30/10/2022

gRPC. A high performance, open source universal RPC framework

Activity 02. DC applications. Report

Svyatoslav Gudymyak Grygorak
55264455D

INDEX

What is RPC? **2**

Introduction to RPC 2

Steps performed during an RPC call 2

Advantages of RPC 2

What is Client Stub? **3**

What are Protocol-Buffers? **3**

Introduction to Protocol-Buffers 3

Why Protocol-Buffers? 3

Who uses Protocol Buffers? 3

Examples 3

What is gRPC? **5**

Introduction to gRPC 5

Principal characteristics 5

When is gRPC used? 6

Challenges of running gRPC services 6

Example of an gRPC implementation **8**

Server side gRPC 9

Client side 10

Alternative solutions, benefits and disadvantages **12**

Introduction 12

REST advantages 12

Open API advantages and disadvantages 13

Benefits of gRPC 13

Downsides of gRPC 13

Conclusions **15**

Links **16**

What is RPC?

Introduction to RPC

Remote Procedure call [0] known as **RPC** is a powerful technique used while constructing **distributed**, **client-server based** applications. The **RPC** is based on the traditional procedure calling, allowing to have both, call procedure and calling procedure not in the same address space. Being in the same or different systems, being connected through a network. Please see the example of the communication below **[1]**.

Steps performed during an RPC call

The **Remote Procedure call** follows the steps **[2]** described below.

1 - The client calls a client stub [3]. The call is a local procedure call, with parameters pushed in the normal way.

2 - The client stub packs ("marshalls") [4] the parameters into a message, this procedure includes converting the representation of the parameters into a standard format and copying each one of the parameters into the message.

3 - The client stub then addresses the message to the transport layer and the message is sent to the remote server machine.

4 - The server's transportation layer passes the message to a server stub [3] which unpacks ("unmashalls") [5] the parameters from the message.

5 - The server stub finally calls the desired server procedure using the regular procedure call mechanism.

6 - The reply from the server side follows the same steps in the reverse order.

Advantages of RPC

Some of the most relevant advantages of using RPC are the following ones [6]:

1 - **RPC** provides **abstraction**, hiding to the user i.e the Network communication process.

2 - **RPC** usually omits many of the protocol layers in order to have better performance.

3 - **RPC** allows the application to be used not only in the local environment, but also in a **distributed environment**.

4 - **RPC** provides an **effortless** code redeveloping and rewriting.

5- In addition **RPC** also supports **Process-oriented** and **Thread-oriented** models.

What is Client Stub?

The **stub** [16] in distributed computing is a piece of code that converts parameters passed between client and server during a Remote Procedure Call.
The stub used in the gRPC is the local object which implements the same methods as the service. the client is able to use this methods on the local object, wrapping the parameters for the call in the appropriate protocol buffer message type - gRPC looks after sending the request(s) to the server and returning protocol buffer response(s).

What are Protocol-Buffers?

Introduction to Protocol-Buffers

Protocol buffers provide a **language-neutral**, **platform-neutral** extensible mechanism for **serializing structured data** in a **forward** and **backward** compatible **way** in a more **efficient** way compared to the standard **JSON** case as the data is **generated** in **less** amount of **time** and it **weighs less,** resulting in a more efficient way of serializing structured data.

Why Protocol-Buffers?

Some of the advantages of using Protocol-buffers are:
- The **compact data storage** provided when using **protocol-buffers**.
- Fast **parsing** of data **backwards** and **forwards**. Which allows updating the Proto definitions without Updating code.
- The **availability** of **usage** in some of the **most used programming languages** such as C++, C#, Java, Kotlin, Objective-C, PHP, Python, Go, Dart and Ruby.
- **Optimized functionality** provided though **automatically generated classes**.

Who uses Protocol Buffers?

Some of the most relevant Protocol Buffer uses are gRPC, Google Cloud Platform and Envoy Proxy.

Examples

In order to use Protocol buffers you need to create a .proto file which defines the data structure and a possible example of a .proto file would be:

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

“””

Compiling the .proto file would result in the following Builder which can be used to create new instances as in the following Java code, but also same result would be when compiling for any of supported languages:

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

Now Protocol Buffer generated classes can be used in order to serialize, share & deserialize the data.

What is gRPC?

Introduction to gRPC

Google's Remote Procedure Call is a technology for implementing RPC APIs and that uses **HTTP 2.0** [7] as its underlying **transport protocol**, **Protocol buffers** [8] as the **Interface description language** (**IDL**) [9].
This technology was initially used and developed by **Google**, which has used a single general-purpose **RPC infrastructure** called **Stubby** in order to connect a big amount of microservices.
In **2015** Google decided to build the next version of **Stubby** and made it open source.
Nowadays the **gRPC** protocol is the second most used protocol over the API communication, just after **openAPI** [10].

Principal characteristics

Some of the most relevant characteristics gRPC has are the ones described below:
- A key feature of **gRPC** when talking about **inter-process communication** is the **transparency principle**. The communication between client and server using gRPC appears to be identical to a local communication, as the Remote calls made using gRPC interact so closely and smoothly even at great distances.
- **Multiplexing** and **bidirectional streaming** is also one of the most relevant **gRPC** features thanks to the **HTTP 2.0** protocol, allowing to have both server and client communications done at the same time, both client and server streams operate independently, so the message delivery time is reduced and possible error messages due to collisions are avoided. This functionality also allows us to send messages in any sequence independently for both client and server.
- High **Scalability** is one of the most important features for all APIs and using **gRPC** allows to realize **multiple communications simultaneously** without affecting the **performance** of each communications as each one of the communications is performed independently in both sides, this allows APIs being able to **scale** as the communication needs increase without having to perform any change to the communication protocol.
- The **gRPC** technology also provides **multi-language support** which provides **flexibility** to the developer in order to implement services in whatever language and framework is desired. Some of the most relevant programming languages supported by gRPC are **C#, Java**, **Go**, **Node.js**, **Python**, **PHP** and **Ruby**.
- The gRPC Framework also provides both **asynchronous** and **synchronous** communications, and can be used in order to provide a synchronous RPC communication, waiting on the client/server response or having an asynchronous communication and this option can be set in the configuration for each gRPC independently.
- As in any real-world scenario, **security** is essential, **gRPC** is not the exception, and has been designed in order to use multiple authentication mechanisms, such as [11]:
    - **SSL/TLS** which is promoted to be used for both, authentication encryption for the server and data exchange encryption between both client and server.

- **ALTS** [12] is also a supported transport **security** mechanism, when running the application on the **Google Cloud Platform** (GCP).
- **Token-based authentication with Google** is also supported as **gRPC** provides a generic mechanism to attach metadata based credential to both requests and responses. This mechanism must be used with SSL/TLS as Google and most language implementations do not allow sending credentials on an unencrypted channel.

- **Performance** of **gRPC** has been evaluated resulting in a **10x faster performance** than an usual **API REST+JSON call**, thanks to the **protobuf** and **HTTP 2.0** as can be seen in the following documentation [13]. This performance increase is due to two main factors, the first one is the usage of the **protobuf** which **serializes the messages** on the server and client sides efficiently, resulting in **small and compact message payloads**. On the other hand **HTTP 2.0 scales** up the performance ranking though **push**, **multiplexing** and **header compression**. The push functionality pushes content from the server before being requested,while multiplexing removes head-of-line blocking and the header compression used by HTTP 2.0 leads to smaller messages which lead to a faster loading.

When is gRPC used?

The gRPC Framework is used in a lot of situations, but the most common ones are the following ones:

- When talking about **large-scale microservices** the first issue we encounter, and that leads to a mis-performance of the microservices is the **Network Latency** which is **crucial** when talking about a **large-scale microservices scenario,** weather this characteristic is essential for this microservices, there are more factors that lead to choose gRPC instead of an alternative Framework such as multi-language implementation, Protobuf and HTTP 2.0 benefits.
- real-time communication is another factor to consider, as having gRPC communication benefits is really important for sharing information between customer and server in the most efficient ways.
- As multi-language environments are supported by gRPC and the simple implementation for each of the supported languages leads to taking into consideration the usage of the gRPC Framework in this occasion.

Challenges of running gRPC services

One of the most relevant challenges when trying to implement **gRPC** to your application is the fact that gRPC is a little bit difficult to implement when trying to use load balancing when running on **Kubernetes** clusters, which is one of the most used systems for **automatic deployment**, **scaling** and **management** of **containerized applications** as having automatic scaling or any other implementation which require a load balancers to be used.

In order to overcome these problems the gRPC blog [14] provides some alternatives which depend on the setup of the application we are implementing the gRPC.

Some of the recommended **load balancers** would be:
- Thick client-side load balancing or Client side LB with ZooKeeper/Etcd/Consul/Eureka for very high traffic between clients and server, and where Clients can be trusted.
- Proxy Load Balancing, L3/L4 LB with GCLB (if using GCP), L3/L4 LB with haproxy, Nginx and If need session stickiness - L7 LB with Envoy as proxy for a traditional Setup, with many clients connecting behind a proxy which need trust boundary between server and client.
- Look-aside Load Balancing and Client-side LB using gRPC-LB protocol when working with microservices with high performance requirements (low latency with high traffic) when the client can be untrusted.
- Finally Service Mesh or built-in LB with Istio or Envoy when Existing service-mesh like setup is using Linkerd or Istio.

Example of an gRPC implementation

This section provides some examples of an **gRPC implementation** using **Python**.
- The first step when using gRPC would be **defining** the **gRPC servicer** and the **request** and **response methods** using the **protocol buffers**.

  - The service definition would look like this on your **.proto** file:

    “””
    ```
    service RouteGuide {
     // (Method definitions not shown)
    }
    ```
    “””

  - After **defining** the **service** you should define **RPC methods**, specifying their **request** and **response types**.
  In this case **gRPC allows** you to define **four kinds** of **service methods** explained below:

    1 - A **simple RPC** where the **client sends** a **request to the server** and **waits** for the **response** to come back.
    Example below:
    “””
    ```
    rpc GetFeature(Point) returns (Feature) {}
    ```
    “””

    2 - A **response-streaming RPC** 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 stream until there are no more messages.
    Example below:
    “””
    ```
    rpc ListFeatures(Rectangle) returns (stream Feature) {}
    ```
    “””

    3 - A **request-streaming RPC** where the client **sends a stream** to the server and the server **keeps reading the stream** until there are **no more messages** available and then the client waits for the **Server response** when finishing reading all the messages.
    Example below:
    “””
    ```
    rpc RecordRoute(stream Point) returns (RouteSummary) {}
    ```
    “””

    4 -
    A **bidirectional-stream** where the client and customer send both data though read-write streams which are independent so the client and server can read and write messages in the order they like the most.
    “””
    ```
    rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
    ```
    “””

The .proto file alos contains the protocol buffer message definition for all the requests and response types used in our service methods. For example this is the “**Point**” message type:
“”””

```
// Points are represented as latitude-longitude pairs in the E7 representation
// (degrees multiplied by 10**7 and rounded to the nearest integer).
// Latitudes should be in the range +/- 90 degrees and longitude should be in
// the range +/- 180 degrees (inclusive).
message Point {
  int32 latitude = 1;
  int32 longitude = 2;
}
```
“”””

In order to generate both client and server code we need to use the grpcio-tools application which generates two code files, for example““file_name”_pb2.py” and ““file_name”_pb2_grp.pyc” files. The obtained files contain the following codes:
- The classes for the messages defined in “file_name”.proto
- The classes for service defined in “file_name”.proto .
- A function for the service defined in “file_name”.proto .

Server side gRPC

The GetFeature function which gets a “Point” form the client and returns the corresponding feature information from it’s database in as Feature can be seen below:
“””
```
def GetFeature(self, request, context):
  feature = get_feature(self.db, request)
  if feature is None:
    return route_guide_pb2.Feature(name="", location=request)
  else:
    return feature
```
“””

In this case the method gets passed a ”file_name”_pb2.Point request for the RPC and a grpc.ServiceContext object that provides the RPC-specific information such as timeout limits and it returns a “file_name”_pb2.Feature response.

The ListFeatures function which is a response-streaming RPC that sends multiple features to the client.
“””
```
def ListFeatures(self, request, context):
  left = min(request.lo.longitude, request.hi.longitude)
  right = max(request.lo.longitude, request.hi.longitude)
  top = max(request.lo.latitude, request.hi.latitude)
  bottom = min(request.lo.latitude, request.hi.latitude)
  for feature in self.db:
    if (feature.location.longitude >= left and
        feature.location.longitude <= right and
        feature.location.latitude >= bottom and
        feature.location.latitude <= top):
      yield feature
```
“””

In this case the message is a “file_name”_pb2.Rectangle within which the client wants to find a Feature. In this case instead of returning a single response, the server yields zero or more responses.

When “file_name” methods are set, the next step would be stating the gRPC server, in order to allow the clients to use the actual service.
“””
```
def serve():
  server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
  route_guide_pb2_grpc.add_RouteGuideServicer_to_server(
      RouteGuideServicer(), server)
  server.add_insecure_port('[::]:50051')
  server.start()
  server.wait_for_termination()
```
“””
This is a code snippet of the gRPC server start.

Please see the following documentation [17] in order to get more information in regards to server side code snippets.

Client side

In order to create the client the following steps should be followed.

Firstly in order to be able to call service methods, we should create a stub, in this case we use our “file_name”_pb2_grpc RouteGuideStub generated from our .proto file.
“””
```
channel = grpc.insecure_channel('localhost:50051')
stub = route_guide_pb2_grpc.RouteGuideStub(channel)
```
“””
For single response RPC methods Python supports both asynchronous and synchronous control flow semantics. For response-streaming RPC methods, calls immediately return an iterator of response values.
The GetFeature function provides which is nearly as a straightforward as calling a local method. After calling this method the RPC waits for the server response and returns the response or raises the exception if needed.
Please see the example below:
“””
```
feature = stub.GetFeature(point)
```
“””
The asynchronous version of the GetFeature function is shown below:
“””
```
feature_future = stub.GetFeature.future(point)
feature = feature_future.result()
```
“””
Calling the ListFeatures feature is similar to working with sequence types as can bee seen below:

“””
for feature in stub.ListFeatures(rectangle):
“””
Please see the following documentation [18] in order to get more information in regards to Client side code snippets.

Alternative solutions, benefits and disadvantages

Introduction

There a three primary ways of using HTTP for APIs, the first one and the least commonly used one is the REST [19] API model, which lets the client use Server side constructed URLs in order to get some information and obtain a following url which is called in order to get more and more information in this case clients do not require to understand the url format, and only form urls from relative and base URLs.
The second most common model for HTTP is gRPC which we have talked about in this report and which uses HTTP 2.0 under the covers, but which is not exposed to the API designer. gRPC-generated stubs and skeletons hide HTTP from both client and server so nobody has to worry about how RPC is being mapped to HTTP, they only have to learn gRPC.
The last one but probably the most popular one is OpenAPI [20] whose most characteristic signature is that clients use the API by constructing URLs from other information. In this case we can observe that OpenAPI is not a REST API and it requires the client to have detailed knowledge of the format of the URLs they use in the requests and to construct URLs that conform to that format from other information. Performing the opposite to an REST API where the clients are completely blind to the formats of the URLs.

REST advantages

The most popular advantages which REST presents are the same World Wide Web itself presents, which are stability, uniformity and universality and one of the most relevant characteristics REST uses is the entity-orientation in the HTTP/REST model, which provides models which are simpler, more regular easier to understand and more stable compared to the RPC models.
Being an entity oriented model this provides REST an overall organization for the system behavior, which contrasts with the RPC APIs which tend to grow organically as one procedure after another are added, each one implementing an action the system can perform.
The best example when using REST results in an advantage for example would be in online shopping, with its products, carts, orders, accounts, and so on. If that capability were expressed using only RPC procedures, it would result in a long, unstructured list of procedures for browsing catalogs of products, adding them to carts, checking out, tracking deliveries, and returning products. This is quickly solved when using REST for its entity-oriented standard behaviors.

Open API advantages and disadvantages

Open API success is founded in two principal characteristics. The first one is that the OpenAPI model is very similar to the standard RPC model with which most programmers are familiar and comfortable with. The second reason could be that it allows programmers to customly map RPC concepts to HTTP requests. This second characteristic brings both benefits and disadvantages as the most notable benefit is that clients are able to use the API just using standard HTTP technologies. This is vital for public APIs which are required to be accessible from almost every programming language and environment without needing to adopt any additional technology. This comes directly related to the main disadvantage OpenAPI has, which is the significant effort it takes to design the HTTP details.

Benefits of gRPC

The gRPC Framework expresses the RPC API in an interface description language (IDL) and so it benefits from a long tradition of RPC IDLs that include DCE IDL [21] and Cobra IDL [22]. The gRPC IDL provides a simple and direct way of defining remote procedure calls.
In addition gRPC uses HTTP 2.0 under the covers, but does not expose any HTTP 2.0 to the API designer or API user as gRPC has already decided how the RPC is mapped to HTTP in order to make life easier for API designers and clients. Contrasting with for example OpenAPI which requires the designer to specify the details on how RPC is expressed on top of the HTTP for their specific API, and the client has to learn that details.
One of the most attractive characteristics gRPC has is that it is very good at generating client-side programming libraries that are intuitive for programers to use and execute efficiently.
Another characteristic of gRPC is the good performance as it uses binary payload which is efficient to be created and to be parsed as it exploits HTTP 2.0 for efficient management of connections.
The last benefit which involves gRPC is that it implements the whole HTTP protocol for both client and server, forcing the client and server to adopt special software that implements gRPC protocol.

Downsides of gRPC

Some of gRPC downsides are the following ones.
The first and most relevant gRPC downside is that in contrast to ordinary API calls which can be done as easy as typing an URL into a browser and which do not require any additional technology, gRPC requires special software on both client and server. gRPC-generated code has to be implemented into the client and server build process. This process may be unusual and onerous in some occasions to someone who is used to working with dynamic languages such as Javascript or Python where the build process, at least on the development machines, may be non-existing.
Google Cloud Endpoints [23] solve this problem allowing gRPC APIs being accessed via HTTP and JSON without any special software, but still this does not fit everyone's needs.

Another downside of gRPC is that it does not allow crawling the entirety of an gRPC API, as it could if it was an REST API, as RPC gives each entity type a different API that requires custom software or metadata to use it. But usually this isn’t critical although it can be useful.
Another important downside that should be taken into consideration is that gRPC does not defines a standard mechanism to prevent data loss, when two clients try to update the same resource at the same time, if you use gRPC you will have to invent your own mechanism to handle this cases while HTTP defines standard Etag and If-Match headers for this purpose and most of the HTTP APIs use these headers.

Finally gRPC also does not implement a mechanism to perform partial updates, so you will likely have to create your own while HTTP provides the PATCH for partial updates, but does not say what a patch should look like or how to apply it.

Conclusions

In conclusion I would like to say that I trust gRPC is a very interesting Framework that you should take into consideration when starting developing a new API.

This Framework still has a lot of disadvantages, it provides some features for which particular usages may result in a great increase of the productivity and may reduce some issues which are generally caused by using standard HTTP APIs.

I also consider that when you have never used this API it may result a bit different and somehow tricky to implement, this is far from being difficult to be implemented and the only place you may encounter some problems is the Load Balancing side which can lead to some errors.

But still I think that with the appropriate approach it won’t be a problem due to the big amount of possible alternatives provided.

Having the RPC directly mapped to the HTTP which does not happen in the OpenAPI may result in some occasions in a limitation, but I believe this sometimes also leads to make it more difficult to be implemented, so in this case it would result in a more effortless approach to this problem.

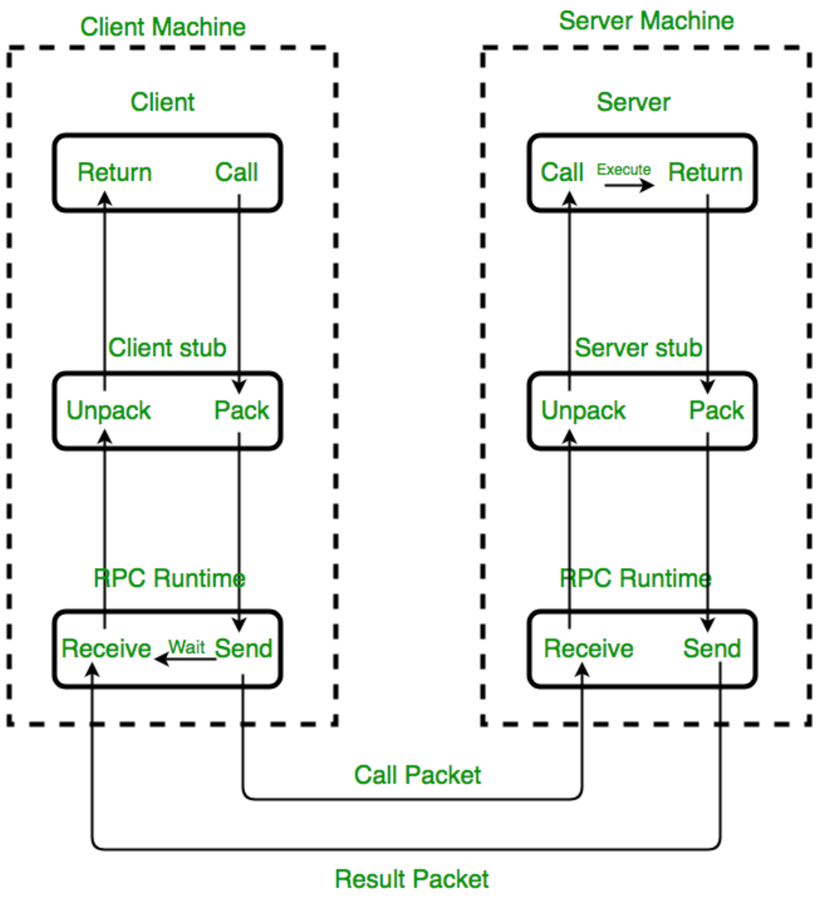
One of REST’s key features which is entity-oriented style can also be partially obtained while using gRPC if you organize your API in an entity-oriented style and standardize the names of the procedures.

Please bear in mind that you can still use gRPC without forcing your clients to use it by using Cloud Endpoints which is not a problem when the API is internal and you can choose both server and client technologies.

As with most of the design challenges faced when trying to develop an API, there are many particular factors to consider, but still I think this Report has provided some light on the gRPC, the technologies it uses and the benefits and disadvantages compared to other similar technologies.

Links

[0] - https://en.wikipedia.org/wiki/Remote_procedure_call
[1] -
https://media.geeksforgeeks.org/wp-content/uploads/operating-system-remote-call-procedure-working.png



Implementation of RPC mechanism

[2] -
https://en.wikipedia.org/wiki/Remote_procedure_call#:~:text=low%2Dlevel%20subsystems.-,Sequence%20of%20events,-%5Bedit%5D
[3] - https://en.wikipedia.org/wiki/Stub_(distributed_computing)
[4] - https://en.wikipedia.org/wiki/Marshalling_(computer_science)
[5] -
https://en.wikipedia.org/wiki/Marshalling_(computer_science)#Unmarshalling:~:text=marshalling%20is%20called-,unmarshalling,-(or%20demarshalling%2C%20similar
[6] -
https://www.tutorialspoint.com/remote-procedure-call-rpc#:~:text=Advantages%20of%20Remote%20Procedure%20Call&text=The%20effort%20to%20re%2Dwrite,by%20RPC%20to%20improve%20performance.
[7] - https://es.wikipedia.org/wiki/HTTP/2
[8] - https://en.wikipedia.org/wiki/Protocol_Buffers

[9] - https://en.wikipedia.org/wiki/Interface_description_language
[10] - https://www.openapis.org/
[11] - https://grpc.io/docs/guides/auth/
[12] -
https://cloud.google.com/docs/security/encryption-in-transit/application-layer-transpo
rt-security
[13] - https://grpc.io/docs/guides/benchmarking/
[14] - https://grpc.io/blog/grpc-load-balancing/
[15] - https://developers.google.com/protocol-buffers/docs/overview
[16] - https://en.wikipedia.org/wiki/Stub_(distributed_computing)
[17] -
https://github.com/grpc/grpc/blob/v1.50.0/examples/python/route_guide/route_guid
e_server.py
[18] -
https://github.com/grpc/grpc/blob/v1.50.0/examples/python/route_guide/route_guid
e_client.py
[19] -https://en.wikipedia.org/wiki/Representational_state_transfer
[20] - https://www.openapis.org/
[21] -
http://odl.sysworks.biz/disk$axpdocsep022/network/dcev30/DEVELOP/apgstyle/Apgst
y12.htm
[22] - https://es.wikipedia.org/wiki/CORBA
[23] -https://cloud.google.com/endpoints/docs/grpc/transcoding