Remote Procedure Call in Python

Advanced Computer Programming

Prof. Luigi De Simone

Summary

- **Argomenti**
  - Richiami su middleware RPC
  - gRPC
  - Esempio di utilizzo di gRPC in Python

- **Riferimenti**
  - https://grpc.io/docs/what-is-grpc/core-concepts/
  - https://grpc.io/docs/languages/python/quickstart/
  - https://grpc.io/docs/languages/python/basics/
  - Kasun Indrasiri, Danesh Kuruppu, *"gRPC: Up and Running"*, O'Reilly Media, Inc.

gRPC

- grpc.io è un middleware RPC **universale** ad **alte prestazioni** e **open source**
- Può essere eseguito in qualsiasi ambiente
  - Framework multilinguaggio e **multipiattaforma**
- Principali scenari di utilizzo
  - Collegare **microservizi** poliglotti che utilizzano lo stile di comunicazione richiesta-risposta
  - Connettere **dispositivi mobili** e browser ai servizi di backend
  - Generare **librerie client efficienti**
- Sviluppato da **Google** e ora un progetto *Cloud Native Computing Foundation (CNCF)*
- **Utilizzato da molte aziende** e in molti sistemi distribuiti
  - Ad esempio, Google, Dropbox, Netflix, Square, etcd, CockroachDB

Caratteristiche principali di gRPC

- **HTTP/2 per il trasporto**
  - Streaming bidirezionale e multiplexing
- ***Protocol buffer* (*protobuf*) usato come IDL**
  - Generazione automatica del codice
  - I *contratti* rigorosi definiti tramite *protobufs* **prevengono gli errori**
- Supporto integrato per l'**autenticazione**, **streaming bidirezionale** e **controllo del flusso**
- Supporto per la **comunicazione bloccante e non bloccante**
  - *Blocking/synchronous stub*: il chiamante attende che il server risponda all'atto di una chiamata RPC; il chiamante ritornerà la risposta ricevuta oppure solleverà un'eccezione
  - *Non-blocking/asynchronous stub*: il chiamante non attende che il server risponda all'atto di una chiamata RPC, ovvero la risposta dal server avviene in maniera asincrona

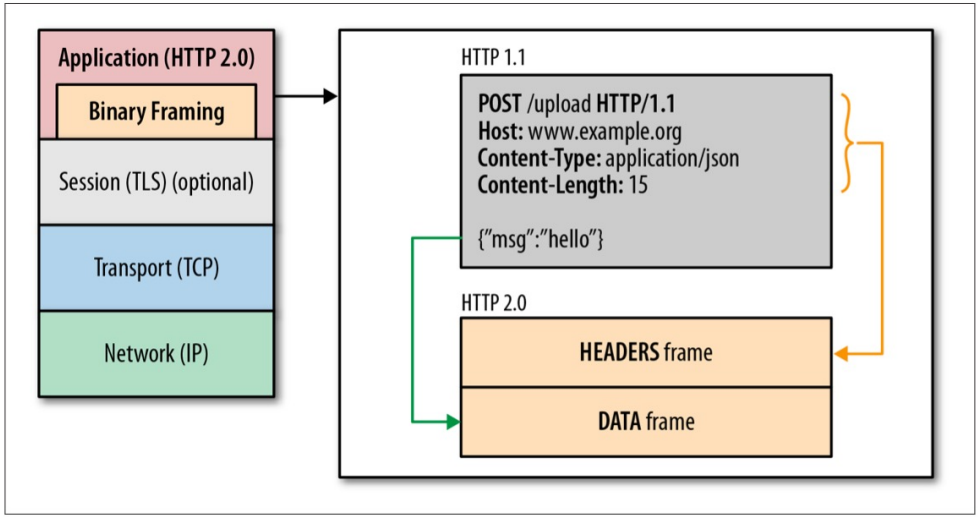
Caratteristiche principali di gRPC

- **Trasporto dei dati e delle richieste su HTTP/2**
  - Idea di base di gRPC: trattare le RPC come ***riferimenti a oggetti HTTP***

- **HTTP/2** è un importante revisione di HTTP che fornisce significativi vantaggi in termini di prestazioni rispetto a HTTP 1.x

Cenni a HTTP/2

- **Binary Framing Layer:** Le richieste/risposte di HTTP/2 sono suddivise in *messaggi di piccole dimensioni* e con un frame in formato binario, rendendo la trasmissione dei messaggi **efficiente**



Cenni a HTTP/2

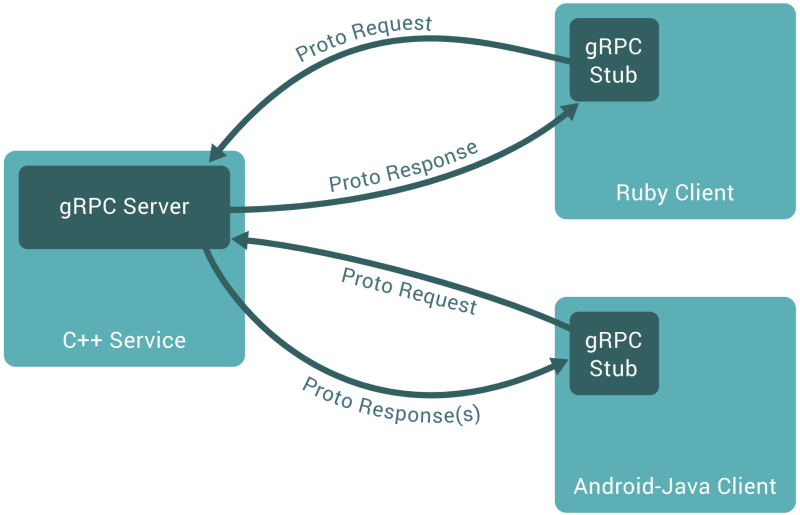
- Dai messaggi di *richiesta/risposta* agli **stream**
  - **Stream**: flusso bidirezionale di byte all'interno di una connessione stabilita, che può trasportare uno o più messaggi.
  - **Message**: sequenza completa di messaggi che corrispondono a una richiesta logica o a un messaggio di risposta.
  - **Frame**: la più piccola unità di comunicazione in HTTP/2, ciascuna contenente un'*intestazione di frame*, che identifica almeno il flusso a cui il frame appartiene

Cenni a HTTP/2

- **Multiplexing richiesta/risposta** (utilizzo di una singola connessione per client): consente un uso efficiente delle connessioni TCP ed evita il blocco della linea di testa a livello HTTP.
- **Supporto nativo per lo streaming bidirezionale**
- **Compressione** dell'intestazione HTTP: per ridurre l'overhead del protocollo
- Ulteriori dettagli in *web.dev/performance-http2*

gRPC Protocol buffer

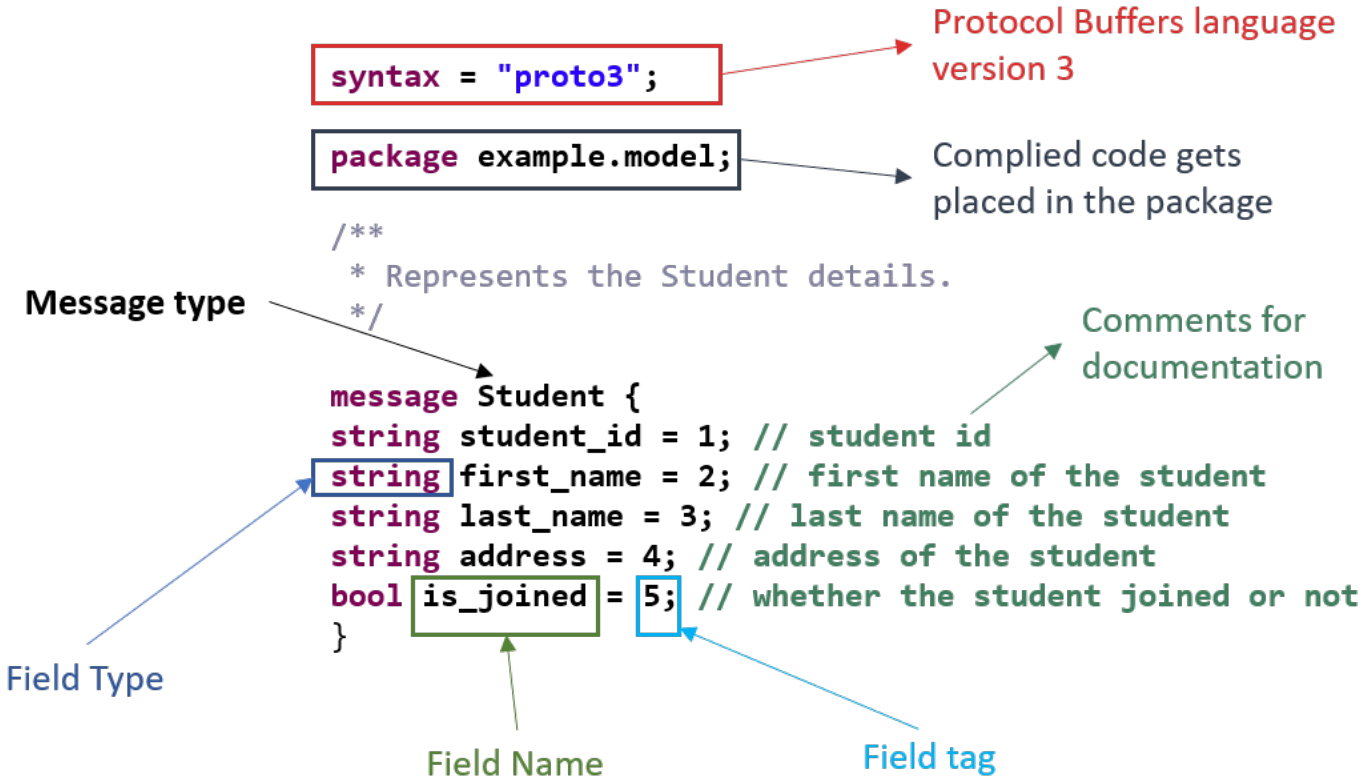
- gRPC utilizza i ***protocol buffer*** (detti anche *protobuf*) come:
  - **IDL per definire l'interfaccia del servizio**
    - generazione automatica di ***stub client*** e ***classi server astratte***
  - **Formato di interscambio dei messaggi**
    - i messaggi di gRPC sono serializzati utilizzando i *protocol buffer*, avendo dei piccoli payload dei messaggi da gestire
- I *protocol buffer* sono basati sul solito modello di Proxy-Skeleton (stub e server)



Protocol buffer

- I *protocol buffer* sono un meccanismo open-source ormai maturo di Google per **serializzare i dati strutturati**
- Usano una **rappresentazione binaria** dei dati
- Descrizione **fortemente tipizzata**
- **I tipi di dati sono strutturati come messaggi**
  - **Messaggio**: piccolo record logico di informazioni contenente una serie di coppie *nome-valore* chiamate **campi**
  - **I campi hanno numeri univoci**, utilizzati per identificare i campi nel formato binario del messaggio

Protocol buffer



Protocol buffer: package

- È possibile aggiungere uno specificatore di **`package`** (opzionale) a un file `.proto` per evitare conflitti di nome tra i tipi di messaggi di un protocollo
- In **Python**, la direttiva package viene **ignorata**, poiché i moduli Python sono organizzati in base alla loro posizione nel file system
  - In ogni caso, è fortemente raccomandato specificare il package per il file `.proto`, perché altrimenti potrebbe portare a conflitti di denominazione nei descrittori e rendere il proto non portabile per altri linguaggi

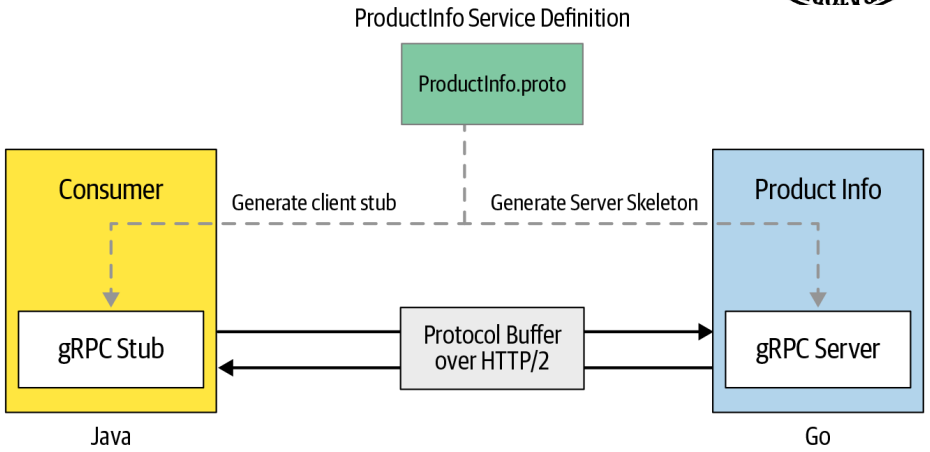
Protocol buffer: Esempio

```
// ProductInfo.proto
syntax = "proto3";
package ecommerce;

service ProductInfo {
   rpc addProduct(Product) returns (ProductID);
   rpc getProduct(ProductID) returns (Product);
}

message Product {
   string id = 1;
   string name = 2;
   string description = 3;
}

message ProductID {
   string value = 1;
}
```



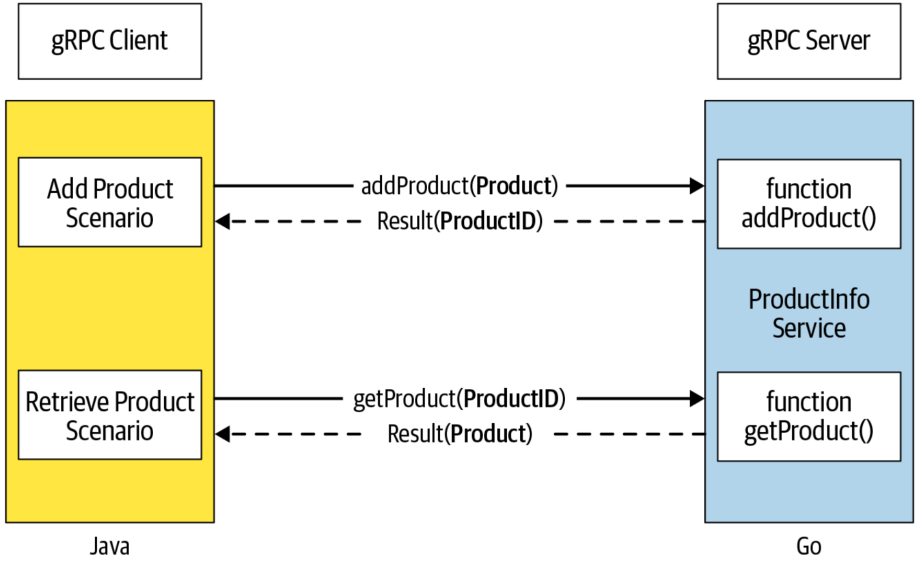
Protocol buffer: Esempio

```
// ProductInfo.proto
syntax = "proto3";
package ecommerce;

service ProductInfo {
   rpc addProduct(Product) returns (ProductID);
   rpc getProduct(ProductID) returns (Product);
}

message Product {
   string id = 1;
   string name = 2;
   string description = 3;
}

message ProductID {
   string value = 1;
}
```



Uso di gRPC

1. **Installazione** (una tantum) **dei package necessari per usare gRPC e dei tool relativi in Python**

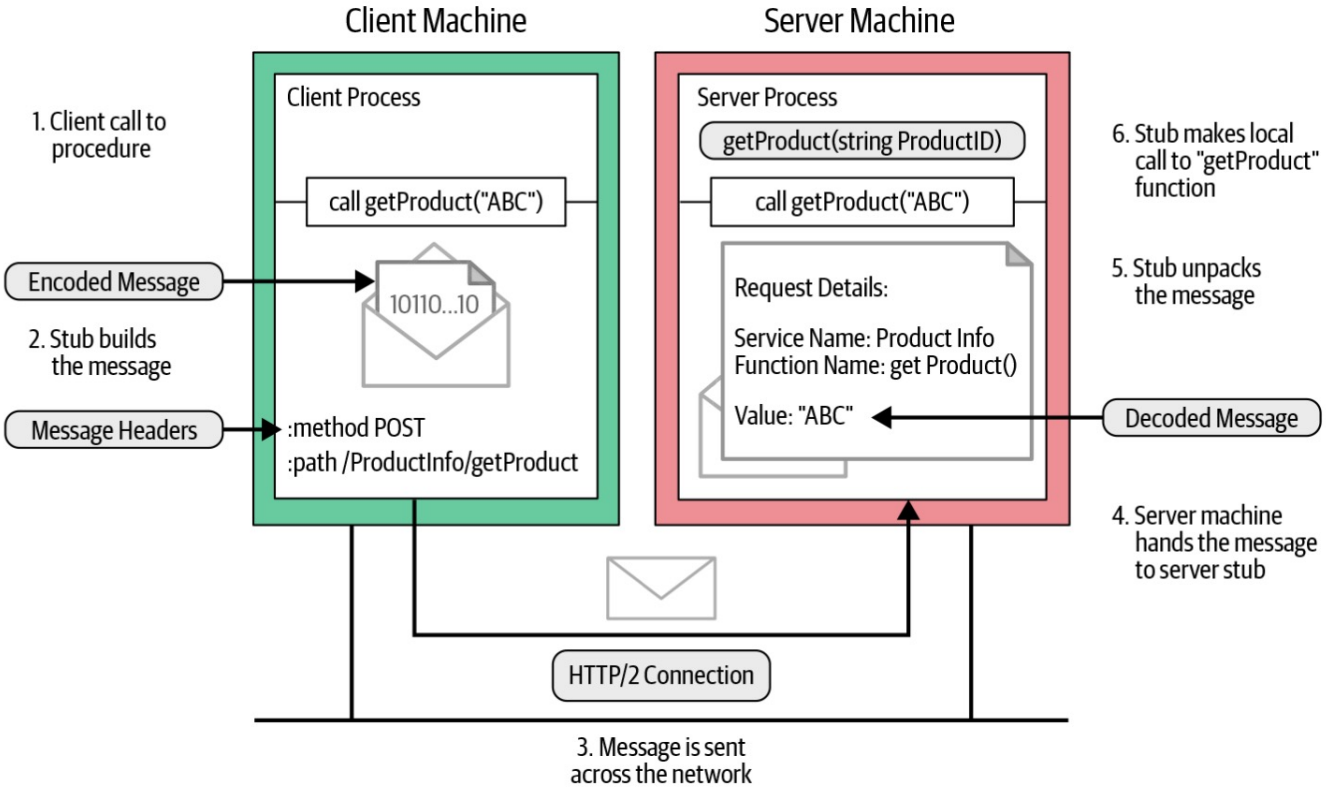
```
$ python –m pip install grpcio
$ python –m pip install grpcio–tools
```

2. **Definire il servizio** (collezione di metodi remoti) e i **tipi di messaggi** che vengono scambiati tra il client e il servizio nel file **.proto**, utilizzando i protobuf come IDL

3. **Generare** il **codice stub** del **server** e del **client** a partire dalla definizione del servizio nel file `.proto`, utilizzando **protoc** (compilatore di *protobuf*) per il linguaggio specifico
   - Ogni linguaggio possiede il proprio compilatore protobuf (e.g., in Python usare il modulo `grpc_tools.protoc`)

4. Usare **gRPCAPI** per il linguaggio scelto (per esempio, Python, Java, etc.) per scrivere successivamente il **client** e il **server**

gRPC workflow



Esempio di gRPC in Python: HELLO WORLD

1. **Definire il servizio** (file helloworld.proto)

```
syntax = "proto3";

package helloworld;

// La definizione del servizio chiamato Greeter
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// Il messaggio di richiesta contiene una stringa che è il nome utente
message HelloRequest {
  string name = 1;
}

// Il messaggio di risposta che contiene una stringa di benvenuto
message HelloReply {
  string message = 1;
}
```

Esempio di gRPC in Python: HELLO WORLD

2. **Compilare la definizione del servizio per ottenere gli stub***

```
$ python -m grpc_tools.protoc --python_out=. --pyi_out=.
                              --grpc_python_out=. helloworld.proto
```

- La compilazione genererà automaticamente i seguenti file:
  - **`helloworld_pb2.py`**: contiene codice *protobuf* per popolare, serializzare, e recuperare tipi di messaggi di richiesta e risposta
  - **`helloworld_pb2_grpc.py`**: contiene
    - L'interfaccia (proxy) utilizzata dai **client** per invocare (*call*) i metodi definiti nel servizio `helloworld`
    - L'interfaccia (skeleton) utilizzata dai **server** per implementare i metodi definiti nel servizio `helloworld`

* NOTA: Per Windows, lanciare **`python -m grpc_tools.protoc -I. ...`**

Esempio di gRPC in Python: HELLO WORLD

2. **Compilare la definizione del servizio per ottenere gli stub**

```
$ python -m grpc_tools.protoc --python_out=. --pyi_out=.
                              --grpc_python_out=. helloworld.proto
```

- **NOTA**:
  - Se il file di specifica (helloworld.proto) è all'interno di una directory, bisogna specificare quest'ultima con il flag `-Idirectory`
    - `python -m grpc_tools.protoc -Iprotos --python_out=. --pyi_out=. --grpc_python_out=. protos/helloworld.proto`

Esempio di gRPC in Python: HELLO WORLD

3. **Creazione del server, in 2 parti**
   a) **Implementare l'interfaccia del servizio** dalla definizione del servizio

```
class Greeter(helloworld_pb2_grpc.GreeterServicer):

def SayHello(self, request, context):

    return helloworld_pb2.HelloReply(message="Hello, %s!" % request.name)
```

   - L'implementazione del metodo SayHello restituisce un messaggio del tipo ***Hello nome_richiesta***

Esempio di gRPC in Python: HELLO WORLD

3. **Creazione del server, in 2 parti**
   b) **Creare ed eseguire un server gRPC** per accogliere le richieste dei client e inviarle all'implementazione del servizio

```
def serve():
        # definisco il porto
        port = "50051"
        server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
        # aggiungo al server l'oggetto istanza del mio servizio Greeter
        helloworld_pb2_grpc.add_GreeterServicer_to_server(Greeter(), server)
        # faccio il bind al porto impostato
        server.add_insecure_port("[::]:" + port)
        # avvio il server
        server.start()
        print("Server started, listening on " + port)
        # attendo che il server termini
        server.wait_for_termination()

if __name__ == "__main__":
        serve()
```

Esempio di gRPC in Python: HELLO WORLD

4. **Creazione del client**
   a) Per chiamare i metodi del servizio, occorre prima **creare un canale gRPC** per comunicare con il server di destinazione usando `insecure_channel`
   b) Abbiamo bisogno di uno **stub client per eseguire le RPC**: lo otteniamo dal file `helloworld_pb2_grpc` generato tramite il file .proto
   c) **Chiamiamo i metodi di servizio sullo stub client**: creiamo e popoliamo un oggetto *protobuf* di richiesta (`HelloRequest`) passando il corpo del messaggio

```
# creo un canale verso il server RPC
with grpc.insecure_channel("localhost:50051") as channel:

        # creo uno stub (GreeterStub, ovvero ${NOMESERVIZIO}Stub) per
        invocare tutti i metodi implementati nel servizio
        stub = helloworld_pb2_grpc.GreeterStub(channel)
        response = stub.SayHello(helloworld_pb2.HelloRequest(name="you"))
```

Aggiornare servizio gRPC

Per aggiornare un servizio bisogna

- Aggiungere il nuovo metodo al file .proto
- Rigenerare il codice gRPC usando protoc
- Aggiornare il codice del server per implementare il nuovo metodo
- Aggiornare il codice del client per invocare il nuovo metodo

Tipi di RPC call in gRPC

- gRPC supporta 4 tipi di RPC call che possono essere definiti nel file `.proto`
- **RPC semplice (unary)**: il client invia una richiesta al server e attende una singola risposta (cioè, unaria), come una normale chiamata di funzione

```
rpc SayHello (HelloRequest) returns (HelloReply) {}
```

Tipi di RPC call in gRPC

- **Streaming RPC lato server** (*response-streaming RPC*): il client invia una richiesta al server e ottiene un **flusso** (`stream`) per leggere una sequenza di messaggi
- **Il client legge dallo stream restituito finché non ci sono più messaggi**
- gRPC garantisce l'ordinamento dei messaggi all'interno di una singola chiamata RPC

```
rpc ListFeatures(Rectangle) returns (stream Feature) {}
```

Tipi di RPC call in gRPC

- **Streaming RPC lato client** (*request-streaming RPC*): il client scrive una sequenza di messaggi e li invia al server, sempre utilizzando uno stream fornito
- Una volta terminata la scrittura dei messaggi, **il client attende che il server li legga tutti e restituisca la risposta**
- gRPC garantisce l'ordinamento dei messaggi all'interno di una singola chiamata RPC

```
rpc RecordRoute(stream Point) returns (RouteSummary) {}
```

Tipi di RPC call in gRPC

- **Bidirectional Streaming RPC**: entrambe le parti inviano una sequenza di messaggi utilizzando un flusso di lettura e scrittura
- **I due flussi operano in modo indipendente**, per cui client e server possono leggere e scrivere nell'ordine che preferiscono
  - Ad esempio, il server potrebbe aspettare di ricevere tutti i messaggi del client prima di scrivere le sue risposte, oppure potrebbe alternativamente leggere un messaggio e poi scriverne uno, o qualche altra combinazione di lettura e scrittura
- L'ordine dei messaggi in ogni flusso viene mantenuto

```
rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
```

Generators e funzione yield

- Per creare le funzioni per lo streaming RPC è necessario l'utilizzo dei **Generator**:
  - Funzioni che ritornano un *iteratore*
    - Un iteratore è un oggetto che contiene un certo numero di elementi su cui è possibile iterare

```
mytuple = ("apple", "banana")       # tupla
myit = iter(mytuple)                # metodo che permette di ottenere un iterator
print(next(myit))                   # apple
print(next(myit))                   # banana
```

  - Per creare un generator è necessario utilizzare il metodo **yield** al posto del metodo *return*
    - È usato per produrre un valore, **senza stoppare la funzione**
    - Le funzioni che contengono la yield sono considerate automaticamente generator

```
yield <expression>
```

Generators e funzione yield

Esempio di generator:

```
def range2iter(n):
    """
    Generator for the squares of numbers 0 to n-1
    Precon: n is an int >= 0
    """

    for x in range(n):
        yield x*x
```

```
>>> a = range2iter(3)
>>> a
<generator object>
>>> next(a) 0
>>> next(a) 1
>>> next(a) 4
```

N.B.: L'utilizzo della `return` al posto di `yield` porterebbe la funzione ad interrompersi al primo ciclo!

Streaming RPC lato server: Esempio

- **Specifica del servizio**: helloworld.proto

```
syntax = "proto3";

package helloworld;

// La definizione del servizio chiamato Greeter
service Greeter {
  // Sends a greeting
  rpc SayHello_v1 (HelloRequest) returns (stream HelloReply) {}
}

// Il messaggio di richiesta contiene una stringa che è il nome utente
message HelloRequest {
  string name = 1;
}

// Il messaggio di risposta che contiene una stringa di benvenuto
message HelloReply {
  string message = 1;
}
```

Streaming RPC lato server: Esempio

- **Server**: Class Greeter in helloworld_server.py

```
class Greeter(helloworld_pb2_grpc.GreeterServicer):

  # Streaming server
  def SayHello_v1(self, request, context):

     for i in range (0, 5):
       print("[server] SayHello method invoked, returning response...")
       yield helloworld_pb2.HelloReply(message="Hello, " + request.name + "! - " + str(i))
```

Streaming RPC lato server: Esempio

- **Client**: Run method in helloworld_client.py

```
def run():

  print("Will try to greet world ...")

  with grpc.insecure_channel("localhost:50051") as channel:

    stub = helloworld_pb2_grpc.GreeterStub(channel)

    # Server streaming
    for response in stub.SayHello_v1(helloworld_pb2.HelloRequest(name="you")):
      print("[CLIENT] SayHello invoked Greeter client received: " + response.message)
```

Streaming RPC lato client: Esempio

- **Specifica del servizio**: helloworld.proto

```
syntax = "proto3";

package helloworld;

// La definizione del servizio chiamato Greeter
service Greeter {
  // Sends a greeting
  rpc SayHello_v2 (stream HelloRequest) returns (HelloReply) {}
}

// Il messaggio di richiesta contiene una stringa che è il nome utente
message HelloRequest {
  string name = 1;
}

// Il messaggio di risposta che contiene una stringa di benvenuto
message HelloReply {
  string message = 1;
}
```

Streaming RPC lato client: Esempio

- **Server**: Class Greeter in helloworld_server.py

```
class Greeter(helloworld_pb2_grpc.GreeterServicer):

  # Streaming client
  def SayHello_v2(self, request_iterator, context):

    names = []
    for request in request_iterator:
      print("[server] SayHello method invoked, with name " + request.name)
      names.append(request.name)

    return helloworld_pb2.HelloReply(message="Hello, " + ' '.join(names) + "!")
```

Streaming RPC lato client: Esempio

- **Client**: Run method in helloworld_client.py

```
def generate_requests():

  names = ['Raf', 'Gigi', 'Pippo', 'Pippozzo']

  for name_to_send in names:
    yield helloworld_pb2.HelloRequest(name=name_to_send)

def run():

  print("Will try to greet world ...")

  with grpc.insecure_channel("localhost:50051") as channel:

    stub = helloworld_pb2_grpc.GreeterStub(channel)

    # Client streaming
    response = stub.SayHello_v2(generate_requests())
    print("[CLIENT] SayHello invoked Greeter client received: " + response.message)
```

Bi-directional streaming RPC: Esempio

- **Specifica del servizio**: helloworld.proto

```
syntax = "proto3";

package helloworld;

// La definizione del servizio chiamato Greeter
service Greeter {
  // Sends a greeting
  rpc SayHello_v3 (stream HelloRequest) returns (stream HelloReply) {}
}

// Il messaggio di richiesta contiene una stringa che è il nome utente
message HelloRequest {
  string name = 1;
}

// Il messaggio di risposta che contiene una stringa di benvenuto
message HelloReply {
  string message = 1;
}
```

Bi-directional streaming RPC: Esempio

- **Server**: Class Greeter in helloworld_server.py

```
class Greeter(helloworld_pb2_grpc.GreeterServicer):

  # Bi-directional streaming
  def SayHello_v3(self, request_iterator, context):

    for request in request_iterator:
      print("[server] SayHello method invoked, returning response...")
      yield helloworld_pb2.HelloReply(message="Hello, " + request.name + "!")
```

Bi-directional streaming RPC: Esempio

- **Client**: Run method in helloworld_client.py

```
def generate_requests():

  names = ['Raf', 'Gigi', 'Pippo', 'Pippozzo']

  for name_to_send in names:
    yield helloworld_pb2.HelloRequest(name=name_to_send)

def run():

  print("Will try to greet world ...")

  with grpc.insecure_channel("localhost:50051") as channel:

    stub = helloworld_pb2_grpc.GreeterStub(channel)

    # Bi-directional streaming
    for response in stub.SayHello_v3(generate_requests()):
       print("[CLIENT] SayHello invoked Greeter client received: " + response.message)
```

Limitazioni di gRPC

- **Supporto limitato nei browser**
  - Mancanza del supporto completo HTTP/2
  - gRPC-Web può fornire il supporto gRPC al browser ma con caratteristiche limitate (solo RPC semplice e streaming server limitato)
- **Formato non *human-readable***
  - Protobuf è efficiente nell'invio e nella ricezione, ma il suo formato binario non è leggibile dall'uomo
  - Gli sviluppatori hanno bisogno di strumenti aggiuntivi (ad esempio, lo strumento a riga di comando gRPC) per analizzare i payload protobuf, scrivere richieste a mano, eseguire il debugging