens

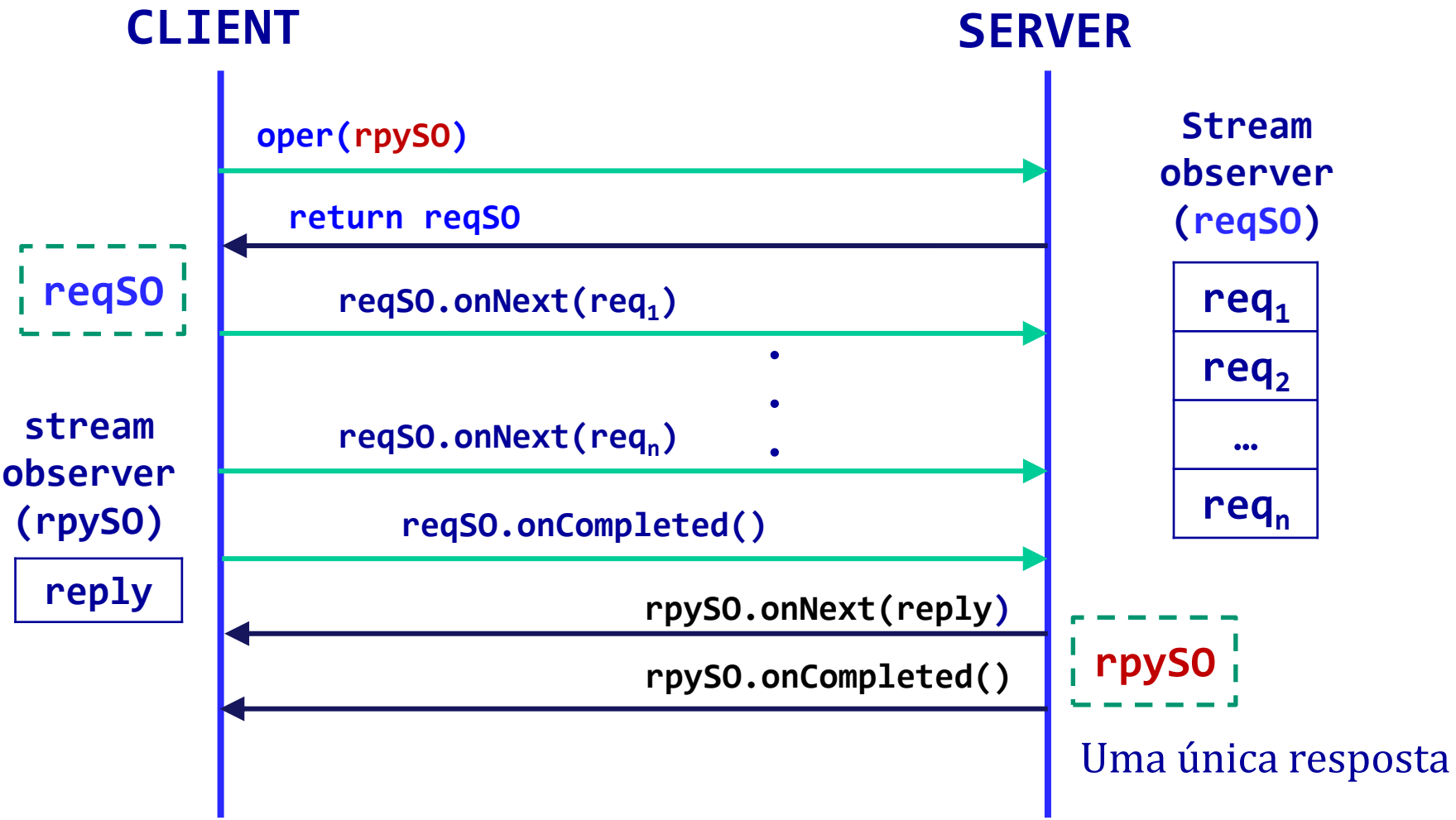
Caso 2: *stream* de servidor (cliente não bloqueante)

```
rpc oper(request) returns (stream Reply)
```



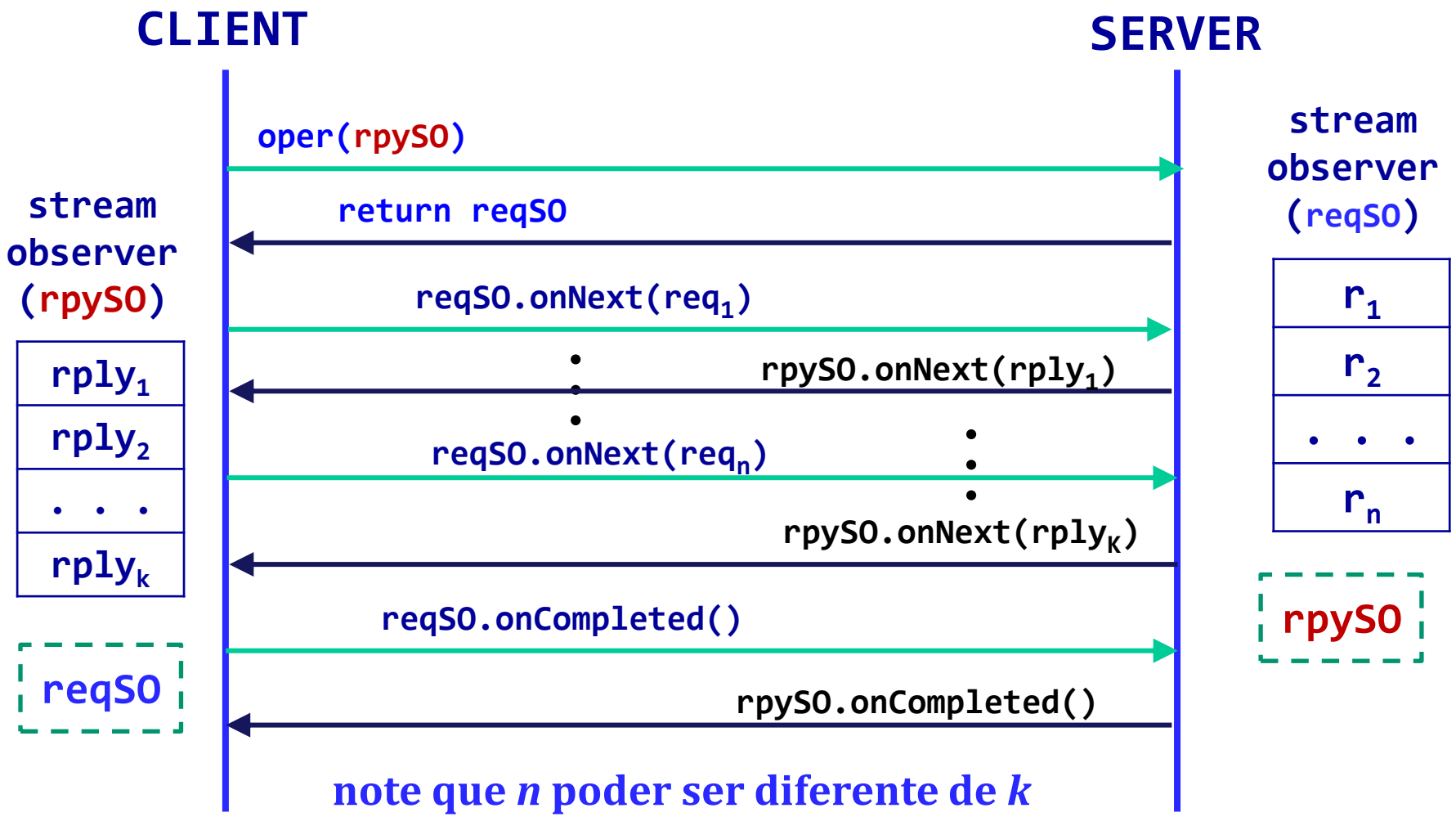
Caso 3: *stream* de cliente (cliente não bloqueante)

```
rpc oper(stream request) returns (Reply)
```



Caso 4: *stream* de cliente e *stream* de servidor

```
rpc oper(stream request) returns (stream Reply)
```



Casos 1 e 2: Implementação do Serviço

- Sintaxe na linguagem *proto*
  - Chamada unária (**caso 1**)

```
rpc oper(Request) returns (Reply)
```

  - *Streaming* do servidor (**caso 2**)

```
rpc oper(Request) returns (stream Reply)
```

- **Assinatura a implementar no serviço (caso 1 e caso 2)**

```
public void oper(
    RequestType request, StreamObserver<ReplyType> responseObserver
)
```

Independentemente do tipo de chamada, a resposta do servidor (caso 1) e as múltiplas respostas (caso 2) é sempre feita sobre um objeto ***StreamObserver<ReplyType>***

Casos 3 e 4: Implementação do Serviço

- Sintaxe na linguagem *proto*
  - *Streaming* do cliente (caso 3)

```
rpc oper(stream Request) returns (Reply)
```

  - *Streaming* do cliente e do servidor (caso 4)

```
rpc oper(stream Request) returns (stream Reply)
```

- **Assinatura a implementar no serviço (caso 3 e caso 4)**

```
public StreamObserver<RequestType> oper(
                    StreamObserver<ReplyType> responseObserver
)
```

Quando há *stream* de cliente a implementação do serviço tem de retornar um objeto *StreamObserver<RequestType>* onde o cliente escreve as várias mensagens do pedido. O servidor coloca as respostas, tal como na caso 2, também num *StreamObserver* (parâmetro *responseObserver*)

Stubs de chamada no cliente

- Os *stubs* de chamada no cliente podem ser bloqueantes, baseados em Java *futures ou* não bloqueantes
  - **Stub Bloqueante**: Só possível nos casos 1 e 2 e porque não faz sentido nos casos com *streaming* de cliente (casos 3 e 4)
    - Chamada unária (**caso 1**)

```
public ReplyType oper(RequestType request)
Chamada: ReplyType rply=blockingStub.oper(request);
```

    - Chamada com *Stream* de servidor (**caso 2**)

```
public Iterator<ReplyType> oper(RequestType request)
Chamada: Iterator<ReplyType> n_Rply=blockingStub.oper(request);
```

  - ***Stub Future***: (*com.google.common.util.concurrent.ListenableFuture<T>*)
    - Só possível no caso 1 por não fazer sentido nos casos de *streaming*

```
public ListenableFuture<ReplyType> oper(RequestType request)
Chamada: ListenableFuture<ReplyType> fut = futStub.oper(request);
```

Revisão do Conceito de Java Future

Um Future representa um resultado de um computação assíncrona, ou seja, uma computação que pode já ter terminado ou não.

```
public interface Future<T> {
    boolean cancel(boolean mayInterruptIfRunning)
    T  get();
    T  get(long timeout, TimeUnit unit);
    boolean isCancelled();
    boolean isDone();
}
```

```
Future<SomeType> future = ... // get Future by starting async task

// do something else, until ready to check result via Future

// get result from Future
try {
    SomeType result = future.get();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

Stubs de chamada no cliente (não bloqueante)

- **Não bloqueante (assíncrono):** a assinatura dos métodos de chamada é igual à assinatura dos métodos de implementação no serviço
  - Casos 1 e 2

```
public void oper(
    RequestType request, StreamObserver<ReplyType> responseObserver
)
Chamada:  noBlockStub.oper(request, rpyStreamObs);
```

  - Casos 3 e 4

```
public StreamObserver<RequestType> oper(
                  StreamObserver<ReplyType> responseObserver
)
Chamada:
 StreamObserver<RequestType> n_Reqs = noBlockStub.oper(rpyStreamObs);
```

Resumo com matriz de casos

Caso \ Stub	Bloqueante	Não bloqueante (assíncrono)	Future (assíncrono)
Caso 1 **Call Unária** **1 Request** **1 Reply**	**1 Request** **1 Reply** ***rpy=case1(req)***	1 Request 1 Reply, obtido no ***StreamObserver rpyStreamObs.*** ***case1(req, rpyStreamObs);***	**1 Request** **1 Reply, obtido por** ***ListenableFuture<Reply>*** ***future=case1(req)*** **Após future *isDone()*,** **fazer *get()* do Reply**
Caso 2 **Streaming Servidor** **1 Request** **1..N Reply**	**1 Request** **1..N Reply,** **obtidos por** ***Iterator<Reply>*** ***nRpys=case2(req)***	1 Request 1..N Reply, obtidos no ***StreamObserver rpyStreamObs.*** ***case2(req, rpyStreamObs);***	Não suportado (não aplicável)
Caso 3 **Streaming Cliente** **1..N Request** **1 Reply**	Não suportado (não aplicável)	***StreamObserver<Request>*** ***requests=case3(rpyStreamObs)*** **Retorna um *StreamObserver*** *onde se* colocam *os* N requests e obtém 1 **Reply** por ***StreamObserver rpyStreamObs***	Não suportado (não aplicável)
Caso 4 **Streaming Cliente e Servidor** **1..N Request** **1..N Reply**	Não suportado (não aplicável)	***StreamObserver<Request>*** ***requests=case4(rpyStreamObs)*** Retorna um ***StreamObserver*** onde se colocam os N requests e obtém N **Reply** por **StreamObserver rpyStreamObs**	Não suportado (não aplicável)

Exceções nos *StreamObserver - onError(Throwable msg);*

- A assinatura do método *OnError* dos *Stream Observers* recebe como argumento um objeto de uma classe *Throwable*
  - Se na chamada *OnError*, fizermos *`new Throwable("mensagem de erro")`* essa mensagem não chega ao destino e aparece a mensagem de status por omissão *UNKNOWN*.
- A biblioteca gRPC tem classes especificas para passar exceções (*StatusException*; *StatusRuntimeException*) e a classe *Status* para tipificar os erros:
  - https://grpc.github.io/grpc-java/javadoc/
- Por exemplo, considere uma operação *findPrimes* em que um *request* tem um intervalo mal definido (endNumber < startNumber)

```
void findPrimes(PrimesInterval request, StreamObserver<Prime> responseObserver)
```

- Podemos gerar um erro de intervalo inválido

```
if (request.getEndNum() < request.getStartNum()) {
   Throwable th=new StatusException(Status.INVALID_ARGUMENT.withDescription("Invalid Interval"));
   responseObserver.onError(th);
   return;
 }
```

- No lado do cliente (implementação do *stream observer*) já recebe uma mensagem completa "INVALID_ARGUMENT: Invalid Interval"

```
public void onError(Throwable throwable) { System.out.println(throwable.getMessage()); }
```

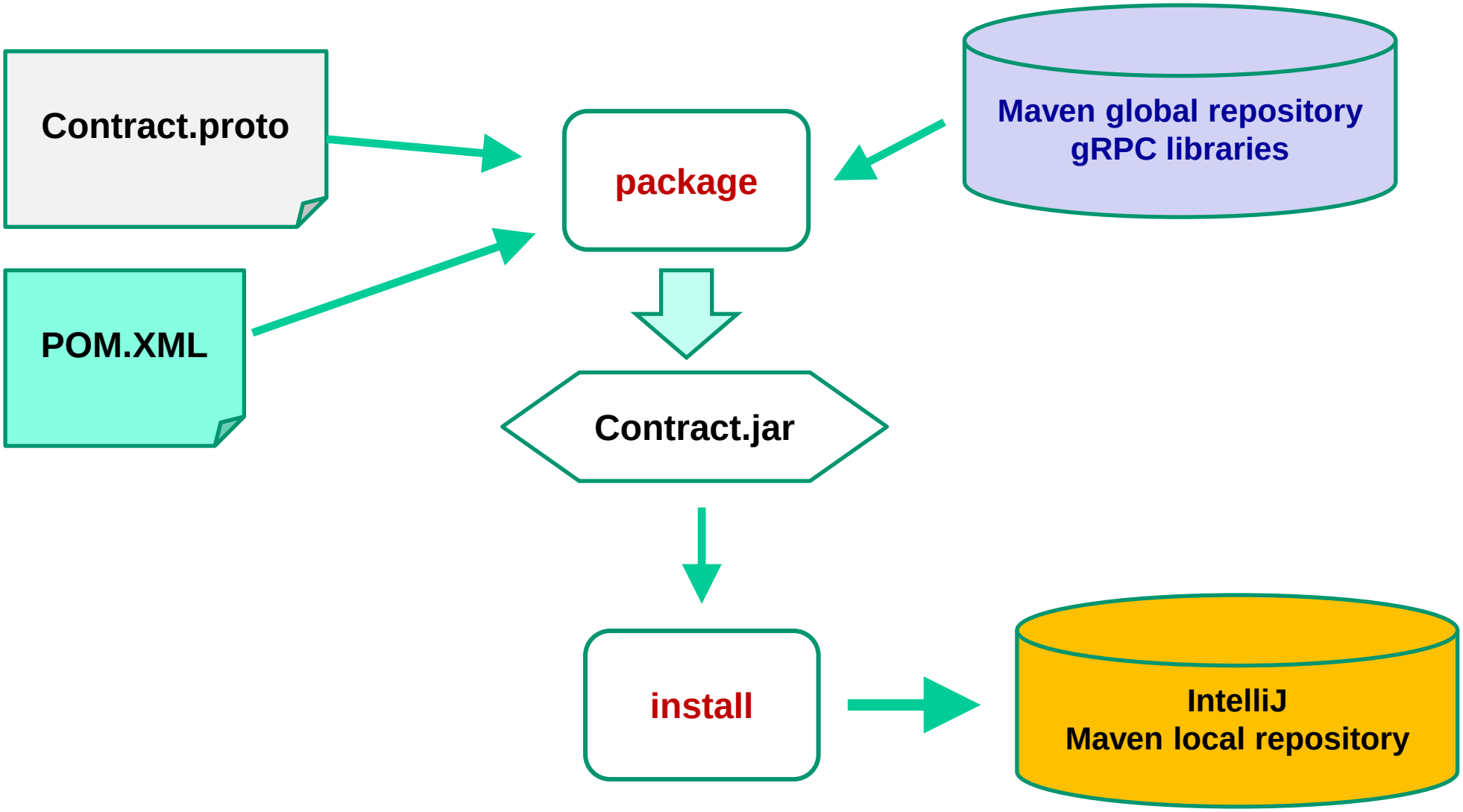
Conclusões

- Interações transparentes entre clientes e servidores com grande flexibilidade e baixa latência tirando partido do protocolo HTTP/2 , nomeadamente *streaming*:
  - Definição de contratos de forma sucinta
  - Canais duplex com suporte de *streaming*
  - Chamadas síncronas e assíncronas
- Suporte para múltiplas linguagens: C++, Java (suporte para Android), Objective-C (iOS), Python, Ruby, Go, C#, Node.js, etc.
- Usado em vários produtos e APIs da Google Cloud e também por muitas outras organizações, tais como:
  - Square, Netflix, CoreOS, Docker, Cockroachdb, Cisco, etc.
- Potencial para suprir algumas desvantagens do protocolo HTTP 1.1, nomeadamente na implementação de REST APIs ou arquiteturas de micro serviços: mais liberdade do que o modelo *request-response* de arquiteturas REST (HTTP + JSON)

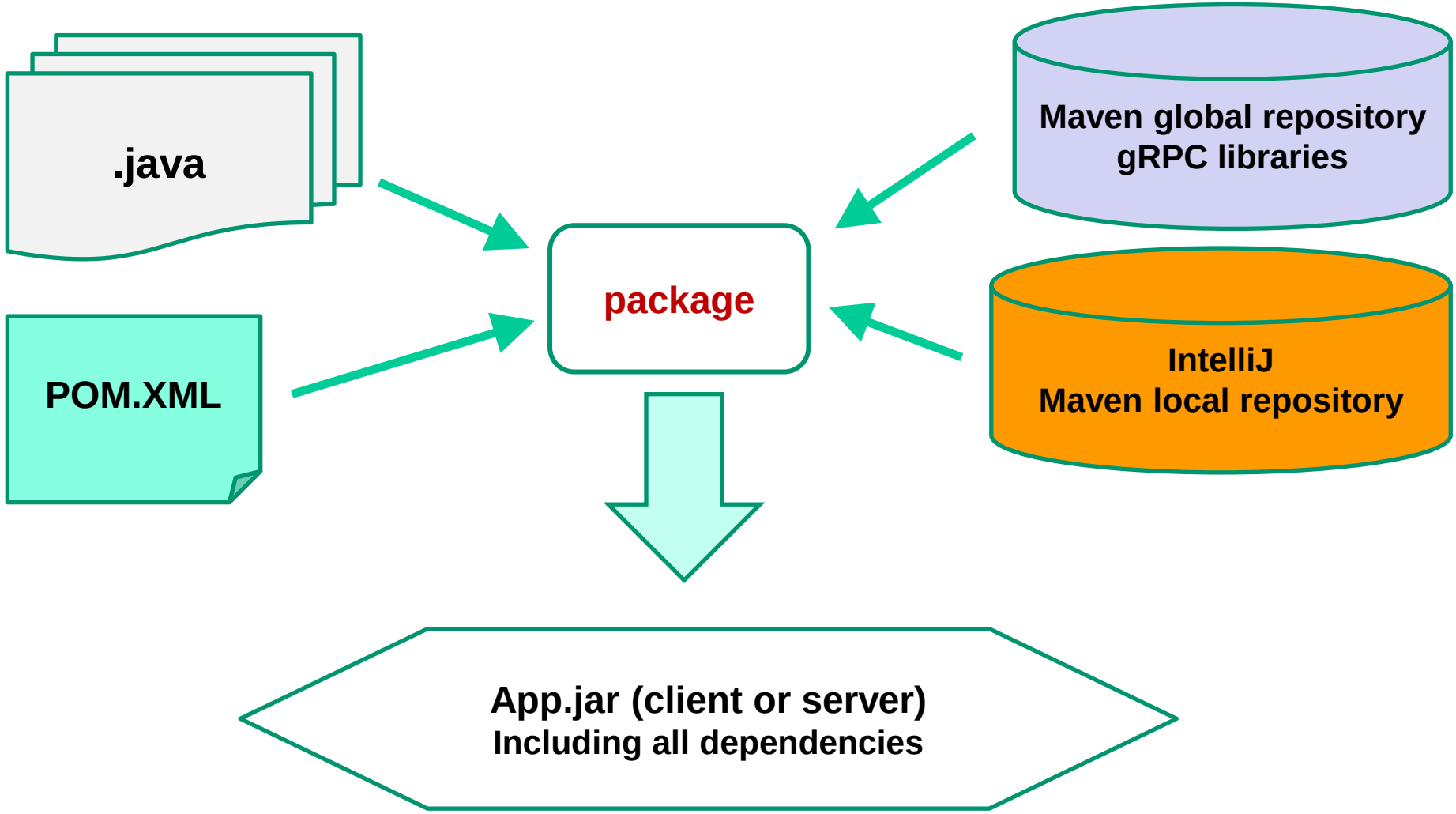
Exemplo base

(demonstração dos 4 casos de operações gRPC)

Desenvolvimento gRPC com projetos Maven e IntelliJ: **Contrato**



Desenvolvimento gRPC com projetos Maven e IntelliJ: Client e Server



Exemplo base: Contrato

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";
option java_multiple_files = true;
option java_package = "rpcstubs";

package baseservice; // package do proto

// Os 4 casos de definição de serviço
service BaseService {
  rpc case1(Request) returns (Reply);
  rpc case2 (Request) returns (stream Reply);
  rpc case3(stream Request) returns (Reply);
  rpc case4(stream Request) returns (stream Reply);

  // Utilização de parâmetro e retorno Void
  rpc pingServer(Void) returns (Reply);
  rpc publishNews(News) returns (Void);
}
```

```
message Request {
    int32 reqID = 1;
    string txt = 2;
}
message Reply {
    int32 rplyID = 1;
    string txt = 2;
}
```

```
message Void {

}

message News {
   google.protobuf.Timestamp ts = 1;
   string texto = 2;
}
```

Exemplo base: Implementação do serviço

```
public class Server extends BaseServiceGrpc.BaseServiceImplBase {

    private static int svcPort=8000;
    public static void main(String[] args) {
       try {
          io.grpc.Server svc=ServerBuilder.forPort(svcPort).addService(new Server()).build();
          svc.start();
             System.out.println("Server started, listening on " + svcPort);
             Scanner scan=new Scanner(System.in); scan.nextLine();
          svc.shutdown();
        } catch (Exception ex) { ex.printStackTrace(); }
    }

    @Override
    public void case1(Request request, StreamObserver<Reply> responseObserver) { ... }
    @Override
    public void case2(Request request, StreamObserver<Reply> responseObserver) { ... }
    @Override
    public StreamObserver<Request> case3(StreamObserver<Reply> responseObserver) { ... }
    @Override
    public StreamObserver<Request> case4(StreamObserver<Reply> responseObserver) { ... }
    @Override
    public void pingServer(Void request, StreamObserver<Reply> responseObserver) { ... }
    @Override
    public void publishNews(New request, StreamObserver<Void> responseObserver) { ... }
}
```

Exemplo base: Implementação do Cliente

```
public class Client {

    private static String svcIP = "localhost";   private static int svcPort = 8000;
    private static ManagedChannel channel;
    private static BaseServiceGrpc.BaseServiceBlockingStub blockingStub;
    private static BaseServiceGrpc.BaseServiceStub noBlockStub;
    private static BaseServiceGrpc.BaseServiceFutureStub futStub;

    static void case1() {  }
    static void case2() {  }
    static void case3() {  }
    static void case4() {  }
    static void pingServer() {  }
    static void publishNews() {  }


    public static void main(String[] args) {
         // Instanciar canal para conexão ao serviço
         // Instanciar Stubs
         // Chamar as operações do serviço
    }
}
```

Exemplo base cliente: Chamar as operações

```
public static void main(String[] args) {
  try {
      channel = ManagedChannelBuilder.forAddress(svcIP, svcPort)
                // Channels are secure by default (via SSL/TLS).
                // For the example we disable TLS to avoid needing certificates.
                .usePlaintext()
                .build();
      blockingStub = BaseServiceGrpc.newBlockingStub(channel);
      noBlockStub = BaseServiceGrpc.newStub(channel);
      futStub = BaseServiceGrpc.newFutureStub(channel);
      // ping Server com argumento Void
      Reply ping = blockingStub.pingServer(Void.newBuilder().build());
      System.out.println(ping.getTxt());
      // invocar as operações disponibilizadas pelo serviço
      case1();   // unário
      case2();   // stream servidor
      case3();   // stream cliente
      case4();   // stream cliente e stream de servidor
      // operação com retorno Void
      long millis = System.currentTimeMillis();
      Timestamp ts = Timestamp.newBuilder().setSeconds(millis / 1000).build();
      blockingStub.publishNews(
              News.newBuilder().setTs(ts).setTexto("tempo chuvoso").build()
      );
  } catch (Exception ex) {  ex.printStackTrace();  }
```

Servidor: Operação *case1*

```
//Implementação no servidor da operação case1
@Override
public void case1(Request request, StreamObserver<Reply> responseObserver) {
    System.out.println("case1 called");
    Reply rply = Reply.newBuilder()
                      .setRplyID(request.getReqID())
                      .setTxt(request.getTxt().toUpperCase()).build();
    responseObserver.onNext(rply);
    responseObserver.onCompleted();
}
```

```
//A mplementação da operação pingServer é idêntica à operação case1
@Override
public void pingServer(Void request, StreamObserver<Reply> responseObserver) {
    System.out.println("pingServer called");
    Reply rply = Reply.newBuilder()
                     .setRplyID(0).setTxt("Server is alive").build();
    responseObserver.onNext(rply);
    responseObserver.onCompleted();
}
```

Cliente: Chamada da operação *case1*

```
// Chamada síncrona da operação case1 com blocking stub
for (int i=0; i < 3;i++) {
   Request req = Request.newBuilder().setReqID(i).setTxt("request " + i).build();
   Reply rply = blockingStub.case1(req);
   System.out.println("Reply(" + rply.getRplyID() + "):" + rply.getTxt());
}
```

Cliente: Chamada da operação *case1*

```
// Chamada assíncrona com non blocking stub
Request req = Request.newBuilder()
                          .setReqID(100)
                          .setTxt("request assincrono ")
                          .build();
ClientStreamObserver replyStreamObserver = new ClientStreamObserver();
noBlockStub.case1(req, replyStreamObserver);
while (!replyStreamObserver.isCompleted()) {
   System.out.println("cliente active");
   Thread.sleep(1*1000);
}
List<Reply> replies=replyStreamObserver.getReplays();
if (replyStreamObserver.OnSuccesss()) {
   for (Reply rpy : replyStreamObserver.getReplays()) {
       System.out.println("Reply for Case1:"+rpy.getRplyID()+":"+rpy.getTxt());
   }
}
```

Nas chamadas assíncronas as respostas são recebidas, por iniciativa do *middleware*, através dos métodos *onNext*(...), *onCompleted*(...) ou *onError*(...) de um objeto *StreamObserver*

Cliente: *StreamObserver<Reply>*

```
public class ClientStreamObserver implements StreamObserver<Reply> {
    private boolean isCompleted=false; private boolean success=false;
    public boolean OnSuccesss() { return success;  }
    public boolean isCompleted() {  return isCompleted;  }
    List<Reply> rplys = new ArrayList<Reply>();
    public List<Reply> getReplays() {  return rplys; }
    @Override
    public void onNext(Reply reply) {
        System.out.println("Reply ("+reply.getRplyID()+"):"+reply.getTxt());
        rplys.add(reply);
    }
    @Override
    public void onError(Throwable throwable) {
        System.out.println("Error on call:"+throwable.getMessage());
        isCompleted=true; success=false;
    }
    @Override
    public void onCompleted() {
        System.out.println("Stream completed");
        isCompleted=true; success=true;
    }
}
```

Cliente: Chamada da operação *case1*

```
// Chamada assíncrona com future
System.out.println("Case1 with Future");
Reply frpy = null;
try {
    Request futRequest=Request.newBuilder()
                              .setReqID(200)
                              .setTxt("invoked with future")
                              .build();
    ListenableFuture<Reply> fut=futStub.case1(futRequest);
    while (!fut.isDone()) {
        System.out.println("waiting futures completed");
        Thread.sleep(1 * 1000);
    }
    frpy = fut.get();
} catch (Exception e) {
    e.printStackTrace();
}
System.out.println("Future reply:RES="+frpy.getTxt());
```

Server: Operação *case2*

```
// Implementação no servidor stream servidor
@Override
public void case2(Request request, StreamObserver<Reply> responseObserver) {
    System.out.println("case2 called: Multiplas respostas função do request ID");
    for (int i=0; i < request.getReqID(); i++) {
        Reply rply = Reply.newBuilder()
                          .setRplyID(request.getReqID())
                          .setTxt(request.getTxt().toUpperCase()+" "+i)
                          .build();
        responseObserver.onNext(rply);
    }
    responseObserver.onCompleted();
}
```

Cliente: Chamada da operação *case2 (síncrona e assíncrona)*

```
// chamada síncrona da operação case2 (stream server) com blocking stub
Request req = Request.newBuilder().setReqID(5).setTxt("request case2").build();
Iterator<Reply> manyRpys = blockingStub.case2(req);
while (manyRpys.hasNext()) {
    Reply rpy = manyRpys.next();
    System.out.println("Reply BlockStub:"+rpy.getRplyID()+":"+rpy.getTxt());
}

// chamada assíncrona da operação case2 (stream server) com non blocking stub
ClientStreamObserver rpyStreamObs = new ClientStreamObserver();
noBlockStub.case2(req, rpyStreamObs);
while (!rpyStreamObs.isCompleted()) {
    System.out.println("Active and waiting for Case2 completed ");
    Thread.sleep(1 * 1000);
}
if (rpyStreamObs.OnSuccesss()) {
  for (Reply rpy : rpyStreamObs.getReplays()) {
     System.out.println("Reply non BlockStub:"+rpy.getRplyID()+":"+rpy.getTxt());
  }
}
```

Server: Operação *case3*

```
//Implementação no servidor stream de cliente
@Override
public StreamObserver<Request> case3(StreamObserver<Reply> responseObserver) {
   System.out.println("case3 called");
   ServerStreamObserverC3 reqs = new ServerStreamObserverC3(responseObserver);
   return reqs;
}
```

Nas operações com stream de cliente o servidor devolve um objeto *StreamObserver* para o cliente colocar *requests* através do método *onNext*(...). Quando o cliente invocar o método *onCompleted*(...) ou *onError*(...), é processada uma resposta e enviada através do método *onNext*(...) do objeto *responseObserver*.

Server: *StreamObserver<Request> case3*

```
public class ServerStreamObserverC3 implements StreamObserver<Request> {
    StreamObserver<Reply> sFinalreply;  String finalText="";

    public ServerStreamObserverC3(StreamObserver<Reply> sreplies) {
        this.sFinalreply=sreplies;
    }
    @Override
    public void onNext(Request request) {
        // More one request
        finalText += request.getTxt() + ":";
    }
    @Override
    public void onError(Throwable throwable) { . . . }
    @Override
    public void onCompleted() {
        Reply rply = Reply.newBuilder()
                    .setRplyID(9999)
                    .setTxt(finalText.toUpperCase()).build();
        sFinalreply.onNext(rply);
        sFinalreply.onCompleted();
    }
}
```

Recebe mútiplos pedidos no método *onNext*(...) e produz uma única resposta quando o cliente chamar o método *OnCompleted*(...)

Server: Operação *case4*

```
//Implementação no servidor stream de cliente e stream de servidor
@Override
public StreamObserver<Request> case4(StreamObserver<Reply> responseObserver) {
   System.out.println("case4 called");
   ServerStreamObserverC4 reqs = new ServerStreamObserverC4(responseObserver);
   return reqs;
}
```

Nas operações com *stream* de cliente o servidor devolve um objeto *StreamObserver* para o cliente colocar *requests a*través do método *onNext(…),* que o servidor processa produzindo múltiplas respostas no método onNext(…) do objeto *responseObserver*. Quando o cliente invocar o método *onCompleted(…)* ou *onError*(…), o servidor termina o processamento e invoca o método *onCompleted(…)* do objeto *responseObserver*.

Server: *StreamObserver<Request> case4*

```
public class ServerStreamObserverC4 implements StreamObserver<Request> {
    StreamObserver<Reply> sreplies;

    public ServerStreamObserverC4(StreamObserver<Reply> sreplies) {
        this.sreplies=sreplies;
    }
    @Override
    public void onNext(Request request) {
        // More one request to process and one more reply
        Reply rply = Reply.newBuilder().setRplyID(request.getReqID())
                                  .setTxt(request.getTxt().toUpperCase()).build();
        sreplies.onNext(rply);
        // pode armazenar múltiplos pedidos e só responder em OnCompleted
    }
    @Override
    public void onError(Throwable throwable) {    }
    @Override
    public void onCompleted() {
        // processar eventuais mensagens de pedido recebidas em OnNext
        // e responder com uma ou mais respostas em onNext(…)
        sreplies.onCompleted();
    }
}
```

Server: Operação *publishNews*

```
//A mplementação da operação publishNews é idêntica à operação case1,
// devolvendo uma resposta Void
@Override
public void publishNews(News request, StreamObserver<Void> responseObserver) {
    System.out.println("publishNews called");
    Timestamp ts = request.getTs();
    System.out.println("News timestamp:“ + ts.getSeconds());

    // De acordo com a lógica de aplicação o servidor poderia armazenar
    // as notícias num repositório

    responseObserver.onNext(Void.newBuilder().build());
    responseObserver.onCompleted();
}
```

Dependências e compilação

- Os projetos gRPC em Java (Contrato, Serviço e Cliente) têm as seguintes dependências (no idioma *Project Object Model* – POM, do Maven)

```
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-protobuf</artifactId>
    <version>1.45.1</version>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-stub</artifactId>
    <version>1.45.1</version>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty-shaded</artifactId>
    <version>1.45.1</version>
</dependency>
```

Protocol buffers

gRPC

Transporte http/2

Dependências e compilação do contrato (.proto)

- A geração dos *stubs* a partir da descrição dos contratos é feita pelo compilador de *proto buffer*
  - O compilador tem um arquitetura de *plugins* onde para cada linguagem há um *plugin* que gera os respetivos artefactos *stub*
- No slide seguinte encontra um excerto de exemplo de configuração do *plugin* para geração de *stubs* Java em projeto Maven:
- Para projetos futuros deve usar, como referência, os ficheiros POM.XML dos 3 projetos em anexo (*05.gRPCBaseProjectCompleto.zip*)

Dependência adicional e compilação do contrato

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

```
...
<plugin>
    <groupId>org.xolstice.maven.plugins</groupId>
    <artifactId>protobuf-maven-plugin</artifactId>
    <version>0.6.1</version>
    <configuration>
        <protocArtifact>com.google.protobuf:protoc:3.5.1-1:exe:
                        ${os.detected.classifier}
        </protocArtifact>
        <pluginId>grpc-java</pluginId>
        <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.18.0:exe:
                        ${os.detected.classifier}
        </pluginArtifact>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>compile</goal>
```