gRPC framework and its applications

Bachelor’s degree Mohammed Fataka

Bachelor’s thesis
2022-2023

Tomas Bata University in Zlín
Faculty of Applied Informatics

Tomas Bata University in Zlin
Faculty of Applied Informatics
Department of Informatics and Artificial Intelligence

Academic year: 2022/2023

ASSIGNMENT OF BACHELOR THESIS
(project, art work, art performance)

Name and surname:	Mohammed Abdulrazzaq Qas Fataka
Personal number:	A19899
Study programme:	B0613A140021 Software Engineering
Type of Study:	Full-time
Work topic:	Framework gRPC a jeho aplikace
Work topic in English:	gRPC Framework and its Applications

Theses guidelines

1. Prepare the literature review of the thesis topic.
2. Introduce modern architectures for client-server communication and categorize them.
3. Describe the advantages of gRPC with older RPC frameworks — SOAP, RMI and CORBA.
4. Implement a demo application using gRPC. Document the implementation step-by-step.
5. Evaluate the performance of the demo application by performance benchmarks.

Recommended resources:

l. GEEWAX, J. J. and SKEET, Jon. API design patterns. Shelter Island, NY: Manning, 2021. ISBN 978-1-61729-585-0.

2. INDRASIRI, Kasun and KURUPPU, Danesh. gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes. First edition. Beijing Boston: O'Reilly, 2020. ISBN 978-1-4920-5833-5

3. LARSSON, Magnus. Hands-on microservices with Spring Boot and Spring Cloud: build and deploy Java microservices using Spring Cloud, Istio, and Kubernetes. Birmingham Mumbai: Packt, 2019. ISBN 978-1-78961-3476.
4. COSMINA, luliana, HARROP, Rob, SCHAEFER, Chris and HO, Clarence. Pro Spring 5: an in-depth guide to the Spring framework and its tools. Fifth edition. New York, NY: Apress, 2017. ISBN 978-1-4842-2807-4.
5. CORBIN, John R. The art of distributed applications: programming techniques for remote procedure calls. New York: Springer-Verlag, 1991. Sun technical reference library. ISBN 978-0-387-97247-3.

Supervisors of bachelor thesis: Ing. Dulik, Ph.D.
Department of Informatics and Artificial Intelligence

Date of assignment of bachelor thesis: October 5, 2022
Submission deadline of bachelor thesis: May 12, 2023

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.
①

doc. Ing. Jiii Ph.D. m.p. prof. Mgr. Roman Ja5ek, Ph.D., DBA m.p.

Dean Head of Department

ABSTRAKT

Cílem této bakalářské práce je provést hloubkovou analýzu frameworku gRPC a jeho různých aplikací a ukázat přednosti gRPC oproti jiným protokolům v oblasti mikroslužeb a aplikací. Práce vysvětlue základní koncepty a způsoby použití gRPC pro komplexní porozumění jeho funkcím. Zaměřuje se na oblast serverových "backend" funkcionalit s využitím jazyka Java a frameworku Spring Boot. Kromě toho tato práce používá reaktivní programovací přístup a nejnovější technologie dostupné v době psaní práce. Dalším významným bodem, kterým se tato práce zabývá, je porovnání protokolu gRPC s jinými protokoly včetně testů, prokázujících vyšší výkon gRPC.
Závěrem lze říci, že tato bakalářská práce je cenným zdrojem informací pro vývojáře, kteří se snaží využít gRPC pro mikroslužby a vývoj aplikací. Prostřednictvím komplexní analýzy a srovnávacích testů prokazuje nadřazenost protokolu gRPC nad ostatními protokoly a poskytuje pádné argumenty pro jeho implementaci v různých vývojových kontextech.
Klíčová slova:

ABSTRACT

This bachelor's thesis aims to provide an in-depth analysis of the gRPC framework and its various applications. The primary objective of this thesis is to demonstrate the superior capabilities of gRPC over other protocols concerning the development of microservices and applications. One of the key aspects that sets this thesis apart is its utilization of Spring Boot to fill the gap in gRPC demonstrations. Additionally, the thesis adopts a reactive programming approach utilizing the latest technologies available at the time of writing. Another significant point addressed in this thesis is benchmarking gRPC against other protocols, demonstrating its superior performance. Furthermore, this thesis delves into the core concepts and usages of the gRPC framework to provide a comprehensive understanding of its functionalities. Specifically, this thesis is written for backend purposes, utilizing the Java language.
In conclusion, this bachelor's thesis is a valuable resource for developers seeking to utilize gRPC for microservices and application development. Through its comprehensive analysis and benchmarking, it establishes the superiority of gRPC over other protocols, making a strong case for its implementation in various development contexts.

Keywords: gRPC, microservices, spring boot, benchmarking, backend, java , protocols

ACKNOWLEDGEMENTS

I am extending my heartfelt appreciation to my family, friends, and professors for their unwavering support during the completion of their thesis. The encouragement, guidance, and motivation provided by these individuals were instrumental in the successful culmination of this academic endeavor.

I also want to express my gratitude towards Mr. Ing. Tomáš Dulík Ph.D. for his significant guidance and advice provided during the duration of this thesis. His proficiency and guidance have played an important role in enhancing my comprehension and evaluation of the topic.

I hereby declare that the print version of my Bachelor's thesis and the electronic version of my thesis deposited in the IS/STAG system are identical.

CONTENTS

INTRODUCTION

In today's world, the ubiquitous presence of millions of devices has led to the widespread use of RPC for various services. As technology advances, expectations of protocols and frameworks also increase. When developing an application or system, the first choice of protocol is often REST, but this can pose challenges when the system or application grows and requires faster and more efficient performance.

In my company, we had been using the traditional REST protocol primarily for communication between our microservices. However, we soon realized that this was causing more trouble than it was worth. REST proved to be inefficient and slow, especially when dealing with large amounts of data. We were constantly faced with bottlenecks and performance issues that affected our overall system's reliability and stability. After thorough research and evaluation of alternative protocols, we decided to replace REST with gRPC. The results were remarkable, I was personally amazed and surprised by its performance. The data transfer speed was significantly improved, and the framework's ease of use made it a more convenient option for our developers. The performance benefits of gRPC were immediate and evident, leading to more streamlined communication between our microservices. We were able to overcome the challenges faced with REST, and our system's overall performance and reliability improved dramatically as a result.

The thesis explores the different client-server communication architectures, APIs, and modern architectures, with a specific focus on gRPC as a new client-server communication architecture. The first section provides a literature review of client-server communication architectures, and API protocols, including the differences between RPC and REST, and why gRPC is a promising new solution. The second section delves into the concept of gRPC, including networking, communication types, and protocol buffers. The final section includes a demo application that provides an overview of gRPC's implementation and usage, as well as its security and authentication features, scalability in distributed systems and microservices, and benchmarking comparison with REST. The thesis concludes with a summary of the advantages of using gRPC over other frameworks, as well as its potential impact on the future of client-server communication architectures.

I. THEORY

1 LITERATURE REVIEW

Client-server communication architectures have been developing for decades, beginning with SOAP and RPC and, more recently, with REST and gRPC. These designs make it simple to create distributed applications by allowing clients and servers to connect via a network.
The RPC protocol enables software to use a distant computer's function or method like a local one. The SOAP protocol, on the other hand, uses XML to exchange structured data between applications. Based on HTTP, the ubiquitous web service architecture known as REST offers clients and servers a quick and easy method to exchange data.

gRPC has become a cutting-edge architecture for client-server communication in recent years. It is an open-source, high-performance framework that supports various programming languages and serializes data using Protocol Buffers. gRPC has become famous for distributed systems because of its built-in capabilities, including streaming, bi-directional communication, and load balancing. Modern architectures like gRPC are becoming more and more critical as the need for faster and more reliable communication between client and server develops. This literature study will provide an in-depth analysis of these architectures, compare them to conventional RPC and SOAP frameworks, and discuss their unique characteristics. It will also look at the difficulties that need to be solved to increase these design's efficacy and future applications.

1.1 Client-server communication architectures

The client-server architecture describes a system that hosts, delivers, and manages most of the requested client resources and services. All requests and services are sent across a network, the networking computer model, or the client-server network under this paradigm.

Client-server architecture is a network application that divides duties and workloads among clients and servers that sit on the same system or are interconnected through a computer network. In a client-server design, on client side several computer or other devices are often linked to a central server through the Internet or another network.

The client can do certain calls or operations with server like making a request call for data, which the server may accepts and fulfills by returning a response that holds data to be returned to the client.

The purpose of this technology is the need of accessibility and performance. Quick-to-act systems in every aspect of IoT, the client-server architecture makes it more efficient. It is coming to a point where it is hard to survive without it. This architecture it has been worked on, and it has been proven that it can provide this need to fulfill the challenges modern services face alongside this technology is getting more significant and evolving [1].

1.1.1 Evolution of client-server architectures

The client-server architecture was born when mainframe computer was debuted, which enabled several users to access a centralized system. Nevertheless, the 1980s experienced the beginning of client-server architecture as networked systems, and personal computers gained popularity [2].

Two-tier architecture consists of a client and a server. In such architecture, the client is the first tier, and the second tier is the database server and application together. In most cases, the second tier is responsible for business logic functionalities. However, this architecture had minor issues regarding security. Thus, three-tier architecture was introduced [3].

SOA was introduced in the late 1990s. It works so that components, services, and functionalities work to accomplish a job or task.
The purpose of this architecture was to avoid a monolithic model for applications. In other words, before the 1990s, applications had to be shut down, fixed issues, and re-deploy whenever one service is not working from many [4].

Modern technology is shifting towards client-server architectures that leverage cloud platforms. Such platforms offer highly flexible and scalable infrastructure, based on the principles of serverless computing and containers. This allows clients to access services from anywhere, while servers can easily adjust their capacity to meet demand.

Lastly, client-server design has progressed from two-tier to three-tier SOA, microservices, and cloud-based systems. Each iteration offered increased scalability, performance, and security over the one before it. Resources may be used thanks to the flexibility and scalability of current designs effectively [5].

1.1.2 Modern architectures

Today's modern architecture is mainly used to the design and organize software systems, with having goal to achieve scalability, flexibility, and maintainability. It encompasses a wide range of approaches, technologies and patterns, many of these are accomplished by the need to handle the increased complexity and scale of modern applications and services.

Some of the key characteristics of modern architecture include:
Decoupling and modularity: Modern architectures are designed to be loosely coupled and highly modular, making it easier to scale, maintain, organize, and evolve the system over time [6].

Cloud-based and distributed: Many modern architectures are built for the cloud, taking full advantage of the benefits of distributed computing and storage, it is good for avoiding physical server and devices that are not needed to be run considering cloud approaches, this characteristic can be achieved in could-native architecture.
Automation and self-healing: Modern architectures often employ automation to handle tasks such as scaling, monitoring, and self-healing, health check in order to minimize manual intervention and monitoring through technical personals.

Resilience and fault-tolerance: Modern architectures are designed to be resilient and fault-tolerant, meaning that they can continue to operate even in the face of failures or other disruptions.

Real-time and event-driven: Many modern architectures are designed to handle real-time and event-driven workloads, meaning that they can respond to changes in the environment quickly and efficiently, thus they can perform needed changes just in time [7].

All of these characteristics are changed by the need to handle the increased complexity and scale of modern applications and services and can be seen in different architectures and technologies such as Microservices, Cloud-Native, Serverless, Event-Driven, and Containerization, these architectures have their own set of trade-offs and specific use cases, and often used together in a hybrid way to make the most of their benefits.

1.2 APIs

After wrapping up exploration of software architectures and client-server architectures, it’s time to focus on another key concept in modern software development: APIs. While client-server architectures provide a framework for enabling communication between disparate systems, APIs take this idea further by providing developers with tools for building highly flexible and interoperable software components. This section will examine APIs, how they work, and the benefits they offer to developers, businesses, and end-users. It is essential to know about API fundamentals an API is a standard that allows software services and components to communicate. An API specification may form routines, data structures, object types, and variable specifications.

APIs are used widely across all other IoT. APIs can be defined as a set of protocols and tools that will allow various applications to communicate with each other. The purpose of APIs is to make creating applications easier that communicate with other applications. Thus it is easier to build complex applications. API is typically used for defining types of requests that can be made and what kind of response can be expected.

Web-based applications interact with each other through web API; an example would be a booking flight web application. This service would likely use web API to communicate with the airline’s system. First, the web application sends requests over the internet to the airline’s server, which is the server in this example. Then the server processes the request and responds to the requesting application.

One of the main advantages of APIs is that they enable developers to reuse existing code and functionality, reducing development time and costs. By enabling various applications to function together seamlessly, APIs can also enhance the user experience. In addition, APIs can give users access to information and services that may not be accessible in other ways. However, designing and maintaining APIs can be challenging. One of the main challenges is ensuring backward compatibility, which means ensuring that older versions of an API continue to work with newer versions. Another challenge is ensuring security, as APIs may expose sensitive data and functionality to unauthorized users [8].

1.2.1 Types of API

There are four APIs: Open API, Partner API, Composite API, and Internal API.
An open API or public API is accessible and serviceable by any outside developer or company. A company that develops and offers an open API will have a business model that includes communicating its apps and data with other companies. Authentication and authorization for open APIs are generally modest. A company may also monetize the API by charging a fee per call to use the open API; they are typically available to anyone who wants to use them and can be accessed over the internet using standard protocols such as HTTP.
A partner API is a way to facilitate business-to-business interactions that are only available to explicitly selected and approved outside developers or API users. Partner API is similar to open API in that they allow external developers to access a program's or service's functionality, but they have more strict usage guidelines. They may require more extensive registration or approval processes. For example, if a company wishes to selectively share customer data with third-party CRM companies, a partner API can connect the internal customer data system with those third-party parties and provide all security requirements; no other API use is authorized.

An internal (or private) API is solely meant for usage within the corporation to link systems, data, and resources. Internal APIs share functionality or data between different organizational applications or systems. As a result, they can improve the efficiency and effectiveness of the company's internal processes.

A Composite API is a type of API that allows developers to access multiple endpoints in a single call. This can simplify the process of interacting with multiple APIs by combining multiple requests into a single call, reducing the number of round trips to the server, and improving performance.

Composite APIs can expose complex or multi-step operations as a single, simple-to-use API. For example, a Composite API that combines data from multiple sources, such as a customer's account information and purchase history, can create a complete customer profile. Composite APIs can also create a higher-level abstraction of multiple underlying APIs, making them easier to use and understand. This can be specifically useful when working with APIs with different interfaces or levels of complexity.
Overall, Composite APIs can help developers to work more efficiently and effectively with multiple APIs by providing a simplified and consistent interface for accessing multiple endpoints.

1.2.2 API Protocols

REST, SOAP, and RPC API protocols facilitate communication between various software applications. Due to its ease of use and capability to specify routes using a URL, REST is one of the most extensively used API protocols. However, REST APIs may only communicate with text and are restricted to the HTTP protocol for data transmission. Text transmissions may be made more sophisticated by formatting options. However, REST and HTTP demand that developers utilize creativity and work within their limitations.

In addition to HTTP, SOAP is a significant API standard that may connect through more internet communication protocols. While it is more constrained and challenging to design and maintain than REST, it is more adaptable. In complicated systems where dependability is more important than speed or usability, SOAP requests consume more bandwidth than REST requests but are more dependable.

GraphQL is a distinct query language with best practices, not a distinct protocol. For example, GraphQL APIs generally feature a single endpoint that may handle infinite data schemas, unlike REST APIs, which have many endpoints to represent various data schemas. This implies that to create a query that combines those elements in whatever order they desire, an API user must be familiar with the data fields that are accessible. Data is then returned in the form of the schema given by the query, and the query is submitted as the payload of an HTTP POST request.

RPC protocols call a method instead of a data resource, which is how they vary from SOAP and REST APIs. They are used in distributed client-server systems when the payload is minimal, and only the parameters are needed to invoke the methods. In addition, RPC APIs are typically private since they need high security and trust between producers and consumers.

Google introduced gRPC, a brand-new variety of RPC, in 2015. Data is serialized and parsed using Protocol Buffers, which offer better flexibility than XML and a quicker response time than the JSON encoding used in REST. The foundation of gRPC is HTTP/2, a revision of the HTTP protocol introduced in 2015.

In conclusion, each API protocol has unique advantages and disadvantages and is better suited for specific applications. For example, RPC is lightweight and perfect for distributed client-server systems, SOAP is trustworthy and predictable, and REST is straightforward [9].

1.3 RPC vs REST

After knowing about API types and API protocols, this section gives closer look at RPC and REST specifically.
Between RPC and REST, there are several key distinctions. The most major distinction is that REST is a software architectural style and RPC is a protocol that allows one program to request a service from another program situated on a separate computer on a network.

RPC involves clients sending requests to a server, which performs a process and delivers the client response. Often, this procedure is asynchronous, which means that the client must wait for a response from the server before it can go on. As a result, RPC is often used for activities that call for precise control and managing massive amounts of data.

The REST architectural style, on the other hand, outlines several guidelines that have to be followed while developing online services. RESTful services employ resources to express an application's state and work with the current protocols and standards of the web, such as HTTP and URLs.

Most stateless services, even RESTful ones, do not save any data about past client requests. Compared to RPC services, they are thus more scalable and simpler to manage. Furthermore, RESTful services are also more adaptable since they can be accessed from any system or computer language that supports HTTP requests.

To sum up, RPC and REST are two alternative methods for managing network communication. REST is a more versatile and scalable technique perfect for building web services that can be accessed from any device or computer language. At the same time, RPC is best suited for managing massive amounts of data and jobs that need precise control [10].

1.3.1 Why RPC

RPC has had tremendous success. The majority of distributed applications written nowadays make use of an RPC runtime, such as gRPC or Apache Thrift . Its simplicity and strong semantics of RPC's programming paradigm are the key to its success. As a result, parameters and return values are immutable by definition. Because no distributed coordination is required, these basic semantics allow for very efficient and reliable implementations while staying suitable for a wide range of distributed applications. RPC's universality also allows for interoperability, any program that knows RPC may communicate with another application that speaks RPC [11].

1.3.2 RPC Types

There are several types of RPC communications from both server-client sides, in the difference of data exchange, way of transfer, scale of requests and response, client or server side's presence in time of connection most common types that companies use for different purposes are defined below which have been used in various fields and systems

Synchronous:

Synchronous is the most common mode of operation. The client makes a call and proceeds when the server responds. The RPC protocol enables the development of client-server applications by utilizing a demand/response protocol with transaction management. The client is blocked until the server responds or a user-defined optional timeout occurs. RPC ensures at-most-once semantics for request delivery. It also ensures that the response obtained by a client is unquestionably that of the server and adequately corresponds to the request (and not to a former request to which the response might have been lost). RPC also permits a client to be unblocked (with an error result) if the server is unavailable or has failed before responding. Finally, this protocol encourages the spread of abortion via the RPC [12].

Asynchronous:

Unlike synchronous, Asynchronous calls do not block clients, and responses are received when needed. Asynchronous RPC calls are classified into two categories based on whether or not they return a value. Most asynchronous RPC systems only support non-returning calls [13].

Nonblocking:

The client makes a call and then goes about its business. Commonly, the server does not respond. To obtain the result, the client must either poll the server or make alternative arrangements, such as callback RPC. Nonblocking RPC does accomplish by setting the RPC timeout value to zero—either by changing the values in the timeout parameter of the client call) to zero or by using client control() to set the default timeout value to zero before performing any RPC queries using the supplied client handle. Because a nonblocking RPC request does not wait for a response from the server, server creation and its response shall be void, so there will be no response for the request [14].

Batching:

Batching is a feature that allows submitting several nonblocking clients calls in a single batch; batching allows a client to send an infinite number of call messages to a server. Batching uses dependable byte stream technologies such as TCP/IP for transport. In the case of batching, the client never waits for a server response, and the server never responds to batched requests. In order to flush the pipeline, a sequence of batch calls generally is concluded by a genuine, non-batched RPC. According to the RPC architecture, clients submit a call message and wait for servers to respond that the call was successful. Thus, while servers are processing a call, clients do not compute. On the other hand, the customer may not desire or require an acknowledgment for every communication delivered. Clients can thus utilize RPC batch features to continue computing while waiting for a response [15].

Broadcast RPC:

RPC clients can broadcast messages to several servers and get all the responses. The client sends a broadcast packet to the network and waits for multiple answers in broadcast RPC-based protocols. Broadcast RPC exclusively employs packet-based protocols for transport, such as User Datagram Protocol/Internet Protocol. Servers implementing broadcast protocols in only case they answer when the request is successful and keep silent while faults occur. Broadcast RPC requires the RPC port mapper service to function. The port map daemon is responsible for converting RPC program numbers into Internet protocol port numbers [16].

Callback RPC:

Callback RPC is a communication protocol that allows a server to initiate a function call on a client. Unlike traditional RPC, where there is a request sent from the client to the server, and the server responds with any form of result, in callback RPC, the server initiates the function call on the client, and the client returns the result to the server. Callback RPCs are often used when the server needs to receive real-time updates or notifications from the client. Thus it can be useful in a wide range of applications, such as real-time communication, online gaming, and distributed systems [17].

1.3.3 gRPC new client-server communication architecture

Although several types of RPC have been used in distributed systems, there are newer architectures developed to address some of the limitations of those traditional approaches. One of these architectures is gRPC, a high-performance, open-source framework for building distributed systems. gRPC uses a new client-server communication architecture designed to be faster and more efficient than traditional RPC protocols. Let us now explore the details of this new architecture and how it differs from the types of RPC that we have discussed.

gRPC is a framework for RPCs based on the standard buffer serialization protocol. However, this is compatible with other widely used RPC protocols, Programming languages, and platforms. gRPC is a popular choice for constructing large-scale distributed systems. GRPC's design allows it to be quick, efficient, and scalable.

gRPC, like standard RPC protocols, enables clients to invoke remote procedures or methods on a server, passing parameters and receiving a response. However, gRPC also enables streaming, bidirectional streaming, and other sophisticated capabilities that enable clients and servers to communicate more complicated and flexibly.

As the underlying transport protocol for gRPC, HTTP/2 offers several advantages over HTTP/1.x. For instance, HTTP/2 provides multiplexing, enabling the transmission of numerous requests and responses over a single connection. Lowering the latency and overhead connected with creating and maintaining many connections can enhance network communication performance.

Overall, gRPC is a very robust and adaptable choice for developing distributed systems since it combines the familiarity and ease of use of older RPC protocols with having the performance and scalability advantages of more recent technologies like HTTP/2 [18].

2 MODERN ARCHITECTURES

Modern client-server communication architectures have developed to offer more effective and adaptable data-sharing methods between systems. The conventional client-server model has been expanded upon and improved to solve the shortcomings of earlier communication designs and satisfy the requirements of modern distributed systems.

GraphQL is a newer client-server communication architecture. It offers a single endpoint through which numerous resources can be retrieved. With GraphQL, clients can only request the data they require, minimizing the amount of data delivered and enhancing performance, in contrast to REST, which mandates that clients obtain a predetermined data set for each request. In addition, GraphQL is a viable option for creating APIs because it is language-neutral and compatible with any backend data source.

There are event-driven architectures for client-server communications in addition to request-response systems. In event-driven architectures, the server notifies the clients when specific events occur. When real-time data processing is necessary, this is helpful. For example, publish-subscribe architecture is one type of event-driven architecture. Customers who subscribe to particular topics are notified when new messages are released. This technique is frequently employed in messaging systems like Apache Kafka and RabbitMQ.

Last, gRPC is a modern client-server communication architecture that relies on RPCs. Standard protocol buffer serialization protocol and HTTP/2 serve as the foundational protocols for gRPC. Being quick, effective, and scalable by design makes it a popular option for creating massively distributed systems. Clients make requests and are answered by servers in a request-response architecture, such as gRPC.

Generally, request-response and event-driven patterns can be used to design distributed systems using contemporary client-server architectures. Each architecture, however, has advantages and disadvantages, and the chosen one will rely on the particular needs of the constructed system.

2.1 Categorization of modern architectures

There are many software architectures build throughout history of software development, and it's essential for developers to understand different software architectures, modern architectures are categorized by their ability to handle complex tasks with a result of high efficiency and scalability. Categorization of these software architectures can be on various criteria for example design principles, use cases, performance and their functionality.
We would look into few of these architectures and explain them in detail.

2.2 Request-Response architecture

Request-response architectures are very common pattern in computer networking a software development, it used widely used between microservices and apps, where a client sends a request to a server and the server sends back a response. The client and server communicate over a network using a specific protocol, such as followings.

HTTP-based:
In this architecture, a web browser (client) sends an HTTP request to a web server (server) to retrieve a webpage or data in various format. The web server responds with the webpage content and other information, such as cookies, header, HTML content, it can be simple plain text.

RESTful APIs:
REST is an architectural pattern for creating web services or APIs. A client sends an HTTP request to a RESTful API to retrieve or manipulate data, and the API sends back a response with the requested data, it comes with different formats of data to be exchanged on.

RPC based architectures:
gRPC, XML-RPC, and JSON-RPC are examples of this category. These architectures are based on the idea of calling a remote function or procedure, and receiving a response, it can be bidirectional or non-bidirectional, it can be also one client to many servers at same time and vice versa [19].

All of these examples follow the basic pattern of a client sending a request and a server sending back a response. The specific details, such as the protocol used, the format of the data, and the nature of the request and response, can vary depending on the specific application or service and its goal to achieve.

2.3 Event driven architecture

Event-driven architectures are a software design pattern where the flow of the application is determined by the occurrence of specific events, there is not much going in application going unless and event is occurred. In this pattern, the system reacts to specific events or changes in its environment, rather than constantly polling for updates, such architectures can be categorized in following.

Web Sockets: web Sockets is a protocol for real-time, two-way communication between a web browser and a server. Unlike REST, WebSocket maintains a persistent connection between the client and server, allowing for low-latency communication.

Pub/Sub: Pub/Sub pattern is a communication pattern in which a publisher sends a message to a topic and any subscriber who is interested in that topic will receive the message, this pattern is used for asynchronous communications [20].

2.4 Microservice architecture

Microservices are commonly used to make an application's functionality separated from one processor, meaning if one of the functionalities of the application does not work, the application should stay alive and functioning because assumingly only one microservice is down, this pattern makes application components run independent and slightly coupled services, each of these components are responsible for executing a task in the application. When it comes to building, deploying, developing, and scaling, each microservice is done independently, allowing teams to work on different services simultaneously without interfering with each other. Thus it makes the application more flexible, scalable, and easier to test, deploy, and maintain [21].

2.5 Broker architecture

Broker architecture facilitates communication between components or services in distributed systems. This architecture has a main central component called BROKER acts as an intermediary between other components and services.

The flow of this architecture is from clients or components sending a request to the broker. Then the broker decides which component or service is appropriate to receive that particular request by routing the request to that component or service.

Broker architecture is typically used for messaging, passing messages from one component or service to another. Thus broker is responsible for routing those messages between each point to ensure reliable message delivery, and supporting advanced features like message queuing, filtering, and transformation.

Usage of this architecture can be seen in messaging systems and frameworks like Apache Kafka, RabbitMQ, and ActiveMQ. It can also be used in other cases, such as service-oriented and microservices architectures [22].

Different architectures have different ability in various criteria as well as their vulnerability, use cases differ from criteria to another, it's also possible to combine several architectures for its best benefit in particular situations.

3 GRPC AND OLDER RPC FRAMEWORKS

It's hard to overlook the importance of communication protocols when looking at current designs in today's software development world. For many years, RPC frameworks have been an essential aspect of developing distributed systems. But, as technology progressed, so did the expectations placed on these frameworks. As a result, a new challenger in the field has emerged: gRPC. Let's look at how gRPC differs from earlier RPC frameworks and what advantages it provides to current designs.

3.1 SOAP, RMI and CORBA

SOAP, RMI, and CORBA are all RPC frameworks that allow disparate systems to communicate across a network. Yet, there are some significant distinctions in their work and underlying protocols.

Similarities:

- These frameworks are intended to allow remote systems to call operations on a distant server as if they were local, abstracting away network connection specifics.
- They all employ a middleware layer to govern client-server connection, allowing efficient and secure data transfer.
- They all support many languages and platforms, making system integration easy.
- They are all built on an object-oriented paradigm, allowing objects and methods to be used across several systems.

SOAP:

- SOAP is a network protocol for exchanging XML-based communications.
- It employs HTTP as a transport protocol. It may be used with other underlying protocols such as SMTP or FTP as well as RPC.
- SOAP is language and platform-neutral, which means it may be utilized with any XML-supporting programming language and platform.
- SOAP is frequently used in web services because it provides a standardized data-sharing method across various systems.[23]

RMI:

- RMI is a Java-based RPC system that allows Java objects to call methods on remote objects on another JVM.
- It exchanges data between the client and server via a binary protocol, which is more efficient than SOAP.
- RMI is confined to Java platforms and requires a JVM to run on both the client and the server.
- RMI allows objects to be sent as arguments, making complicated data structures easier to operate.

CORBA:

- CORBA is a language-agnostic, platform-agnostic middleware that allows diverse systems to communicate easily.
- It describes the interface of objects using an object-oriented IDL, allowing clients to access distant objects through a standard interface.
- CORBA supports various transport protocols such as TCP/IP, IIOP, and DCE.
- Various programming languages are supported by CORBA, including C++, Java, and Python.

While SOAP, RMI, and CORBA all serve the same purpose of allowing remote procedure operations, they differ in their underlying protocols, platform/language support, and amount of abstraction. Thus, developers shall pick the framework that can be fulfilling the best demands of their particular project [24].

3.2 Advantages of gRPC over older RPC frameworks

Compared to more traditional RPC frameworks like CORBA and RMI, gRPC is superior.

- gRPC's performance is excellent because it is built on top of a lightweight and efficient transport protocol—HTTP/2—and a binary serialization format called Protocol Buffers. As a result, more connections and data streams may be processed simultaneously, which improves performance and speeds up communication compared to earlier frameworks.

- gRPC is compatible with many other systems and languages. These systems and languages include C++, Java, Python, Ruby, etc. In addition, it works with mobile and web-based clients, making it a great option for today's dispersed systems.

- gRPC is very scalable; it can process massive volumes of data and several connections simultaneously. Furthermore, it facilitates dynamic scalability by supporting load balancing and service discovery.

- Code is automatically created in different languages thanks to gRPC's IDL for creating the service contract. As a result, it simplifies and shortens the time required to write client and server code while decreasing the likelihood of mistakes.

- Safety gRPC's support for SSL/TLS encryption and authentication helps keep client-server exchanges safe. In addition, authentication protocols such as OAuth2, JWT, and others are supported.

Overall, gRPC is way more advanced than previous RPC frameworks like CORBA and RMI, particularly in speed, scalability, and portability [25].

4 CONCEPT OF GRPC

As the shortcomings of traditional RPC frameworks become more obvious, software engineers have been adopting more modern alternatives like gRPC. So what is gRPC, and why is it considered the wave of the future for RPCs?

gRPC is a contemporary open-source RPC framework that may run in any environment. It's programmable support for load balancing, tracing, health monitoring, and authentication may efficiently link services within and across data centers. It is also valuable for the final mile of distributed computing, connecting devices, mobile apps, and browsers to backend services.

A client program can call a method on a server application on a separate computer as if it were a local object with gRPC, making it easier to construct distributed applications and services. gRPC, like many RPC systems, is built around creating a service, specifying the methods that may be called remotely, along with their parameters and return types. The server implements this interface and launches a gRPC server to process client requests. On the client side, a stub (also known as a client in specific languages) exposes the same methods as the server.

gRPC clients and servers can run and communicate in various contexts, from Google servers to the user's desktop. It is possible to build gRPC in any of the gRPC-supported languages. Users may develop a gRPC server in Java with clients in Go, Python, or Ruby, for example. Furthermore, the most recent Google APIs will have gRPC versions of their interfaces, allowing users to seamlessly integrate Google functionality into their applications [26].

Each of the gRPC-supported languages is made up of multiple layers that allow users to customize their applications. There are two types of languages: those that use the pure language itself (for example, Java, Go, or C#) and those that use C-core. Most languages, such as Python and Ruby, are designed as layers on top of the C-core framework. In C-core-based languages, the application calls a stub of the current language.

The calls are intercepted and handed to a library, which converts them into C language calls; gRPC is based on the Protocol Buffers data serialization format and uses the HTTP/2 protocol for transport. It encodes RPC calls as HTTP/2, encrypts them, and delivers them to the network [27].

gRPC allows for efficient and stable communication between services. It also features bi-directional streaming, flow control, and flow control and is designed to work well in high-latency or unreliable network environments.

4.1 Networking in gRPC

When it comes to gRPC, networking is a critical component of its functionality. Essentially, gRPC uses the RPC protocol for establishing a stable connection between the client and server. This protocol allows for the exchange of data between the two parties, which is transmitted over the network.

Networking in gRPC is handled via RPC protocol and built on top of HTTP/2. RPC protocol is powerful enough to provide a stable connection between server and client, as well as their data exchange from receiving and responding from the provided connection by RPC. For example, when a client calls for a request from the server, the client will trigger a request sent through the network to the server; the request contains methods definitions such as name, parameters, and metadata which holds all information for authentication and security. From the other side, the server receives the request and does some computation or functions for its purpose, and then sends the response back to the client via RPC's provided network.
What makes gRPC dominant is that it provides unique features; one of them is live bi-directional streams from server and client; it is handy for situations when fast actions need to be performed in such short time situations; it can be video or gaming; thus both client and server are able to exchange or send data at the same time and in the same network connection.

One of the keys that make gRPC perform very well in data exchange is its serialization and deserialization of data, which is done by protocol buffers and is very efficient; protocol buffers are language and platform-neutral data formats [28].

4.2 Communication types of gRPC

As gRPC being efficient network communication provider, it has to have various communication types, gRPC allows you to define four different types of service methods which are built under different types of communication *Unary RPCs, Server streaming RPCs, Client streaming RPCs, Bidirectional streaming RPCs*

The Unary model is a request-response communication structure where a single request is sent from client to server, then server processes the request and sends a response back to client, in unary model calls can be synchronous and blocking or asynchronous and non-blocking, depending on its purpose, an example for that method would be as bellow [29].

```
rpc SayHello(HelloRequest) returns (HelloResponse);
```

Unlike the unary model, the server streaming model is a type of gRPC communication where the client sends a single request to the server, and the server processes the request and sends a stream of responses to the client; this mechanism works by sending a large amount of data from server to client in chunks rather than sending it all in one; this makes client be able to process each chunk upon their arrivals and not wait until an entire response has arrived, this type is handy for cases where the client is a massive consumer of data.

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);
```

The client streaming model is the opposite of the server streaming model, where in this model, the client is sending a stream of requests to the server in chunks; thus, the server can start processing them upon the first chunks arrival rather than waiting for the entire request to arrive, this model is type handy for cases where the server is a massive consumer of data from its clients.

```
rpc LotsOfGreetings(stream HelloRequest) return (HelloResponse);
```

The bidirectional streaming model works as the combined version of both client and server stream models, in which both sides submit a message sequence via a read-write stream. Clients and servers can do any related operations in any order they want because the two streams operate independently: for example, the server could wait to receive all client messages before sending its responses, or it could alternately receive a chunk and then respond to that specific chunk, or some other combination of requests and responses. Messages in each stream are kept in their original sequence, which makes both sides to have unique identification of any item in the stream or any chunk [30].

```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);
```

4.3 Protocol buffers

gRPC depends heavily on Protocol Buffers, often known as protobufs, therefore familiarity with these data structures is essential. In reality, gRPC's strength lies in the fact that it employs protobufs as its data serialization format, making it a useful tool for facilitating rapid and reliable communication between clients and servers.

Protocol Buffers is developed by Google, as a data format for exchanging structured data between various computer languages. Data transfers utilizing protobufs are quicker and more effective because they employ a small binary rather than text-based formats like XML or JSON.

In order to start with protobufs, first must specify the structure of the user's data in a **.proto** file before the user can use protobufs. The variable names, types, and ordering of the fields in the data are specified in this file. Then, it is ready to quickly serialize and deserialize data into the compact binary format by creating code in a chosen programming language using the Protocol Buffers compiler.

Protobufs are faster and smaller than text-based formats, making them perfect for sending vast amounts of data across a network. Furthermore, protobufs are backward-compatible, allowing you to expand your data structure without causing old code to malfunction. In short, everyone who wants to store and share structured data between different programming languages effectively should consider protobufs. They provide a quick and straightforward to use and maintain solution.[31]

4.4 gRPC server definition

After knowing what's Protobufs and types of communications we can develop various types of gRPC service definitions in order to establish communication between server-client.
Protoc is first step to take in order to use Protobuf to generate definitions of gRPC service
It can be installed in all operating systems, following is how to install Protoc
Linux:

```
$ apt install -y protobuf-compiler
$ protoc --version  # Version must be 3+
```

Mac:

```
$ brew install protobuf
$ protoc --version  # Version must be 3+
```

IDL for gRPC is protobufs, but this can be changed to other alternatives if desired.
There are 3 syntax versions for **.proto** files, each of them has its own way of defining data and objects, current latest syntax version is 3. Following are simple message payloads that can be used for demonstration of various types of gRPC service definition.

Fields in proto syntax 3 start by type-of-field + name-of-field + order-of-field

```
syntax = "proto3";

message GreetingRequest {
  string requestMessage = 1;
}

message GreetingResponse {
  string responseMessage = 1;
}
```

This is simple Unary service, and it will be defined as RPC service accepting payload of **GreetingRequest** for server client is receiving **GreetingResponse** back.

```
service GreetingService {
  rpc greet (GreetingRequest) returns (GreetingResponse);
}
```

In order to convert previous Unary example to Server-stream RPC it has to be defined as following.

```
service GreetingService {
  rpc LotsOfGreeting(GreetingRequest) returns (stream GreetingResponse);
}
```

Client-streaming RPC is as following.

```
service GreetingService {
  rpc LotsOfGreetings(stream GreetingRequest) returns (GreetingResponse);
}
```

And finally bidirectional is as following:

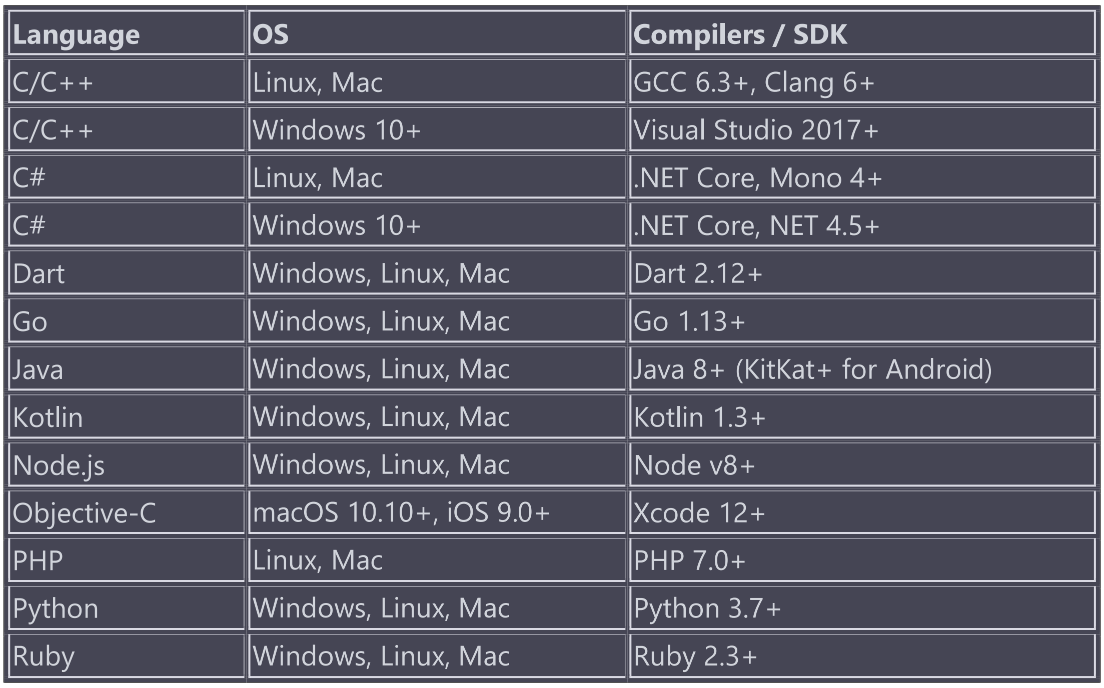
```
service                          GreetingService                          {
  rpc BidiGreet(stream GreetingRequest) returns (stream GreetingResponse);
}
```

4.5 gRPC programming languages and libraries

Various programming languages and systems support gRPC, and appropriate gRPC libraries are offered for every language. C++, Java, Python, and Go are among the most well-liked programming languages for gRPC development, although there are libraries for languages like PHP, Ruby, and Rust. Developers can create gRPC services and clients using the tools, APIs, and utilities offered by each language library.

The fact that gRPC supports various systems and architectures is one benefit of utilizing it. Operating platforms like Linux, macOS, Windows, Android, and iOS can all run gRPC. Additionally, because gRPC supports various programming languages, it is simple for programmers to create microservices that can communicate with one another regardless of the language in which they are written. Below is table of full official supported languages and libraries by gRPC.

Table 1 Supported programming languages in gRPC



Language	OS	Compilers / SDK
C/C++	Linux, Mac	GCC 6.3+, Clang 6+
C/C++	Windows 10+	Visual Studio 2017+
C#	Linux, Mac	.NET Core, Mono 4+
C#	Windows 10+	.NET Core, NET 4.5+
Dart	Windows, Linux, Mac	Dart 2.12+
Go	Windows, Linux, Mac	Go 1.13+
Java	Windows, Linux, Mac	Java 8+ (KitKat+ for Android)
Kotlin	Windows, Linux, Mac	Kotlin 1.3+
Node.js	Windows, Linux, Mac	Node v8+
Objective-C	macOS 10.10+, iOS 9.0+	Xcode 12+
PHP	Linux, Mac	PHP 7.0+
Python	Windows, Linux, Mac	Python 3.7+
Ruby	Windows, Linux, Mac	Ruby 2.3+

II. DEMO APPLICATION

5 APPLICATION STRUCTURE AND OVERVIEW

The present study outlines the application for programming implementation of gRPC, developed under the Spring Boot framework, utilizing the Java programming language and Gradle as its build automation tool. The application comprises three key modules: COMMON, SERVER, and CLIENT. The **common** module is responsible for constructing common developments for client and server modules thus it is imported by both modules. In contrast, the SERVER module acts as an executable microservice that encompasses all aspects related to gRPC server development. Lastly, the CLIENT module is an executable module responsible for gRPC client developments. The application is publicly available on GitHub at https://github.com/hammafataka/gRPC

The scenario for this application would be UTB university has several applications that deal with user management. These applications communicate with the user management microservice to access user information and manage user roles and permissions. To ensure efficient communication between these applications and the user management microservice, gRPC can be used as a high-performance and scalable communication protocol. The user management microservice exposes a gRPC API, defining the available methods for accessing user information, validating user credentials, and managing user roles and permissions. Each application built as a microservice includes a gRPC client that connects to the user management microservice, making fast and efficient communication possible. Additionally, gRPC supports code generation for multiple programming languages, making it easy to integrate with different microservices built with different technologies.

The **common** module, previously referenced, is a non-executable module that serves as a shared module for utility, constants, and protobuf classes. Both the client and server modules import the **common** module. The client module, in contrast, is an executable module that functions as a gRPC client microservice. This module holds gRPC client services, allowing for the execution of gRPC calls. Additionally, it contains resources intended for the demonstration and initiation of gRPC calls via REST, as illustrated in Figure 1. It is important to note that these resources are not designed for any other use of REST.

The **server** module is an executable module that functions as a gRPC server microservice. Its primary function is to house gRPC service implementations that interact with data access layers based on specific business logic. The module also includes a controller intended for person, which serves solely for benchmarking gRPC against REST. Furthermore, it has a **PersonService**, which interacts with repositories responsible for conducting database operations, as demonstrated in Figure 1.

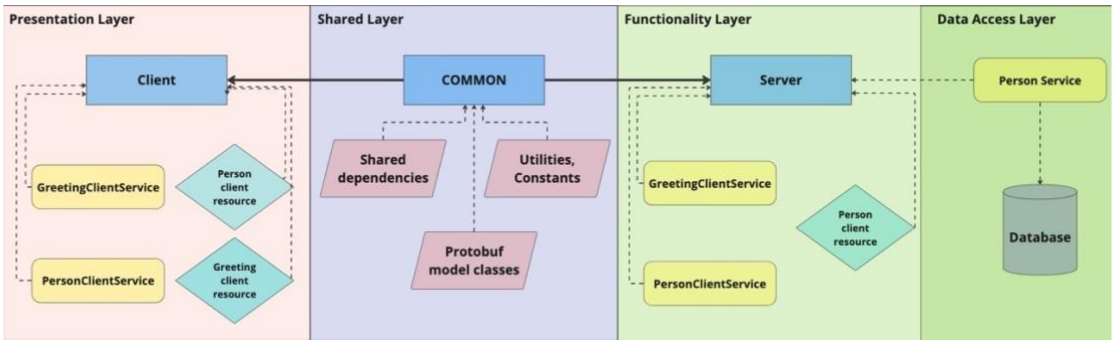


Figure 1 Application modules high level functionality

Client module has two services for gRPC demonstration, equipped with their own resources purely for triggering gRPC calls. Figure 2 shows more details about client module.

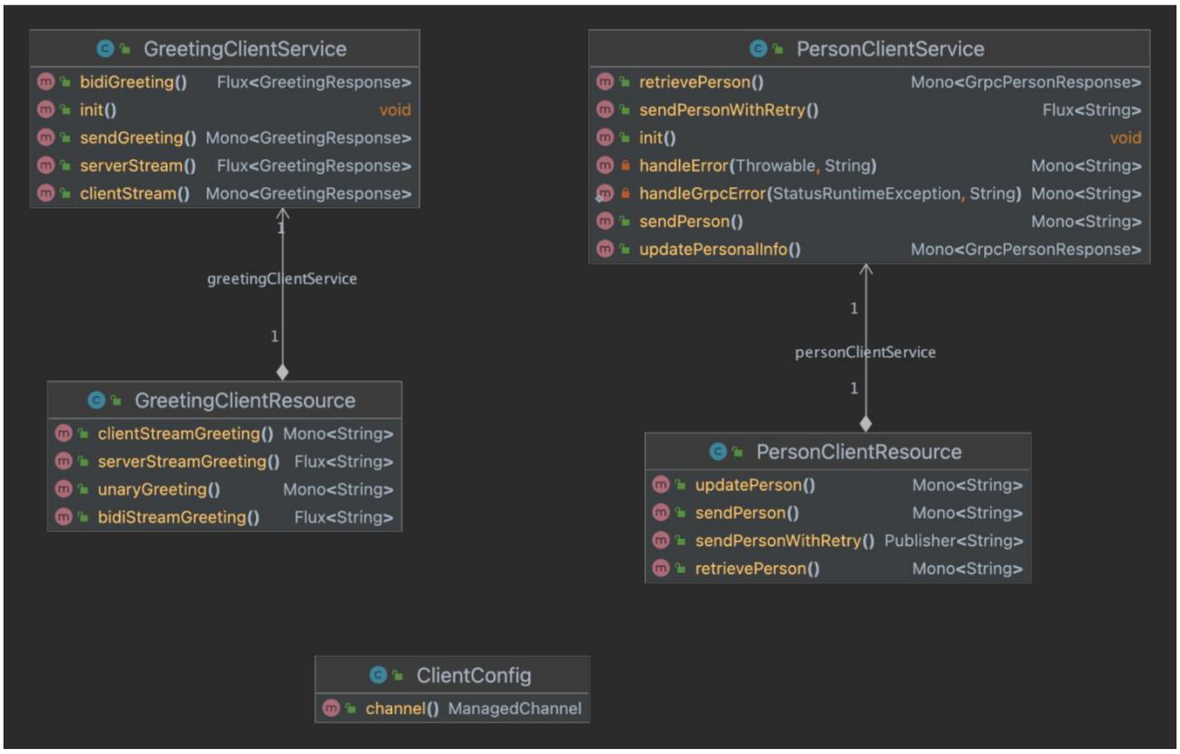

GreetingClientService
bidiGreeting() Flux<GreetingResponse>
init() void
sendGreeting() Mono<GreetingResponse>
serverStream() Flux<GreetingResponse>
clientStream() Mono<GreetingResponse>
PersonClientService
retrievePerson() Mono<GrpcPersonResponse>
sendPersonWithRetry() Flux<String>
init() void
handleError(Throwable, String) Mono<String>
handleGrpcError(StatusRuntimeException, String) Mono<String>
sendPerson() Mono<String>
updatePersonalInfo() Mono<GrpcPersonResponse>
greetingClientService
1
1
personClientService
1
1
GreetingClientResource
clientStreamGreeting() Mono<String>
serverStreamGreeting() Flux<String>
unaryGreeting() Mono<String>
bidiStreamGreeting() Flux<String>
PersonClientResource
updatePerson() Mono<String>
sendPerson() Mono<String>
sendPersonWithRetry() Publisher<String>
retrievePerson() Mono<String>
ClientConfig
channel() ManagedChannel

Figure 2 Client module class diagram

On the opposite end, the server module comprises four distinct services, including PersonService, which houses the business logic of person functionalities, **PersonGrpcService**, functioning as the gRPC implementation for the person service, **GreetingGrpcService**, also serving as a gRPC implementation for the greeting service, and **PersonRestService**, designed for REST functionalities of the person service. It is essential to note that the latter is intended solely for benchmarking purposes. The server module facilitates communication with the database in two ways. One of the approaches involves connecting to a MySQL database, while the other utilizes a static implementation of a repository that functions as a dummy repository, facilitating benchmarking without the inclusion of database communication. Additional information on this can be found in Figure 3.

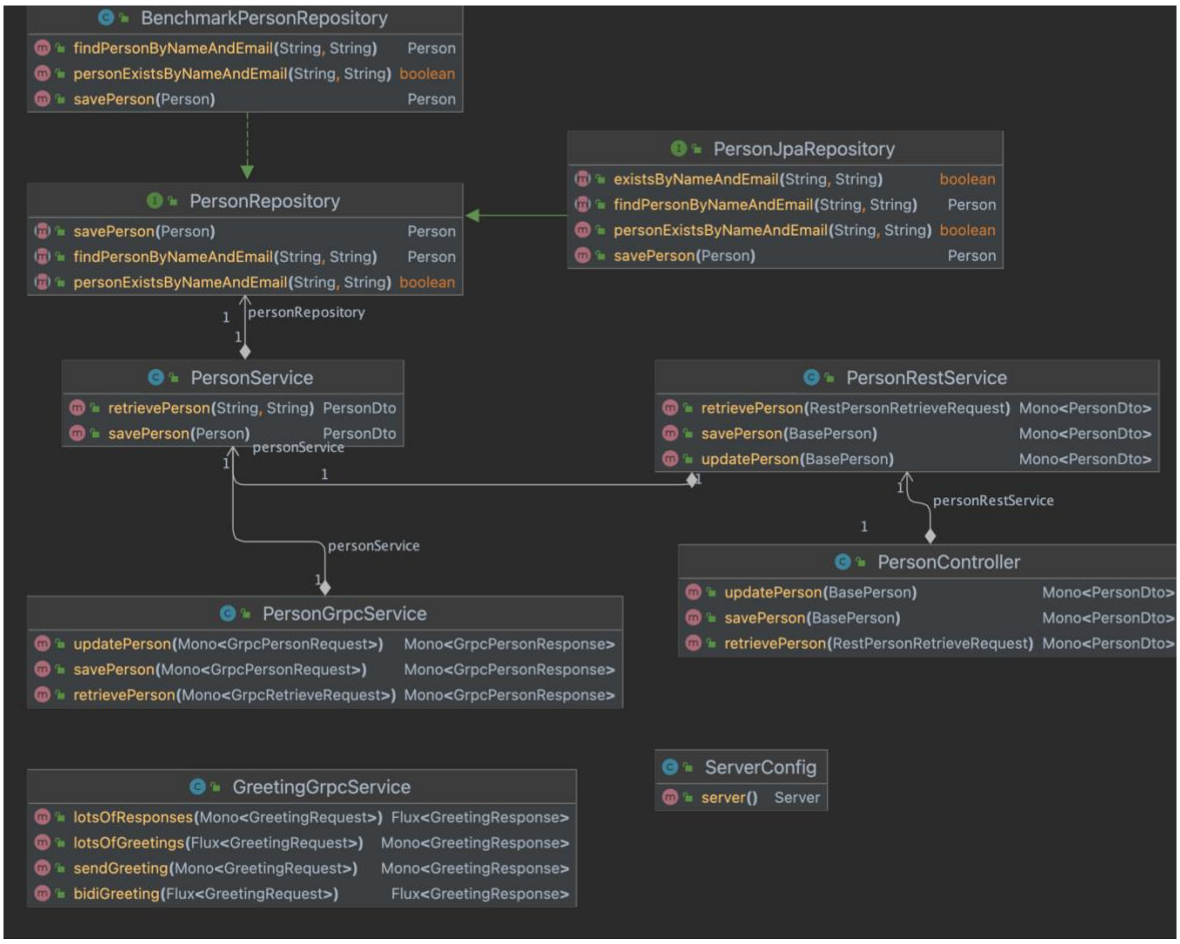

BenchmarkPersonRepository
findPersonByNameAndEmail(String, String) Person
personExistsByNameAndEmail(String, String) boolean
savePerson(Person) Person
PersonJpaRepository
existsByNameAndEmail(String, String) boolean
findPersonByNameAndEmail(String, String) Person
personExistsByNameAndEmail(String, String) boolean
savePerson(Person) Person
PersonRepository
savePerson(Person) Person
findPersonByNameAndEmail(String, String) Person
personExistsByNameAndEmail(String, String) boolean
1
personRepository
1
PersonService
retrievePerson(String, String) PersonDto
savePerson(Person) PersonDto
personService
1
1
PersonRestService
retrievePerson(RestPersonRetrieveRequest) Mono<PersonDto>
savePerson(BasePerson) Mono<PersonDto>
updatePerson(BasePerson) Mono<PersonDto>
1
1
personRestService
1
personService
1
PersonController
updatePerson(BasePerson) Mono<PersonDto>
savePerson(BasePerson) Mono<PersonDto>
retrievePerson(RestPersonRetrieveRequest) Mono<PersonDto>
PersonGrpcService
updatePerson(Mono<GrpcPersonRequest>) Mono<GrpcPersonResponse>
savePerson(Mono<GrpcPersonRequest>) Mono<GrpcPersonResponse>
retrievePerson(Mono<GrpcRetrieveRequest>) Mono<GrpcPersonResponse>
ServerConfig
server() Server
GreetingGrpcService
lotsOfResponses(Mono<GreetingRequest>) Flux<GreetingResponse>
lotsOfGreetings(Flux<GreetingRequest>) Mono<GreetingResponse>
sendGreeting(Mono<GreetingRequest>) Mono<GreetingResponse>
bidiGreeting(Flux<GreetingRequest>) Flux<GreetingResponse>

Figure 3 Server module class diagram

5.1 Gradle and plugin managements

It is noteworthy to mention that the described application is designed to operate on Java version 17, Spring Boot version 3.0.2, and Gradle version 8.0.1. The demo application utilizes Gradle as a plugin management system and build automation tool. The Gradle build tool is user-friendly and efficient in building projects, publishing, executing tasks, and managing complex application structures. The syntax used by Gradle build files is based on Groovy, which is unambiguous and easy to comprehend compared to other build automation tools.

Maintainability, usability, extendibility, performance, and flexibility are the cornerstones of Gradle, a popular build tool that is well-known for fixing problems that other build tools, such as Maven and ANT, have. Projects involving a wide range of technologies, including Java, Android, and Groovy, benefit greatly from its adaptability. Gradle is well-known for its lightning-quick performance, nearly twice as fast as Maven, and for its support of a wide range of IDEs, which makes for a more pleasant user experience. For those who are more comfortable in the terminal, Gradle also includes a command-line interface with helpful tools like Gradle tasks and Command line completion.

5.2 Libraries and plugins

The application under consideration depends on Lombok, an essential plugin for eliminating Java boilerplate code. In scenarios that require a Logger, several constructors must be created, such as a constructor with all arguments, a constructor with no arguments, a constructor with only final declared variables, and constructors that add behavior upon creation. Additionally, getter, setters, and toString methods need to be included.

Furthermore, it is crucial to mention the significance of Project Reactor in the application, as its reactive programming approach relies on this tool. The program flow utilizes non-blocking reactive stream specifications built upon the JVM. Project Reactor is primarily used to accomplish this and offers various benefits, such as providing non-blocking streams that can be altered or handled at any stage of their behavior.

6 GRPC API USAGE

With all its support for various programming languages and libraries, gRPC has huge advantage when it comes to its API usage, the best way to go through it is to start from its IDL (Protobuf), server-client implementation and how gRPC can handle in failure states with its error handling capability.

6.1 Protocol buffers

Protocol buffers are very useful for serialize and deserialize objects (messages), it uses Protobuf API for both operations, in order to achieve that we need to do following steps, which is done in Java programming language.

First step is to define a schema in a **.proto** file, the schema specifies the fields and types of the data that will be serialized and deserialized. The extension of **.proto** file is making **protoc** compiler to compile it and generate Java source code. Generated Java code from the **.proto** file is used by **protoc** compiler will have serialization and deserialization within the generated Java code.

Second step will be adding Protobuf to Spring application by importing *com.google.protobuf* plugin as well as configuring it, and then run `/gradlew generateProto ` or `gradle generateProto` and you have generated java source code for every correctly defined schema in **.proto** file.

build.gradle configuration would be as following

```
plugins {
    id "com.google.protobuf" version "0.9.2"
}

import org.apache.tools.ant.taskdefs.condition.Os

def archSuffix = Os.isFamily(Os.FAMILY_MAC) ? ':osx-x86_64' : ''
protobuf {
    protoc {
        artifact = "com.google.protobuf:protoc:3.20.1$archSuffix"
    }
    plugins {
        grpc {
            artifact = "io.grpc:protoc-gen-grpc-java:1.43.2"
        }

        reactor {
            artifact = "com.salesforce.servicelibs:reactor-grpc:1.2.3"
        }
    }
    generateProtoTasks {
        all().each { task ->
            task.builtins {
                java {
                    option "annotate_code"
                }
            }
            task.plugins {
                grpc {}
                reactor {}
            }
        }
    }
}
```

An example of this would be first to define a schema like following.

```
syntax = "proto3";

package iam.mfa.grpc.api.data;

option java_multiple_files = true;
option java_package = "iam.mfa.grpc.api.data";

message GrpcPersonResponse{
  string id = 1;
  string name = 2;
  int32 age = 3;
  string  email = 4;
  string lifeIntro = 5;
}
message GrpcPersonRequest{
  string name = 1;
  int32 age = 2;
  string  email = 3;
  string lifeIntro = 4;
}
message GrpcRetrieveRequest{
  string name = 1;
  string email = 2;
}



service PersonService{
  rpc savePerson(GrpcPersonRequest) returns (GrpcPersonResponse);
  rpc updatePerson(GrpcPersonRequest)  returns (GrpcPersonResponse);
  rpc retrievePerson(GrpcRetrieveRequest) returns (GrpcPersonResponse);
}
```

Run following command.

`gradle generateProto`

By default, generated java sources codes will be under **/build/generated/source/proto/**

Create Person object through builder that is also generated by Protoc compiler.

```
final var personRequest = GrpcPersonRequest.newBuilder()
        .setName("hamma")
        .setAge(23)
        .setEmail("m_fataka@utb.hz")
        .build();
```

6.2 Server-client implementation

Main concept of gRPC is server-client communication thus most of its setting, configuration is done upon configuring server-client configurations. By default, when generating gRPC java services they have no implementation, service classes must be extended by server service classes in order to have implementations for its service methods. On the other side of clients there is no default client listeners for gRPC services, clients' services must listen through stubs of gRPC and react to the event. An example of this would be a client makes a request then it is packed to stub through to be sent to RPC runtime, and from this point clients RPC runtime will wait until it receives response or terminate when reaches timeout.

Server will receive call of request through server's RPC runtime and it is sent to stub, then stub unpacks request for server to execute call/process request, after that server generates response to be return to stub packed, and stub sends back to server's RPC runtime thus client at this point can receive response through its RPC runtime and passed to its stub, finally client's stub will unpack response and the process will reach its final step. Detailed information is shown in Figure 4.

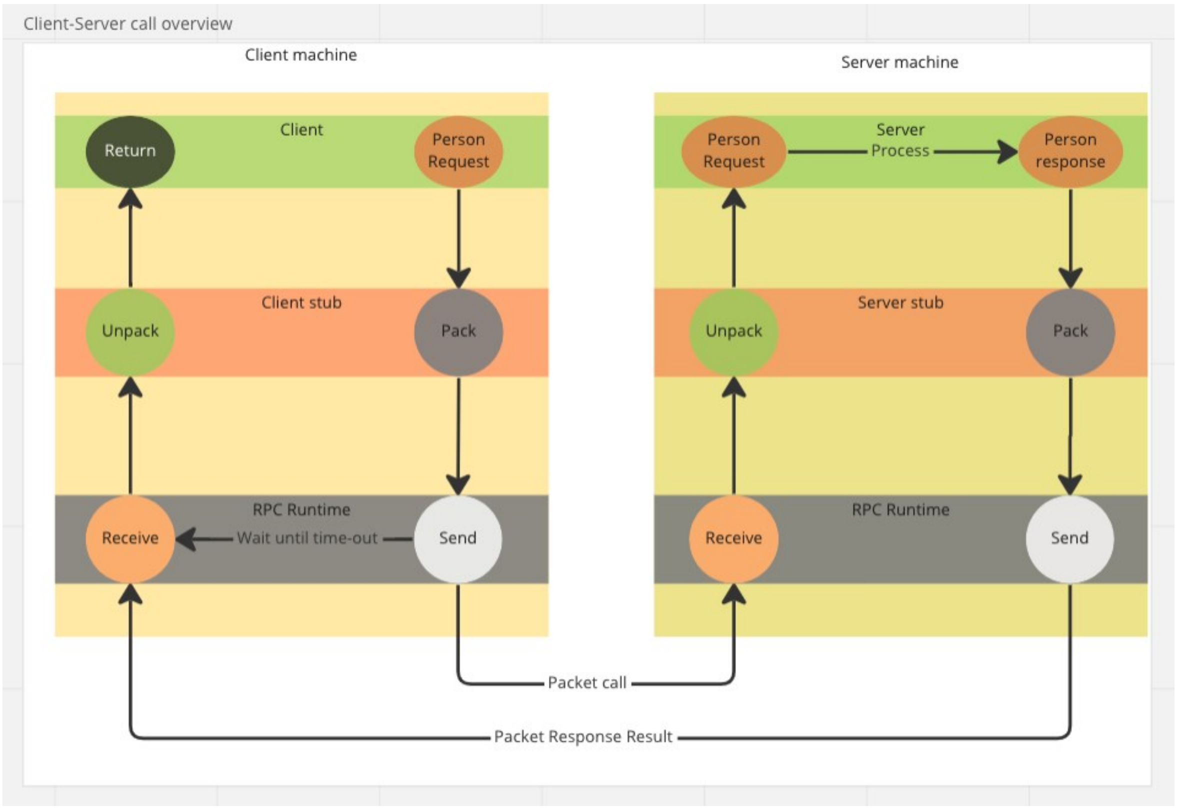


Figure 4 Client-server call overview

From a programming perspective, the initial step involves configuring the server configuration. To illustrate, a simple Server configuration is defined as a bean to facilitate its automatic injection when a service is added to the server as shown below.

Server configuration:

```
@RequiredArgsConstructor(onConstructor_ = @Autowired)
public class ServerConfig {

    @Bean
    public Server server() {
        return ServerBuilder.forPort(6969)
                .build();
    }
}

```

The subsequent step in creating simple server-side communication entails annotating service classes with **@GRpcService** to enable their injection, similar to a Spring bean. Additionally, it is necessary to extend **ReactorPersonSenderGrpc.PersonSenderImplBase**, which provides an abstract implementation for Person server implementations generated by Protobuf.

Once the class skeleton is created, the **sendPerson()** method must be overridden within the **ReactorPersonServiceGrpc.PersonServiceImplBase**, which has the same method definition as defined in **Person.proto**. The method's implementation can then be completed or simple which can be added to **personService** for business logic as shown below.

```
@Slf4j
@GRpcService
@RequiredArgsConstructor(onConstructor_ = @Autowired)
public                class                PersonGrpcService                extends
ReactorPersonServiceGrpc.PersonServiceImplBase {
    private final PersonService personService;

    @Override
    public Mono<GrpcPersonResponse> savePerson(Mono<GrpcPersonRequest> request) {
       return request.doOnNext(personRequest -> log.trace("Received person request
[{}]", personRequest))
                .map(personRequest -> {
                    final var person = Person.fromGrpc(personRequest);
                    return personService.savePerson(person).toGrpc();
                })
                  .doOnSuccess(personResponse  ->  log.trace("responded  with  [{}]",
personResponse));
    }

}
```

Similarly, the client-side implementation requires configuration. The first step is to configure the client's channel as a Spring Boot bean, enabling it to connect to the stub when creating client implementation and subsequently call and connect to the server through RPC.

The channel configuration is straightforward, comprising only the host and port of the server on which it will run. For this purpose, gRPC provides **ManagedChannels**, which handle all necessary client channel configurations. For example, connecting to localhost and port 6969 as defined below for the server-side.

```java
@Configuration
public class ClientConfig {

    @Bean
    public ManagedChannel channel() {
        return ManagedChannelBuilder.forAddress("localhost", 6969)
                .usePlaintext()
                .build();
    }

}
```

The client implementation must contain an injected **ManagedChannel** that can be passed to the gRPC stub to send client calls to the server.

It is recommended to initialize the stub only after the application is fully ready, meaning that Spring Boot beans are ready to be used. This can be achieved by adding an event listener to the class that executes the specified method upon event time.

The final step in the client implementation is to call the pre-defined method inside the stub, which is generated by protobufs, as well as adding events for successful responses before sending the request call, after the request is ready to be sent, and for the timeout of waiting for the response to arrive as shown below.

```java
@Slf4j
@Service
@RequiredArgsConstructor(onConstructor_ = @Autowired)
public class PersonClientService {
    private final ManagedChannel channel;
    private ReactorPersonSenderGrpc.ReactorPersonSenderStub stub;

    @EventListener(ApplicationReadyEvent.class)
    public void init() {
        stub = ReactorPersonSenderGrpc.newReactorStub(channel);
    }

    public Mono<String> sendPerson(String id) {
        return stub.savePerson(GrpcPersonRequest.newBuilder()
                        .setName("hamma")
                        .setAge(23)
                        .setEmail("m_fataka@utb.cz")
                        .setLifeIntro("my name is hamma, living in zlin, studying software engineering!")
                        .build()
                )
                  .doOnNext(personResponse -> log.trace("received person response [{}]", personResponse))
                .timeout(Duration.ofSeconds(10));
    }
}
```

6.3 gRPC error handling

it's significantly important to handle errors appropriately and correctly to ensure that application behaves predictably and gracefully thus gRPC has provided `StatusRuntimeException` which hold metadata for any behavior that can be predicted and handled.

Status codes and status messages are used by gRPC to handle errors. Status messages offer a human-readable explanation of the mistake, whereas status codes are numerical numbers that represent the results of a gRPC call.

Understanding the various status codes that might be returned by a gRPC request can help you handle issues with gRPC:

- OK: The call completed successfully.
- CANCELLED: The call was cancelled by the client.
- UNKNOWN: An unknown error occurred.
- INVALID_ARGUMENT: The client provided an invalid argument.
- DEADLINE_EXCEEDED: The call timed out before completing.
- NOT_FOUND: The requested resource was not found.
- ALREADY_EXISTS: The requested resource already exists.
- PERMISSION_DENIED: The client does not have permission to perform the requested action.
- UNAUTHENTICATED: The client is not authenticated.
- RESOURCE_EXHAUSTED: The resource being accessed has been exhausted.
- FAILED_PRECONDITION: The client did not meet a precondition for the call.
- ABORTED: The call was aborted.
- OUT_OF_RANGE: The client provided an argument that is out of range.
- UNIMPLEMENTED: The requested operation is not implemented.
- INTERNAL: An internal server error occurred.
- UNAVAILABLE: The server is currently unavailable.
- DATA_LOSS: Data was lost or corrupted during the transmission.

This will make both client and server be safe and process gracefully in case of an error produce when a problem arises making server ready to handle errors that are raised by application code from client, and client ready to handle errors that are raised by the server.

gRPC error handling provides generic error handle alongside of specific codes, in any mentioned cases it's very useful to provide detailed message in error occurrence. Having detailed messages can help developers diagnose and fix the problem. It should include information about the error code, the location of the error, and any relevant context or information that can help developers understand the cause of the error.

Using metadata is another useful way to share information about error event occur so server or client have detailed resources to address error.

gRPC has ability also to intercept and recover or handle error at various point of process if the issue is recoverable according to code implementation. Such cases can happen for missing not having enough resources or field validations etc., in that case simple handling can be saving record to database and marking as failed item or simply logging the error to address the issue on server, for example client sends person object but email does not exists and server has pattern combination of email which can be id + name, thus sever can handle this issue by assigning email to person, save as failed to database or simply logging it.

Sometimes errors can be transient, and can be resolved by retrying the request after a short delay. gRPC provides retrying and timeout mechanisms to handle these types of errors automatically.

Java example for sending person from client would look like below for **PersonClientService** class, with time-out and error handler.

```java
    public Mono<String> sendPerson() {
        return stub.savePerson(request)
                .doOnNext(personResponse -> log.trace("received person response [{}]", personResponse))
                .map(GrpcPersonResponse::toString)
                .onErrorResume(error -> handleError(error, request.getName()))
                .timeout(Duration.ofSeconds(10));
    }
    private Mono<String> handleError(final Throwable error, final String name) {
        if (error instanceof final StatusRuntimeException gRPCError) {
            return handleGrpcError(gRPCError, name);
        }
        final var rootCause = ExceptionUtils.getRootCause(error);
        if (rootCause instanceof final StatusRuntimeException gRPCError) {
            return handleGrpcError(gRPCError, name);
        }
        log.error("non-gRPC StatusRuntimeException error occurred, message [{}]", error.getMessage(), error);
        return Mono.error(error);
    }

    private static Mono<String> handleGrpcError(StatusRuntimeException gRPCError, String name) {
        final var errorStatus = gRPCError.getStatus();
        final var errorCode = errorStatus.getCode();
        var message = "status " + errorCode + " handling not implemented yet, message: " + errorStatus.getDescription();

        return switch (errorCode) {
            case FAILED_PRECONDITION -> {
                message = "server has failed preconditions for person name, message: " + errorStatus.getDescription();
                yield Mono.just(message);
            }
            case ALREADY_EXISTS -> {
                message = "person with name" + name + "already exists";
                yield Mono.just(message);
            }
            case CANCELLED -> {
                message = "call was not sent to server, it was canceled on client call, message: " + errorStatus.getDescription();
                yield Mono.just(message);
            }
            case DEADLINE_EXCEEDED -> {
                message = "call exceeded expected time call, message: " + errorStatus.getDescription();
                yield Mono.just(message);
            }
            case PERMISSION_DENIED -> {
                message = "one of fields of person with name " + name + " is not valid, message: " + errorStatus.getDescription();
                yield Mono.just(message);
            }
            default -> {
                log.trace("error from server received, {}", message);
                log.error("unrecoverable error received, messgae [{}]", gRPCError.getMessage(), gRPCError);
                yield Mono.error(gRPCError);
            }
        };
    }
```

It is possible to incorporate retry functionality in a gRPC service by utilizing Flux from the Project Reactor library. The reason for this approach is due to the inability to directly subscribe to a gRPC service multiple times concurrently with Mono. To implement the retry functionality, the **Flux.defer()** method can be employed, which allows the reactive stream to wait and depend concurrently on all sub-streams inside the Flux object before completing the call. The corresponding Java code is presented below.

```java
public Flux<String> sendPersonWithRetry() {
    final var personRequest = GrpcPersonRequest.newBuilder()
            .setName("hamma")
            .setAge(23)
            .setEmail("m_fataka@utb.hz")
            .build();

    return Flux.defer(() -> stub.savePerson(personRequest)
                    .doOnNext(personResponse -> log.trace("received person response [{}]", personResponse))
                    .map(GrpcPersonResponse::toString)
                    .timeout(Duration.ofSeconds(10)))
            .retryWhen(Retry.fixedDelay(3, Duration.ofSeconds(5)))
            .onErrorResume(error -> handleError(error, personRequest.getName()));
}
```

In the above example, the stub sends the request once. In the event of an error occurring, the Flux reactive chain will retrigger the call up to three times at an interval of five seconds. After retrying three times, the call will finish with either a success or a failure, which can be handled if the error is predictable and recoverable. This mechanism enables gRPC to gracefully handle transient errors, and it can prove useful in numerous situations.

On the server side, we can add conditions and filters to requests that have been made, such as field validation at business logic, deadline exceeding, checking for the existence of an item, or permission for particular operations. In such cases, the server should be fully equipped to use gRPC **StatusRuntimeException** with a predefined status and a description of the error event that can address its cause.

For instance, we can implement a simple approach to checking the registration deadline, validating email suffixes, blocking certain suffixes, and checking for existing records in the PersonService class. The Java source code for this implementation would resemble the following:

```
    private final PersonRepository personRepository;
    private static final String BLOCKED_ID_SUFFIX = "BL-000";
    private final LocalDateTime registrationDate = LocalDateTime.of(2023, 3, 1, 0, 0);

    public PersonDto savePerson(final Person person) {
        Preconditions.checkNotNull(person);

        final var name = person.getName();
        final var email = person.getEmail();
        final var exists = personRepository.personExistsByNameAndEmail(name, email);
        if (name.contains(BLOCKED_ID_SUFFIX)) {
            throw new StatusRuntimeException(Status.PERMISSION_DENIED.withDescription("Person is in blocked list"));
        }
        if (LocalDateTime.now().isAfter(registrationDate)) {
            throw new StatusRuntimeException(Status.DEADLINE_EXCEEDED.withDescription("registration date is already expired"));
        }
        if (exists) {
            throw new StatusRuntimeException(Status.ALREADY_EXISTS.withDescription("person is already registered with name:" + name));
        }
        if (!email.endsWith("@utb.cz")) {
            throw new StatusRuntimeException(Status.FAILED_PRECONDITION.withDescription("person is not from utb"));
        }
        return personRepository.savePerson(person).asDto();
    }

    public PersonDto retrievePerson(final String name, final String email) {
        Preconditions.checkNotNull(name);
        Preconditions.checkNotNull(email);
        return personRepository.findPersonByNameAndEmail(name, email).asDto();
    }
```

Above code is sample of error being passed from server to client with particular status and description.

6.4 gRPC streaming

There are three different forms of streaming RPCs in gRPC: bidirectional streaming, server streaming, and client streaming. The information obtained for this topic references core concepts, architecture and lifecycle [30].

Implementation of streams will be based on **Greeting.proto** as base model for implementing all types of streaming:

Greeting.proto:

```
syntax = "proto3";

package iam.mfa.grpc.api.data;

option java_multiple_files = true;
option java_package = "iam.mfa.grpc.api.data";


message GreetingRequest {
  string greeterName = 1;
  int32 age = 2;
}


message GreetingResponse{
  string result = 1;
  int32 age = 2;

}


service GreetingSender{
  rpc sendGreeting(GreetingRequest) returns (GreetingResponse);
  rpc lotsOfResponses(GreetingRequest) returns (stream GreetingResponse);
  rpc lotsOfGreetings(stream GreetingRequest) returns (GreetingResponse);
  rpc bidiGreeting(stream GreetingRequest) returns (stream
GreetingResponse);
}
```

Above model is simple unary RPC, that sends single request to server and receives single response back.

The server-streaming approach in gRPC involves the server returning a stream of messages to the client in response to a request. After sending all of its messages, the server includes any optional trailing metadata and status information in its response to the client. The server's processing is complete once the client has received all of the messages.

Following is implementation of server-streaming in a gRPC client service.

```
public Flux<GreetingResponse> serverStream() {
    return stub.lotsOfResponses(GreetingRequest.newBuilder()
                    .setGreeterName("hamma")
                    .build()
            ).doOnNext(personResponse -> log.trace("received greeting response [{}]", personResponse))
            .timeout(Duration.ofSeconds(10));
}
```

Server-side gRPC service:

```
@Override
public Flux<GreetingResponse> lotsOfResponses(Mono<GreetingRequest> request) {
    return request.doOnNext(greetingRequest -> log.trace("greeting request received [{}]", greetingRequest))
            .flatMapIterable(greetingRequest -> {
                        final var greetingResponse = GreetingResponse.newBuilder()
                                .setResult("hello " + greetingRequest.getGreeterName())
                                .build();
                        return List.of(greetingResponse, greetingResponse, greetingResponse);
                    }
            )
            .doOnNext(greetingResponse -> log.trace("greeting responded with [{}]", greetingResponse));
}
```

Client-streaming: Instead of sending a single message to the server, the client sends a stream of messages using client-streaming RPCs. The server replies with one message, status information, and optional trailing metadata. Typically, this occurs following the server's receipt of all client messages. For such streaming **Greeting.proto** would change as below:

Client-side service would change to:

```
public Mono<GreetingResponse> clientStream() {
    final var firstPerson = GreetingRequest.newBuilder()
            .setGreeterName("hamma")
            .build();

    final var secondPerson = GreetingRequest.newBuilder()
            .setGreeterName("eliza")
            .build();

    final var publisher = Flux.just(firstPerson, secondPerson);
    return stub.lotsOfGreetings(publisher)
            .doOnNext(personResponse -> log.trace("received greeting response [{}]", personResponse))
```

```
                .timeout(Duration.ofSeconds(10));
    }
```

Server-side service would change to:

```
    @Override
    public Mono<GreetingResponse> lotsOfGreetings(Flux<GreetingRequest> request) {
        return request.doOnNext(greetingRequest -> log.trace("greeting request received [{}]", greetingRequest))
                .collectList()
                .map(list -> {
                    final var responseStream = list.stream()
                            .map(GreetingRequest::getGreeterName)
                            .collect(Collectors.joining(",", "hello ", ""));
                    return GreetingResponse.newBuilder()
                            .setResult(responseStream)
                            .build();
                });
    }
```

Bidirectional streaming: In bidirectional streaming RPCs, the call is started by the client calling the method and providing the server with the method's information, name, and deadline. The server can decide whether to return its initial metadata or to hold off until the client begins streaming messages. Application-specific stream processing occurs on both the client and server sides. The two streams are separate, so the client and server can read and write messages in any sequence. For such streaming **Greeting.proto** would change as below:

Client-side service would change to:

```
    public Flux<GreetingResponse> bidiGreeting() {
        final var firstPerson = GreetingRequest.newBuilder()
                .setGreeterName("hamma")
                .build();
        final var secondPerson = GreetingRequest.newBuilder()
                .setGreeterName("eliza")
                .build();
        final var publisher = Flux.just(firstPerson, secondPerson);
        return stub.bidiGreeting(publisher)
                .doOnNext(personResponse -> log.trace("received greeting response [{}]", personResponse))
                .timeout(Duration.ofSeconds(10));
    }
```

Server-side service would change to:

```
    @Override
    public Flux<GreetingResponse> bidiGreeting(Flux<GreetingRequest> request) {
        return request.doOnNext(greetingRequest -> log.trace("greeting request received [{}]", greetingRequest))
                .map(greetingRequest -> GreetingResponse.newBuilder()
                        .setResult("hello " + greetingRequest.getGreeterName())
                        .build()
                )
                .doOnNext(greetingResponse -> log.trace("greeting responded with [{}]", greetingResponse));
    }
```

7 GRPC SECURITY AND AUTHENTICATION

After knowing gRPC API usage on protocol buffers, implementation of server-client on a fundamental level, and handling predictable issues and errors, it is crucial to make gRPC securely transport data between the server(s) and client(s) and not worry about security breaks.

As with all other data transport protocols, gRPC permits the use of a range of security protocols, from the most fundamental to the most advanced. gRPC allows authentications in multiple ways, including token-based, OAuth, and Authorization based authentication, which makes gRPC provide robust security and authentication mechanism to avoid any worrying about security breaks. Moreover, it is possible to use multiple authentications and security mechanisms [32].

7.1 Transport layer security (TLS)

Multiple approaches exist for configuring TLS/SSL for gRPC in Spring Boot. These include defining the certificate and key for gRPC within the application's properties, as well as utilizing **GrpcAutoConfigurer.java** possesses the capability to automatically configure TLS/SSL for either the server or client. This can be achieved through the utilization of code similar to the example provided application.yaml

```yaml
grpc:
  port: 6969
  security:
    cert-chain: server.pem
    private-key: private.key
```

The aforementioned configuration can be applied to both client and server. Alternatively, TLS/SSL may be implemented programmatically when configuring the gRPC server for the server-side or the gRPC channel for the client-side.
For the programmatic approach, a JKS file can be used as the keystore for the server, and gRPC will automatically load all certificates within it. Custom properties for the keystore file can be added, as demonstrated below:

```
1.  grpc:
2.    port: 6969
3.    security:
4.      key-store-path: path/to/jks
5.      key-store-password: password
```

Server configuration:

```
1.  @Configuration
2.  public class ServerConfig {
3.
4.      @Value("${grpc.security.key-store-path}")
5.      private String trustStorePath;
6.      @Value("${grpc.security.key-store-password}")
7.      private String trustStorePassword;
8.
9.      @Bean
10.     public Server server() {
11.         final var serverCredentials = TlsServerCredentials.newBuilder()
12.                 .keyManager(SslUtils.buildKeyManagerFactory(trustStorePath,
    trustStorePassword).getKeyManagers())
13.                 .build();
14.         return NettyServerBuilder.forPort(6969, serverCredentials)
15.                 .build();
16.     }
17. }
```

The code demonstrated above utilizes **TlsServerCredentials.java**, which is a provided server credential by the gRPC library. It is important to note that once server credentials are configured, they cannot be modified on the builder level. For instance, any additional security configurations cannot be added after line 14 in the code.
An example for that would be:

```
18.         return NettyServerBuilder.forPort(6969, serverCredentials)
19.                 .useTransportSecuirty()
20.                 .build();
```

On client side programmatically configuring TLS/SSL is slightly different from server-side configurations since we need to tell gRPC to not use HTTP2. As mentioned before we can configure it through properties within *application.yaml*, or configuring it programmatically like following **application.yaml:**

```
1.  grpc:
2.    security:
3.      trust-store-path: path/to/jks
4.      trust-store-password: password
```

Client channel configuration:

```
@Configuration
public class ClientConfig {
    @Value("${grpc.security.trust-store-path}")
    private String trustStorePath;
    @Value("${grpc.security.trust-store-password}")
    private String trustStorePassword;

    @Bean
    @SneakyThrows
    public ManagedChannel channel() {
        final var trustManagerFactory =
SslUtils.buildTrustManagerFactory(trustStorePath, trustStorePassword);
        return NettyChannelBuilder.forTarget("127.0.0.1:6969")
                .sslContext(GrpcSslContexts
                        .configure(SslContextBuilder.forClient().trustManager(trus
tManagerFactory), SslProvider.JDK)
                        .build()
                )
                .usePlaintext()
                .build();
    }
}
```

The above code snippet demonstrates the usage of the **.sslContext()** method to configure SSL context. **GrpcSslContext.java**is provided by gRPC to configure SSL contexts using **SslContextBuilder.java** by passing the trust store file and password.

7.2 OAuth and JWT

Besides TLS/SSL mechanisms, gRPC provides OAuth and JWT authentication. JWT-based authentication is based on a token that holds security information such as role, expiration date, issued date, issuer, and username. It is also possible to add custom information that can also be added to the JWT token. JWT tokens are considered one of the modern applications' most secure and commonly used security mechanisms.

First step to starting implementation of JWT mechanism by importing JWT library by *io.jsonwebtoken* to gradle.build we need two dependencies as below:

```
api 'io.jsonwebtoken:jjwt-api:0.11.5'
runtimeOnly 'io.jsonwebtoken:jjwt-impl:0.11.5'
runtimeOnly 'io.jsonwebtoken:jjwt-jackson:0.11.5'
```

And define a common constant to be imported for both client and server side.

```
@UtilityClass
public class SecurityConstants {
    public static final String JWT_SIGNING_KEY = "eyJhbGciOiJIUzI1NiJ9.eyJSb2xlIjoiY2xpZW50IiwiSXNzdWVyIjoiaGFtbWEiLCJVc2VybmFtZSI6ImNsaWVudF8xIiwiZXhwIjoxNjgwMjEwNzMwLCJpYXQiOjE2ODAyMTA3MzB9.jHChEFhC90rhf2HYpuk_KqcdH5wIwNptBn7mBON_ZKg";
    public static final String BEARER_TYPE = "Bearer";

    public static final Metadata.Key<String> AUTHORIZATION_METADATA_KEY = Metadata.Key.of("Authorization", ASCII_STRING_MARSHALLER);
    public static final Context.Key<String> CLIENT_ID_CONTEXT_KEY = Context.keyWithDefault("clientId", ”Not found”);

}
```

The undermentioned class will serve as a builder for creating the interceptor, and **CallCredentials.java** will be utilized for both the client and the server. The next step involves implementing CallCredentials for the client, which can be passed to gRPC stubs. These credentials carry the required data that will be propagated to the server via request metadata for each RPC.

```
@NoArgsConstructor(staticName = "of")
public class ClientBearerToken extends CallCredentials {

    @Override
    public void applyRequestMetadata(final RequestInfo requestInfo, final Executor appExecutor, final MetadataApplier applier) {
        appExecutor.execute(() -> {
            try {
                final var headers = new Metadata();
                headers.put(SecurityConstants.AUTHORIZATION_METADATA_KEY, String.format("%s %s", SecurityConstants.BEARER_TYPE, buildJwtToken()));
                applier.apply(headers);
            } catch (Throwable e) {
                applier.fail(Status.UNAUTHENTICATED.withDescription("failed to apply request metadata").withCause(e));
            }
        });
    }

    @Override
    public void thisUsesUnstableApi() {

    }

    private String buildJwtToken() {
        final var secretKey = Keys.hmacShaKeyFor(SecurityConstants.JWT_SIGNING_KEY.getBytes(StandardCharsets.UTF_8));
        return Jwts.builder()
                .setId("client_1")
                .signWith(secretKey)
                .compact();
    }
}
```

In the aforementioned implementation, lines 26 to 28 are dedicated to creating headers and setting the header with a designated key. Meanwhile, the value of the header is constructed between lines 40 and 47 using the JWT library, which takes a secret token from constants that have been previously defined. Following this, the ClientBearerToken can be passed as CallCredentials to the client stub within the client service, as exemplified below:

```java
@Slf4j
@Service
@RequiredArgsConstructor(onConstructor_ = @Autowired)
public class PersonClientService {
    private final ManagedChannel channel;
    private ReactorPersonServiceGrpc.ReactorPersonServiceStub stub;

    @EventListener(ApplicationReadyEvent.class)
    public void init() {
        stub = ReactorPersonServiceGrpc.newReactorStub(channel)
                .withCallCredentials(ClientBearerToken.of());
    }
```

The client is now equipped with JWT authentication. It sends a JWT token for each call to PersonStub. Next, we will implement a ServerInterceptor on the server side to enable the server to read the headers received from the calls.

```java
public class ServerAuthenticationInterceptor implements ServerInterceptor {
    private final JwtParser jwtParser = Jwts.parserBuilder()
            .setSigningKey(SecurityConstants.JWT_SIGNING_KEY.getBytes(StandardCharsets.UTF_8))
            .build();

    @Override
    public <ReqT, RespT> ServerCall.Listener<ReqT> interceptCall(final ServerCall<ReqT, RespT> call, Metadata headers, final ServerCallHandler<ReqT, RespT> next) {
        final var key = headers.get(SecurityConstants.AUTHORIZATION_METADATA_KEY);
        final var nonValidKey = nonValidKey(key);

        if (Objects.nonNull(nonValidKey)) {
            call.close(nonValidKey, headers);
            return new NoopServerCall.NoopServerCallListener<>();
        }
        try {
            final var token = key.substring(SecurityConstants.BEARER_TYPE.length()).trim();
            final var claims = jwtParser.parseClaimsJws(token);
            final var context = Context.current().withValue(SecurityConstants.CLIENT_ID_CONTEXT_KEY, claims.getBody().getId());
            return Contexts.interceptCall(context, call, headers, next);
        } catch (Exception e) {
            call.close(Status.UNAUTHENTICATED.withDescription(e.getMessage()).withCause(e), headers);
            return new NoopServerCall.NoopServerCallListener<>();
        }
    }
    private Status nonValidKey(final String key) {
        if (key == null) {
```

```
27.             return Status.UNAUTHENTICATED.withDescription("Missing authorization
    token");
28.         }
29.         if (!key.startsWith(SecurityConstants.BEARER_TYPE)) {
30.             return Status.UNAUTHENTICATED.withDescription("Authorization type is
    unknown");
31.         }
32.         return null;
33.     }
34. }
```

The server interceptor is equipped with a JWT parser that can verify the validity of the JWT token received from the client and inspect its claims. If the client is authorized and authenticated, the regular call will proceed as expected. However, if the client is not authorized or authenticated, the current call will be terminated with a moral status, and NoopServerCall will be invoked to perform no action on the server side. Apart from the authentication process on line 38, client details are extracted and added to the current call context, which can be utilized in future implementations, such as client id.

```
1.  package iam.mfa.grpc.server.sevice.grpc;
2.
3.  imports ...
4.
5.  @Slf4j
6.  @RequiredArgsConstructor(onConstructor_ = @Autowired)
7.  @GRpcService(interceptors = ServerAuthenticationInterceptor.class)
8.  public class PersonGrpcService extends
    ReactorPersonServiceGrpc.PersonServiceImplBase {
9.      private final PersonService personService;
10.
11.     @Override
12.     public Mono<GrpcPersonResponse> savePerson(Mono<GrpcPersonRequest> request) {
13.         final var clientId =
    SecurityConstants.CLIENT_ID_CONTEXT_KEY.get(Context.current());
14.         return request.doOnNext(personRequest -> log.trace("Received person
    request [{}] from client [{}]", personRequest, clientId))
15.                 .map(personRequest -> {
16.                     final var person = Person.fromGrpc(personRequest);
17.                     return personService.savePerson(person).toGrpc();
18.                 })
19.                 .doOnSuccess(personResponse -> log.trace("responded with [{}]",
    personResponse));
20.     }
```

To add the server interceptor, we can pass it to the **@GrpcService()** annotation, which will automatically apply the interceptor to the **PersonGrpcService**. As mentioned earlier, by extracting the client id from the JWT claims and adding it to the current context in the interceptor, we can now retrieve the client id at the **PersonGrpcService**, as shown in lines 10 and 11.

8 GRPC PERFORMANCE AND SCALABILITY IN DISTRIBUTED SYSTEMS AND MICROSERVICES

The gRPC framework does more than just providing security and authentication to make its performance the best, gRPC framework has been specifically developed to cater to the requirements of high-performance services. The technology under consideration relies on the HTTP/2 protocol, which offers many performance advantages compared to the conventional HTTP/1.1. These benefits include a binary protocol, the ability to multiplex numerous requests on a single connection, and header compression. To optimize the performance of gRPC, adherence to several recommended practices is advised. It is recommended to reuse gRPC channels when initiating gRPC calls. The practice of channel reuse enables multiplexing calls via an extant HTTP/2 connection. Furthermore, streaming RPCs can manage a prolonged logical sequence of information. Streams can effectively circumvent the need for repeated RPC invocations. However, their implementation should be limited to cases where they confer significant advantages in terms of performance or operational ease to the underlying application logic [33].

8.1 gRPC load balancing

Load balancing refers to distributing network traffic across multiple servers intending to enhance the efficiency and dependability of services. The implementation of load balancing in gRPC can be achieved through either a proxy or on the client side. Proxy load balancing involves the client issuing RPCs to a Load Balancer proxy. The Load Balancer distributes the RPC to an available backend server that executes the necessary logic to serve the call. Client-side load balancing refers to a technique where the client possesses knowledge of multiple backend servers and subsequently selects one to use for each RPC. The end-user receives load reports from the servers located at the backend and subsequently applies load-balancing algorithms. The feature of client-side load balancing enables gRPC clients to distribute load among the currently available servers efficiently [34].

In this section, we will discuss the implementation of client-side load-balancing, which does not require a load-balancing proxy server. The initial step is to define a custom multiple address NameResolver.Factory by extending the NameResolverProvider class. The full class is presented below.

```
1.  @RequiredArgsConstructor(staticName = "of")
2.  public class MultipleAddressNameResolverProvider extends NameResolverProvider {
3.      private final List<EquivalentAddressGroup> addresses;
4.
5.      @Override
6.      public NameResolver newNameResolver(URI targetUri, NameResolver.Args args) {
7.          return new NameResolver() {
8.              @Override
9.              public String getServiceAuthority() {
10.                 return "noAuthority";
11.             }
12.             @Override
13.             public void start(Listener2 listener) {
14.                 listener.onResult(ResolutionResult.newBuilder().setAddresses(addre
    sses).setAttributes(Attributes.EMPTY).build());
15.             }
16.
17.             @Override
18.             public void shutdown() {}
19.         };
20.     }
21.
22.     @Override
23.     public String getDefaultScheme() {
24.         return "multiTarget";
25.     }
26.
27.     @Override
28.     protected boolean isAvailable() {
29.         return true;
30.     }
31.
32.     @Override
33.     protected int priority() {
34.         return 6;
35.     }
36. }
```

It is important to note that we set the priority to 6 at line 53 because the Netty library provides two other default name resolvers, with priorities of 3 and 5. By setting our custom name resolver to 6, the gRPC client channel will pick it as the resolver for the channel. Once we have our custom resolver, we can apply it to our client configuration as shown below:

```java
@Configuration
public class ClientConfig {

    @Value("#{'${grpc.client.targets}'.split(',')}")
    private List<String> targets;


    @PostConstruct
    public void initNameResolver() {
        final var addresses = targets.stream()
                .map(target -> {
                    final var splitTarget = target.split(":");
                    final var host = splitTarget[0];
                    final var port = splitTarget[1];
                     return new EquivalentAddressGroup(new InetSocketAddress(host, Integer.parseInt(port)));
                })
                .toList();
        final var multipleAddressNameResolverProvider = MultipleAddressNameResolverProvider.of(addresses);
        NameResolverRegistry.getDefaultRegistry().register(multipleAddressNameResolverProvider);
    }
    @Bean
    public ManagedChannel channel() {
        return NettyChannelBuilder.forTarget("localhost")
                .defaultLoadBalancingPolicy("round_robin")
                .useTransportSecurity()
                .usePlaintext()
                .build();
    }
}
```

On client application.yaml we put following as well so property can be picked up by spring.

```yaml
grpc:
  client:
    targets: localhost:6960,localhost:6961,localhost:6962
```

In the server-side implementation, three servers are created and run on different ports. The server's name, distinguished by its port, is added to the response to enable checking on the client which server responded to the call. The configuration is shown below.

```
@Slf4j
@Configuration
@RequiredArgsConstructor(onConstructor_ = @Autowired)
public class ServerConfig {
   private final PersonService personService;

    @PostConstruct
    public List<Server> servers() {
        final var servers = new ArrayList<Server>();
        final var numOfServers = 3;
        final var executorService = Executors.newFixedThreadPool(numOfServers);
        IntStream.range(0, 3)
                .forEach(index -> {
                    final var port = 6960 + index;
                    final var serverName = "Server_" + port;
                    executorService.submit(() -> {
                        final var server = ServerBuilder
                                .forPort(port)
                                       .addService(new  GreetingService(serverName, personService))
                                .build();
                        try {
                            server.start();
                             log.trace("{} server started, listening on port: {}", serverName, server.getPort());
                            server.awaitTermination();
                            servers.add(server);
                        } catch (IOException e) {
                            throw new RuntimeException(e);
                        } catch (InterruptedException e) {
                            Thread.currentThread().interrupt();
                            throw new RuntimeException(e);
                        }
                    });

                });
        return servers;
    }
}
```

And endpoint is created which will execute 3 calls at the same time, we can check on client side which server responded to each call, so let’s create simple endpoint like below:

```
    @GetMapping(path = "send/people")
    public Publisher<String> sendPeople() {
        return Flux.range(0, 3)
                .flatMap(index -> personClientService.sendPerson(String.valueOf(index)))
                .map(list -> String.join(", ", list));
    }
```

After triggering endpoint one or two times, we can check below.

```
2023-04-30T03:57:36.341+02:00      TRACE      20127      ---      [ault-executor-0] i.m.g.c.service.PersonClientService      : received person response [id: "af7d983b-0807-4a97-b4e9-41b7beb07823"
name: "hamma0"
age: 23
email: "mfataka@utb.cz"
lifeIntro: "my name is hamma, living in zlin, studying software engineering!"
serverName: "Server_6962"
]
2023-04-30T03:57:36.341+02:00      TRACE      20127      ---      [ault-executor-2] i.m.g.c.service.PersonClientService      : received person response [id: "10322bb2-d9e1-4e8d-ac6f-34969eb75904"
name: "hamma1"
age: 23
email: "mfataka@utb.cz"
lifeIntro: "my name is hamma, living in zlin, studying software engineering!"
serverName: "Server_6960"
]
2023-04-30T03:57:36.342+02:00      TRACE      20127      ---      [ault-executor-1] i.m.g.c.service.PersonClientService      : received person response [id: "af03b4c3-154b-4f6c-9f06-4ad87f1e7c8b"
name: "hamma2"
age: 23
email: "mfataka@utb.cz"
lifeIntro: "my name is hamma, living in zlin, studying software engineering!"
serverName: "Server_6961"
]
```

The above log messages reveal that three calls were executed, and each call was received by a different server, thanks to the implemented load balancer. Typically, during the initial call of the endpoint, calls are routed to one available server, and this is due to the load balancer initially directing calls to a single available server before the actual load balancing algorithm is applied.

8.2 gRPC service orchestration

Service orchestration refers to the process of organizing and administering various services in order to achieve a specific goal. Service orchestration involves coordinating interactions, dependencies, and relationships among services to optimize their collective performance and efficacy.

The main goal of gRPC service orchestration involves coordinating and managing multiple gRPC services to attain a particular business objective. This task can be accomplished through a range of techniques, such as service discovery, load balancing, and API gateway management. Service discovery is the process of automatically identifying available services within a network. The usage of gRPC enables service discovery via various tools, including but not limited to Kubernetes, Consul, and other tools. These tools facilitate the automatic registration and discovery of gRPC services.

Load balancing refers to evenly distributing incoming requests among multiple service instances to achieve optimal performance and high availability. gRPC offers inherent load balancing capabilities via various algorithms, including round-robin, most minor connections, and random [34].

An API gateway is managing access to multiple gRPC services through a singular entry point, is essential to API gateway management. The feature mentioned above enables customers to gain entry to numerous gRPC services via a consolidated interface while furnishing an additional stratum of security and governance.

To summarize, the process of gRPC service orchestration encompasses synchronizing and administering numerous gRPC services to attain a particular business objective through service detection, load distribution, and API gateway oversight [35].

9 GRPC VS. REST BENCHMARKING

Following an in-depth understanding of gRPC, it is only logical to establish its performance via benchmarking. This will be achieved through a comparison of gRPC against REST, using two operations: sending a person object to the server and retrieving a person object from the server.

To eliminate data access layers from the equation, the spring profile will be switched to "benchmark," utilizing static values for the repository and avoiding any database communication altogether.

A PersonController is implemented explicitly for this purpose, with REST calls received by the controller and gRPC calls by the PersonGrpcService.
Benchmarking will be conducted on both protocols, utilizing the same input and output configurations. Apache JMeter is the preferred tool for benchmarking as it can generate results, response time, throughput, and track all requests. Additionally, it has plugins that support benchmarking for gRPC services. For this particular benchmark, version 5.5 of Apache JMeter, the latest version available at present, will be utilized.

It is worth mentioning that Apache JMeter enables listeners to be added to the benchmarking process, providing full details about the benchmark process and results.

Hardware specifications of system used for benchmarking:

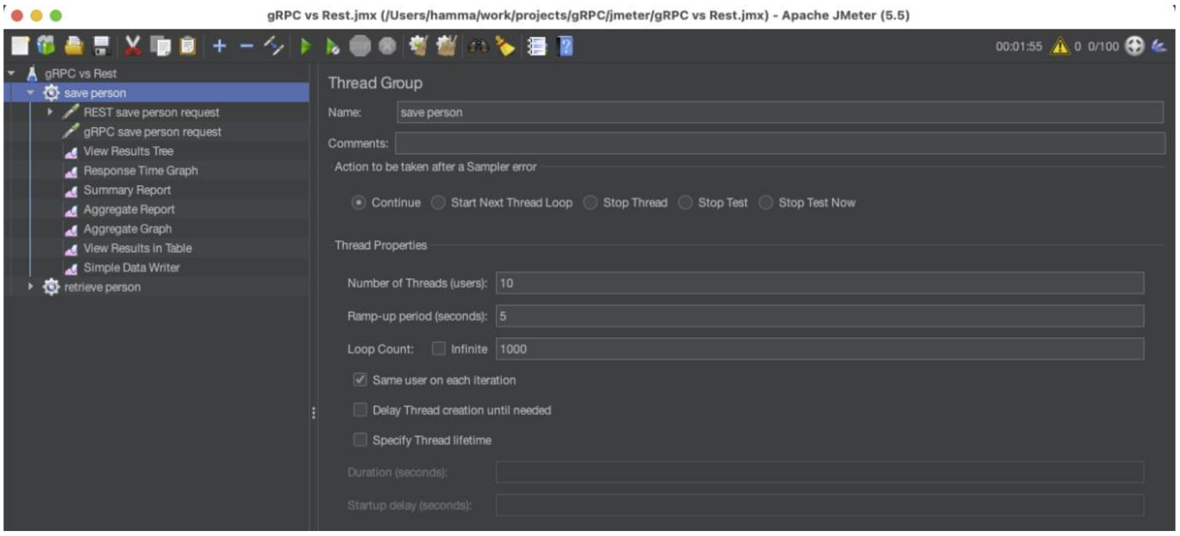
- macOS Ventura 13.3.1
- Apple M1 Pro chip
- 16 GB Memory

The benchmarking process is primarily focused on latency, and throughput is not included in the comparison.

9.1 Sending object benchmark

For the initial Person sending benchmark, the configurations involve using ten threads (users) with a count of 2500, with the same thread utilized in each iteration. Further details regarding the thread configuration can be found in Figure 5Figure 5. The payload used is large enough to impact performance and will be used for both protocols as shown below.

```
{
    "name": "Mohammed fataka",
    "age": 23,
    "email": "m_fataka@utb.cz",
    "lifeInfo": "i am Mohammed Fataka aka hamma fataka, i live in beautiful city of zlin in czech republic! i love programming, playing voleyball and ice cream! i am 23 years old and i am doing my bachelor in software engineering in university of tomase bate! i am a dog lover so if you ave one then we are certainly friends"
}
```


gRPC vs Rest.jmx (/Users/hamma/work/projects/gRPC/jmeter/gRPC vs Rest.jmx) - Apache JMeter (5.5)

00:01:55 0 0/100

- gRPC vs Rest
  - save person
    - REST save person request
    - gRPC save person request
    - View Results Tree
    - Response Time Graph
    - Summary Report
    - Aggregate Report
    - Aggregate Graph
    - View Results in Table
    - Simple Data Writer
  - retrieve person

Thread Group

Name: save person

Comments:

Action to be taken after a Sampler error

[✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 10

Ramp-up period (seconds): 5

Loop Count: ☐ Infinite 1000

☑ Same user on each iteration

☐ Delay Thread creation until needed

☐ Specify Thread lifetime

Duration (seconds):

Startup delay (seconds):

Figure 5 “Save person” JMeter thread group configuration

The benchmarking process will be conducted in three stages. The first stage has already been mentioned. The remaining two stages are as follows:

- Second stage with 100 threads, 5 seconds warm-up, and 5000 loop count.
- Third stage with 150 threads, 5 seconds warm-up, and 5000 loop count.

The response time for the first round of benchmarking is shown in Figure 6. It indicates that gRPC initially performed very well, with response times remaining between 1-3 milliseconds. In contrast, REST's response time grew rapidly.

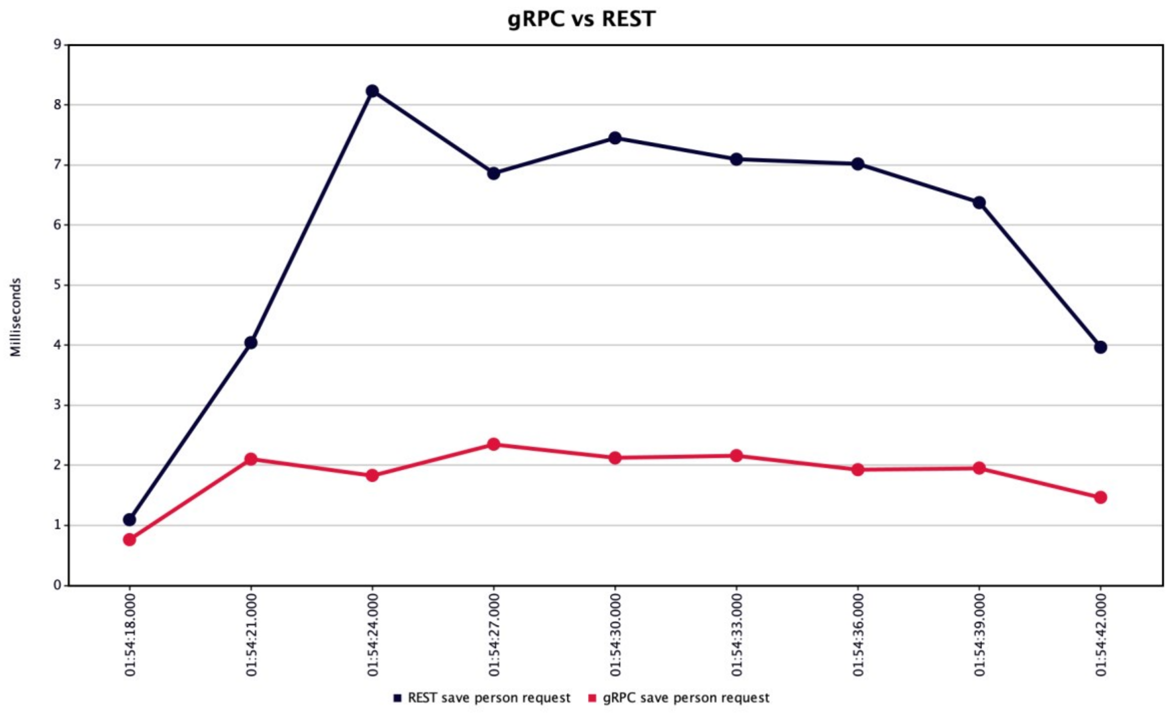
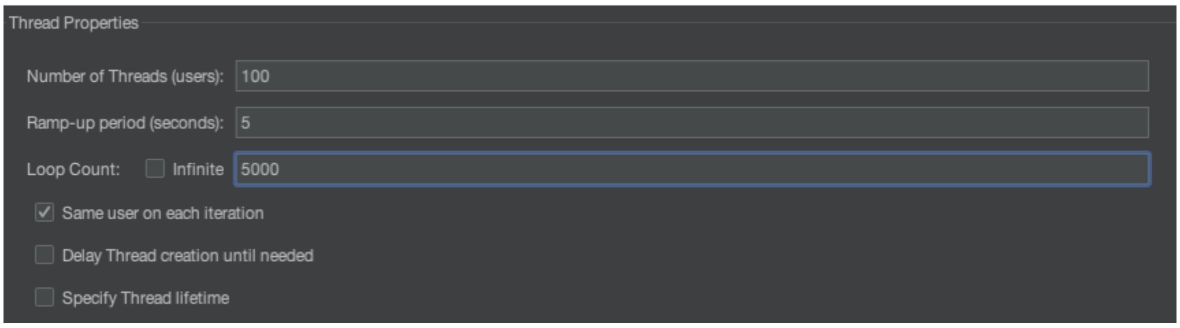


Figure 6 “Save person” first round result

For the second round of benchmarking, the thread properties will change to 100 threads with 5000 counts. These changes are illustrated in Figure 7.


Thread Properties

Number of Threads (users): 100

Ramp-up period (seconds): 5

Loop Count: ☐ Infinite 5000

☑ Same user on each iteration

☐ Delay Thread creation until needed

☐ Specify Thread lifetime

Figure 7 Thread properties for save person second round

As expected, gRPC continues to maintain a response time between 1-3 milliseconds while REST response time grows again, reaching 20 milliseconds. This trend is shown in Figure 8, which is a continuation of the first and second rounds.

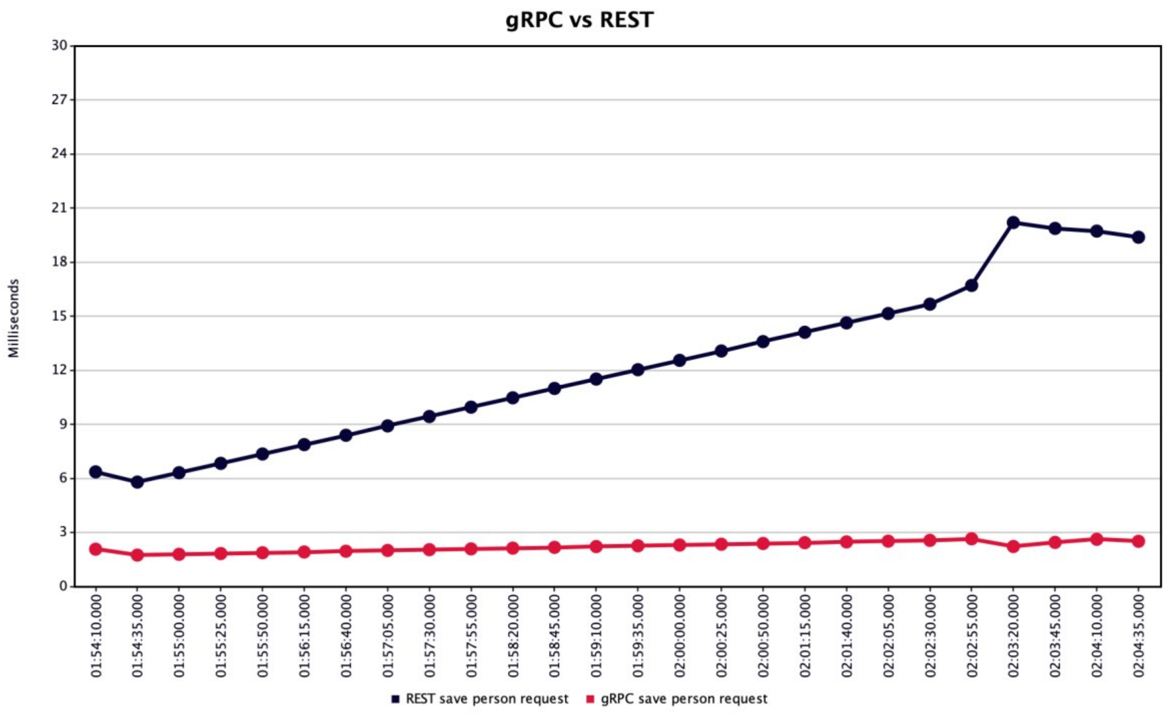
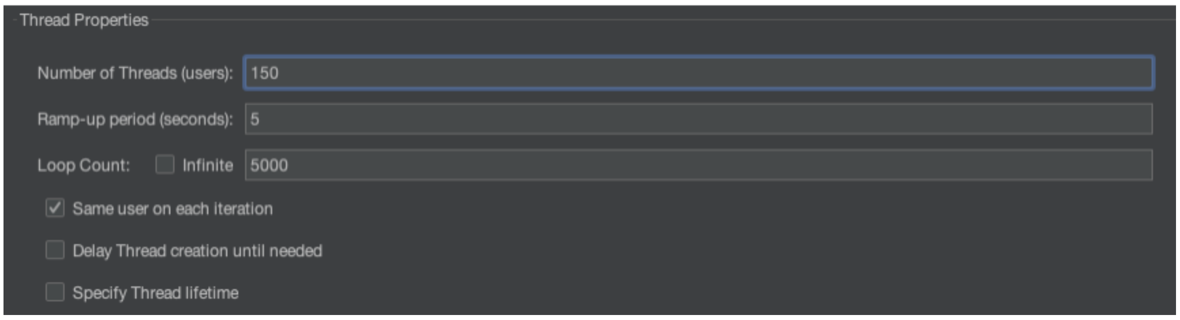


Figure 8 “Save person” second round result

For the third round of benchmarking, the thread properties will change to 150 threads with 5000 counts. These changes are illustrated in Figure 9Figure 9.


Thread Properties

Number of Threads (users): 150

Ramp-up period (seconds): 5

Loop Count: ☐ Infinite 5000

☑ Same user on each iteration

☐ Delay Thread creation until needed

☐ Specify Thread lifetime

Figure 9 Thread properties for “save person” third round

The results of the third round of benchmarking are displayed in Figure 10. As shown, the response time for REST rapidly grew to almost 40 milliseconds, while the response time for gRPC remained between 1-3 milliseconds.

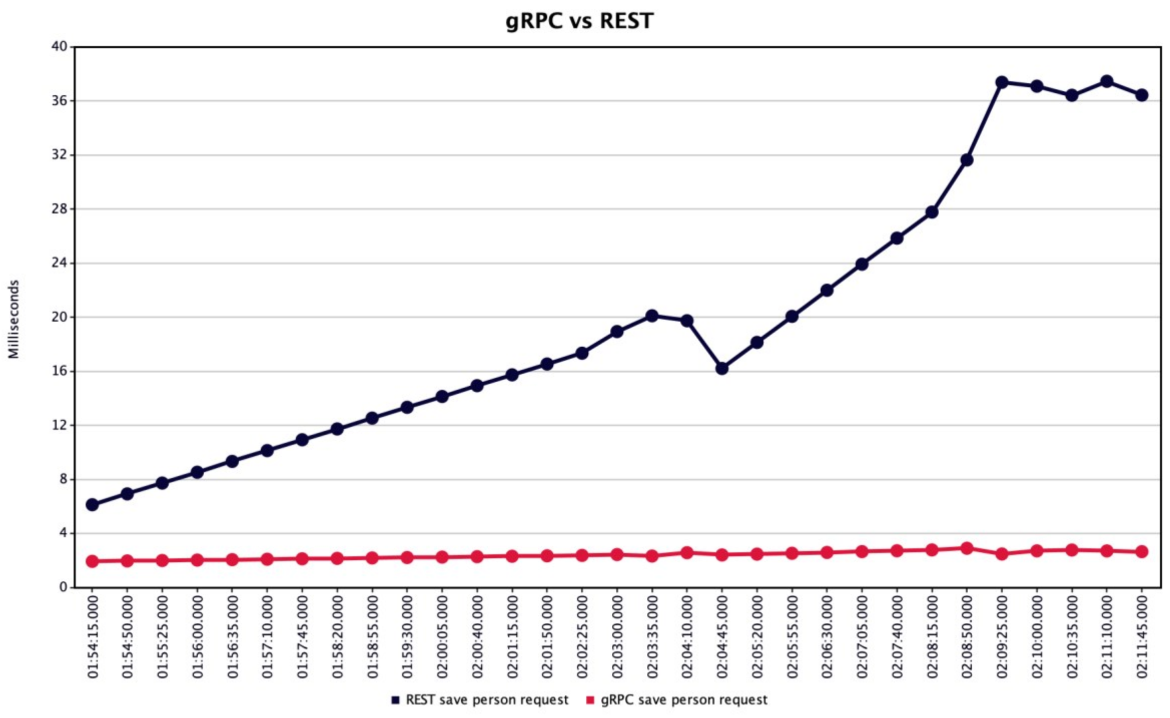
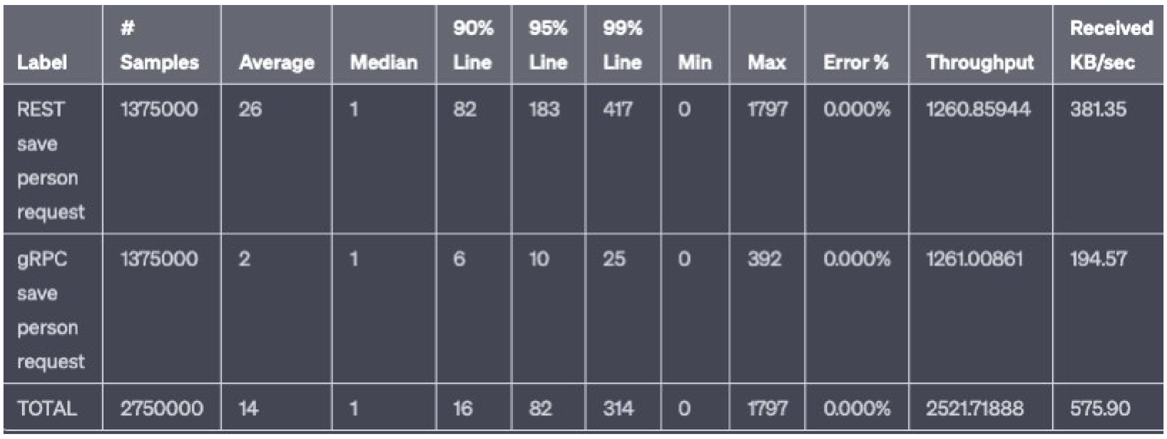


Figure 10 “Save person” third round result

In summary, Figure 11 presents a table with the benchmarking results of both gRPC and REST requests. The table includes the following information:

- Label: name of the request.
- Samples: total number of executions.
- Average: The average elapsed time in milliseconds.
- Median: The median value, which separates the higher half of a data sample from the lower half.
- 90% Line: 90% Percentile, A percentile (or a centile) is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall.
- 95% Line: 95% Percentile.
- 99% Line: 99% Percentile.
- Min: The minimum elapsed time.
- Max: The maximum elapsed time.
- Errors %: The percentage of errors, calculated as errors / (errors + samples) * 100.
- Throughput: The number of samples per second.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/sec
REST save person request	1375000	26	1	82	183	417	0	1797	0.000%	1260.85944	381.35
gRPC save person request	1375000	2	1	6	10	25	0	392	0.000%	1261.00861	194.57
TOTAL	2750000	14	1	16	82	314	0	1797	0.000%	2521.71888	575.90

Figure 11 “Save person” aggregate result

9.2 Receiving object benchmark

In the second part of the benchmark, the goal is to receive an object from the server using both gRPC and REST, with a difference in the amount of data being sent in the request and received in the response. The same configuration used in the first benchmark is applied here, with three stages:

1. First stage with 10 threads, 5 seconds warm-up, and 1000 loop count.
2. Second stage with 100 threads, 5 seconds warm-up, and 5000 loop count.
3. Third stage with 150 threads, 5 seconds warm-up, and 5000 loop count.

The payload used for retrievePerson() on both protocols is as follows:

```
{
    "name": "Mohammed fataka",
    "email": "m_fataka@utb.cz"
}
```

This benchmark is designed to demonstrate whether gRPC can outperform REST in handling requests that require larger data to be sent in the response with less data in request payload.

At the first round of benchmark, 10 threads were applied with a 1000 loop count. In Figure 12, the results are pretty close on both protocols, with gRPC ranging between 0-2 milliseconds and REST ranging between 1-5 milliseconds.

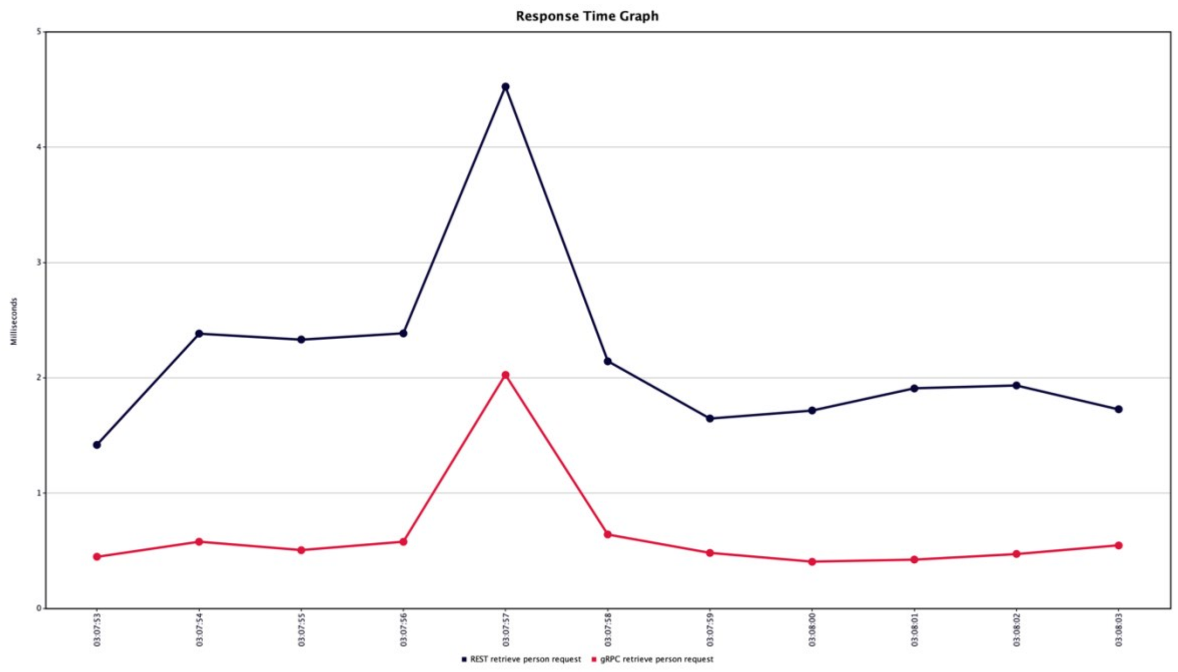


Figure 12 “Retrieve person” first round result

Figure 13 is a graph that shows the results of the second round of the benchmark, including the results from the first round. As shown in the graph, the response time for REST has rapidly grown and reached almost 45 milliseconds, while gRPC has remained between 0-6 milliseconds.

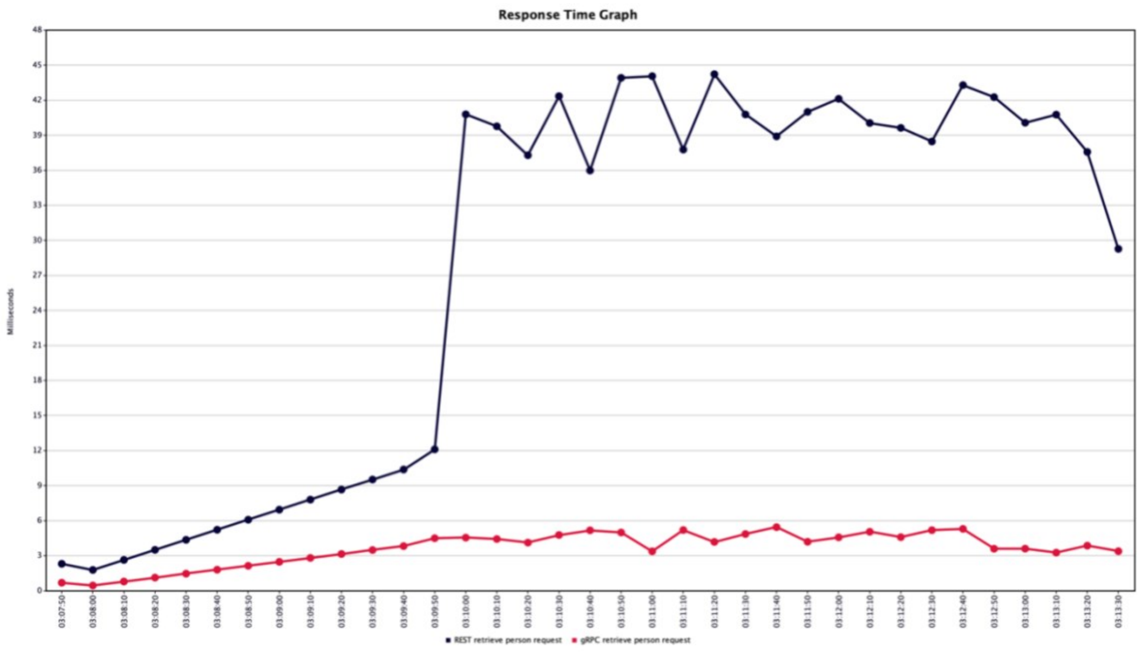


Figure 13 “Retrieve person” second round result

The results of the third round of benchmarking are displayed in Figure 14. As shown, the response time for REST rapidly grew to almost 75 milliseconds, while the response time for gRPC remained between 1-10 milliseconds.

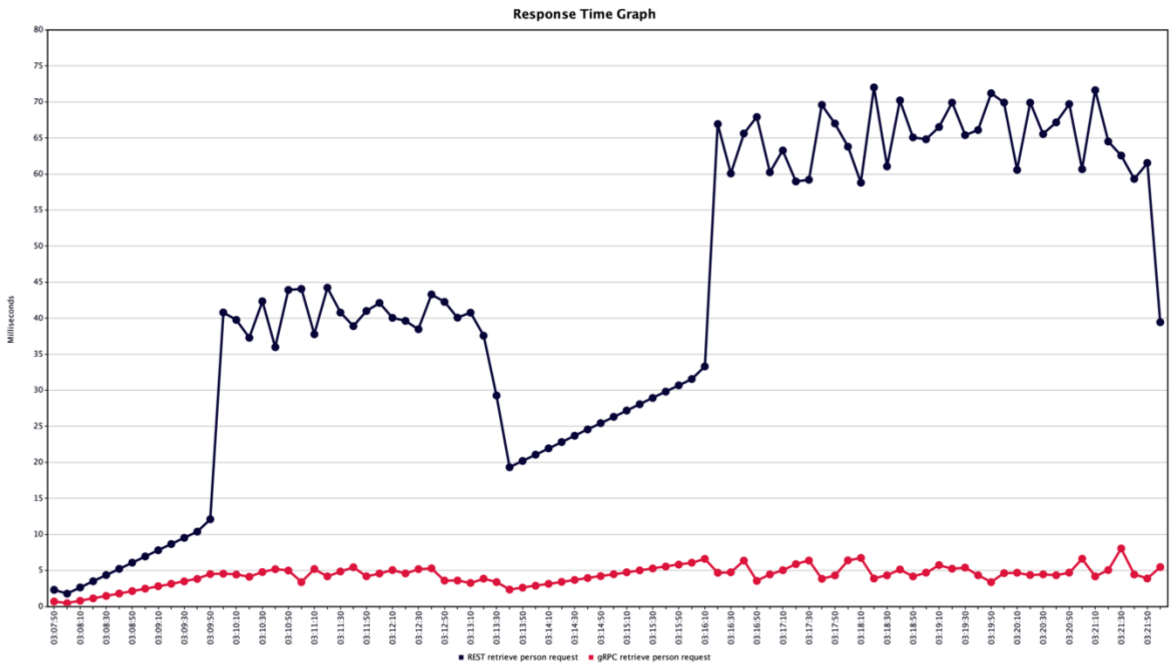
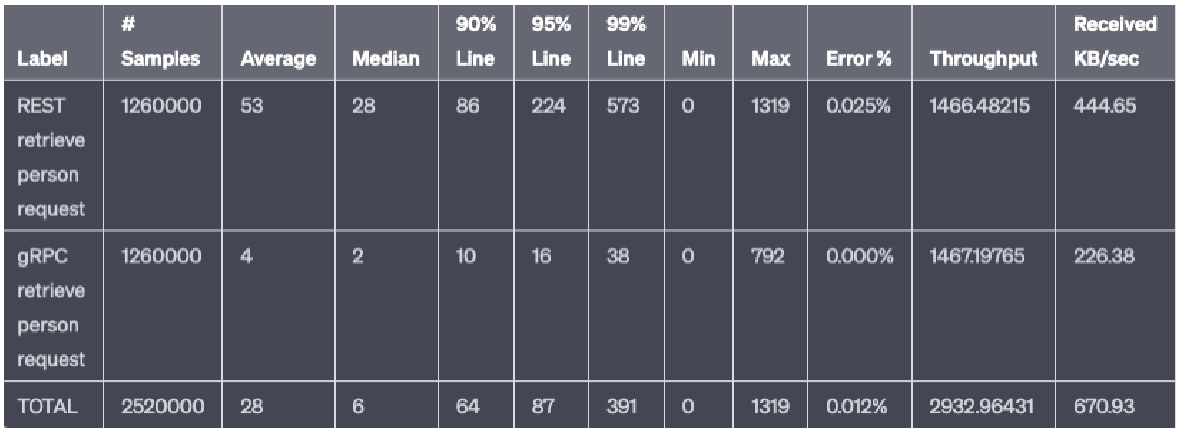


Figure 14 “Retrieve person” third round result

In summary, Figure 15 presents a table with the benchmarking results of both gRPC and REST requests. The table includes detailed information about responses for both protocols.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/sec
REST retrieve person request	1260000	53	28	86	224	573	0	1319	0.025%	1466.48215	444.65
gRPC retrieve person request	1260000	4	2	10	16	38	0	792	0.000%	1467.19765	226.38
TOTAL	2520000	28	6	64	87	391	0	1319	0.012%	2932.96431	670.93

Figure 15 “Retrieve person” aggregate result

The benchmarks compared the performance of gRPC and REST API protocols in a variety of scenarios, including sending and receiving data with varying payload sizes and levels of concurrency. The benchmarks consistently showed that gRPC outperformed REST in terms of both response time and throughput.

In the first round of benchmarks, which involved sending data, gRPC had significantly lower response times and higher throughput than REST, regardless of the size of the payload or the number of concurrent requests.

In the second round of benchmarks, which is about receiving data with a small request payload and large response payload, gRPC again outperformed REST, with significantly lower response times and higher throughput.

The benchmarks showed that, overall, gRPC is a more effective protocol than REST, particularly in circumstances involving high levels of concurrency and big data payloads. It's crucial to remember that each protocol's performance will vary depending on its particular use case and the network infrastructure at its foundation.

CONCLUSION

The objective of this thesis was to demonstrate the potential of utilizing gRPC framework in modern technologies and to assess its congruity with present-day expectations.
Although gRPC is based in the notion of RPC, alternative frameworks depend on REST or other protocols. The gRPC framework has gained significant traction among developers due to its superior performance, speed, and ease of use, particularly in modern client-server architectures, such as microservices.

gRPC provides developers with various benefits due to its emphasis on security, streaming, and protobufs. The outcomes of benchmarking demonstrate that gRPC outperforms REST in terms of both object transmission and retrieval benchmarks, highlighting its capacity for constructing systems that are highly efficient and scalable.

It's crucial to keep in mind, though, that gRPC might not always be the best solution. REST may be a better option if the application has strict needs for backward compatibility. Moreover, gRPC might not be the best option when synchronous communication is necessary, or when the client and server are loosely coupled and follow different development cycles.

The knowledge I have of gRPC has grown significantly as a result of this thesis, which also revealed some of its advantages and weaknesses. I have high hopes that this effort will help readers better understand gRPC and give them useful advice on how to use it.
The thesis intends to encourage a deeper understanding of the technology's promise and limitations by exposing its inner workings, empowering developers to make thoughtful decisions about the technology's adoption. In the end, it is hoped that this thesis would aid gRPC's expansion and advancement as well as its adoption in diverse settings, resulting in client-server communication architectures that are more reliable and effective.

BIBLIOGRAPHY

1. What is Client-Server Architecture? Everything You Should Know | Simplilearn. *Simplilearn.com*. Online. 22 April 2022. [Accessed 1 March 2023]. Available from: https://www.simplilearn.com/what-is-client-server-architecture-articleThis article will explain client-server architecture, show you a client-server model, and illustrate the advantages of client-server architecture. Click here to learn more.

2. Client-Server Technology | Software Architecture: Basic Training | InformIT. Online. [Accessed 2 March 2023]. Available from: https://www.informit.com/articles/article.aspx?p=169547&seqNum=4

3. DRAGUN, N., JARAK, R. and MEDVED, Damir. *Documentation and inventory system based on four-tier architecture*. . 2004. ISBN 978-953-96769-9-3. It can be stated for a fact that relational databases are still the most efficient way of storing large amounts of data. Naturally, this fact caused necessity for developing systems that would allow high level customization of user interface still with efficient data acquiring. Architectures of such systems are becoming more and more modular. This paper describes a model of designing such multitier, multiuser system which deals with large amounts of data

4. What is service-oriented architecture? Online. [Accessed 2 March 2023]. Available from: https://www.redhat.com/en/topics/cloud-native-apps/what-is-service-oriented-architectureService-oriented architecture (SOA) is a type of software design that makes software components reusable using service interfaces that use a common communication language over a network.

5. RUPASINGHE, Hansini. EVOLUTION OF CLIENT/SERVER ARCHITECURES. *Medium*. Online. 17 May 2021. [Accessed 1 March 2023]. Available from: https://hansinirup.medium.com/evolution-of-client-server-architecures-58e531fbf1a6What is Client Server Architecture?

6. Modularity in modern software development. Online. [Accessed 1 March 2023]. Available from: https://www.linkedin.com/pulse/modularity-modern-software-development-cas-nouwensWritten by Cas Nouwens by order of Avans University of Applied Sciences, 's Hertogenbosch Projectmembers: Dylan Schaafstra, Maarten van Alebeek, Koen Fransen, Nina Barkat, Rick Septer, Yorick de Jong Published in: 2019 Cover photo by Andre A. Xavier on Unsplash Abstract Modularity is an old concept,

7. WILLIAM. What is Cloud Native Architecture? Characteristics of a Cloud Native App. *ClickIT*. Online. 19 October 2021. [Accessed 2 March 2023]. Available from: https://www.clickittech.com/devops/cloud-native-architecture/Cloud native architecture is an innovative software development approach that is specially designed to leverage the cloud computing model leverage the cloud computing model

8. Application programming interface. Online. Available from: https://basicknowledge101.com/pdf/km/Application%20programming%20interface.pdf

9. Types of APIs | Types Of API Calls & REST API Protocol. *Stoplight*. Online. [Accessed 5 March 2023]. Available from: https://stoplight.io/api-typesDiscussing different types of APIs, alongside protocols and standards, such as Open APIs, Internal APIs, Partner APIs, Compostie APIs, RESTFUL,JSON-RPC, XML-RPC, and SOAP.

10. CHARBONEAU, Tyler. What's the Difference Between RPC and REST? | Nordic APIs |. *Nordic APIs*. Online. 22 March 2022. [Accessed 5 March 2023]. Available from: https://nordicapis.com/whats-the-difference-between-rpc-and-rest/REST and RPC handle web-based communications in different ways. Here, we compare and contrast REST and RPC to see where each one excels.

11. RPC is Not Dead: Rise, Fall and the Rise of Remote Procedure Calls. Online. [Accessed 5 March 2023]. Available from: http://dist-prog-book.com/chapter/1/rpc.html#remote-procedure-calls

12. IBM Documentation. Online. 10 January 2023. [Accessed 5 March 2023]. Available from: https://www.ibm.com/docs/en/cics-ts/6.1?topic=routing-types-remote-procedure-callThese are the five types of remote procedure call.

13. SHARMA, Gajendra. Asynchronous RPC Interface in Distributed Computing System. *Information Technology Journal*. Online. 15 December 2021. Vol. 21, no. 1, p. 1–7. [Accessed 8 March 2023]. DOI 10.3923/itj.2022.1.7. Asynchronous RPC Interface in Distributed Computing System

14. CORBIN, John. The Art of Distributed Applications : Programming Techniques for Remote Procedure Calls. In : *The Art of Distributed Applications : Programming Techniques for Remote Procedure Calls*. New York : Springer-Verlag, 1991. chapter 6. ISBN 978-0-387-97247-3.

15. IBM Documentation. Online. 20 January 2023. [Accessed 5 March 2023]. Available from: https://www.ibm.com/docs/en/aix/7.2?topic=features-batching-remote-procedure-callsBatching allows a client to send an arbitrarily large sequence of call messages to a server. Batching typically uses reliable byte stream protocols, such as TCP/IP, for its transport. When batching, the client never waits for a reply from the server, and the server does not send replies to batched requests. Normally, a sequence of batch calls should be terminated by a legitimate, nonbatched RPC to flush the pipeline.

16. IBM Documentation. Online. 20 January 2023. [Accessed 5 March 2023]. Available from: https://www.ibm.com/docs/en/aix/7.1?topic=features-broadcasting-remote-procedure-callsIn broadcast RPC-based protocols, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses only packet-based protocols, such as User Datagram Protocol/Internet Protocol (UDP/IP), for its transports.

17. chapter 4: REMOTE PROCEDURE CALL. In : TANENBAUM, Andrew S. and STEEN, Maarten van, *Distributed systems: principles and paradigms*. 2nd ed. Upper Saddle RIiver, NJ : Pearson Prentice Hall, 2007. ISBN 978-0-13-239227-3. QA76.9.D5 T36 2007

18. What is gRPC Protocol? Meaning, Architecture, Advantages . Online. [Accessed 6 March 2023]. Available from: https://www.wallarm.com/what/the-concept-of-grpcgRPC is one of the latest developer approaches to API design that promises to solve problems that other design styles have failed to address.

19. REST or Events? Choose the right communication style for your microservices. *Habr*. Online. 30 June 2021. [Accessed 8 March 2023]. Available from: https://habr.com/en/post/565506/Microservices Architecture is a well-known pattern for

building a complex system that consists of loosely coupled modules. It provides better scalability, and it is easier to develop the system in...

20. Event-Driven Architecture and Pub/Sub Pattern Explained. *AltexSoft*. Online. [Accessed 8 March 2023]. Available from: https://www.altexsoft.com/blog/event-driven-architecture-pub-sub/Whatever the industry, business workflows often rely on distributed software systems. That's why it is crucial to establish the interconnectivity and interope

21. UDANTHA, madhuka. Microservice Architecture and Design Patterns for Microservices. *Medium*. Online. 26 June 2019. [Accessed 8 March 2023]. Available from: https://medium.com/@madhukaudantha/microservice-architecture-and-design-patterns-for-microservices-e0e5013fd58aMicroservices can have a positive impact on your enterprise. Therefore it is worth to know that, how to handle Microservice Architecture…

22. STRENG, Sebastian. The Broker Pattern & how it works. *Medium*. Online. 3 March 2023. [Accessed 8 March 2023]. Available from: https://blog.devgenius.io/clean-architecture-s-broker-pattern-10bc08f57753The more complex software becomes, the more likely it is that it will also "grow". The number of users can increase over time, which drives…

23. MAUREEN. Difference Between RPC and SOAP | Difference Between. Online. [Accessed 9 March 2023]. Available from: http://www.differencebetween.net/technology/internet/difference-between-rpc-and-soap/Difference Between RPC and SOAP RPC vs SOAP Communication is of vital importance in any field be it in business, politics, personal relationships, and even in saving lives. Another area where communication proves to be of utmost importance is through a computer network. Without proper communication avenues, a typical service requester and service provider cannot function in full. In the

24. David Chappell. Online. [Accessed 9 March 2023]. Available from: http://chappellassoc.com/writing/article_Sense_Dist_Objects.php

25. GRPC — Remote Procedure Calls' Choice For The 2020's? – Avenga. Online. 2 July 2020. [Accessed 9 March 2023]. Available from: https://www.avenga.com/magazine/grpc-remote-procedure-calls/gRPC is an open source high performance RPC framework developed by Google originally. Now it's an incubating project of the Cloud Native Computing Foundation.

26. Introduction to gRPC. *gRPC*. Online. [Accessed 12 March 2023]. Available from: https://grpc.io/docs/what-is-grpc/introduction/An introduction to gRPC and protocol buffers.

27. gRPC on HTTP/2 Engineering a Robust, High-performance Protocol. *gRPC*. Online. 20 August 2018. [Accessed 12 March 2023]. Available from: https://grpc.io/blog/grpc-on-http2/In a previous article, we explored how HTTP/2 dramatically increases network efficiency and enables real-time communication by providing a framework for long-lived connections. In this article, we'll look at how gRPC builds on HTTP/2's long-lived connections to create a performant, robust platform for inter-service communication. We will explore the relationship between gRPC and HTTP/2, how gRPC manages HTTP/2 connections, and how gRPC uses HTTP/2 to keep connections alive, healthy, and utilized.

28. 4. gRPC: Under the Hood - gRPC: Up and Running [Book]. Online. [Accessed 12 March 2023]. Available from: https://www.oreilly.com/library/view/grpc-up-and/9781492058328/ch04.htmlChapter 4. gRPC: Under the Hood As you have learned in previous chapters, gRPC applications communicate using RPC over the network. As a gRPC application developer, you don't need to … - Selection from gRPC: Up and Running [Book]

29. VINS. gRPC Unary API. *Vinsguru*. Online. [Accessed 12 March 2023]. Available from: https://www.vinsguru.com/grpc-unary-api/This tutorial explains how to implement a gRPC Unary API in Java & make gRPC blocking unary call and gRPC asynchronous unary calls.

30. Core concepts, architecture and lifecycle. *gRPC*. Online. [Accessed 12 March 2023]. Available from: https://grpc.io/docs/what-is-grpc/core-concepts/An introduction to key gRPC concepts, with an overview of gRPC architecture and RPC life cycle.

31. Overview. *Protocol Buffers Documentation*. Online. [Accessed 12 March 2023]. Available from: https://protobuf.dev/overview/Protocol Buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.

32. Authentication. *gRPC*. Online. [Accessed 2 April 2023]. Available from: https://grpc.io/docs/guides/auth/An overview of gRPC authentication, including built-in auth mechanisms, and how to plug in your own authentication systems.

33. Performance Best Practices. *gRPC*. Online. [Accessed 8 April 2023]. Available from: https://grpc.io/docs/guides/performance/A user guide of both general and language-specific best practices to improve performance.

34. gRPC Load Balancing. *gRPC*. Online. 15 June 2017. [Accessed 8 April 2023]. Available from: https://grpc.io/blog/grpc-load-balancing/This post describes various load balancing scenarios seen when deploying gRPC. If you use gRPC with multiple backends, this document is for you.A large scale gRPC deployment typically has a number of identical back-end instances, and a number of clients. Each server has a certain capacity. Load balancing is used for distributing the load from clients optimally across available servers.

35. gRPC overview | API Gateway Documentation. *Google Cloud*. Online. [Accessed 13 April 2023]. Available from: https://cloud.google.com/api-gateway/docs/grpc-overview

LIST OF ABBREVIATIONS

RPC	Remote Procedure Call.
gRPC	Google Remote Procedure Call.
REST	Representational State Transfer.
Protobuf	Protocol Buffers.
API	Application Programming Interface.
JVM	Java Virtual Machine.
IDL	Interface Definition Language.
Proto	Protobuf Compiler.
XML	Extensible Markup Language.
JSON	JavaScript Object Notation.
CRM	Customer relationship management.
SOA	Service-oriented Architecture.
HTTP	Hypertext Transfer Protocol.
TCP/IP	Transmission Control Protocol/Internet Protocol.
Pub/Sub	Publish/Subscribe.
SQL	Structured Query Language.
CORBA	Common Object Request Broker Architecture.
HTML	Hyper Text Markup Language.
JWT	JSON Web Token.
OAuth	Open Authorization.
SOAP	Simple Object Access Protocol.
TLS	Transport Layer Security.
URL	Uniform Resource Locator.
YAML	Yet Another Markup Language.
XML-RPC	Extensible Markup Language remote procedure call.
IDE	Integrated Development Environments.
Iot	Internet of things

LIST OF FIGURES

LIST OF TABLES

APPENDICES

APPENDIX P I: APPENDIX TITLE