

# Microservices: Use of REST and gRPC over SOAP

Marcel Gredler  
Computer Science  
FH Technikum Wien  
Vienna, Austria  
se19m025@technikum-wien.at

Florencia Cavallin  
@Florecia (department)  
@Florecia (uni)  
@Florecia (city), Argentina  
fcavallin@itba.edu.ar

**Abstract**—Since the induction of Web Services into modern applications, their usage has grown. Together with them the general public was first introduced to the use of protocols like the Simple Object Access Protocol (SOAP) and Representational State Transfer (REST).

In more recent years a new trend started emerging in the development of applications, in which the application is split into a set of small services that use a communication protocol to interact with another. This trend is known as the Microservice Architecture and it favors the use of REST, a Message Bus or other stateless communication protocols over SOAP.

This paper gives an overview of the Microservice Architecture, explains why the use of REST is favored over SOAP and how the new gRPC Remote Procedure Calls (gRPC) framework may support them.

**Index Terms**—services, rest, grpc, microservices

## I. INTRODUCTION

Nowadays there exist many different paradigms and architectures that can be used to create applications. One of these architectural styles is the Service Oriented Architecture (SOA), in which the functionality is split between multiple services that may be run locally or be distributed around the network.

One methodology that heavily uses this paradigm and that is in widespread use, are web services [1]. Within the use of web services applications can benefit by reusing the functionality of already existing other services. Because of this it is possible to reduce the need of rewriting existing functionality into every application. Another benefit is the possibility of sharing information between all such web services. To achieve all these benefits, web services traditionally employ the Simple Object Access Protocol (SOAP) or Representation State Transfer (REST) protocols [1].

Another methodology that became widespread between 2010 and 2020 are microservices. Where web services focused on offering a full service to another application or user, microservices are built around the offering and using of capabilities [2]. Each capability is thereby built and offered as its own service and all communication is performed through an Application Programming Interface (API). A set of microservices can therefore build an application or be used to provide capabilities to other applications.

Since microservices are using lightweight protocols [2] to communicate with each other, a protocol like REST is preferred over SOAP. One of the biggest advantages of REST over SOAP is the smaller payload, as SOAP may require

the services to exchange ten times more bytes between each other in comparison to REST [1]. Since microservices are built around capabilities and interaction between each other, a smaller payload allows for better use of the available bandwidth. Additionally, changing the provisioning in SOAP may require changes on the clients, whereas REST does not [1]. This is another important advantage, as microservices are generally designed to be independently deployable [2], which would be broken, if the deployment of one service requires the deployment of another.

With the from 2015 slowly emerging HTTP/2 standard [3] another communication framework has been developed: gRPC Remote Procedure Calls (gRPC) [4]. While previously microservices only used REST, this new framework allows for the use of both gRPC and REST to interact with another. The gRPC framework thereby offers the ability to build so called gRPC REST Gateways [5]. These gateways are proxies for that expose the gRPC capability through REST. This means that other services are able to use them with gRPC - if they so support this too - or use the REST API. The latter is especially useful in networks where HTTP/2 traffic is not yet fully supported, if for example used loadbalancer only support HTTP/1.1 yet.

This paper gives an introduction of the mentioned protocols SOAP, REST and gRPC. Furthermore it explains what microservices are and how they compare to a classical service oriented architecture. Lastly, the document explains the benefit of using HTTP/2 and gRPC in conjunction with a microservice architecture.

To achieve this goal, the paper is structured as follows: Sections II to V cover the explanation about SOAP, REST and gRPC respectively. Section VI is about microservices and their comparison to the service oriented architecture. This is followed by section VII which contains the benefit of using HTTP/2 and gRPC and REST. And lastly, the document finished by providing a conclusion about the use of gRPC and REST in microservices.

## II. SOAP

## III. REST

## IV. HTTP/2

The state of the web report of the HTTP archive [6] suggests that between December 2010 and 2020, the amount

of kilobytes (KB) required to load a web page increased by about 318 percent. In other words, in 2010 a page consisted of a median 487.9 KB, whereas in 2020 a web page consists of 2042.3 KB. Such an increase in size is a challenge for both the network we use and the protocols our communication is built upon.

HTTP/1.1 tries to solve the problem of increased page sizes by allowing request pipeline and through the use of multiple parallel open connections. While this was a viable approach for many years, an IETF working group published the newer HTTP/2 specification RFC 7540 [3] in 2015, with an update regarding TLS 1.3 having been published in RFC 8740 [7]. This new protocol version promises to solve the problems of HTTP/1.1 that prevent it from being more efficient.

Protocol version HTTP/2 moves away from sending text over the web and instead replaces it with binary data. RFC 7540 [3] and the IETF HTTP working group [8] explain this decision through the argument that binary data is more efficient to parse, as it less error prone and does not require to keep traffic of special characters in the data.

Furthermore HTTP/2 reworks the way semantic information is send through the web (see RFC 7540 [3] for details) with the goal of allowing multiple requests to be sent through the same connection. Based on the documents [3] [8] this decisions has been to make the HTTP/2 protocol more in line with how TCP connections work. For traffic to flow through an TCP connection, a handshake has to be done first between the two participants. In HTTP/1.1 this meant that for each parallel request a newly opened connection had to go through this very handshake again, instead of being able to use the handshake from the first request, overall increasing the overhead a user had to perform to download the content of a page. HTTP/2 on the other hand only opens one TCP connection minimizing this overhead. Additionally, the header of the HTTP/2 protocol itself are compressed while being send through the network, further reducing the size of each message.

Lastly, HTTP/2 introduces a new feature called server push [3]. With this feature it is possible for a server to push additional information to the clients, based on their previous requests, without having to wait for them to open a request for this data themselves. Though introducing the ability to send "false-positive" data (i.e. data that is not used by the user in the end), this new feature allows the server to provide the user with "future" information without delay (from user perspective), which helps in providing a better user experience.

TABLE I  
HTTP/1.1 VS HTTP/2

HTTP/1.1	HTTP/2
text-based protocol	binary protocol
one request per TCP connection	multiple requests per TCP connection
no header compression	header compression
only client may request data	client may request data and server may push additional data

To further highlight these core differences between how

HTTP/1.1 and HTTP/2 treat their traffic, table I provides a comparison of the two versions, reiterating the information provided by this section in a compact format.

The claims of performance improvement by use of HTTP/2 instead of HTTP/1 can be validated empirically. Corbel, Stephan and Omnes perform in their paper "HTTP/1.1 pipelining vs HTTP2 in-the-clear: Performance comparison" [9] such an empirical study. They compare the time to download web pages by using the same infrastructure and server, with only the protocol version differing in the data-sets. As a result of their study they were able to identify that HTTP/2 truly does perform better than HTTP/1 for downloading content.

## V. gRPC

The gRPC Remote Procedure Calls (gRPC) framework [4] is a technology initially developed by Google and since having been become an open source framework maintained by the Cloud Native Computing Foundation (CNCF). It is a remote procedure call (RPC) framework that has been developed in modern times (2015 and onwards) and is backed by the HTTP/2 protocol. It has been created with the idea of a general purpose RPC infrastructure and microservice architecture in mind. This means that the created RPC framework was required to be language agnostic and allow for the fast and responsive intercommunication between a multitude of service instances.

Above requirements to the RPC protocol are reflected by the following principles of the gRPC design specification [10]:

- **Services not objects:** gRPC allows for the communication with services and is not an object oriented.
- **Messages not References:** Client and server exchange messages between them and do not have references to objects.
- **Interoperability & Reach:** "The wire protocol must be capable of surviving traversal over common internet infrastructure." [10]
- **General Purpose:** The framework should be usable with most use-cases.
- **Performant:** The framework should not perform significantly worse when compared to a framework designed for a specific use-case.
- **Payload Agnostic:** gRPC needs to supports different message types (protocol buffers, JSON, XML, Thrift) and compression mechanisms.
- **Streaming:** The framework allows for server responses and clients requests to be send in a stream.
- **Blocking:** gRPC supports synchronous communication.
- **Non-Blocking:** gRPC supports asynchronous communication.

As with other RPC frameworks, gRPC allows the client to perform operations as if they were run on the clients machine. For this the client needs to be aware of method definition and location of the remote server. To describe these methods gRPC uses protocol buffers [11] as the default interface description language (IDL), but it is possible to use other formats (like JSON) through the use of gRPC extensions. Protocol buffers

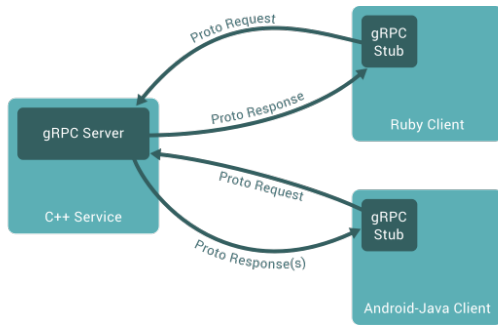


Fig. 1. Abstract example of different clients using gRPC (Image downloaded from <https://grpc.io/docs/what-is-grpc/introduction/> in January 2021)

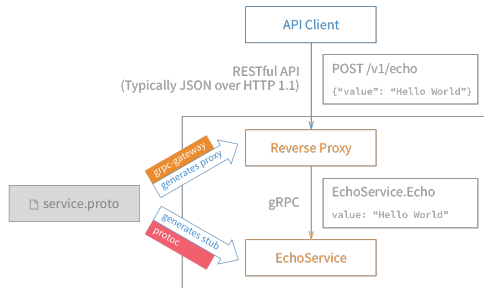


Fig. 2. Example use of created rest-proxy to communicate with gRPC server (Image downloaded from <https://grpc.io/blog/coreos/> in January 2021)

themselves are a format developed by Google and used to serialize and deserialize structured data. Google describes this format as "think XML, but smaller, faster and simpler" [11].

A high-level abstraction of the gRPC intercommunication is shown in figure 1. Within this image the gRPC server is written using C++ code. But because of the language agnostic of gRPC an android phone with a Java client is able to use communicate with this server, the same way a ruby client is able to communicate with the server.

```
1 service EchoService {
2   rpc Echo(EchoMessage) returns (EchoMessage) {
3     option (google.api.http) = {
4       post: "/v1/echo"
5       body: "*"
6     };
7   }
8 }
```

Listing 1. Small sample proto buffer specification (Full example available at <https://grpc.io/blog/coreos/>)

Though gRPC has many advantages in its favor, the framework has the problem that it is requiring the HTTP/2 streaming functionality and the protocol HTTP/2 itself to work. As HTTP/2 is still relatively new, not all companies have their infrastructure supporting HTTP/2. As a backwards compatibility to HTTP/1.1 and for general automation support (i.e. no need to import and use stub code), gRPC IDL information can be enhanced with REST options, as shown in listing 1. These changes will cause the compilation to create a rest-proxy in addition to the actual gRPC server. When a client now wants

to communicate through REST, it may talk with the rest-proxy, which in turn will forward the requests through gRPC to the actual gRPC server. Figure 2 illustrates this communication path and additionally highlights how the created services are based on the same IDL specification.

As previously mentioned, this makes it possible to have gRPC server and services running within an HTTP/1.1 infrastructure. By HTTP/1.1 infrastructure is hereby meant that not all network components within the infrastructure support the HTTP/2 protocol and have the connections fall back to HTTP/1.1 when trying to communicate through them. Such components could be router, loadbalancer or even the web-server.

## VI. MICROSERVICES

As the purpose of this paper is to identify the use of REST and gRPC over SOAP in microservice architectures, it is necessary to declare the definition that is used for the term microservice. The internet has multiple possible uses of the term microservice and they may vary on the blogs or scientific papers visited. For the purpose of this document, the definition of the national institute of standard and technology (NIST) is used:

*"Microservices: A microservice is a basic element that results from the architectural decomposition of an applications components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communications protocol and a set of well-defined APIs, independent of any vendor, product or technology."* - NIST [2]

TABLE II  
COMPARISON OF SOA AND MICROSERVICES (KARMEL, ANIL CHANDRAMOULI, RAMASWAMY IORGA, MICHAELA, 2016)

Service Oriented Architecture	Microservice
Self-contained, monolithic services	Small, decomposed, isolated and independently deployable services
Communications between services occur through an enterprise service bus	Communications between services occur through lightweight, standard communications protocols and interfaces
Stateful and requires mapping of service dependencies when changes are introduced	Stateless and less fragile when changes are introduced
Longer start/stop times	Quick start/stop times
Built around services	Built around capabilities

Looking at this definition it can be seen that microservice architectures are based on service oriented architectures, but they drive the decomposition of applications even further. Document "Nist definition of microservices, application containers and system virtual machines" [2] provides us with a comparison between a "standard" SOA and microservice architecture, see table II. Based on this comparison it can be seen that microservices decompose an application not just into services but into capabilities. These capability services are to be stateless and allow for independent, fast deployments.

In section VII the paper focuses on how these differences between SOA and microservices have brought about the trend of using REST and gRPC instead of SOAP in microservice architectures.

## VII. COMPARISON OF PROTOCOLS

TABLE III  
COMPARISON BETWEEN SOAP AND REST (WAGH & THOOL, 2015)

SOAP	REST
Changing services in SOAP web provisioning often means a complicated code change on the client side.	Changing services in REST web provisioning not requires any change in client side code
SOAP has heavy payload as compared to REST	REST is definitely lightweight as it is meant for lightweight data transfer over a most commonly known interface, - the URI It
It requires binary attachment parsing.	It supports all data types directly.
SOAP web services always return XML data.	While REST web services provide flexibility in regards to the type of data returned.
It consumes more bandwidth because a SOAP response could require more than 10 times as many bytes as compared to REST.	It consumes less bandwidth because its response is lightweight.
SOAP request uses POST and require a complex XML request to be created which makes response-caching difficult.	Restful APIs can be consumed using simple GET requests, intermediate proxy servers / reverse-proxies can cache their response very easily.
SOAP uses HTTP based APIs refer to APIs that are exposed as one or more HTTP URIs and typical responses are in XML / JSON. Response schemas are custom per object	REST on the other hand adds an element of using standardized URIs, and also giving importance to the HTTP verb used (i.e. GET / POST / PUT etc)
Language, platform, and transport agnostic.	Language and platform agnostic
Designed to handle distributed computing environments.	Assumes a point-to-point communication model not for distributed computing environment where message may go through one or more intermediaries
Harder to develop, requires tools.	Much simpler to develop web services than SOAP
False assumption: SOAP is more secure. SOAP use WS-Security. WS-Security was created because the SOAP specification was transport-independent and no assumptions could be made about the security available on the transport layer.	REST assumes that the transport will be HTTP (or HTTPS) and the security mechanisms that are built-in to the protocol will be available

// use of table III as basis for the three protocols

// use microservice definition and table II as basis for ms requirements

// I commented the rows that are (based on my opinion) not relevant for our comparison

I would say our arguments should be (as to why microservices use rest):

- 1) support of data types

- 2) payload size / bandwidth
- 3) deployment frequency: MS is deploy fast and often; the REST where changes are easier is therefore preferred
- 4) point-to-point communication (based on MS definition)
- 5) easier to develop REST than SOAP

These 5 points are the main points based on what I can find in the tables and definitions. Additionally, it nicely correlates to my personal experiences to why people prefer REST than SOAP (making me feel that my observation is valid; your opinion please :) )

-ı afterwards I would make the observation how gRPC can help / support. For this I would just take the "REST" column of table III and compare it with a "gRPC" column. And then make some observations based on that.

## VIII. CONCLUSION

We split the heavens and conquer the earth!

## REFERENCES

- [1] F. Halili and E. Ramadani, "Web services: a comparison of soap and rest services," *Modern Applied Science*, vol. 12, no. 3, p. 175, 2018.
- [2] A. Karmel, R. Chandramouli, and M. Iorga, "Nist definition of microservices, application containers and system virtual machines," National Institute of Standards and Technology, Tech. Rep., 2016.
- [3] M. Belshe, R. Peon, and M. Thomson, "Hypertext transfer protocol version 2 (http/2)," Internet Requests for Comments, Internet Engineering Task Force (IETF), RFC 7540, 5 2015. [Online]. Available: <https://tools.ietf.org/html/rfc7540>
- [4] gRPC Authors, "gRPC," 2020. [Online]. Available: <https://grpc.io/>
- [5] B. Philips and L. Cary, "gRPC with REST and Open APIs," 2016. [Online]. Available: <https://grpc.io/blog/coreos/>
- [6] S. Souders, "Report: State of the Web," 2010. [Online]. Available: <https://httparchive.org/reports/state-of-the-web>
- [7] D. Benjamin, "Using tls 1.3 with http/2," Internet Requests for Comments, Internet Engineering Task Force (IETF), RFC 8740, 2 2020. [Online]. Available: <https://tools.ietf.org/html/rfc8740>
- [8] I. H. W. Group, "HTTP/2 Frequently Asked Questions," 2021. [Online]. Available: <https://http2.github.io/faq/>
- [9] R. Corbel, E. Stephan, and N. Omnes, "HTTP/1.1 pipelining vs HTTP2 in-the-clear: Performance comparison," in *2016 13th International Conference on New Technologies for Distributed Systems (NOTERE)*, 2016, pp. 1–6.
- [10] L. Ryan, "gRPC Motivation and Design Principles," 2015. [Online]. Available: <https://grpc.io/blog/principles/>
- [11] Google, "Protocol Buffers," 2021. [Online]. Available: <https://developers.google.com/protocol-buffers>