Esercitazione di Sistemi Distribuiti e Pervasivi

Comunicazione tra processi

Gabriele Civitarese
gabriele.civitarese@unimi.it

EveryWare Lab
Università degli Studi di Milano
-
Docente: Claudio Bettini

Slide realizzate a partire da versioni precedenti di Letizia Bertolaja, Sergio Mascetti, Dario Freni e Claudio Bettini

Copyright

Alcune slide di questo corso sono in parte personalizzate da quelle fornite dall'autore del testo di riferimento e distribuite dalla casa editrice (Distributed Systems: Principles and Paradigms, A. S. Tanenbaum, M. Van Steen, Prentice Hall, 2007). Tutte le altre slide sono invece di proprietá del docente. Tutte le slide sono soggette a diritto d'autore e quindi non possono essere ri-distribuite senza consenso. Lo stesso vale per eventuali registrazioni o videoregistrazioni delle lezioni.

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). All the other slides are from the teacher of this course. All the material is subject to copyright and cannot be redistributed without consent of the copyright holder. The same holds for audio and video-recordings of the classes of this course.

Machine to machine communication

Comunicazione tra processi

- In un sistema distribuito, processi su diverse macchine (connesse da una rete) necessitano di comunicare
- La comunicazione avviene tramite scambio di **messaggi**
- La coordinazione tra processi in rete consiste quindi in sole due operazioni:
  - Trasmissione di messaggi
  - Ricezione di messaggi
- I messaggi vengono scambiati tramite socket

Come rappresentiamo i messaggi?

- A livello applicativo i dati sono memorizzati in opportune strutture dati
- A livello di rete sono comunicati come stream di byte
- Problema: i processi possono essere eterogenei e memorizzare in modo diversi i dati. Ad esempio:
  - Big endian VS Little endian
  - Diverse rappresentazioni dei float
  - ASCII vs Unicode
  - ...
- È quindi necessario concordare un formato per rappresentare i dati

Marshalling e unmarshalling

Trasmissione dei messaggi

- Il trasmettitore effettua il **marshalling**: i dati da trasmettere vengono assemblati in un formato adatto per la trasmissione
- Il ricevente effettua l'**unmarshalling**: i dati ricevuti vengono tradotti in opportune strutture dati
- Uso di formato di dati esterno concordato per garantire l'interoperabilità
- Vedremo tre diversi formati:
  - XML
  - JSON
  - Protocol Buffers

XML (eXtensible Markup Language)

- É uno standard di memorizzazione delle informazioni
- Linguaggio di markup generalizzato (è possibile creare vari tipi di dialetti)
- Human & machine readable
- Presenti svariati strumenti per la validazione (XMLSchema) e per le interrogazioni (XPath/XQuery)

XML, un esempio

```
<researcher>
  <name>Gabriele</name>
  <surname>Civitarese</surname>
  <publishedPapers>
     <paper>
        <title>activity recognition in smart homes</title>
        <year>2014</year>
      </paper>
      <paper>
        <title>activity recognition again</title>
        <year>2015</year>
     </paper>
   </publishedPapers>
</researcher>
```

JSON (JavaScript Object Notation)

- Standard più recente
- Nato per Javascript, ma ne è indipendente
- Human & machine readable
- Piú compatto e leggibile di XML
- Offre la possibilità di descrivere array
- La trasmissione è quindi più rapida a parità di contenuto informativo
- Ha ormai soppiantato XML per la comunicazione in rete!

JSON, un esempio

```
{
  name : Gabriele,
  surname : Civitarese,
  publishedPapers : [
    {title : activity recognition in smart homes , year : 2014},
    {title : activity recognition again , year : 2015}
   ]
}
```

Vedi http://json.org/ per la definizione esatta degli oggetti JSON

JSON: esempio di codifica

```
public class Researcher {
...
  public String toJSONString() throws JSONException {
    JSONObject researcher = new JSONObject();
    researcher.put("name", this.name);
    researcher.put("surname", this.surname);
    JSONArray publications = new JSONArray()
    for (Paper p : this.publishedPapers) {
      JSONObject publication = new JSONObject();
      publication.put("title", p.getTitle());
      publication.put("year", p.getYear());
      publications.put(publication);
      }
    researcher.put("publishedPapers", publications);
    return researcher.toString()
  }
 ...
```

JSON: esempio di decodifica

```
public class Researcher {
...
  public Researcher(String jsonString) throws JSONException {
    JSONObject input = new JSONObject(jsonString);
    this.setName(input.getString("name"));
    this.setSurname(input.getString("surname"));
    JSONArray array = input.getJSONArray("publishedPapers");
    for (int i=0; i<array.length(); i++){
      JSONObject current = array.getJSONObject(i);
      String title = current.getString("title");
      int year = current.getInt("year");
      Paper p = new Paper(title, year);
      this.publishedPapers.add(p);
  }
...
}
```

Gson

- Libreria di Google usata per convertire oggetti Java in rappresentazione JSON e viceversa
- Funziona con qualsiasi tipo di oggetto Java pre-esistente (anche oggetti dei quali non si ha il codice)
- Nessun utilizzo di annotazioni e supporto di tipi generici
- Semplice da utilizzare:
  - metodo *toJson*() per passare da un oggetto ad una stringa JSON
  - medoto *fromJson*() per passare da una stringa Json ad un'istanza di un oggetto

Per maggiori info...

`https://code.google.com/p/google-gson/`

Come trasmettiamo i dati veri e propri?

- I dati sono rappresentati nella loro codifica testuale
  - Nessun problema per stringhe, numeri e boolean
- Bisogna trovare una codifica apposita per dati binari. Es di codifica: Base64
  - Si usa per trasformare una stringa di bit in una stringa di caratteri ASCII
  - Idea:
    - divido stringa bit in blocchi da 6 bit (=64 valori)
    - rappresento ogni valore con una lettera

Google's Protocol Buffers

- Meccanismo per serializzare dati strutturati proposto da Google
- Piú compatto e rapido rispetto a XML/JSON
- Dati rappresentati in formato binario
  - quindi più efficiente
- É sufficiente definire uno schema della struttura dei dati:
  - Viene automaticamente generato il codice per marshalling e unmarshalling (in base al linguaggio usato)
  - Dopodichè è possibile leggere/scrivere questi dati su/da diversi stream
  - Altamente interoperabile e ci sono librerie per svariati linguaggi

Come usare ProtocolBuffers

1. Definire tramite file con estensione *.proto* il formato dei messaggi da serializzare
   - É necessario definire i campi previsti e il loro tipo
   - I file *.proto* devono essere condivisi tra tutti i programmi che intendono comunicare
2. Lanciare il compilatore Protocol Buffer per generare automaticamente il codice per marshalling e unmarshalling
   - *protoc –proto_path=src –java_out=build/gen src/foo.proto*
3. Dopodichè è possibile scrivere il codice per mandare e ricevere messaggi

Messaggi

- Ogni tipo di messaggio che si intende modellare va specificato con la keyword *message*
  - Ogni *message* é un record che contiene diversi campi
  - Puó contenere a sua volta altri *message*
- Ogni campo di un messaggio deve essere definito con una delle seguenti keywords:
  - **required**: il campo é obbligatorio
  - **optional**: il campo puó essere omesso
  - **repeated**: il campo puó essere ripetuto da 0 a piú volte
    - Va pensato come un array di dimensione dinamica

Tipi scalari

.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
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

- Ogni campo definito in un messaggio ha un **tag univoco**
- I tag sono usati per identificare i campi nel formato binario
  - Usati per il match dei campi nelle operazioni di marshalling e unmarshalling
  - I tag da 1 a 15 vengono rappresentati con un byte (da utilizzare per campi piú frequenti)
  - I tag da 16 a 2047 necessitano 2 bytes
- Aiutano la back-compatibilitá della definizione dei messaggi
  - Sistemi che non hanno aggiornato la definizione dei messaggi possono ignorare i nuovi campi
- I tag non devono essere cambiati una volta che il messaggio é effettivamente usato in un sistema!

Protocol Buffers: un esempio

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

        enum ResearcherType {
                PHDSTUDENT = 0;
                POSTDOC = 1;
                ASSISTANTPROFESSOR = 2;
        }

        message Paper {
                required string title = 1;
                required int32 year = 2;

        }

        repeated Paper paper = 3;
        optional ResearcherType type = 4;
}
```

Costruire un oggetto

Sfruttiamo il codice che è stato generato automaticamente...

```
Researcher researcher =
        Researcher.newBuilder()
                .setName("Gabriele")
                .setSurname("Civitarese")
                .setType(Researcher.ResearcherType.POSTDOC)
                .addPaper(Researcher.Paper.newBuilder()
                        .setTitle("Activity Recognition")
                        .setYear(2014))
                .addPaper(Researcher.Paper.newBuilder()
                        .setTitle("Activity Recognition Again")
                        .setYear(2015))
                .build();
```

Getters

- Oltre ai setter, ProtocolBuffer fornisce automaticamente metodi per accedere ad ogni campo
- Ad esempio:
  - *researcher.getName()* restituisce una stringa
  - *researcher.getPaperList()* restituisce una lista di oggetti Paper

Comunicazione tramite Socket

Come comunichiamo messaggi Protocol Buffers?

- Come già introdotto, il codice generato automaticamente si occupa di marshalling e unmarshalling per la comunicazione dei dati
- **Lato client** è sufficiente utilizzare il metodo *writeTo()* per wrappare lo stream di output della socket verso il server:
  - *researcher.writeTo(socket.getOutputStream());*
- In modo analogo, **lato server** è necessario richiamare il metodo *parseFrom()* per wrappare lo stream di input
  - *researcher.parseFrom(socket.getInputStream());*

Riferimenti per Protocol Buffer

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

Esercizio

Statistiche studenti

- Sviluppare un'applicazione server *Università* che riceve dati da applicazioni client *Studente*
- L'applicazione *Studente* invia tramite socket all'applicazione *Universitá* informazioni relative ad uno specifico studente:
  - Generalitá: Nome, Cognome, Anno di Nascita, Residenza (campo composto)
  - Lista degli esami passati. Per ogni esame si vuole memorizzare il nome dell'esame, il voto ottenuto e la data di verbalizzazione
- L'applicazione *Università* si limita a ricevere il messaggio da socket e a stampare su schermo le statistiche dello studente
- Sviluppare la soluzione utilizzando sia JSON che Protocol Buffers per lo scambio dei messaggi
  - Nel caso di JSON consiglio l'uso della libreria Gson
- Confrontare quanti bytes vengono effettivamente risparmiati utilizzando Protocol Buffers invece che JSON

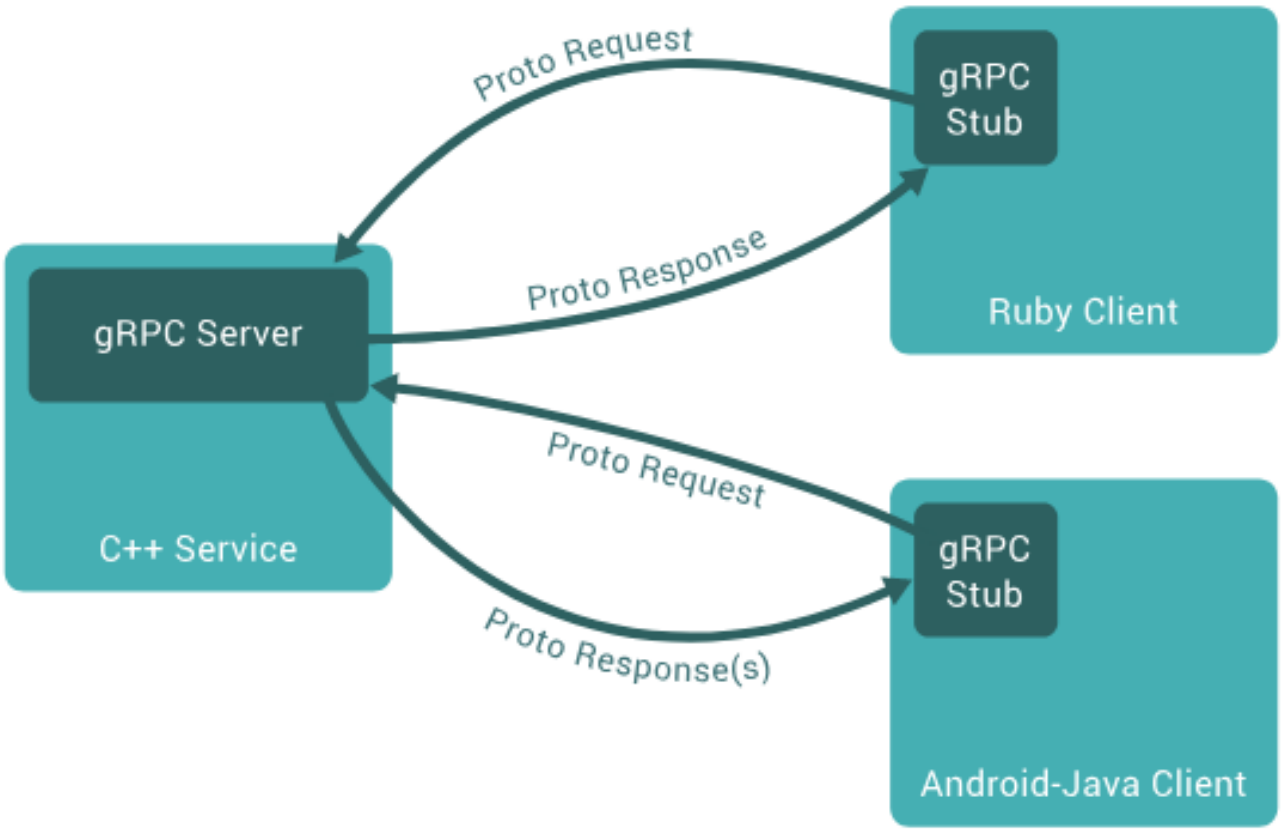
RPC: Ripasso

- Paradigma per servizi di rete
- Invece di mandare e ricevere messaggi esplicitamente, il client invoca un servizio remoto tramite una chiamata di una procedura locale
- La procedura locale (*stub*) nasconde tutti i dettagli della comunicazione di rete
- Internamente, la libreria di RPC effettua il *marshalling*, trasmettendo al server i parametri del metodo *stub*
- Il server effettua l'*unmarshalling*, invoca il metodo e invia l'output al client
- Il client non avverte di effettuare una chiamata di rete, ma bensì una chiamata locale

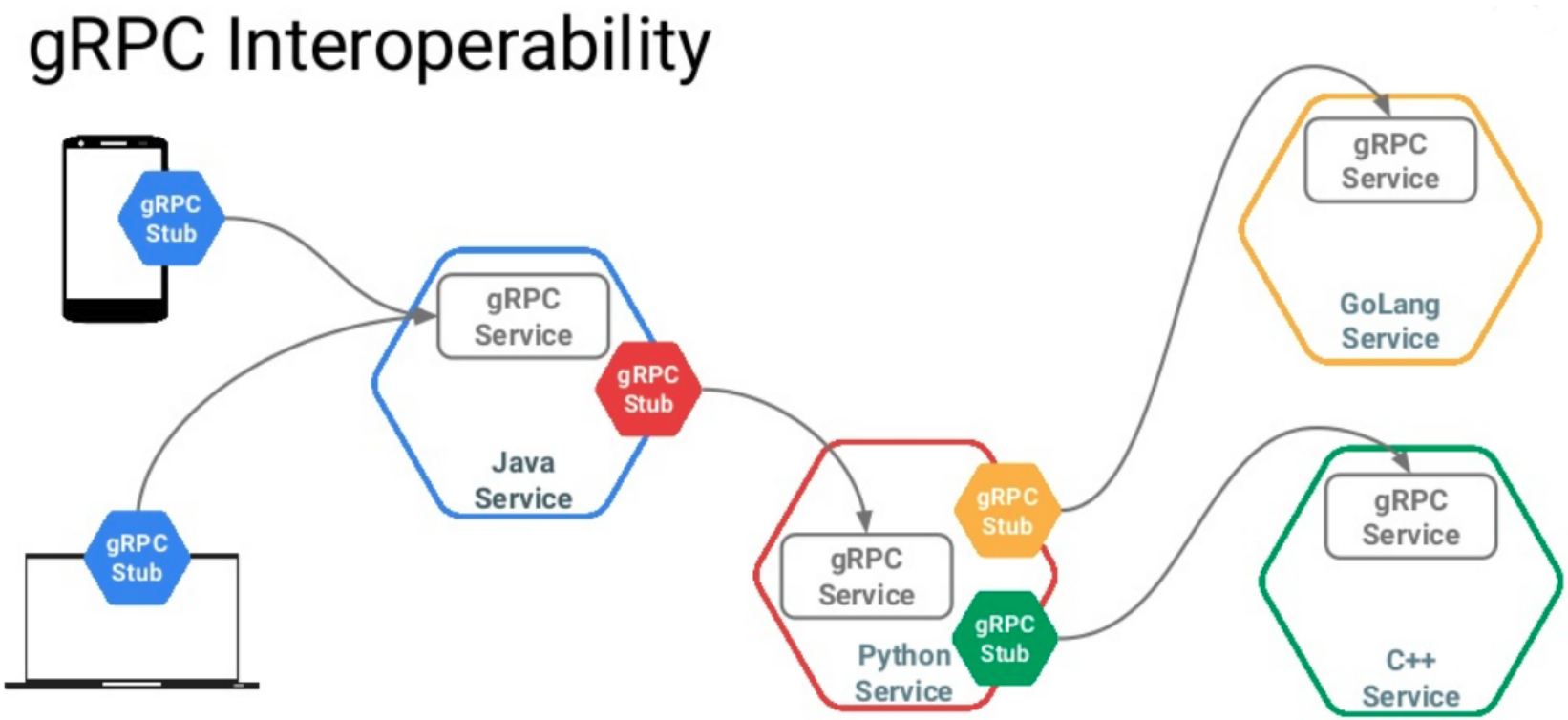
gRPC: a gRPC Remote Procedure Call framework

- Sistema di RPC sviluppato e usato ampiamente da Google
  - ...e da altri big tipo Netflix
- Definizione dei servizi tramite **Protocol Buffer**
  - nome dei metodi, argomenti, tipo del valore restituito, ...
- Lato server è necessario implementare l'interfaccia e far girare un server gRPC per gestire le chiamate dei client
- Lato client vengono semplicemente chiamati metodi *stub* che sono esattamente quelli forniti dal server

Architettura generale



Alta interoperabilità



Vantaggi

- Efficienza di comunicazione garantita da Protocol Buffer (binario)
- Il canale di comunicazione si basa su *HTTP2*:
  - Sfrutta il *multiplexing*: multiple richieste e risposte possono essere ricevute in maniera asincrona con una sola connessione TCP
  - Stream bi-direzionali: una volta stabilita la connessione è possibile inviare più messaggi in entrambe le direzioni
- Fortemente tipato
- Più sofisticato di REST che si basa solo su verbi HTTP

Esempio di definizione

```
// 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;
}
```

Sincrono VS Asincrono

- É possibile sfruttare i metodi RPC sia in modalità sincrona che asincrona
  - Dipende dall'applicazione
- Le chiamate sincrone sono bloccanti:
  - Il client chiama il metodo stub e si sblocca solo quando riceve la risposta dal server
  - Approssimazione più vicina all'astrazione di una procedura RPC
- I sistemi distribuiti sono di natura asincrona, e gRPC ci permette di avere stub asincroni
  - Il client chiama il metodo stub senza bloccare il thread corrente
  - In maniera asincrona verrà ricevuta la risposta del server

Tipi di RPC

1. **Unary RPC**: Il client manda una singola richiesta e riceve una singola risposta
2. **Server streaming RPC**: Il client manda una singola richiesta e il server risponde con uno stream di risposte. Il client conclude quando riceve tutte le risposte.
3. **Client streaming RPC**: Il client invia uno stream di richieste e il server risponde con una singola risposta (non necessariamente dopo aver ricevuto tutte le richieste)
4. **Bidirectional streaming RPC**: Quello che succede dipende dall'applicazione. Client e server possono scrivere in qualsiasi ordine, gli stream sono indipendenti.

Tipi di RPC a confronto

- **Unary RPC**:

  ```
  rpc Greeter (HelloRequest) returns (HelloResponse) {}
  ```

- **Server streaming RPC**:

  ```
  rpc LotsOfReplies (HelloRequest) returns (stream HelloResponse) {}
  ```

- **Client streaming RPC**:

  ```
  rpc LotsOfGreetings (stream HelloRequest) returns (HelloResponse) {}
  ```

- **Bidirectional streaming RPC**:

  ```
  rpc BiHello (stream HelloRequest) returns (stream HelloResponse) {}
  ```

Compilazione

- Per compilare la definizione del servizio per generare codice stub per client e server è necessario eseguire il comando:
  - *protoc –plugin=protoc-gen-grpc-java=build/exe/java_plugin/protoc-gen-grpc-java –grpc-java_out="$OUTPUT_FILE" –proto_path="$DIR_OF_PROTO_FILE" "$PROTO_FILE"*
- Lato server, per ogni servizio definito nel file proto è necessario implementarne la logica
- Lato client è sufficiente specificare l'indirizzo del servizio ed effettuare chiamate locali ai metodi stub

Greeter: lato server

GreeterImpl.java

```
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();
        }

}
```

Lanciare il servizio

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

server.start();

System.out.println("Server started!");

server.awaitTermination();
```

Importante!

- Di default, il *ServerBuilder* è implementato con un pool di thread che gestiscono le richieste
- Ogni chiamata gRPC è quindi asincrona!
- Sarà quindi necessario gestire i problemi di concorrenza come visto per server multi-thread

Greeter: lato client

Chiamata RPC bloccante

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

La classe *StreamObserver*

- Per comunicare in maniera asincrona su stream, si usa la classe *StreamObserver*
- Serve a ricevere notifiche asincrone da uno stream di messaggi
- Utilizzato sia da client stub che dalle implementazioni dei servizi per comunicare su stream
- Mette a disposizione tre metodi handler:
  - *onNext(V value)*: riceve un valore dallo stream
  - *onError(Throwable t)*: riceve un errore verificatosi sullo stream
  - *onCompleted()*: riceve la comunicazione che lo stream è completato correttamente
- Chi riceve lo stream deve *implementare* questi metodi
- Chi invia lo stream deve chiamare gli *stub* di questi metodi
  - Sono metodi RPC!

Esempio: Greeting stream

Definizione .proto

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

Implementazione server

```
public void streamGreeting(HelloRequest request,
                           StreamObserver<HelloResponse> responseObserver){

        HelloResponse response = HelloResponse.newBuilder()
                        .setGreeting("Hello there, "+request.getName())
                                .build();

        responseObserver.onNext(response);
        responseObserver.onNext(response);
        responseObserver.onNext(response);
        responseObserver.onCompleted();
}
```

Esempio: Greeting stream client

```
final ManagedChannel channel = ManagedChannelBuilder
                .forTarget("localhost:8080").usePlaintext(true).build();
GreetingServiceStub stub = GreetingServiceGrpc.newStub(channel);
HelloRequest request = HelloRequest.newBuilder()
        .setName("Gianni").build();

stub.streamGreeting(request, new StreamObserver<HelloResponse>() {
        //this hanlder takes care of each item received in the stream
        public void onNext(HelloResponse helloResponse) {
                System.out.println(helloResponse.getGreeting());
        }
        //if there are some 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();
        }
});
```

Riferimenti

- What is gRPC?
- gRPC Basics - Java
- Introduzione a HTTP/2

Esercizio

- Sviluppare un servizio utilizzando *grpc*
- Il servizio deve offrire diversi metodi RPC:
  - *SimpleSum*: dati due numeri interi, restituisce la loro somma (provare sincrono e asincrono)
  - *RepeatedSum*: dato un numero n e un numero t, il servizio restituisce uno stream di t numeri. Il primo valore dello stream è n, il secondo è $n + n$, il terzo $n + n + n$, ..., fino ad arrivare a t valori.
  - *StreamSum*: il client manda in stream coppie di numeri da sommare e il server risponde con la somma