الـجـمـهـوريــة الجـزائـريــة الـديمـقراطـيـة الـشـعبيــة

PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

وزارة التعليم العالي والبحث العلمي

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

جامعة عمّار ثليجي بالأغواط

AMAR THELIDJI UNIVERSITY OF LAGHOUAT

كلية العلوم

FACULTY OF SCIENCES

قسم الرياضيات والإعلام الآلي

Department of MATHEMATICS AND COMPUTER SCIENCE

Master's Degree Thesis

Domain : Mathematics and Computer Science

Field : Computer Science

Option : Networking and Distributed Systems

Presented by:

Benarous Mohammed Okba

Betaimi Oussama Abderaouf

THEME

Configuration of gRPC Java with Envoy proxy for web clients.

In front of members of jury :

Mr Allaoui Taher	**President**
Mr Chaibe Nourddine	**Examiner**
Mr Bensaad Lahcen	**Supervisor**

Academic Year 2018/2019

DEDICATION

We dedicate this work to our family and friends, to all the respondents, to our professor and to all the people who make this study possible. Most especially to our group mates, this is the fruit of our work!

Okba ; Oussama

ACKNOWLEDGEMENTS

This work was developed by the grace of Allah who gave us the knowledge and wit to finish and established this work

We would like to thank the following people who helped to make this study possible.

We would like to thank all the people who supported us, our families, friends, and classmates. Especially our parents who helped us with everything, to our friends who believed that we can finish the study despite of all the struggles, depression, and stress they experienced in the making of this work.

Special thanks to Mr Bensaad Lahcen, who patiently taught us everything we need to know. Thank you we appreciate sincerely.

مـلـــخـــص

يستخدم نداء الإجراء البعيد الخاص بجوجل (بالإنجليزية google Remote Procedure Calls) في الأجهزة المحمولة وعملاء الويب لأنه يمكن أن ينشئ مكتبات للعديد من الأنظمة. إنه إطار حديث وسريع وفعال، ويستند على HTTP/2، ويعمل عبر عدة لغات وأنظمة أساسية. هناك أيضًا أعمال جارية لتطوير مكتبة جافا سكريبت لاستخدامها في المتصفحات.

نداء الإجراء البعيد الخاص بجوجل يجعل الخدمات الصغيرة تتحدث مع بعضها البعض بسرعة الضوء. وهو الخيار الأفضل لإنشاء تطبيقات الويب المستندة إلى خادم العميل والمستعرض.

علاوة على ذلك، يمكننا العثور على gRPC في أي مكان تقريبًا لديك جهازي كمبيوتر متصلان عبر شبكة: الخدمات الصغيرة، وتطبيقات خادم/عميل، وواجهات برمجة التطبيقات، وتطبيقات الويب المستندة إلى المستعرض.

عند استعماله في الويب، تواجه المطورين أو المستخدمين عدة مشاكل تتمثل في صيانة وإدارة الوسيط.

في هذا العمل، نضع مقدمة إلى gRPC ونوضح كيف تقارن وتتباين مع التقنيات المماثلة، ثم نشرح Protocol Buffers، وهي تقنية أساسية لاستخدام gRPC. في النهاية، نركز على وسيط الويب gRPC ومشكلاته ونقدم بعض الحلول الممكنة باستخدام لغة الجافا.

Abstract

gRPC is used in last mile of computing in mobile and web client since it can generate libraries for various of systems. It is modern, fast and efficient framework, build on top of HTTP/2, and works across languages and platforms. There is also work underway to develop a JS library for use in browsers.

gRPC is Remote Procedure Call (RPC) system. It makes the microservices talk to each other at light-speed. gRPC is the best choice for the creation of client-server and browser-based web applications.

Beyond that, we can find gRPC almost anywhere you have two computers communicating over a network: Microservices, Client-Server Applications, Integrations and APIs, Browser-based Web Applications.

In web gRPC, the developers or the users faced some issues with the configuration, maintenance and managing of the proxy (envoy).

In this thesis, we introduce to gRPC and explain how it compares and contrasts with similar technologies, then we explain Protocol Buffers, a key technology for using gRPC. In the end, we focus on gRPC web proxy and its issues and we provide some possible solutions.

Table of Contents

General Introduction 2
Chapter I. Introduction to RPC systems 4
I.1. Request-response protocols 4
I.2. Client–Server Computing 6
I.3. Remote Procedure Calls 6
I.4. RPC systems 8
I.5. Streaming 9
I.6. HTTP2 overview 10
Chapter II. Generality about gRPC 16
II.1. What is gRPC 16
II.2. How gRPC works 16
II.3. gRPC Architecture 17
II.4. Where we find gRPC 18
II.5. Protocol buffers 20
II.6. gRPC vs REST 23
Chapter III. gRPC web proxy 25
III.1. gRPC web 25
III.2. Advantages of using gRPC-Web 26
III.3. gRPC web implementation with proxy 26
III.4. Proposed solution 32
General Conclusion 34
Bibliography 35

List of Figures

Figures of chapter I

Figure I.1 : Diagram shows an enterprise's network 5
Figure I.2 : pattern shows how the networking protocols works 6
Figure I.3 : Execution of a remote procedure call (RPC). 7
Figure I.4 : HTTP 1.1 pipelining. 11

Figures of chapter II

Figure II.1 : The use of gRPC with an online retail application 17
Figure II.2 : Example of a file.proto 21
Figure II.3 : Example of a person in XML 22
Figure II.4 : Example of a person in protocol buffer 22
Figure II.5 : Manipulating a protocol buffer file 22
Figure II.6 : Manipulating a XML file 22

Figures of chapter III

Figure III.1 : A diagram shows the use of gRPC via proxy 25
Figure III.2 : helloworld.proto file 27
Figure III.3 : HelloworldServer.java file 28
Figure III.4 : GreeterImpl.java file 28
Figure III.5 : envoy.yaml file 29
Figure III.6 : envoy.Dockerfile 29
Figure III.7 : client.js file 30
Figure III.8 : index.html file 30
Figure III.9 : The implementation result 32
Figure III.10 : Java proxy sequence diagram 33

List of Tables

Table I.1 RPC systems. 8

Table II.1 The languages supported by the core gRPC and Protocol Buffers projects. . 20

Table II.2 The difference among REST and gRPC..... 23

GENERAL INTRODUCTION

General Introduction

Over the years, Google has been using a single general-purpose RPC infrastructure called Stubby to connect the large number of microservices. In 2015, Google created a new efficient and secure framework uses HTTP/2 and Protocol Buffers. This technology provides us to have the ability of making a micro-services communication, client-Server Applications and browser-based web Applications between any two computers communicating over a network. [1]

The problem is the JavaScript code client cannot utilize HTTP/2.0, so that is why cannot communicate with the server directly, unless if there is a tool between the client and the server allowing them to communicate. However, this middleware might reduce the performance, also it is hard to install it and maintain it and manage it.

Our thesis objective is to solve this problem by the elimination of the middleware (proxy), and adding the first part of a built-in java support included in our java server to do the same work as the middleware.

Our thesis is organize in the following way: in the first chapter, we give a brief introduction about distributed systems, request-response protocols and Remote procedure calls systems. Next, in the second chapter, we explain the gRPC and how it works, and then we mention its layers. The last chapter we implement grpc-web with proxy, finally we propose a solution.

CHAPTER I.

INTRODUCTION TO RPC SYSTEMS

Chapter I. Introduction to RPC systems

In 1960’s the ideas of connecting numerous computers to form a network came out. Shortly after, the ARPANET (the precursor to the Internet) was created. In the 1970’s, email was invented, which became the most widely used distributed application on the ARPANET. Most of the network's power in these early days was to share information by sending data from one computer to another on the country's other side.

In the 1970’s, distributed computing became its own field of computer science. Therefore, the study of however multiple pcs may be wont to solve larger issues than one computer might solve was of great interest. This fledgling Internet connected the computing resources of multiple universities and government organizations, creating a large pool of compute power.

Eventually, the ARPANET grew into the Internet that we know today, connecting millions of pcs worldwide. It powered the rise of the World Wide Web in the 1990’s, and today the Internet is a utility, much like electricity or water. It is available almost everywhere in most of the world. With more recent innovations in mobile computing and embedded systems, the Internet has become an integral component in modern life, not just in business.[2]

I.1. Request-response protocols

Various networking protocols have been developed for sharing information from one point of the Internet to the other. Almost all of the protocols use the TCP/IP Internet protocol suite. This has enabled the free flow of information across the globe, and it has also had an immeasurable impact on modern business and commerce, which rely greatly on computer networks and distributed computing.[3]

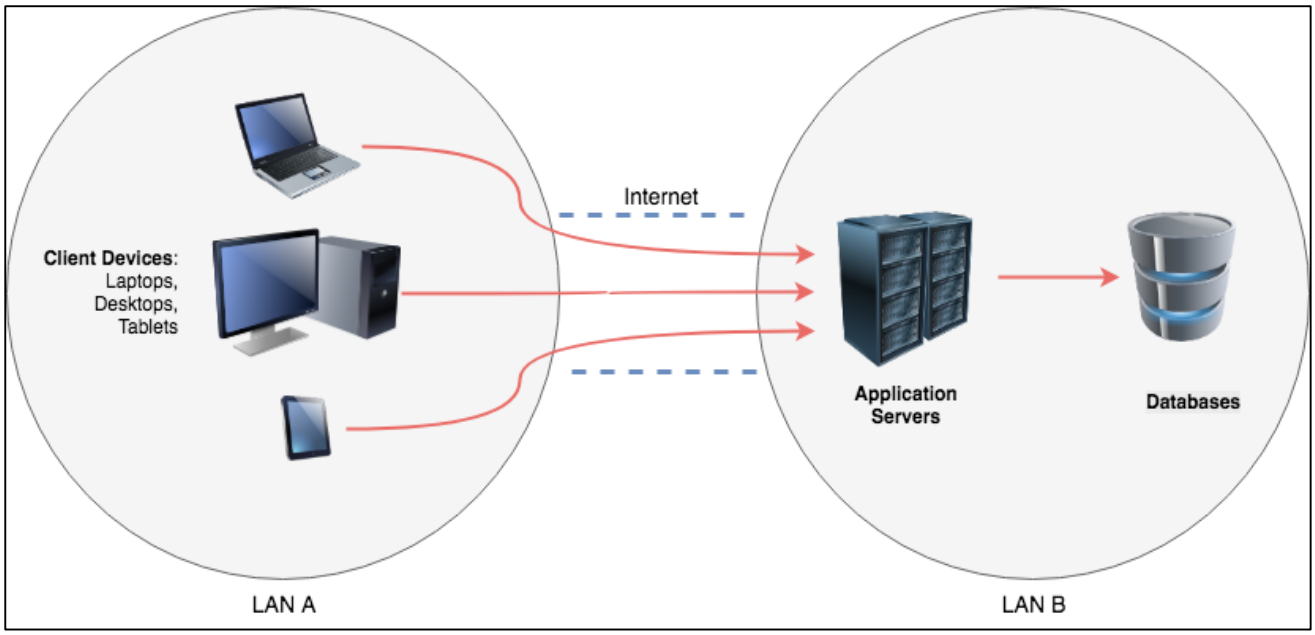

Figure I.1 : Diagram shows an enterprise's network

This diagram in Figure I.1 shows a very simple but typical arrangement. In enterprise settings, the two halves will likely be on the same network, or perhaps a VPN is used to securely connect clients to the servers. For e-Commerce, SaaS (Software as a Service) and other Internet offerings, the servers might be hosted in the provider's datacenter or even with a third-party provider like Amazon Elastic Compute Cloud (EC2), Google Cloud Platform, or Microsoft Azure. The main components of interest here are clients, on the left, and servers, on the right.[2]

A *client* is a program that initiates communication-usually by creating a TCP connection. It may be an end-user program, and initiating communication to request information or resources that it presents to a user. In web applications, for example, the client is a web browser such as Chrome or Firefox.[4]

A *server* is a program that accepts these client connections and in turn processes their requests. There are many kinds of servers. The diagram shows an application server, so named because it serves data to the client application. (It is also called a web server if it uses HTTP as the communication protocol). As shown in the diagram, the application server may involve other computers in order to serve responses. In that case, the server also acts as a client. In the diagram, the application server acts as a database client, requesting information from the database server.[4]

Most networking protocols follow this pattern (Figure I.2) :

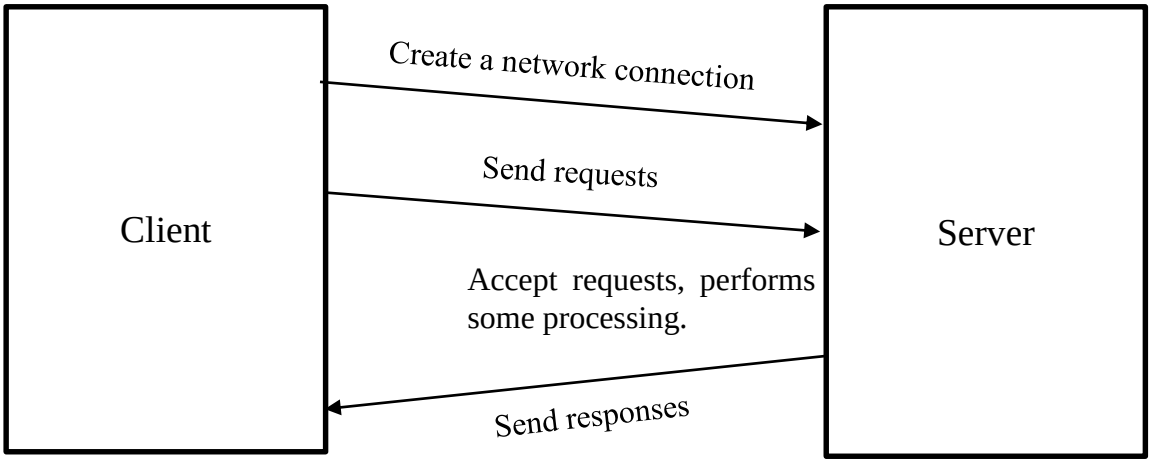

Figure I.2 : pattern shows how the networking protocols works

This request-response flow was conceived in the early days of networking in the 1960's and has been foundational to distributed computing ever since.

I.2. Client–Server Computing

Many of today's systems act as server systems to satisfy requests generated by client systems. This form of specialized distributed system, called a client–server system.

Server systems can be widely classified as compute servers and file servers:

- The compute-server system provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server running a database that answers client demands for data is an example of such a system.
- The file-server system provides a file-system interface that allows clients to create, update, read, and delete files. An example of such a system is a web server that provides files to clients running web browsers.[5]

I.3. Remote Procedure Calls

A remote procedure call (RPC) is a client–server mechanism that enables an application on one machine to make a procedure call to code on another machine. The client calls a local procedure a stub routine that packs its instructions into a message and

sends them via the network to a particular server process. The client-side stub routine then blocks. At the same time, the server unpacks the message, calls the procedure, following the return results into packs as a message, and sends them back to the client. The client stub unblocks, receives the message, unpacks the results of the RPC, and returns them to the caller. This packing of arguments is sometimes called marshaling. The client stub code and the descriptors necessary to pack and unpack the arguments for an RPC are compiled from a specification written in Interface Definition Language (Figure I.3). [6]

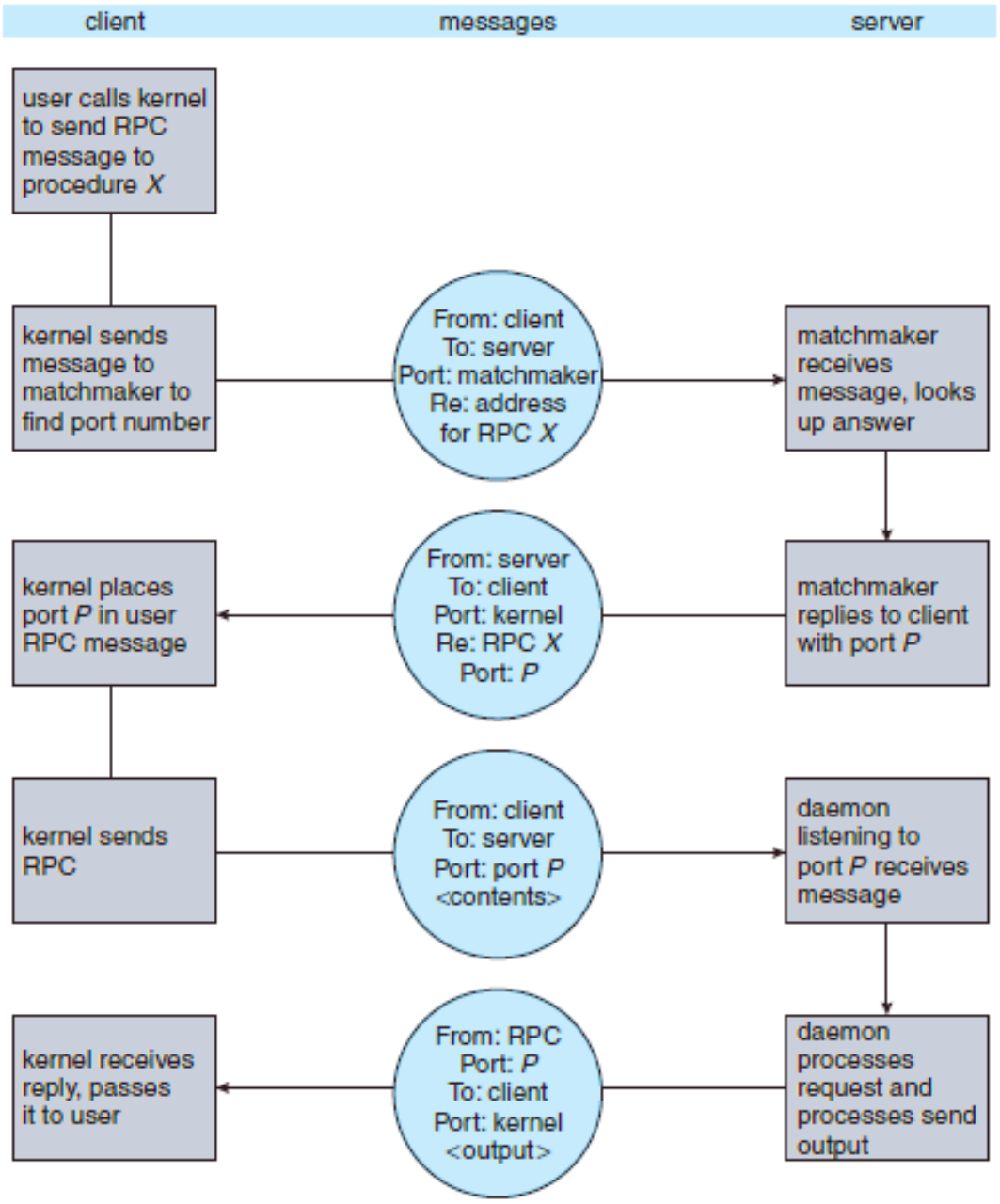

Figure I.3 : Execution of a remote procedure call (RPC).

Chapter I. Introduction to RPC systems

RPC is not a new programming term: proposals for remote procedure call semantics were written in the 70's, and practical RPC implementations appeared in the 80's, such as the Network File System (NFS).[6]

I.4. RPC systems

There are too many RPC systems to realistically review them all, but the table below enumerates the more well-known systems. Some are older technologies that are no longer widely in use (or they are in use primarily in legacy and enterprise systems). Some are newer technologies, like gRPC. They are listed oldest to newest, showing the history and evolution of RPC in (Table I.1). [2]

RPC system	Transport Language	support	Notes
Sun/ONC RPC	Custom TCP or UDP	Various (Mainly C)	Early standard for RPC. Used in NFS (Network File System). Custom "Rpcgen" IDL.
CORBA	Custom TCP	Various	Early standard for RPC and distributed objects. Custom IDL.
DCOM	Custom TCP	Various (Windows Only)	Distributed objects. Proprietary Microsoft serialization. Supports distributed garbage collection. Custom IDL.
Java RMI	Custom TCP	Java	Distributed objects. Uses Java serialization.
XML-RPC, SOAP	HTTP 1.1	Various	Specifies the communication protocol, not the client and server programming models. Uses XML for message encoding.
JSON-RPC	HTTP 1.1	Various	Specifies the communication protocol, not the client and server programming models. Uses JSON for message encoding. Technically a subset of REST.
Apache Thrift	Custom TCP	Various	Custom IDL that is similar to protobuf. No streaming.
Cap'n Proto	Custom TCP	Various (Mainly C++)	Distributed objects. Promise pipelining. Strives for zero-overhead serialization. Custom IDL.
Twirp	HTTP 1.1	Various	No streaming. Can use JSON as encoding. Uses Protobuf as IDL.

Table I.1 RPC systems.

Some notable innovations of RPC systems over the years are the "distributed object" paradigm, Cap'n Proto's "promise pipelining".

I.4.1. Distributed object system

To explain a distributed object system, we first need to talk about a key constraint in non-distributed-object RPC systems: the operations that a server exposes and the implementations to which they are bound are fixed. When a server starts up, its service implementations are registered to be exposed by the network server. In an OOP paradigm, this amounts to the service implementations being singletons. To contrast, in a distributed object system, the server can dynamically allocate new objects at runtime whose methods are also exposed via RPC. The existence of these objects can be communicated to clients as RPC return values. A typical RPC result might be a value that is serialized to bytes and transmitted to the client. But with distributed objects, the system might just serialize and transmit "handles," not the actual values. These handles are then used to create new stubs that are returned to the client application as the RPC result. Calling methods on these stubs in turn issues RPCs, which result in invoking methods on the underlying instances in the server.[7]

I.4.2. Cap'n Proto's

Cap'n Proto's "promise pipelining" permits clients to pipeline many requests without waiting on a server response, even when the arguments to one RPC may depend on the result of an earlier one! It does this by allocating a promise "handle" for each RPC in the client. So, even before the server could have received the first request, the client has a handle to the first response. It can then refer to that promise in a subsequent request, and send that to the server. The server is responsible for resolving these handles: once the result for the first RPC is computed, the server machinery can bind it to the handle in the second RPC and then begin computing a result for the second RPC. This feature is an optimization to lower the latency for multi-step operations, because it can greatly reduce the time of waiting on messages to transit the network.[8]

I.5. Streaming

Streaming allows a request or response to have an arbitrarily large size, such as operations that require uploading or downloading a huge amount of information. Most

RPC systems require that the arguments of an RPC be represented as data structures in memory that are then serialized to bytes and sent on the network. When a very large amount of data must be exchanged, this can mean significant memory pressure on both the client process and the server process. And it means that operations must typically impose hard limits on the size of request and response messages, to prevent resource exhaustion. Streaming alleviates this by allowing the request or response to be an arbitrarily long sequence of messages. The cumulative total size of a request or response stream may be incredibly large, but clients and servers do not need to store the entire stream in memory. Instead, they can operate on a subset of data, even as little as just one message at a time.[9]

Some RPC systems supports full-duplex bidirectional streams. Bidirectional means that the client can use a stream to upload an arbitrary amount of request data and the server can use a stream to send back an arbitrary amount of response data, all in the same RPC. The novel part is the "full-duplex" part. Most request-response protocols, including HTTP 1.1 are "half-duplex." They support bidirectional communication (HTTP 1.1 even supports bidirectional streaming), but the two directions cannot be used at the same time. A request must first be fully uploaded before the server begins responding; only after the client is done transmitting can the server then reply with its full response.[9]

I.6. HTTP2 overview

HTTP/2 is the newer standard for internet communications that address common pitfall of HTTP/1.1 on modern web pages.[1] Before we talk about HTTP/2 let's cover some HTTP/1.1 request.

I.6.1. How HTTP/1.1 works

HTTP (Hypertext Transfer Protocol) is the most expanded way to make two machines communicate. It is a protocol for requesting files over a network, where the payload (both request and response) is transferred as plain text (ASCII).

- HTTP/1.1 was realized in 1997. It has worked for many years
- HTTP/1.1 opens an new TCP connection to a server at each request

- It does not compress headers (which are plaintext)
- It only works with Request/Response mechanism (no server push)
- HTTP was originally composed of two commands :
  - GET: to ask for content
  - POST: to send content
- Nowadays, a web page loads 80 assets (html, css, images,…) on average
- Headers are sent at every request and are PlainText (heavy size)
- These inefficiencies add latency and increase network packet size

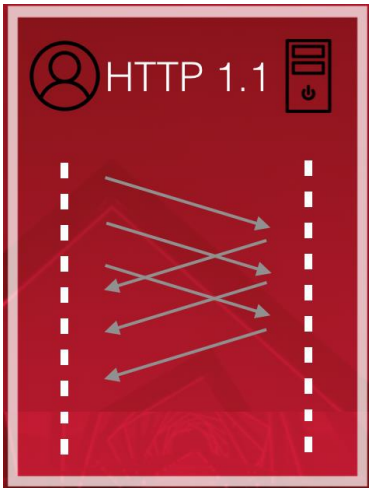

Figure I.4 : HTTP 1.1 pipelining.

The Figure I.4 to introduce the notion of Head-of-Line (HOL) blocking, which is a problem that persists with HTTP/1.1. HOL is a degradation of the performance that occurs when a bunch of packets are blocked in the FIFO (First Input First Output) buffer of a switch, because the first packet cannot be forwarded. The reason why a packet cannot be forwarded is that the destination port is busy. If this happens, all of the subsequent requests are blocked, even with the request pipelining introduced by HTTP/1.1. [10]

I.6.2. How HTTP/2 works

- HTTP/2 was released in 2015. It has been battled tested for many years, and was before that tested by Google under the name SPDY
- HTTP/2 supports multiplexing: The client and server can push messages in parallel over the same TCP connection. This greatly reduces latency

- HTTP/2 supports server push: Servers can push streams (multiple messages) for one request from the client. This saves round trips (latency)
- HTTP/2 supports header compression : these have much less impact on the packet size
- HTTP/2 is secure (SSL is not required but recommended by default) [11]

HTTP/2 introduces multiple notions and they will be detailed in the following:[12]

Frames

A frame in HTTP/2 is the smallest unit data representation. It contains:

- A frame Header that normally only contains the identifier of the stream that it belongs to.
- The data content, which can have different formats depending on the kind of data that it contains.

Here we have some types of frames:

- **DATA: (type=0x0)** Contains the payload of the requests or the responses.
- **HEADERS: (type=0x1)** Equivalent to HTTP/1.x headers, it is used to open a stream and it basically contains a header block fragment.
- **PRIORITY: (type=0x02)** Specifies the sender advised priority of the stream.
- **RST_STREAM (type=0x03)** Sent for immediate termination of a stream purposes.
- **SETTINGS: (type=0x04)** Its purpose is to send the desired configuration of the connection that needs to be established.
- **PUSH_PROMISE: (type=0x5)** Notification that a sender wants to begin to send a stream.
- **PING: (type=0x06)** Intends to check the state of a given connection.
- **GOAWAY (type=0x07)** Initiate a shutdown connection or signal errors.
- **WINDOW_UPDATE (type=0x08)** Used to implement flow control, as we'll describe later on.
- **CONTINUATION (type=0x09)** Used to continue a sequence of header block fragments

Messages

A message is a sequence of frames representing a request or a response.

Streams

A stream is a bidirectional flow of bytes that in a given connection may carry one or more messages. Each stream is identified by an integer that will be written on the header of any frame.

Note that in a single TCP connection, several streams may be active concurrently at the same time.

Streams have lifecycles and they are represented by transitions among states. These are all the possible states:

- **Idle:** Initial state for any opened stream.
- **Open:** State where both peers may send or receive frames at any moment.
- **Reserved:** One of the peers, having the stream in Idle state, sent a PUSH_PROMISE frame in order for the server to begin to push messages to the client.
- **Half Closed:** One of the peers finished to send frames.
- **Closed:** Both peers agreed to terminate the connection.

Multiplexing

As mentioned before, HTTP/1.x doesn't allow you to do multiple parallel requests in the same TCP connection. At most, with HTTP/1.1 you'll be able to do multiple requests, but the responses will need to be received in the same order, inducing Header-of-Line blocking.

HTTP/2 and its binary convention allows you to multiplex requests and responses in the same connection so you can take full advantage of the network's resources.

Flow control

HTTP/2 multiplexed frames unleash the potential to take full advantage of the network resources, but without a flow control you lose any sense of traffic congestion. Any peer sending data needs to know the ingestion capabilities of the receiver, otherwise the frames can be lost. If the receiver is busy doing other stuff, it needs to be able to communicate to the sender to slow down the cadence.

TCP protocol already considers flow control by communicating the `receive window` of each peer. This is the equivalent buffer size of each end point to hold incoming data. Each TCP `ACK` signal contains an updated `receive window` in function of the receiver availability.

The problem with TCP flow control is that it doesn't have enough Application Level granularity. Multiple streams in the same TCP connection prevent the optimizing of the receiving flow for each pair of stream-applications.

So in addition to TCP flow control, HTTP/2 uses the `WINDOW_UPDATE` frame type (type=0x8) to advertise the stream-application receive window size.

Server push

Server push is the capability of HTTP/2 to initiate the transfer of data frames from the server side, instead of having to wait for the request of the client. This is obviously a big win, for instance when browsers need to download multiple resources from a webpage, where HTTP/1.x is needed to open one TCP connection for every one of them.

Headers

In HTTP/1.x headers are systematically sent in any request or response, in plain text. This is a big waste of bandwidth that HTTP/2 improves by applying HPACK compression on the `HEADER` frames.

As said before, the purpose of HTTP/2 is to improve performance by reinventing the encoding, but keeping the syntax. This is to preserve the headers as they are, and it's just the way they will be transferred.

CHAPTER II.

GENERALITY ABOUT GRPC

Chapter II. Generality about gRPC

II.1. What is gRPC

First of all, let's talk about what the letters mean. It is an acronym after all. In the introduction, we talked about RPC: Remote Procedure Calls. This is the programming idiom that gRPC presents. So the letter "g" in "gRPC" is because this technology was created by google as an open source evolution of their RPC technology named Stubby.

gRPC uses HTTP/2 for transport, Protocol Buffers as the interface description language.[13]

II.2. How gRPC works

In gRPC client application a client can directly call methods on a server application on a different machine as if it were a local object. Because gRPC is based on RPC technology but implemented on top of the modern technology stacks such as HTTP2.

gRPC has ability to define a service contract using the gRPC Interface Definition Language (IDL). So, on the server side, the server implements this interface and runs the server to handle client calls. On the client side, the client has a stub that provides the same methods as the server.[14]

Figure II.1. illustrates an example of gRPC where we have an online retail application contain of inventory and product search service. The contract for the inventory service is defined using gRPC IDL in `inventory.proto` file. So, for a developer first to define all the business capabilities using the service for the inventory service and then generate the service side code from the proto file. In the same way the client side code (stub) can be generated using the same proto file.[15]

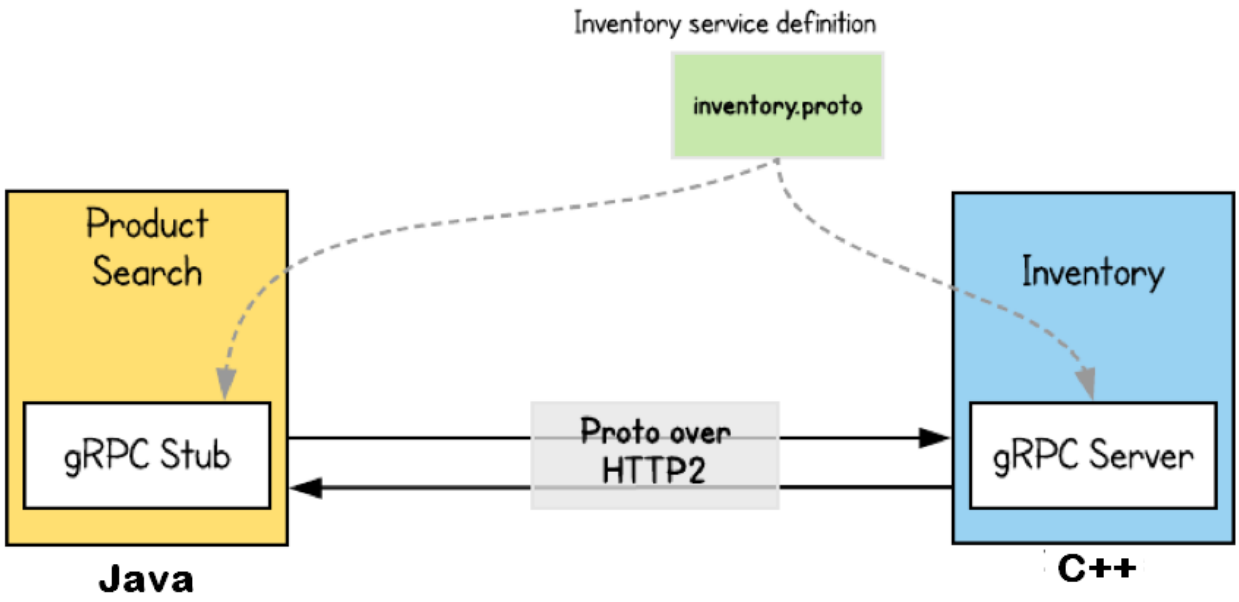

Figure II.1 : The use of gRPC with an online retail application

II.3. gRPC Architecture

The architecture of gRPC is divided to three layers:

- **The transport layer:** is the lowest layer, gRPC uses HTTP/2 at this layer, because HTTP/2 provides the same basic semantics as HTTP/1.1, but more efficient and more secure.
- **The framework layer:** or the channel layer, at this layer the channel defines calling and implements the mapping of an RPC calls, and also the gRPC call consists of client-provided service name and method name, optional request metadata and zero or more request messages. A call is completed when the server provides optional response header metadata, zero or more response messages, and response trailer metadata. The trailer metadata indicates the final disposition of the call. Also, at this layer the message is just a sequence of zero or more bytes because there is no know knowledge of interface constraints, data types, or message encoding.
- **The last layer is the stub:** is where interface constraints and data types are defined because it is an application layer.[16]

II.4. Where we find gRPC

As we saw, the gRPC is request-response protocol for RPC that uses Protocol Buffers to define interfaces and messages. However, what about where and why you would use gRPC, or what programming languages can be used with gRPC.

The answer of the "where" is easy: you can leverage gRPC almost anywhere you have two computers communicating over a network:

- **Microservices:** gRPC works as a tool to servers in service oriented environments. That work was original problem of Stubby (predecessor of gRPC) when solved by wiring together microservices. It is well-suited for a wide variety of arenas: from medium and large enterprises systems all the way to "web-scale" eCommerce and SaaS offerings.[17]
- **Client-Server Applications:** gRPC works also in Client-Server applications, where the client application runs on desktop or mobile devices. It uses HTTP/2. This means you get improved response times and longer battery life.[9]
- **Integrations and APIs:** gRPC is also a way to offer APIs over the Internet, for integrating applications with services from third-party providers. As an example, many of Google's Cloud APIs are exposed via gRPC.[9]
- **Browser-based Web Applications:** we also use gRPC in web. But, the browser cannot directly utilize gRPC since gRPC has a strict requirement for HTTP/2. However, there is a tool for exposing your gRPC APIs for example ENVOY proxy.[9]

For each of the above situations where you might use gRPC, there are alternatives. In fact, REST and JSON can handle all of these situations. So why would we use gRPC instead?

There are serval dimensions where gRPC wins out over the others:

- **Performance/Efficiency:** HTTP/2 is much less verbose thanks largely to header compression, and it supports multiplexing many requests over a single connection. Protocol Buffers, unlike JSON, were designed to be both compact

on the wire and efficient for computers to parse. The result is that gRPC can reduce resource usage, resulting in lower response times compared to using REST and JSON. This also means reduced network usage and longer battery life for clients running on mobile devices.

- **Productive Programming Model:** The programming model with gRPC is simple to understand and leads to developer productivity. Defining interfaces and canonical message formats in an IDL means a lot of boilerplate code is auto-generated.
- **Streaming:** One of gRPC's best things or best features is full-duplex bidirectional streaming (multiple request, and multiple response). While the great majority of RPCs will be simple "unary" operations (single simple request, and single response).
- **Security:** gRPC was designed with security in mind. HTTP/2 is stricter than HTTP 1.1. It allows only TLS (Transport Level Security, sometimes called SSL) 1.2 or higher, and numerous cipher suites are blacklisted because they provide disqualified security.[18]

Now we know that gRPC is well-suited and why it is a good fit. So, What programming languages can you use with gRPC ?

The Table II.1 shows the officially languages supported by the core gRPC and Protocol Buffer project [19] :

Language	Notes
C++	Based on the C core implementation. One of the original languages supported by *protoc*.
Java	One of the original languages supported by *protoc*.
Python	One of the original languages supported by *protoc*.
Go	Support is provided via a protoc plugin named *protoc-gen-go*. Reflection support is limited.
Ruby	Based on the C core implementation. Lacks reflection support.
C#	Based on the C core implementation. Lacks reflection support.
JavaScript	Node.js only (not for browsers). Two implementations: one based on the C core, another that is pure JavaScript.
Objective-C	Based on the C core implementation. Lacks reflection support. No support for gRPC servers.
PHP	Based on the C core implementation. Lacks reflection support. No support for gRPC servers.

Table II.1 The languages supported by the core gRPC and Protocol Buffers projects.

Now, we have to take a detour to talk about a technology that is imperative to know in order to use gRPC is Protocol Buffers.

gRPC uses Protocol Buffers as a solution to all of these known issues:

- Formal contracts
- Code generation
- Bandwidth optimization

II.5. Protocol buffers

Protocol buffers are a mechanism for serializing structured data, like XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even

update your data structure without breaking deployed programs that are compiled against the "old" format. [20]

II.5.1. How do they work?

You specify how you want the information you're serializing to be structured by defining protocol buffer message types in `.proto` files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs. Here's a very basic example of a `.proto` file that defines a message containing information about a person [20]:

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Figure II.2 : Example of a file.proto

II.5.2. Why not just use XML?

Protocol buffers have many advantages over XML for serializing structured data. Protocol buffers:

- are simpler
- are 3 to 10 times smaller
- are 20 to 100 times faster
- are less ambiguous

- generate data access classes that are easier to use programmatically

For example, let's say you want to model a `person` with a `name` and an `email`. In XML, you need to do:

```
<person>
  <name>John Doe</name>
  <email>jdoe@example.com</email>
</person>
```

Figure II.3 : Example of a person in XML

While the corresponding protocol buffer message is:

```
person {
  name: "John Doe"
  email: "jdoe@example.com"
}
```

Figure II.4 : Example of a person in protocol buffer

Also, manipulating a protocol buffer is much easier:

```
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

Figure II.5 : Manipulating a protocol buffer file

Whereas with XML you would have to do something like:

```
cout << "Name: "
     << person.getElementsByTagName("name")->item(0)->innerText()
     << endl;
cout << "E-mail: "
     << person.getElementsByTagName("email")->item(0)->innerText()
     << endl;
```

Figure II.6 : Manipulating a XML file

However, protocol buffers are not always a better solution than XML, for instance, protocol buffers would not be a good way to model a text-based document with markup

(e.g. HTML), since you cannot easily interleave structure with text. In addition, XML is human-readable and human-editable; protocol buffers, at least in their native format, are not. XML is also – to some extent – self-describing. A protocol buffer is only meaningful if you have the message definition (the `.proto` file). [20]

II.6. gRPC vs REST

The Table II.2 shows a simple comparison between gRPC and REST[18]:

gRPC	REST
Protocol Buffers – smaller, faster	JSON – text based, slower, bigger
HTTP/2 (lower latency) – from 2015	HTTP/1.1 (higher latency) – from 1997
Bidirectional and Async	Client => Server requests only
Stream support	Request/response support only
API oriented – "What" (no constraints – free design)	CRUD oriented (Create Retrieve Update Delete / POST GET PUT DELETE)
Code generation through Protocol Buffers in any language	Code generation through OpenAPI / Swagger
RPC Based	HTTP verbs based

Table II.2 The difference among REST and gRPC.

CHAPTER III.

GRPC WEB PROXY

Chapter III. gRPC web proxy

III.1. gRPC web

gRPC-Web is a javascript library where we can communicate to the gRPC service via web-browser. And the gRPC-Web clients connect to gRPC services via a special gateway proxy.

The basic idea is to have the browser send normal HTTP requests and have a small proxy in front of the gRPC server to translate the requests and responses to something the browser can use.

The Web client library implements a different protocol than the native gRPC protocol. This protocol is designed to make it easy for a proxy to translate between the protocols as this is the most likely deployment model.[21]

The gRPC web support both HTTP/1.1 and HTTP/2, when the web client can only support HTTP/1.1, it is impossible to implement the HTTP/2 gRPC spec because there is simply no browser API with enough fine-grained control over the requests, that's where we find the of proxy. [22]

The Figure III.1 shows the use of gRPC via proxy. [23]

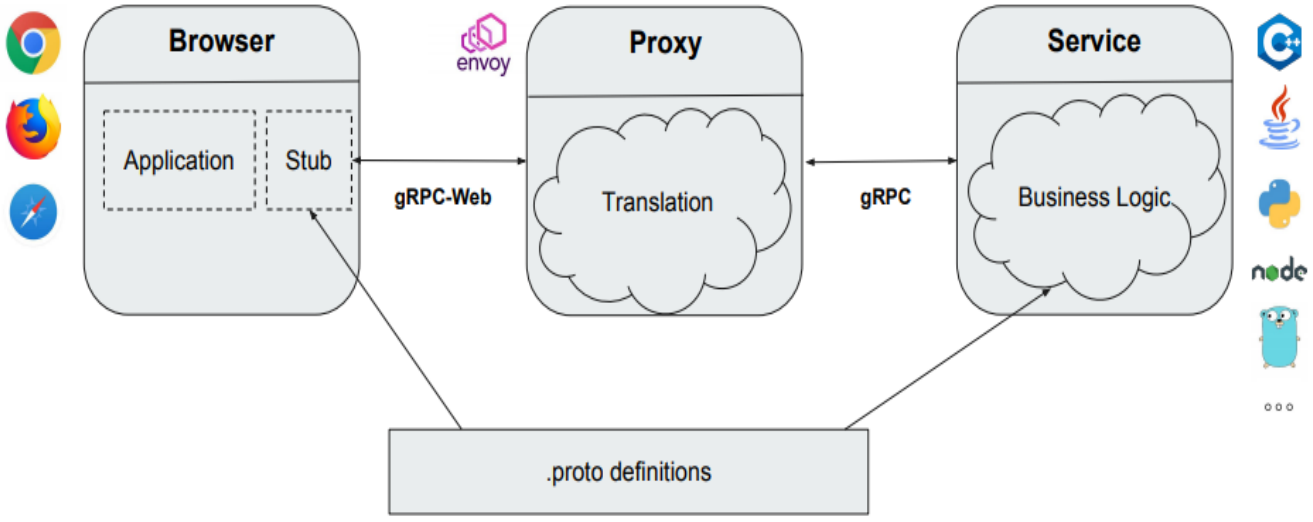

Figure III.1 : A diagram shows the use of gRPC via proxy

III.2. Advantages of using gRPC-Web

There are many features of using gRPC-Web, we mention some of them:

- **End-to-end gRPC:** Enables you to craft your entire RPC pipeline using Protocol Buffers. Imagine a scenario in which a client request goes to an HTTP server, which then interacts with 5 backend gRPC services. There's a good chance that you'll spend as much time building the HTTP interaction layer as you will building the entire rest of the pipeline.
- **Tighter coordination between frontend and backend teams:** With the entire RPC pipeline defined using Protocol Buffers, you no longer need to have your "microservices teams" alongside your "client team." The client-backend interaction is just one more gRPC layer amongst others.
- **Easily generate client libraries:** With gRPC-Web, the server that interacts with the "outside" world, i.e. the membrane connecting your backend stack to the internet, is now a gRPC server instead of an HTTP server, that means that all of your service's client libraries can be gRPC libraries. You no longer need to write HTTP clients for all languages. [24]

III.3. gRPC web implementation with proxy

The teams at Google and Improbable both went on to implement the spec in two different repositories.

- The Improbable gRPC-Web client is implemented in TypeScript and available on npm. There is also a Go proxy available, both as a package that can be imported into existing Go gRPC servers, and as a standalone proxy that can be used to expose an arbitrary gRPC server to a gRPC-Web frontend.

- The Google gRPC-Web client is implemented in JavaScript using the Google Closure library base. It is available on npm. It originally shipped with a proxy implemented as an NGINX extension16, but has since doubled down on an Envoy proxy HTTP filter, which is available in all versions since v1.4.0.[25]

We decided to use the second method to implement the gRPC web with Envoy proxy, as described in the following steps:

Define the Service

First, let's define a gRPC service using protocol buffers in `helloworld.proto` file. Here we define a request message, a response message, and a service with one RPC method: `SayHello`.

```
syntax ="proto3";
package helloworld;

option java_package = "com.proto.helloworld";
option java_multiple_files = true;

service Greeter {
    rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
    string name = 1;
}

message HelloReply {
    string message = 1;
}
```

Figure III.2 : helloworld.proto file

Implement the Service

Then, we need to implement the gRPC Service in Java (`HelloworldServer.java`). Here, we receive the client request, and we can access the message field via `request.getName()`. Then we construct a nice response and send it back to the client via `responseObserver.onNext(reply)`.

```java
package com.okba.grpc.server;

import io.grpc.Server;
import io.grpc.ServerBuilder;

import java.io.IOException;

public class HelloworldServer {
    public static void main(String[] args) throws IOException, InterruptedException {
        System.out.println("Server grpc Started....");
        Server server = ServerBuilder.forPort(9090)
                .addService(new GreeterImpl())
                .build();
        server.start();
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            server.shutdown();
            System.out.println("Successfully stopped the server");
        }));
        server.awaitTermination();
    }
}
```

Figure III.3 : HelloworldServer.java file

```java
package com.okba.grpc.server;

import com.proto.helloworld.GreeterGrpc;
import com.proto.helloworld.HelloReply;
import com.proto.helloworld.HelloRequest;
import io.grpc.stub.StreamObserver;

public class GreeterImpl extends GreeterGrpc.GreeterImplBase {
    @Override
    public void sayHello(HelloRequest request, StreamObserver<HelloReply> responseObserver) {
        String string = request.getName();
        String result = "hellow" + string;
        HelloReply reply = HelloReply.newBuilder()
                .setMessage(result)
                .build();
        responseObserver.onNext(reply);
        responseObserver.onCompleted();
    }
}
```

Figure III.4 : GreeterImpl.java file

Configure the Proxy

Next up, we need to configure the Envoy proxy to forward the browser's gRPC-Web requests to the backend. We put this in an `envoy.yaml` file. Here we configure Envoy to listen at port `:8080`, and forward any gRPC-Web requests to a cluster at port `:9090`.

```yaml
admin:
  access_log_path: /tmp/admin_access.log
  address:
    socket_address: { address: 0.0.0.0, port_value: 9901 }

static_resources:
  listeners:
  - name: listener_0
    address:
      socket_address: { address: 0.0.0.0, port_value: 8080 }
    filter_chains:
    - filters:
      - name: envoy.http_connection_manager
        config:
          codec_type: auto
          stat_prefix: ingress_http
          route_config:
            name: local_route
            virtual_hosts:
            - name: local_service
              domains: ["*"]
              routes:
              - match: { prefix: "/" }
                route:
                  cluster: greeter_service
                  max_grpc_timeout: 0s
              cors:
                allow_origin:
                - "*"
                allow_methods: GET, PUT, DELETE, POST, OPTIONS
                allow_headers: keep-alive,user-agent,cache-control,content-type,content-transfer-encoding,custom-header-1
                max_age: "1728000"
                expose_headers: custom-header-1,grpc-status,grpc-message
                enabled: true
          http_filters:
          - name: envoy.grpc_web
          - name: envoy.cors
          - name: envoy.router
  clusters:
  - name: greeter_service
    connect_timeout: 0.25s
    type: logical_dns
    http2_protocol_options: {}
    lb_policy: round_robin
    hosts: [{ socket_address: { address: localhost, port_value: 9090 }}]
```

Figure III.5 : envoy.yaml file

To run Envoy (for later), we will need a simple Dockerfile (`envoy.Dockerfile`).

```
FROM envoyproxy/envoy:latest
COPY ./envoy.yaml /etc/envoy/envoy.yaml
CMD /usr/local/bin/envoy -c /etc/envoy/envoy.yaml
```

Figure III.6 : envoy.Dockerfile

Client Code

Now, we are ready to write some client code in a `client.js` file.

```
const {HelloRequest,HelloReply} = require('./helloworld_pb.js');
const {GreeterClient} = require('./helloworld_grpc_web_pb.js');

var client = new GreeterClient('http://' + window.location.hostname + ':8080',
                               null, null);

// simple unary call
var request = new HelloRequest();
request.setName('okbaa');

client.sayHello(request, {}, (err, response) => {
  console.log(response.getMessage());
});
```

Figure III.7 : client.js file

The classes `HelloRequest`, `HelloReply` and `GreeterClient`, we import here are generated by the `protoc` generator utility (which we will cover in the next section) from the `helloworld.proto` file we defined earlier.

Then we instantiate a `GreeterClient` instance, set the field in the `HelloRequest` protobuf object, and we can make a gRPC call via `client.sayHello()`, just like how we defined in the `helloworld.proto` file.

Finally a simple `index.html` file.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>gRPC-Web Example</title>
<script src="./dist/main.js"></script>
</head>
<body>
</body>
</html>
```

Figure III.8 : index.html file

Generate Protobuf Messages and Client Service Stub

To generate the protobuf messages and client service stub class from our `.proto` definitions, we need the `protoc` binary and the `protoc-gen-grpc-web` plugin.

So we have both `protoc` and `protoc-gen-grpc-web` installed, we can now run this command:

```
$ protoc -I=. helloworld.proto \
  --js_out=import_style=commonjs:. \
  --grpc-web_out=import_style=commonjs,mode=grpcwebtext:.
```

After the command runs successfully, we should now see two new files generated in the current directory:

- `helloworld_pb.js`: this contains the `HelloRequest` and `HelloReply` classes
- `helloworld_grpc_web_pb.js`: this contains the `GreeterClient` class

These are also the 2 files that our `client.js` file imported earlier in the example.

Compile the Client JavaScript Code

Next, we need to compile the client side JavaScript code into something that can be consumed by the browser.

```
$ npm install
$ npx webpack client.js
```

Run the Example

We are ready to run the Hello World example. The following set of commands will run the 3 processes all in the background.

1. We run the Java gRPC Service. This listens at port `:9090`.
2. We run the Envoy proxy. The `envoy.yaml` file configures Envoy to listen to browser requests at port `:8080`, and forward them to port `:9090`

   ```
   $ docker build -t helloworld/envoy -f ./envoy.Dockerfile .
   $ docker run -d -p 8080:8080 --network=host helloworld/envoy
   ```

3. We run the simple Web Server. This hosts the static file `index.html` and `dist/main.js` we generated earlier.

   ```
   $ python2 -m SimpleHTTPServer 8081 &
   ```

4. When these are all ready, you can open a browser tab and navigate to: `localhost:8081`
5. Open up the developer console and we should see the following printed out:

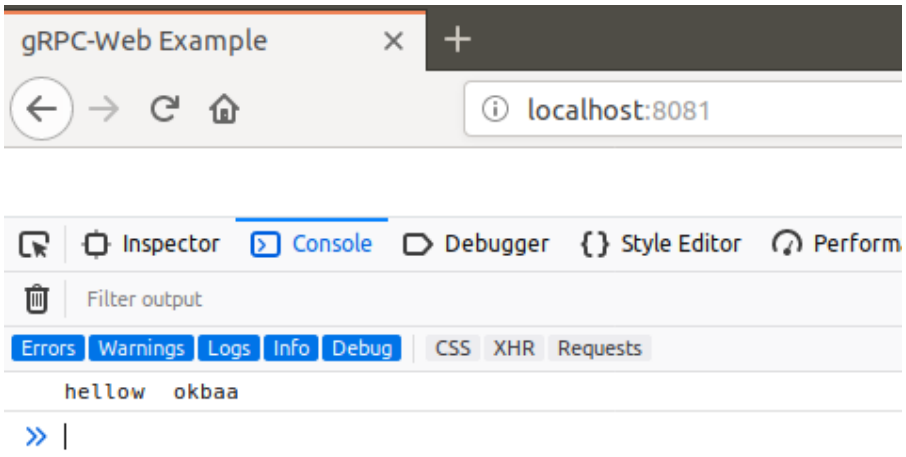

Figure III.9 : The implementation result

III.4. Proposed solution

After we created our server in java, and captured all the http packets in the last implementation with Wireshark, We propose to add a built-in support that is doing the same work of the Envoy proxy, included in the server and with the same language.

The first step is built-in support reads the request of the web client (HTTP/1.1 request) via a class called `ClientListner.java`, the role of this listener is to read the request (Method, Request header, request body) of the browser client and save it.

The second step is to send the HTTP/1.1 to the server, but with gRPC over HTTP/2.0 specification, via a method called SendToServ(httpRequest). So now, the server will accept the request, because it is an HTTP/2.0 request, and now the server will send an HTTP/2.0 response to our java proxy.

The last step is that our java proxy receive the HTTP/2.0 response, via a listener and reverse it into HTTP/1.1 response.

Finally, our java proxy will send the HTTP/1.1 to the client.

The Figure III.10 shows a sequence diagram for the interaction between the client and java proxy and the java server.

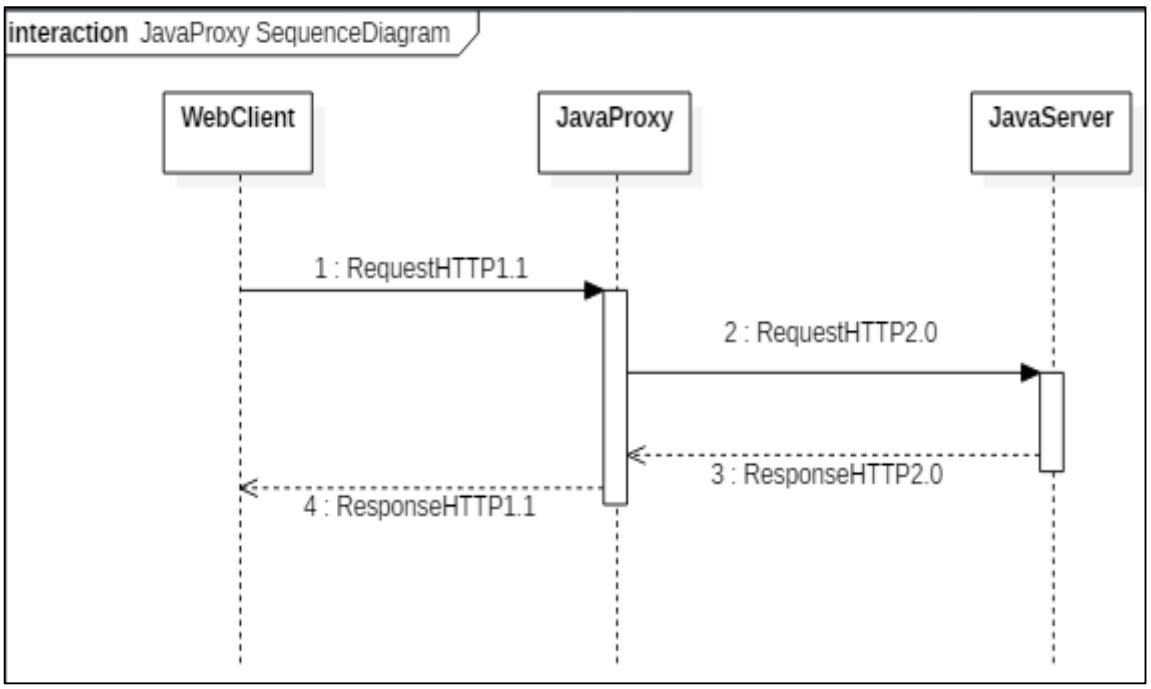

Figure III.10 : Java proxy sequence diagram

General Conclusion

In this thesis, we explained the details of what gRPC is and what differentiates it from similar technologies. We also showed what makes gRPC a good fit for numerous applications and what languages can be used with gRPC, and how gRPC use Proto Buffers and HTTP/2 in the good way.

As a complement to our work, we implemented gRPC web with java proxy, then we analyzed the traffic of HTTP packets before and after get in the Envoy proxy, in order to add a built-in support in web gRPC.

In the programming part, we created just the part of how our java proxy received the http 1.1 request from the browser client, and resend it to our server in http 2 specification.

Nevertheless, we did not finish the second step when the java proxy receive http2 response from the server and resend it to web client in http 1.1 specification.

Although we might not have covered everything, we tried to provide all the basic, necessary and most needed information related to this topic.

We hope this thesis encourages and helps anyone interested, to get to know and pursue this innovation further.

Bibliography

[1] "Introducing gRPC, a new open source HTTP/2 RPC Framework," *Google Developers Blog*. .
[2] J. Humphries, *Practical gRPC*. 2018.
[3] P. E. Green, *Computer Network Architectures and Protocols*. 1982.
[4] "TCP Client/Server Communication - I/O Device Guide - InterSystems IRIS Data Platform 2019.2." [Online]. Available: https://irisdocs.intersystems.com/irislatest/csp/docbook/DocBook.UI.Page.cls?KEY=GIOD_tcp. [Accessed: 01-Jun-2019].
[5] C. Y. Subhash, *Introduction To Client Sever Computing*. New Age International, 2009.
[6] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*, Ninth edition. Hoboken, NJ: Wiley, 2013.
[7] Z. Tari and O. Bukhres, *Fundamentals of distributed object systems: the CORBA perspective*. New York: Wiley, 2001.
[8] "Cap'n Proto: Introduction." [Online]. Available: https://capnproto.org/. [Accessed: 02-Jun-2019].
[9] B. E. Press, "Is gRPC the Future of Client-Server Communication?," *Bleeding Edge Press*, 19-Jul-2018. .
[10] P. J. Leach, T. Berners-Lee, J. C. Mogul, L. Masinter, R. T. Fielding, and J. Gettys, "Hypertext Transfer Protocol -- HTTP/1.1." [Online]. Available: https://tools.ietf.org/html/rfc2616. [Accessed: 11-Jun-2019].
[11] "RFC 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2)." [Online]. Available: https://datatracker.ietf.org/doc/rfc7540/. [Accessed: 11-Jun-2019].
[12] "Overview Of HTTP/2 Protocol." [Online]. Available: https://www.c-sharpcorner.com/article/http2-protocol/. [Accessed: 02-Jul-2019].
[13] A. Says, "Introduction to gRPC," *Container Solutions*, 01-Mar-2017. .
[14] CNCF [Cloud Native Computing Foundation], *Intro: gRPC-Web - Stanley Cheung & Wenbo Zhu, Google*. .
[15] "Build Real-World Microservices with gRPC," *The New Stack*, 27-Nov-2018. .
[16] "API layers | Google Cloud APIs." [Online]. Available: https://googleapis.github.io/google-cloud-dotnet/docs/guides/api-layers.html. [Accessed: 18-Jun-2019].
[17] Side, "How do we leverage gRPC in a microservices architecture," *Medium*, 19-Mar-2019. .
[18] S. P, "How gRPC is convinced to be chose over REST," *Sankar P*, 07-Mar-2018. .
[19] "gRPC." [Online]. Available: https://grpc.io/docs/. [Accessed: 07-Jun-2019].
[20] "Protocol Buffers," *Google Developers*. [Online]. Available: https://developers.google.com/protocol-buffers/. [Accessed: 18-Jun-2019].
[21] *gRPC for Web Clients. Contribute to grpc/grpc-web development by creating an account on GitHub*. grpc, 2019.

Bibliography

[22] A. Punnen, “Interface GRPC with Web using GRPC-Web and Envoy (possibly the best way forward),” *Hacker Noon*, 09-Aug-2018. [Online]. Available: https://hackernoon.com/interface-grpc-with-web-using-grpc-web-and-envoy-possibly-the-best-way-forward-3ae9671af67. [Accessed: 18-Jun-2019].

[23] L. Perkins, “Envoy and gRPC-Web: a fresh new alternative to REST,” *Envoy Proxy*, 01-Nov-2018. [Online]. Available: https://blog.envoyproxy.io/envoy-and-grpc-web-a-fresh-new-alternative-to-rest-6504ce7eb880. [Accessed: 18-Jun-2019].

[24] “gRPC.” [Online]. Available: https://grpc.io/blog/grpc-web-ga/. [Accessed: 08-Jun-2019].

[25] “gRPC.” [Online]. Available: https://grpc.io/blog/state-of-grpc-web/. [Accessed: 08-Jun-2019].