Remote Procedure Call – gRPC

Introducere in gRPC si “protocol buffers”.

gRPC poate folosi “protocol buffers” ca Interface Definition Language (**IDL**) si ca format al mesajelor.

Overview

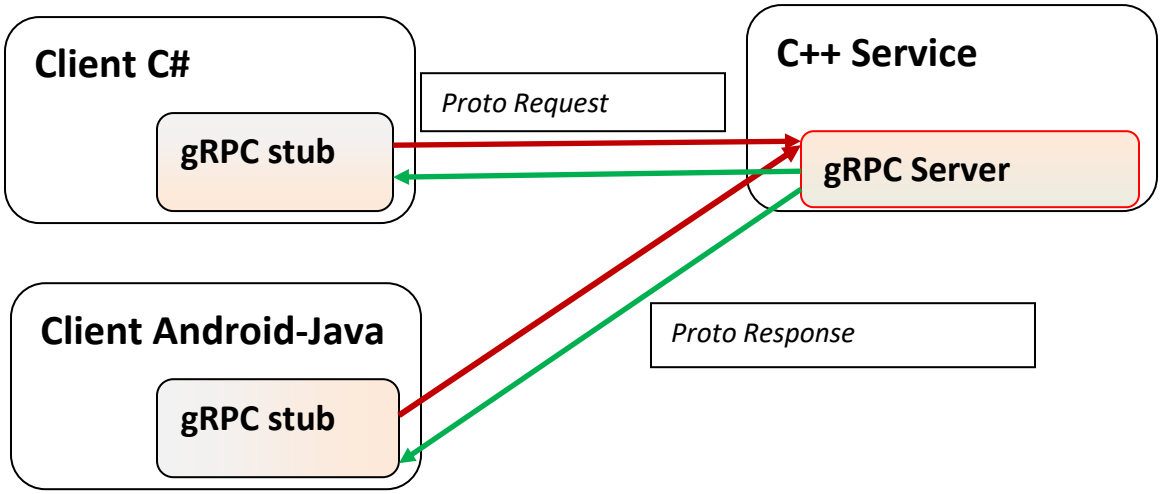
In gRPC, o aplicatie client poate apela in mod direct o metoda din aplicatia server, aplicatie ce poate fi pe o alta masina, ca si cum acea metoda ar fi pe un obiect din aplicatia client. Se pot crea astfel aplicatii distribuite si servicii.
gRPC este bazat pe ideea definirii unui serviciu, specificand metodele ce pot fi apelate remote. Metodele sunt implementate de clase din server.

Pe partea de server:

- Se definesc contractele de serviciu (metode in final) si mesajele intr-o interfata (fisiere cu extensia .proto).
- Serverul implementeaza aceasta interfata si ruleaza un “gRPC server” pentru a intercepta apeluri de la clienti.

Pe partea de client:

- Clientul are un “stub” (il putem asemana cu proxy) ce furnizeaza aceleasi metode ca si serverul. Metodele nu sunt implementate.



Clientii si serverele gRPC pot rula si comunica in diverse medii (platforme). Clientul poate fi scris intr-un limbaj, serverul in alt limbaj in timp ce WCF cere Windows-to-Windows.

Lucrand cu Protocol Buffers

Protocol Buffers, open source dezvoltat de Google, mecanism pentru serializarea datelor structurate.

Structura datelor, structurate ca un mesaj, se defineste intr-un fisier cu extensia .proto. Conventiile sunt foarte putine: tip data, ordinea in cadrul structurii (mesajului), scalar sau lista (array, colectie). Fiecare mesaj este o inregistrare logica de informatii ce contine o serie de perechi "nume-valoare", perechi numite "fields". Aceasta structura este compilata (compilator protoc) si se genereaza clase in limbajul specificat de noi.
Exemplu:

```
syntax = "proto3";
option csharp_namespace = "GrpcServiceEfCore";

// ClientService service definition.
service ClientServiceRPC {
  rpc GetClients(msgEmpty) returns (msgClients){};
  rpc InsertClient(msgClient) returns (msgKeyId){};
  rpc GetClientById(msgKeyId) returns (msgClient){};
}
// Definitii mesaje
message msgEmpty {}

message msgClient{
    int32 client_Id = 1;
    string name = 2;
    string email = 3;
}
// Colectie (array)
message msgClients{
    repeated msgClient  wclient = 1;
}

message msgKeyId {
    int32 id = 1;
}
```

In exemplul de mai sus am definit patru mesaje si un serviciu, *ClientServiceRPC* ce contine trei metode. Observam tipul campului, numele campului si ordinea campului in cadrul mesajului.

Mesajul

```
message msgClients{
    repeated msgClient  wclient = 1;
}
```

defineste in fapt o colectie de mesaje (structuri) de tip msgClient.
Ce se genereaza?

```
  /// <summary>Field number for the "wclient" field.</summary>
   public const int WclientFieldNumber = 1;
   private static readonly pb::FieldCodec<global::GrpcServiceEfCore.msgClient>
             _repeated_wclient_codec =
         pb::FieldCodec.ForMessage(10,global::GrpcServiceEfCore.msgClient.Parser);
   private readonly pbc::RepeatedField<global::GrpcServiceEfCore.msgClient>
          wclient_ = new pbc::RepeatedField<global::GrpcServiceEfCore.msgClient>();
   [global::System.Diagnostics.DebuggerNonUserCodeAttribute]
   public pbc::RepeatedField<global::GrpcServiceEfCore.msgClient> Wclient
          {
                  get { return wclient_; }
          }
```

Pentru fiecare mesaj se genereaza o clasa. In cadrul mesajul de mai sus pentru campul ***wclient*** s-a definit proprietatea publica ***Wclient*** (capitalizare prima litera). Tipul *RepeatedField* este echivalentul pentru definirea unei colectii.
In exemplul de mai sus am definit un serviciu ce contine trei metode.

```
// ClientService service definition.
service ClientServiceRPC {
  rpc GetClients(msgEmpty) returns (msgClients){};
  rpc InsertClient(msgClient) returns (msgKeyId){};
  rpc GetClientById(msgKeyId) returns (msgClient){};
}
```

Metodele accepta mesaj si returneaza mesaj.
Compilatorul *protoc* creaza pentru acest serviciu clasa (observati sufixul *Base*):

```
/// <summary>Base class for server-side implementations
/// of ClientServiceRPC
</summary>
[grpc::BindServiceMethod(typeof(ClientServiceRPC),
                  "BindService")]
public abstract partial class ClientServiceRPCBase
{ ... }
```

Clasa ce va implementa metodele definite de “service *ClientServiceRPC*” va trebui sa fie derivata din *ClientServiceRPCBase*.

Protocol buffer – reprezinta un mecanism pentru serializarea datelor structurate.

Cum lucreaza? Vezi ce genereaza protoc.

Versiunea actuala pentru Protocol Buffers este 3, ***proto3***. Este in lucru si e posibil sa apara schimbari in API aferent acesteia.

Definirea unui tip de mesaj. Vezi exemplele de mai sus.
Se specifica prima data versiunea pentru protocol, apoi spatiul de nume si in continuare interfata pentru serviciu si structurile mesajelor.

Specifying Field Types

- Scalari – 0..1 valori
- Enumerari.
- Colectii : Repeated. 0..n valori.

Atribuire numere la campuri

Fiecare camp are un numar unic, numar ce identifica campul in cadrul mesajului.
Intervalull [19000-19999] este rezervat.
Numerele pot fi in intervalul [1, 536870911 = 2^{29} - 1] cu exceptia intervalului indicat mai sus.
Se recomanda folosirea numerelor din intervalul [1,15] pentru ca caestea se reprezinta pe un byte.

Specifying Field Rules

- singular
- repeated

Un fisier proto poate contine mai multe mesaje.

Reserved Fields

Folosit pentru a pastra compatibilitatea cu mesaje vechi folosite in alte aplicatii.

Cuvant rezervat; **reserved**.

```
message Foo {
  reserved 2, 15, 9 to 11;
  reserved "foo", "bar";
}
```

Scalar Value Types

Am selectat pentru tipul din C#.

.proto Type	Notes	C# Type
double		double
float		float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	long
uint32	Uses variable-length encoding.	uint
uint64	Uses variable-length encoding.	ulong
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	long
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	ulong
sfixed32	Always four bytes.	int
sfixed64	Always eight bytes.	long
bool		bool
string	A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than 2^{32}.	string

bytes	May contain any arbitrary sequence of bytes no longer than 2^{32}.	ByteString

Valori implicite

Se aplica atunci cand nu se atribuie valori campului respectiv si depind de tipul campului.

- For strings, the default value is the empty string.
- For bytes, the default value is empty bytes.
- For bools, the default value is false.
- For numeric types, the default value is zero.
- For enums, the default value is the **first defined enum value**, which must be 0.
- For message fields, the field is not set. Its exact value is language-dependent. See the generated code guide for details.

The default value for repeated fields is empty (generally an empty list in the appropriate language).

Nu exista posibilitatea de a determina daca valoarea unui camp este cea implicita sau nu.
Cand un camp nu este completat cu o valoare, nu se transmite prin retea valoarea implicita.
Valoarea implicita va fi completata la primirea mesajului (server / client).

Enumerari

Pot fi definite in corpul unui mesaj sau in exterior.
Analizam exemplul de mai jos.

```
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;
}
```

Definitia unui enum trebuie sa contina o contanta ce are valoarea 0 (zero), si aceasta este valoarea implicita in cazul unui enum.

Primul element trebuie sa aiba valoarea zero, probleme de compatibilitate cu alte versiuni.

You can define aliases by assigning the same value to different enum constants. To do this you need to set the allow_alias option to true, otherwise the protocol compiler will generate an error message when aliases are found.

```
enum EnumAllowingAlias {
  option allow_alias = true;
  UNKNOWN = 0;
  STARTED = 1;
  RUNNING = 1;
}
enum EnumNotAllowingAlias {
  UNKNOWN = 0;
  STARTED = 1;
  // RUNNING = 1;  // Uncommenting this line will cause a compile error inside Google and a warning message outside.
}
```

Importing Definitions

```
import "myproject/other_protos.proto";
```

```
// new.proto
// All definitions are moved here
// old.proto
// This is the proto that all clients are importing.
import public "new.proto";
import "other.proto";
// client.proto
import "old.proto";
// You use definitions from old.proto and new.proto, but not other.proto
```

Updating A Message Type

Maps

Daca dorim sa cream o asociere, pereche cheie-valoare, putem folosi map.

```
map<key_type, value_type> map_field = N;
```

... unde key_type poate fi orice tip integral (scalar cu exceptia lui floating points si bytes). Enum nu e permis de asemenea.
value_type poate fi orice cu exceptia unui alt map.

Exemplu:

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

- Map fields nu pot fi *repeated*.
- Ordinea din map nu este stabilita. Nu o putem folosi in regasirea sau transmiterea mesajelor.

Sintaxa map este echivalenta cu:

```
message MapFieldEntry {
  key_type key = 1;
  value_type value = 2;
}

repeated MapFieldEntry map_field = N;
```

Packages

Prevenire coliziune de nume.

```
package foo.bar;
message Open { ... }
```

```
message Foo {
  ...
  foo.bar.Open open = 1;
  ...
}
```

In **C#** the package is used as the namespace after converting to PascalCase, unless you explicitly provide an option csharp_namespace in your .proto file. For example, Open would be in the namespace Foo.Bar.

Definirea serviciilor

Interfata serviciului se defineste in fisierul .proto.
Compilatorul protoc va genera o clasa ce va avea sufixul **Base**. Pentru fiecare mesaj genereaza o alta clasa. Vezi partea de inceput a cursului.

```
syntax = "proto3";
```

```
option csharp_namespace = "GrpcServiceEfCore";

// ClientService service definition.
service ClientServiceRPC {
  rpc GetClients(msgEmpty) returns (msgClients){};
  rpc InsertClient(msgClient) returns (msgKeyId){};
  rpc GetClientById(msgKeyId) returns (msgClient){};
}
// Definitii mesaje
message msgEmpty {}

message msgClient{
    int32 client_Id = 1;
    string name = 2;
    string email = 3;
}
// Colectie (array)
message msgClients{
    repeated msgClient  wclient = 1;
}

message msgKeyId {
    int32 id = 1;
}
```

JSON Mapping – [optional]

Proto3 supports a canonical encoding in JSON, making it easier to share data between systems. The encoding is described on a type-by-type basis in the table below.
If a value is missing in the JSON-encoded data or if its value is null, it will be interpreted as the appropriate default value when parsed into a protocol buffer. If a field has the default value in the protocol buffer, it will be omitted in the JSON-encoded data by default to save space. An implementation may provide options to emit fields with default values in the JSON-encoded output.

proto3	JSON	JSON example	Notes
message	object	{"fooBar": v,"g": null,…}	Generates JSON objects. Message field names are mapped to lowerCamelCase and become JSON object keys. If

			the json_name field option is specified, the specified value will be used as the key instead. Parsers accept both the lowerCamelCase name (or the one specified by the json_name option) and the original proto field name. null is an accepted value for all field types and treated as the default value of the corresponding field type.
enum	string	"FOO_BAR"	The name of the enum value as specified in proto is used. Parsers accept both enum names and integer values.
map<K,V>	object	{"k": v, …}	All keys are converted to strings.
repeated V	array	[v, …]	null is accepted as the empty list [].
bool	true, false	true, false	
string	string	"Hello World!"	
bytes	base64 string	"YWJjMTIzIT8kKiYoKSctPUB+"	JSON value will be the data encoded as a string using standard base64 encoding with paddings. Either standard or URL-safe base64 encoding with/without paddings are accepted.
int32, fixed32, uint32	number	1, -10, 0	JSON value will be a decimal number. Either numbers or strings are accepted.
int64, fixed64, uint64	string	"1", "-10"	JSON value will be a decimal string. Either numbers or strings are accepted.
float, double	number	1.1, -10.0, 0, "NaN", "Infinity"	JSON value will be a number or one of the special string values "NaN", "Infinity", and "-Infinity". Either numbers or strings are accepted. Exponent notation is also accepted.

Any	object	{"@type": "url", "f": v, … }	If the Any contains a value that has a special JSON mapping, it will be converted as follows: {"@type": xxx, "value": yyy}. Otherwise, the value will be converted into a JSON object, and the "@type" field will be inserted to indicate the actual data type.
Timestamp	string	"1972-01-01T10:00:20.021Z"	Uses RFC 3339, where generated output will always be Z-normalized and uses 0, 3, 6 or 9 fractional digits. Offsets other than "Z" are also accepted.
Duration	string	"1.000340012s", "1s"	Generated output always contains 0, 3, 6, or 9 fractional digits, depending on required precision, followed by the suffix "s". Accepted are any fractional digits (also none) as long as they fit into nano-seconds precision and the suffix "s" is required.
Struct	object	{ … }	Any JSON object. See struct.proto.
Wrapper types	various types	2, "2", "foo", true, "true", null, 0 , …	Wrappers use the same representation in JSON as the wrapped primitive type, except that null is allowed and preserved during data conversion and transfer.
FieldMask	string	"f.fooBar,h"	See field_mask.proto.
ListValue	array	[foo, bar, …]	
Value	value		Any JSON value
NullValue	null		JSON null
Empty	object	{}	An empty JSON object

JSON options

A proto3 JSON implementation may provide the following options:

- **Emit fields with default values**: Fields with default values are omitted by default in proto3 JSON output. An implementation may provide an option to override this behavior and output fields with their default values.
- **Ignore unknown fields**: Proto3 JSON parser should reject unknown fields by default but may provide an option to ignore unknown fields in parsing.
- **Use proto field name instead of lowerCamelCase name**: By default proto3 JSON printer should convert the field name to lowerCamelCase and use that as the JSON name. An implementation may provide an option to use proto field name as the JSON name instead. Proto3 JSON parsers are required to accept both the converted lowerCamelCase name and the proto field name.
- **Emit enum values as integers instead of strings**: The name of an enum value is used by default in JSON output. An option may be provided to use the numeric value of the enum value instead.

Options

Individual declarations in a .proto file can be annotated with a number of *options*. Options do not change the overall meaning of a declaration, but may affect the way it is handled in a particular context. The complete list of available options is defined in google/protobuf/descriptor.proto. Some options are file-level options, meaning they should be written at the top-level scope, not inside any message, enum, or service definition. Some options are message-level options, meaning they should be written inside message definitions. Some options are field-level options, meaning they should be written inside field definitions. Options can also be written on enum types, enum values, service types, and service methods; however, no useful options currently exist for any of these.
Here are a few of the most commonly used options:

- java_package (file option): The package you want to use for your generated Java classes. If no explicit java_package option is given in the .proto file, then by default the proto package (specified using the "package" keyword in the .proto file) will be used. However, proto packages generally do not make good Java packages since proto packages are not expected to start with reverse domain names. If not generating Java code, this option has no effect.
```
option java_package = "com.example.foo";
```
- java_multiple_files (file option): Causes top-level messages, enums, and services to be defined at the package level, rather than inside an outer class named after the .proto file.
```
option java_multiple_files = true;
```
- java_outer_classname (file option): The class name for the outermost Java class (and hence the file name) you want to generate. If no explicit java_outer_classname is specified in the .proto file, the class name will be constructed by converting the .proto file name to camel-case (so foo_bar.proto becomes FooBar.java). If not generating Java code, this option has no effect.
```
option java_outer_classname = "Ponycopter";
```

- optimize_for (file option): Can be set to SPEED, CODE_SIZE, or LITE_RUNTIME. This affects the C++ and Java code generators (and possibly third-party generators) in the following ways:
- SPEED (default): The protocol buffer compiler will generate code for serializing, parsing, and performing other common operations on your message types. This code is highly optimized.
- CODE_SIZE: The protocol buffer compiler will generate minimal classes and will rely on shared, reflection-based code to implement serialialization, parsing, and various other operations. The generated code will thus be much smaller than with SPEED, but operations will be slower. Classes will still implement exactly the same public API as they do in SPEED mode. This mode is most useful in apps that contain a very large number .proto files and do not need all of them to be blindingly fast.
- LITE_RUNTIME: The protocol buffer compiler will generate classes that depend only on the "lite" runtime library (libprotobuf-lite instead of libprotobuf). The lite runtime is much smaller than the full library (around an order of magnitude smaller) but omits certain features like descriptors and reflection. This is particularly useful for apps running on constrained platforms like mobile phones. The compiler will still generate fast implementations of all methods as it does in SPEED mode. Generated classes will only implement the MessageLite interface in each language, which provides only a subset of the methods of the full Message interface.

```
option optimize_for = CODE_SIZE;
```

- cc_enable_arenas (file option): Enables arena allocation for C++ generated code.
- objc_class_prefix (file option): Sets the Objective-C class prefix which is prepended to all Objective-C generated classes and enums from this .proto. There is no default. You should use prefixes that are between 3-5 uppercase characters as recommended by Apple. Note that all 2 letter prefixes are reserved by Apple.
- deprecated (field option): If set to true, indicates that the field is deprecated and should not be used by new code. In most languages this has no actual effect. In Java, this becomes a @Deprecated annotation. In the future, other language-specific code generators may generate deprecation annotations on the field's accessors, which will in turn cause a warning to be emitted when compiling code which attempts to use the field. If the field is not used by anyone and you want to prevent new users from using it, consider replacing the field declaration with a reserved statement.

```
int32 old_field = 6 [deprecated=true];
```

Custom Options

Protocol Buffers also allows you to define and use your own options. This is an **advanced feature** which most people don't need. If you do think you need to create your own options, see the Proto2 Language Guide for details. Note that creating custom options uses extensions, which are permitted only for custom options in proto3.

gRPC Concepts

An introduction to key gRPC concepts, with an overview of gRPC architecture and RPC life cycle.

Not familiar with gRPC? First read What is gRPC?. For language-specific details, see the Quick Start, tutorial, and reference documentation for your language of choice.

[End Optional]

Definire serviciu

Reluam exemplul de la inceputul cursului.

```
syntax = "proto3";
option csharp_namespace = "GrpcServiceEfCore";

// ClientService service definition.
service ClientServiceRPC {
  rpc GetClients(msgEmpty) returns (msgClients){};
  rpc InsertClient(msgClient) returns (msgKeyId){};
  rpc GetClientById(msgKeyId) returns (msgClient){};
}
// Definitii mesaje
message msgEmpty {}

message msgClient{
    int32 client_Id = 1;
    string name = 2;
    string email = 3;
}
// Colectie (array)
message msgClients{
    repeated msgClient  wclient = 1;
}

message msgKeyId {
    int32 id = 1;
}
```

Care sunt metodele?
Parametrii metodelor?
Ce se genereaza?

Tipuri de metode in serviciu [Tipuri de cereri]

1. **Unary RPC**s where the client sends a single request to the server and gets a single response back, just like a normal function call. [Request/Response – transmit/primesc un tip de data. **{0..1}/{0..1}**]

   ✓ **rpc** SayHello(HelloRequest) **returns** (HelloResponse);

2. **Server streaming** RPCs where the client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages. gRPC guarantees message ordering within an individual RPC call.[**{0..1} / {0..n}**]

   ✓ **rpc** LotsOfReplies(HelloRequest) **returns** (**stream** HelloResponse);

3. **Client streaming** RPCs where the client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them and return its response. Again gRPC guarantees message ordering within an individual RPC call.{**{0..n} / {0..1}**]

   ✓ **rpc** LotsOfGreetings(**stream** HelloRequest) **returns** (HelloResponse);

4. **Bidirectional streaming** (**comunicare bidirectionala, duplex**) RPCs where both sides send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like: for example, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes. The order of messages in each stream is preserved. [**{0..n} / {0..m}**]

   ✓ **rpc** BidiHello(**stream** HelloRequest) **returns** (**stream** HelloResponse);

Synchronous vs. asynchronous

RPC life cycle

In this section, you'll take a closer look at what happens when a gRPC client calls a gRPC server method. For complete implementation details, see the language-specific pages.

Unary RPC

First consider the simplest type of RPC where the client sends a single request and gets back a single response.

1. Once the client calls a stub method, the server is notified that the RPC has been invoked with the client's metadata for this call, the method name, and the specified deadline if applicable.
2. The server can then either send back its own initial metadata (which must be sent before any response) straight away, or wait for the client's request message. Which happens first, is application-specific.
3. Once the server has the client's request message, it does whatever work is necessary to create and populate a response. The response is then returned (if successful) to the client together with status details (status code and optional status message) and optional trailing metadata.
4. If the response status is OK, then the client gets the response, which completes the call on the client side.

Server streaming RPC

A server-streaming RPC is similar to a unary RPC, except that the server returns a stream of messages in response to a client's request. After sending all its messages, the server's status details (status code and optional status message) and optional trailing metadata are sent to the client. This completes processing on the server side. The client completes once it has all the server's messages.

Client streaming RPC

A client-streaming RPC is similar to a unary RPC, except that the client sends a stream of messages to the server instead of a single message. The server responds with a single message (along with its status details and optional trailing metadata), typically but not necessarily after it has received all the client's messages.

Bidirectional streaming RPC

In a bidirectional streaming RPC, the call is initiated by the client invoking the method and the server receiving the client metadata, method name, and deadline. The server can choose to send back its initial metadata or wait for the client to start streaming messages.
Client- and server-side stream processing is application specific. Since the two streams are independent, the client and server can read and write messages in any order. For example, a server can wait until it has received all of a client's messages before writing its messages, or the server and client can play "ping-pong" – the server gets a request, then sends back a response, then the client sends another request based on the response, and so on.

Deadlines/Timeouts

gRPC allows clients to specify how long they are willing to wait for an RPC to complete before the RPC is terminated with a DEADLINE_EXCEEDED error. On the server side, the server can query to see if a particular RPC has timed out, or how much time is left to complete the RPC. Specifying a deadline or timeout is language specific: some language APIs work in terms of timeouts (durations of time), and some language APIs work in terms of a deadline (a fixed point in time) and may or maynot have a default deadline.

RPC termination

In gRPC, both the client and server make independent and local determinations of the success of the call, and their conclusions may not match. This means that, for example, you could have an RPC that finishes successfully on the server side ("I have sent all my responses!") but fails on the client side ("The responses arrived after my deadline!"). It's also possible for a server to decide to complete before a client has sent all its requests.

Cancelling an RPC

Either the client or the server can cancel an RPC at any time. A cancellation terminates the RPC immediately so that no further work is done.
Warning
Changes made before a cancellation are not rolled back.

Metadata

Metadata is information about a particular RPC call (such as authentication details) in the form of a list of key-value pairs, where the keys are strings and the values are typically strings, but can be binary data. Metadata is opaque to gRPC itself - it lets the client provide information associated with the call to the server and vice versa.
Access to metadata is language dependent.

Channels

A gRPC channel provides a connection to a gRPC server on a specified host and port. It is used when creating a client stub. Clients can specify channel arguments to modify gRPC's default behaviour, such as switching message compression on or off. A channel has state, including connected and idle.
How gRPC deals with closing a channel is language dependent. Some languages also permit querying channel state.