Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Benchmarking and performance analysis of communication protocols

A comparative case study of gRPC, REST, and SOAP

MARKUS NEWTON HEDELIN

Benchmarking and performance analysis of communication protocols

A comparative case study of gRPC, REST, and SOAP

MARKUS NEWTON HEDELIN

Degree Programme in Information and Communication Technology
Date: June 19, 2024

Supervisor: Mohit Daga
Examiner: Elena Troubitsyna
School of Electrical Engineering and Computer Science
Swedish title: Mätning och prestandaanalys av kommunikationsprotokoll
Swedish subtitle: En jämförande fallstudie av gRPC, REST och SOAP

Abstract

The evolution of distributed systems, microservices, and IoT applications has elevated the importance of efficient communication protocols and architectures. While gRPC, with its bidirectional streaming and protocol buffers, presents a compelling alternative to traditional REST and other communication protocols, a systematic performance comparison is needed. Previous published comparative studies exist where the targeted formats have been gRPC versus REST, or where more protocols were included, but the review was limited. A more in-depth empirical study involving scalability across various types of data transfers as well as the impact of security on performance is lacking.

This thesis addresses the aforementioned gap by thoroughly analyzing gRPC's latency, throughput, and scalability performance against widely used communication protocols in a local setting. Through benchmarking and empirical evaluations, the study aims to provide actionable insights into the strengths and limitations of gRPC contra REST and SOAP, aiding developers and organizations in making informed decisions when selecting communication protocols for a microservice system architecture.

By simulating real-world clients interacting with microservices implemented with SOAP, REST, and gRPC and assessing their request throughput and latency across different message shapes, the results show that gRPC outperforms both SOAP and REST. The performance advantage exhibited by gRPC is notably pronounced for smaller messages, although for larger messages, the improvement over REST is only marginally better. SOAP, on the other hand, consistently displayed the poorest performance among the three protocols across all experiments. The results also show that gRPC handles scaling clients the best and its performance is least impacted by enabling the security protocol TLS.

Keywords

gRPC, REST, SOAP, Application Programming Interfaces, API Architecture, Communication Protocol, Distributed Systems, Microservice, Benchmark, Performance, Latency, TLS

Sammanfattning

Utvecklingen av distribuerade system, mikrotjänster och IoT-applikationer har ökat betydelsen av effektiva kommunikationsarkitekturer och -protokoll. Medan gRPC, med sin tvåvägsströmning och Protocol Buffers, presenterar ett övertygande alternativ till traditionella REST och andra kommunikationsprotokoll, behövs en systematisk prestandajämförelse. Tidigare publicerade jämförande studier finns där de utvlada formaten har varit gRPC kontra REST, eller där fler protokoll ingick, men där granskningen varit begränsad. En mer djupgående empirisk studie som involverar skalbarhet över olika typer av dataöverföringar samt påverkan av säkerhet på prestanda saknas.

Denna avhandling adresserar det tidigare nämnda gapet genom att noggrant analysera gRPC:s latens-, genomströmnings- och skalbarhetsprestanda mot väletablereade kommunikationsprotokoll. Genom prestandamätning och empiriska utvärderingar i en lokal miljö syftar studien till att tillhandahålla handlingsbara insikter i gRPC:s styrkor och begränsningar jämfört med REST och SOAP, vilket hjälper utvecklare och organisationer att fatta informerade beslut vid val av kommunikationsprotokoll för en mikrotjänstarkitektur.

Genom att simulera verkliga klienter som interagerar med mikrotjänster implementerade med SOAP, REST och gRPC och bedöma deras förfrågningsgenomströmning och latens över olika meddelandeformer, visar resultaten att gRPC presterar bättre än både SOAP och REST. Prestandafördelen som visas av gRPC är tydligt märkbar för mindre meddelanden, medan för större meddelanden är förbättringen över REST bara marginellt bättre. SOAP, å andra sidan, visade konsekvent sämst prestanda bland de tre protokollen över alla experiment. Resultaten visar också att gRPC hanterar uppskalning av klienter bäst och dess prestanda påverkas minst av att säkerhetsprotokollet TLS är aktiverat.

Nyckelord

gRPC, REST, SOAP, Applikationsgränssnitt, API-arkitektur, Kommunikationsprotokoll, Disitribuerade System, Mikrotjänst, Mätning, Prestanda, Latens, TLS

Acknowledgments

I am deeply grateful to my examiner, Elena Troubitsyna, for granting me the opportunity to undertake this independent thesis. I extend my great appreciation to my supervisor, Mohit Daga, for his rigorous guidance throughout the entirety of my work. Lastly, I wish to express my profound gratitude to my beloved cat, whose companionship provided immeasurable moral support during the long hours spent at my desk.

Stockholm, Sweden, June 2024
Markus Newton Hedelin

Contents

List of Figures

List of Tables

List of acronyms and abbreviations

API	Application Programming Interface
CRUD	Create, Read, Update, and Delete
CTS	Crowdsourcing-based Testing System
FTP	File Transfer Protocol
gRPC	gRPC Remote Procedure Call
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
JSON	JavaScript Object Notation
REST	Representational State Transfer
ROA	Resource-Oriented Architecture
RPC	Remote Procedure Call
RPS	Requests Per Second
SDG	Sustainable Development Goal
SDN	Software Defined Networking
SMTP	Simple Mail Transfer Protocol
SOA	Service-oriented Architecture
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UI	User Interface
UN	United Nations
W3C	The World Wide Web Consortium
XML	Extensible Markup Language

Chapter 1

Introduction

In this thesis, we present an empirical investigation of the performance of Application Programming Interface (API) communication protocols. gRPC Remote Procedure Call (gRPC) is compared to Representational State Transfer (REST) and Simple Object Access Protocol (SOAP) in a local microservice environment. The thesis examines how latency, throughput, and scalability differ between gRPC, REST, and SOAP by benchmarking a program with typical business logic implemented with all three protocols. Additionally, the impact of security overhead on performance introduced by Transport Layer Security (TLS) is investigated. The primary objective of this research endeavor is to furnish actionable insights into the comparative strengths and limitations of gRPC, REST, and SOAP. By doing so, it aspires to empower developers and organizations with informed perspectives, facilitating judicious choices in the selection of communication protocols. Moreover, this study strives to contribute meaningful progression of distributed systems knowledge, fostering the evolution of more efficient and scalable systems amidst the dynamic technological landscape.

1.1 Background

In the landscape of distributed systems and communication protocols, the quest for optimization and efficiency has led to the exploration of alternatives to traditional REST and other API architectures and communication protocols. As the landscape evolves, it becomes imperative to conduct systematic performance comparisons to guide informed decision-making in the selection of communication protocols. Previous studies have performed comparative analyses, notably focusing on gRPC versus REST [1, 2]. However, the existing literature lacks a detailed case study of the nuances of gRPC's latency, throughput, and scalability performance compared to widely used API protocols and architectures. The existing gap regards a direct comparison of gRPC's performance compared to the historically used SOAP and REST in a delimited local microservice setting, employing a load-testing tool that emulates real-world client behavior. This thesis aims to bridge the identified gap by undertaking an examination of the three protocols' performance characteristics in such an environment. Furthermore, prior research is missing the effect transport security has on performance. This thesis addresses this by assessing performance both with and without TLS enabled.

gRPC emerges as a compelling alternative to the traditional communication protocols REST and SOAP. One key factor contributing to gRPC's superiority lies in its efficient use of the HTTP/2 protocol, enabling multiplexing and asynchronous communication [3]. This capability translates into reduced latency and enhanced throughput, in theory, addressing performance bottlenecks often associated with REST and SOAP.

With the performance analysis in this thesis, the main goal is to offer insights that can guide developers and organizations in making informed choices when selecting communication protocols for their distributed systems. Through analytical benchmarking of a rudimentary microservice, this study aspires to uncover the core strengths and limitations inherent in gRPC, REST, and SOAP. Thereby contributing to the knowledge base essential for the development of more efficient and scalable systems in the dynamic and ever-evolving technological landscape.

To underline the significance of this research, it is essential to recognize the role that communication protocols play in the performance of distributed systems. The choice of an appropriate architecture/protocol has significant implications for latency, throughput, and overall system efficiency. As technology advances and the demands on distributed systems intensify, there is a compelling need for research that goes beyond superficial comparisons. For example, the growing use of Software Defined Networking (SDN) exposes the security and efficiency discrepancies between protocols [4]. This thesis positions itself as a response to this imperative, aiming not only to address existing gaps in the literature but also to provide a foundation for future advancements in distributed systems knowledge.

Building upon the foundation laid by prior comparative studies, this research incorporates a more refined case study methodology. The study aims to surpass the limitations of prior research and offer a nuanced perspective. The novelty of this thesis lies in directly comparing the performance of gRPC, REST, and SOAP in a local microservice setup, simulating real-world business logic with various types of messages. It assesses not just the conventional metrics of throughput and latency but also evaluates the scalability of each protocol and TLS's impact within this specific environment. Prior work has either not included all three protocols, has involved a less comprehensive benchmark suite with less variance in message shapes and/or limited review of different amounts of clients, or has not included an analysis of security versus performance. Incorporating these missing parameters into this thesis aligns with the broader objective of advancing the understanding of distributed systems, ultimately fostering the development of communication protocols that are not only effective but also resilient and adaptable to the challenges presented by contemporary technological environments.

In consolidating existing knowledge and providing new insights, this thesis strives to serve as a valuable resource for researchers, developers, and organizations navigating the complex terrain of distributed systems. By combining an analytical approach with a focus on practical implications, the study aspires to provide a foundation for evidence-based decision-making in the selection of communication protocols, thereby shaping the trajectory of future advancements in the field.

1.2 Problem Description

The contemporary industry's fast-phased shift toward microservice architecture and the expansion of cloud services sets new requirements for software developers to implement and maintain API architectures and protocols. Whether you are a developer, an accidental architect, or a solutions or enterprise architect, it is becoming crucial to possess knowledge of API's usability, scalability, and best practices for your given domain [5]. The *microservice architecture* offers the capacity to confront this business challenge. Its essence lies in attaining speed and safety at scale while affording the flexibility to selectively adopt technology for solution implementation [6]. However, the selection and decision-making of the most appropriate technology are complex matters with several factors to include in the process. This thesis aims to deliver concrete results that serve as guidance in this decision-making by comparing API communication protocols in a practical setting with a focus on standardized performance metrics, scalability, and security overhead's impact on performance.

1.2.1 Scientific issues

In a scientific context, an empirical understanding of the efficiency, scalability, and reliability of diverse communication protocols is needed to improve and suggest novel distributed system designs. This thesis seeks to contribute to the foundational empirical research that constitutes the scientific cornerstone of this domain.

1.2.2 Ethics and Sustainability

Although this thesis does not explicitly address ethical or sustainability issues, it is within the thesis's scope to examine how the selection of a highly efficient communication protocol may affect sustainability. As such, a reflection of this potential impact is included in this thesis.

1.2.3 Research Question

How does the performance of a local microservice, concerning throughput, latency, and scalability, implemented with gRPC compare to the traditional API formats, REST and SOAP, across diverse client workloads and with transport security?

1.2.4 Goals

The primary goal of this thesis is to produce the valuable contributions mentioned in Section 1.2 (Problem). To achieve these goals, the thesis has been partitioned into the following objectives:

- Compile knowledge from existing literature about API communication architectures/protocols and well-established comparative methods for these.
- Perform a case study involving:
  - Implement microservices with the communication protocols gRPC, REST, and SOAP based on an existing program.
  - Conduct relevant experiments utilizing the open-source web service load-testing tool Locust, combining its existing features with the findings in the compiled domain knowledge.
  - Perform an evaluation and analysis of the experiment results.

- Draw relevant conclusions from the evaluation and analysis, contributing to future research on related issues.

1.3 Research Methodology

This research employs an empirical methodology to assess the performance of a microservice across various communication protocols, including gRPC, REST, and SOAP. The primary approach is a case study that involves two phases: experiments & data collection and data analysis. In the first phase, a rudimentary program inspired by previous studies [7, 1, 8] will be utilized for the microservice that will be benchmarked. Benchmarking relies on standardized metrics such as latency, throughput, and request success rates. To facilitate the collection of this data, an open-source web service performance testing tool, Locust [9], is used for the experiments. In the second phase, the collected data is analyzed following a comparative procedure that lays the foundation for a discussion and evaluation of the protocols' performance.

1.4 Delimitations

This project's scope includes only the aforementioned benchmarking setting based on existing software, which limits the experimental breadth of implemented tests and services. To perform the experiments, this project utilizes an existing open-source load-testing tool — Locust [9]. Thus, this project does not involve implementing a benchmarking platform from scratch. Nor will the microservice be implemented from scratch, since it is based on existing programs. As this is the case, the outcome of this study does not involve new insights into various microservice implementations or benchmarking constructions. Furthermore, this study does not investigate different network scenarios as the experiments are conducted locally, which further delimits the research.

1.5 Structure of the thesis

Chapter 2 presents the relevant theoretical background information about the communication protocols investigated in this thesis, as well as prior work related to this research. Chapter 3 presents the methodology and method used to approach, study and evaluate the benchmarking and the performance comparison performed in this project. In Chapter 4, the implementation and benchmarking setup for the experiments are explained. In Chapter 5, the results from the benchmarks and the performance evaluation are presented. Lastly, Chapter 6 concludes this thesis with a conclusion of the work and a discussion of related future work.

Chapter 2

Background

This chapter presents basic theoretical background information about communication protocols in general, and the protocols within the scope of this thesis, see Section 2.1. Additionally, this chapter examines how the performance of communication protocols is analyzed in Section 2.2. Finally, related works are explored in Section 2.3.

2.1 API communication architectures and protocols

The term API stands for *Application Programming Interface* and is an old term that surfaced around 80 years ago. A term that has been used for such a substantial period of time tends to be overused and its meaning branched into different definitions depending on the context. There is no exception for *API*. Today, at its core, an API represents an abstraction of the underlying implementation. It is also represented by a specification that introduces types, which enables developers to understand the specifications and use tooling to generate code in multiple languages to implement an API consumer. An API has defined semantics or behavior to effectively model the exchange of information. Effective API design enables an extension to customers or third parties for a business integration [5].

In this thesis, the term *API* is used in the context of *communication protocols* and *communication architectural styles* for interacting with microservices. *Microservices* being an architecture that advocates for the decomposition of software systems into a collection of self-governing services. Each service plays a distinct role within the broader software ecosystem, fulfilling a specific function and contributing to the system's overall functionality [10]. Communication API in this context acts as the conduit for communication between software components. With a set of defined protocols, the interfaces govern the exchange of data and the invocation of functionalities exposed by the microservices [11]. A diagram illustrating the location of this API layer in a simplified typical setting is presented in Figure 2.1.

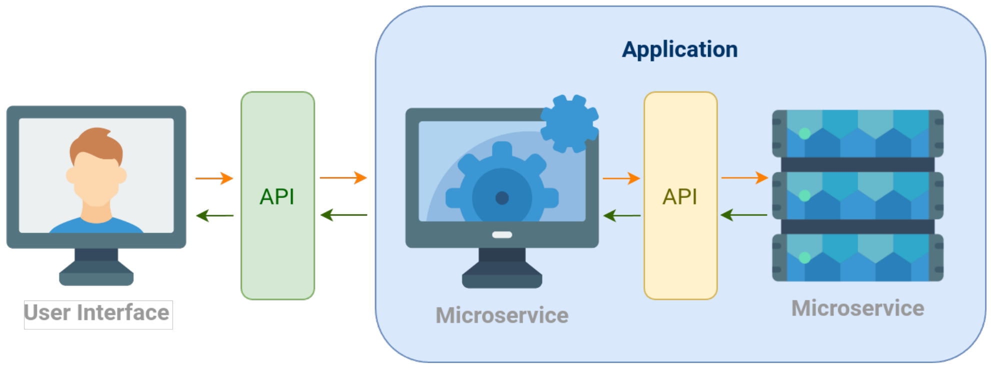


Figure 2.1: Illustration of the communication API layer in a typical microservice setting.

The API protocol for communication between the user interface and the application and the protocol for service-to-service communication do not need to be the same. In this thesis, the only communication API present is the one between the benchmark program and the microservice.

Numerous options for architectures, protocols and styles for communication APIs are available today, all with different optimal use cases. The differences between these architectures lie in the underlying network protocols, data format, data compression technique, available operations, and more [12]. As a developer, choosing the right architecture or protocol for a given task can be difficult because of the vast amount of similar alternatives. However, there are a handful of architectural styles and frameworks more popular than others. The focus of this degree project will be three popular architectures/protocols: REST, SOAP, and gRPC.

2.1.1 Data Exchange

An alternative way to specify the particular use case addressed by this thesis concerning the API is within the framework of *data exchange*. The process of data exchange involves taking structured data from a source schema and converting it into a target schema. The process ensures that the target data accurately represents the information present in the source data [13]. Over the years, data formats have been standardized and as a result, an architectural design has emerged that places APIs at the core of application architecture. Modern application architectures favor client interaction through well-defined APIs that encapsulate server-side functionalities [14].

2.1.2 SOAP

A traditional well-established API communication protocol is SOAP. It is primarily used as a standardized packaging protocol for messages shared between applications. The SOAP specification does not define anything more than an Extensible Markup Language (XML)-based envelope for the information to be transmitted, as well as a set of rules for data exchange, i.e. to convert application and platform-specific data types to XML representations. The design of SOAP makes it suitable for a wide range of application messaging and integration models. For the most part, this has contributed to its increasing popularity [15].

2.1.2.1 SOAP and XML

SOAP is an application of XML. Thus, SOAP is heavily dependent on XML standards such as XML Schema, XML Namespaces, etc. for its definition and operation. The basis of SOAP relies on *XML messaging*, where applications exchange data using XML documents. The underlying idea is that regardless of which operating system, programming language, or other technical details the applications are running, XML can serve as an uncoupled message encoding that all applications involved can utilize to mutually agree on the information shared [15].

2.1.2.2 XML

The Extensible Markup Language (XML) is a data format that defines a specific category of data objects called XML documents. These documents consist of building blocks known as entities, which can contain either processed or raw data. Processed data, further categorized as character data or markup, forms the core content of the document. Markup itself acts as a descriptive layer, encoding information about the document’s storage layout and its inherent logical structure. Importantly, XML provides a mechanism for enforcing constraints on both the storage format and the logical organization of the data within these documents. This is how The World Wide Web Consortium (W3C) introduces XML in their documentation [16].

The documentation also lists the design goals for XML which are deemed necessary to implement and use it. However, this thesis will not address the technical details of XML in any deeper sense. It is assumed that the reader is familiar with the concepts of XML on a higher level to grasp its importance in the context of addressing SOAP as a communication protocol.

2.1.2.3 SOAP Messaging

A SOAP message is composed of an *envelope* featuring an optional header and a mandatory body. The composition can be seen in Figure 2.2. The header contains segments of information relevant to the processing of the message, including configurations for routing and delivery, authentication or authorization declarations, and transaction contexts. The body encapsulates the specific message designated for delivery and processing. Any content that is expressible in XML syntax is allowed for inclusion within the message body [15].

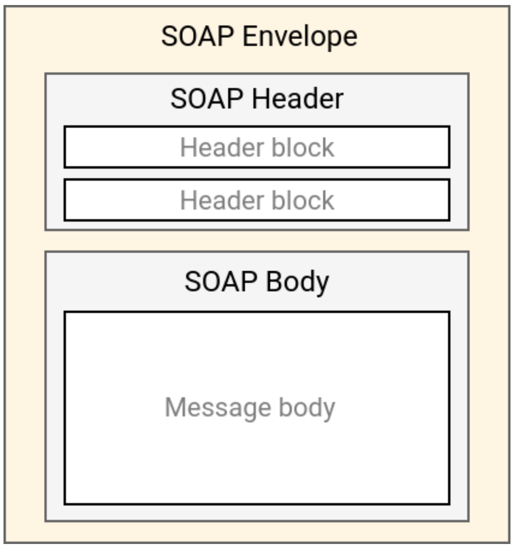


Figure 2.2: SOAP message structure.

2.1.2.4 SOAP Transportation

SOAP communication is commonly conducted using Hypertext Transfer Protocol (HTTP) as the transport protocol for synchronous data exchange and File Transfer Protocol (FTP) and Simple Mail Transfer Protocol (SMTP) for asynchronous interactions [14]. HTTP is, however, the most preferred transport protocol by a significant majority. Since HTTP is a request-response-based protocol, it pairs well with SOAP's Remote Procedure Call (RPC) (request-response) conventions (illustrated in Figure 2.3). The frequent use of SOAP over HTTP has resulted in the SOAP specification incorporating an extra section for HTTP specifically, directly within the specification. It provides detailed guidance on how the semantics of the SOAP message exchange model align with HTTP [15]. In a Service-oriented Architecture (SOA), a common setting is to use HTTP for the transport of messages, XML for description of data, and SOAP for invocation [17].

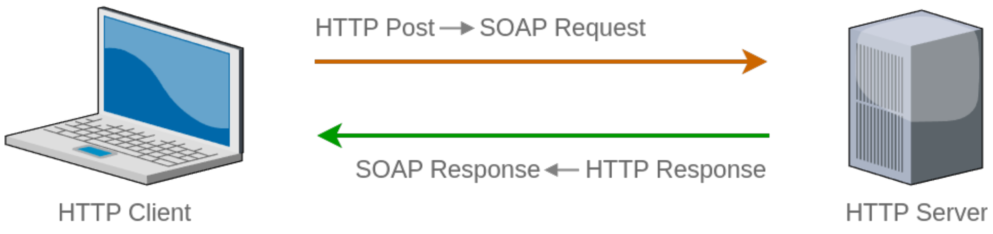


Figure 2.3: SOAP over HTTP.

2.1.3 REST

Representational State Transfer (REST) is a *resource-oriented* architectural style for building distributed systems. The term was coined by Roy Fielding in his PhD dissertation, published in 2000 [18]. The central concept lies in the data, not in actions or invocations. The key aspect of REST is its incorporation of a resource-centric view of applications. As opposed to SOAP, REST is not limited to XML, it also supports JavaScript Object Notation (JSON), plain text, and more. Generally, the REST architecture is much lighter compared to SOAP. It does not require formats like headers to be included in the message, as is the case for SOAP. Instead, it parses JSON by default: a human-readable language designed to allow easier parsable and faster data exchange by a machine.

In similarity with SOAP, REST is intended (and most commonly implemented as such) to utilize the standard HTTP methods to facilitate the functionality of the architecture. A simple illustration is presented in Figure 2.4.

Even though REST is liberated of the requirements of SOAP's message formats, its core is a set of constraints obligatory to follow when developing a *RESTful* service [17].

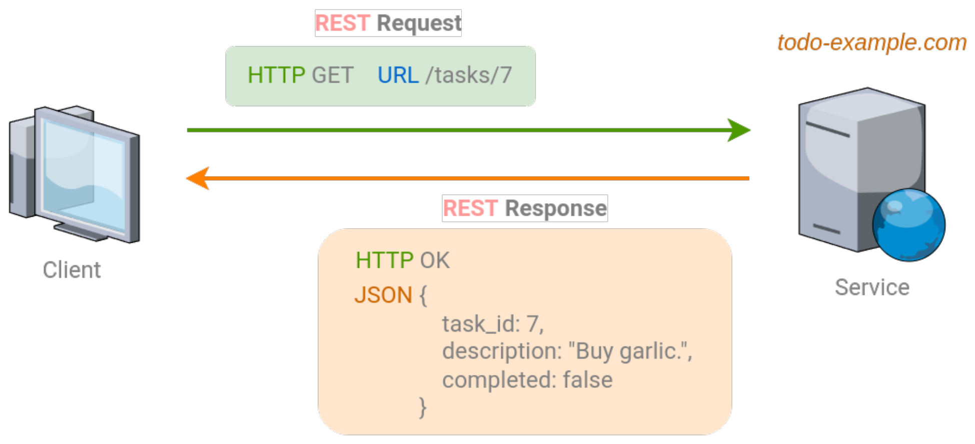


Figure 2.4: REST over HTTP.

2.1.3.1 RESTful constraints

When a REST application successfully implements certain mandatory constraints the system is deemed *RESTful*. The claim is that this results in a highly scalable system that is allowed, thanks to the self-describing representation formats, to grow organically and in a decentralized manner. This involves considering the following set of constraints:

1. *Resource Identification:* Every resource pertinent to an application and its state should be assigned distinctive and consistent identifiers. These identifiers must be global to ensure they can be dereferenced regardless of the context. It's crucial to note that the term *resource* in this context extends beyond the static entities an application interacts with; it also comprises all the information necessary to discuss these entities, including transactional documents like orders.

2. *Uniform Interface:* All interactions should be developed around a uniform interface, offering a comprehensive and functionally adequate

set of methods for engaging with resources. This principle stands in clear contrast to RPC, where functionality exposure primarily involves defining a set of remotely invocable methods. In the realm of REST, the concept of methods that can be called doesn't exist. Instead, RESTful services expose resources, and interactions with these resources can only employ the uniform interface or a subset of it.

3. *Self-Describing Messages:* When dealing with resources through the uniform interface, REST insists on using representations that capture the essential aspects of these resources. These representations should be designed thoughtfully to give involved parties a clear understanding of resources or relevant states just by inspecting them. Any changes in resources or states are conveyed by swapping these representations through the uniform interface. To meet this requirement, the uniform interface should provide a way to *label* representations during information exchanges. This eliminates the need for external information or pre-arranged agreements to make sense of a received representation. It is crucial to note that in this context "self-describing" is not about deep semantic meanings but simply means that you don't need extra information to process a representation exchanged through the uniform interface.

4. *Hypermedia Driving Application State:* The exchanged representations should be interlinked, enabling an application that comprehends a representation to locate and understand these links. The semantics of these links are dictated by the representation, allowing the application to utilize them as they lead to other identified resources accessible through the uniform interface. The presence of links is vital, as it facilitates the exposure of new resources and enables applications to navigate specific state transitions. This constraint allows loose coupling, as identifiers can be discovered at runtime and interacted with through the uniform interface, eliminating the need for any prior agreements between interacting parties.

5. *Stateless Interactions:* This last constraint implies that every interaction between a client and a server must be entirely self-contained. There should be no reliance on the client state, commonly known as a *session* maintained on the server. Such sessions should not influence an interaction based on both the exchanged representation and the client-associated session. While any interaction can lead to a change in a

resource, subsequent interactions with that resource will only reflect the altered resource state. It is important to distinguish this change in resource state from a server-maintained client session. The server is only required to keep track of resource states and not client sessions. This constraint plays a crucial role in ensuring server scalability is constrained solely by the number of concurrent client requests, not by the overall number of clients with which they interact. [19]

2.1.4 gRPC

In gRPC, a client application can seamlessly invoke a method on a server application located on a different machine, mirroring the simplicity of calling a method on a local object. This streamlined approach facilitates the development of distributed applications and services. Like many RPC systems, gRPC revolves around the concept of defining a service, specifying the remotely callable methods along with their parameters and return types. On the server side, the server implements this service interface and operates a gRPC server to manage incoming client calls. Meanwhile, on the client side, a stub, or a "client" (commonly referred to in different languages), provides access to the same methods available on the server.

gRPC uses *protocol buffers* by default, Google's open source mechanism for data exchange, i.e. serializing structured data [20].

Protocol buffers serve as gRPC's Interface Definition Language (IDL). When developing a gRPC application, the initial step involves defining a service interface. This interface definition encapsulates details on how your service is intended to be consumed. It outlines the methods that consumers are permitted to call remotely, specifies the parameters and message formats for invoking those methods, and contains other relevant details necessary for interaction with the service. This service interface is defined in the protocol buffer for the intended service; in a proto file. The proto file is the origin of the code generated by a *protoc* compiler, for both client and server. This way, involved parties only need the proto file - containing all necessary information about the service - to generate viable code to start acting as either a client or server. The client can generate code in one language and the server in another, as a result of protocol buffers being language-agnostic and platform-neutral.

In contrast to both SOAP and REST, gRPC's network communication is performed over HTTP/2.

To visualize the process of defining and developing a service using gRPC, an example of the implementation of a product info service is displayed in Figure 2.5. There, the service is defined in the proto file `ProductInfo.proto`, which both server and client use for generating code in their hosted languages: Go and Java, respectively [21].

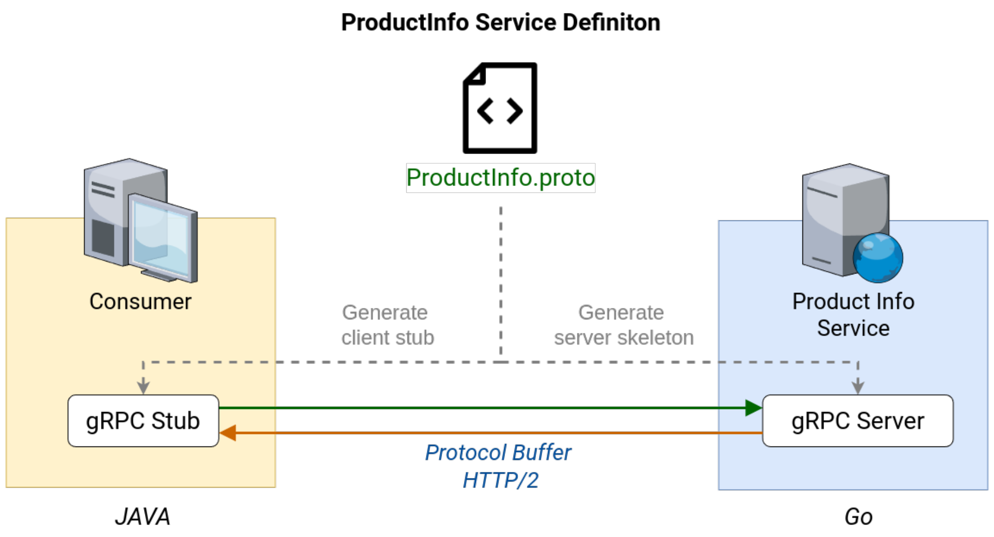


Figure 2.5: An overview of an example service implemented with gRPC. Image is adapted from [21].

2.1.4.1 HTTP/2

Up to this point, the network communication protocol HTTP has been frequently mentioned, and the reader is assumed to possess basic knowledge of this protocol. However, despite having been launched several years ago, HTTP/2 is not as commonly known. Without diving deeper into HTTP/2 in general, it is relevant to this thesis to examine the functionality this protocol provides that gRPC depends on.

HTTP/2 stands as the second significant iteration of the internet protocol HTTP, introduced to address issues related to security, speed, and other concerns found in the preceding version, HTTP/1.1. While retaining all the core features of HTTP/1.1, HTTP/2 implements them more efficiently. As a result, applications developed using HTTP/2 exhibit enhanced speed, simplicity, and robustness. Within HTTP/2, the communication between a client and a server is performed over a singular Transmission Control Protocol (TCP) connection, capable of accommodating numerous bidirectional flows of bytes. This was not possible in the previous version, where multiple HTTP requests/responses had to take place in parallel TCP connections (see example in Figure 2.6) [21]. gRPC leverages HTTP/2 as its transport protocol for transmitting numerous messages across the network over a single TCP connection. This utilization of HTTP/2 significantly contributes to gRPC's status as a high-performance RPC framework [22].

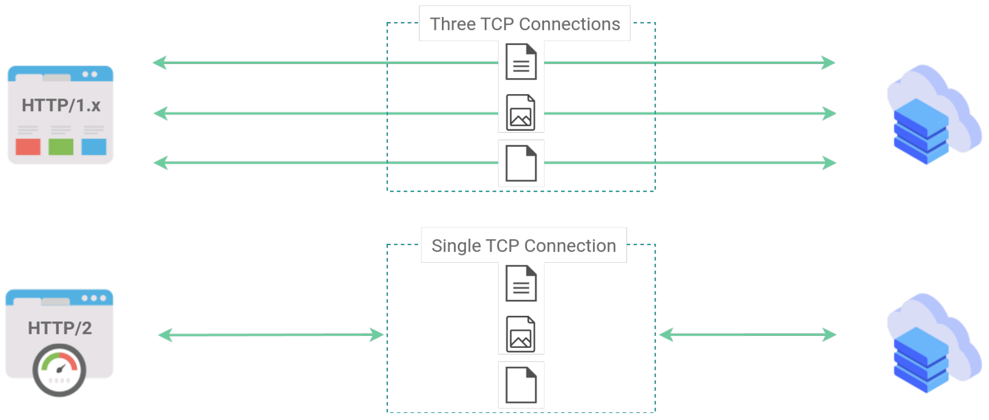


Figure 2.6: HTTP/2 accommodates several requests over a single TCP connection, as opposed to HTTP/1.x. Based on image from [23].

2.1.5 Evolution of SOAP to REST to gRPC

In the history of inter-process communication, there has been a rather clear evolution of communication protocols. SOAP was established first, with its possibility to specify a service interface and utilization of XML's message format. However, the expanding complexity involved in SOAP's message format and specifications presented difficulties in the agile development of distributed systems. A response to this was REST, the even more popular successor that introduced a Resource-Oriented Architecture (ROA), which became de facto standard for microservice communication. REST arrived with a human-readable message format, JSON, which proved to be appropriate for development, but also for its overall compatibility with HTTP. Although REST is still popular today, it entails its own complications. For one, the human-readable text format pairs well with HTTP, but is inefficient in the context of service-to-service communication. This is illustrated in Figure 2.7. gRPC provides a significantly more efficient data exchange with its binary-based protocol buffers over HTTP/2, which is one of the reasons it has grown in popularity when developing large-scale distributed applications [21]. All three protocols remain prevalent today, prompting this thesis to explore how gRPC fares against its predecessors, SOAP and REST, in a targeted microservice environment.

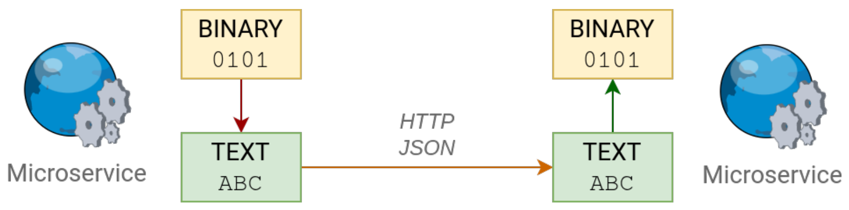


Figure 2.7: REST applications have to convert binary data to textual representation, send it over HTTP/1.x, then the receiving entity parses the text and finally converts it back to binary format.

2.2 Communication Protocol Performance Analysis

To assess the performance of communication protocols, standardized performance metrics must be gathered and analyzed for the application employing these protocols. This thesis focuses on runtime quality attributes concerning performance and scalability. Performance refers to the system's responsiveness in executing actions within a specified timeframe, which can be measured in terms of latency or throughput. Scalability indicates the system's capacity to accommodate increased loads without compromising performance, or its ability to be easily expanded [24]. Measuring performance for communication protocols often goes hand in hand with inspecting network performance metrics.

A prevalent method for collecting performance data involves utilizing established benchmarking or load-testing frameworks tailored to the specific application under scrutiny. In this study, the load-testing tool Locust is employed, which is examined in Section 2.2.2.

2.2.1 Key Performance Metrics

Common key network performance metrics are *latency, bandwidth & throughput* and *error rate*. Descriptions for each metric are presented in Table 2.1. These metrics translate well to performance metrics for communication protocols, which is why they are employed in this thesis.

Table 2.1: Key performance metrics for networks and communication protocols [25].

Metric	Description
Latency	The delay in transmitting data across a network and receiving a response is assessed through latency metrics. Low latency means faster data transmission, while high latency equals slower data exchange. In the context of microservices, this involves measuring the time between a sent request and a received response from the service.
Bandwidth & Throughput	Bandwidth refers to the maximum data transfer capacity in a network, while throughput reflects performance by measuring the number of data transfers within a set timeframe.
Error Rate	A measurement of the error frequency in requests being sent over a specific period. This indicates the reliability of the service under evaluation.

2.2.2 Locust

Locust is the primary tool for collecting performance data to enable the analysis in this thesis. Locust is an open-source load and performance testing tool that offers a developer-friendly approach to testing HTTP and various other protocols. It enables users to define tests using standard Python code that can be executed either through the command line or via its web-based user interface. This allows testers to gain real-time insights into various performance metrics such as throughput, response times, and errors, which can then be exported for subsequent analysis.

What makes this Locust versatile is its ability to test various application features by manipulating the load. Test scenarios are scripted in Python, enabling the generation of distributed system loads with master and slave nodes. Configuration adjustments, like modifying the number of users and test duration, can be performed directly within the web User Interface (UI).

The testing process usually involves swarming a web application with numerous requests, and the system under test can be implemented in any programming language, providing complete customization with various features. Moreover, Locust incorporates a built-in module for displaying test results in the web UI. This UI offers evaluation features such as response time analysis, exception analysis, graphical analysis, and error analysis.

For focused results, the web UI can be disabled during the test, printing only the outcomes at the conclusion. Alternatively, tests can run in headless mode, revealing results solely at the test's conclusion [9, 26].

2.2.3 Security versus Performance

Ensuring the security of APIs against malicious activities is a crucial consideration for developers during system implementation. This entails implementing mechanisms such as authentication, authorization, input validation, and encryption. However, it is important to acknowledge that security measures can impact performance. The additional steps involved in securing data transfers, such as encryption, often lead to increased overhead, affecting both throughput and latency. While this thesis does not delve deeply into API security, the impact of security measures on performance is pertinent to this investigation due to its significant trade-off. One common approach to securing API data transfers is through *Transport Layer Security (TLS)*. This thesis focuses on examining the performance impact introduced by enabling TLS.

The TLS protocol involves two primary phases: the handshake and the data transfer. In the handshake phase, the client and server exchange information about their cryptographic capabilities and establish keys for encrypting the data to be sent. Following this, the data transfer occurs, where the data is divided into records and securely transmitted between the client and server using the previously exchanged cryptographic keys. TLS can be configured for one-way or two-way authentication. One-way authentication involves the server sending its certificates to the clients for them to verify, while two-way authentication adds the reverse process for the server to verify the clients [27]. In this thesis, one-way TLS is configured for the microservice and its clients.

2.3 Related works

Various research studies have delved into this topic. This section provides a concise overview of a selected subset of those studies, related to this project, outlined in Subsection 2.3.1. Additionally, Subsection 2.3.2 highlights different implementations of microservices for similar comparative analysis. Finally, Subsection 2.3.3 examines benchmark suites utilized in related work.

2.3.1 Comparative Studies

In 2020, a comparative analysis was conducted to assess the request throughput of microservices utilizing both REST and gRPC within a cluster setting [1]. The evaluation involved load-testing a product information system comprising multiple microservices. Results revealed that, under specific conditions, gRPC demonstrated superior request throughput, particularly with fewer concurrent virtual users. Additionally, the study highlighted the performance implications of synchronous versus asynchronous communication. In contrast to this previous study, the focus of this thesis is to isolate the communication protocols within a local environment utilizing a single microservice. This approach aims to provide a clearer understanding of the protocol impact without the influence of network-related variables. Furthermore, this thesis's test suite includes various message types with differing sizes to the microservice, rather than a single message format, enriching the analysis.

In another related study conducted in 2022 [2], all three protocols explored in this thesis - SOAP, REST, and gRPC - were claimed to be compared alongside Websocket and GraphQL. However, it is worth noting that SOAP was not included in the testing phase. Despite this, the comparison of gRPC and REST yielded valuable insights. For the performance analysis, clients and servers for each architecture/protocol were hosted on the same machine. The tests, though limited to only three cases - inserting an entry, fetching a single entry, and fetching multiple entries from a database exposed with different API protocols - produced interesting results. The study found that gRPC performed better in terms of latency, while REST consumed less memory on the host machine. In addition to including SOAP in the experiments, this thesis draws inspiration from this latency evaluation by benchmarking with different request types in a local environment. Moreover, this thesis extends the scope by including scenarios with various numbers of concurrent clients sending those requests to further assess scalability. Specifications of the benchmark suite are elaborated in Chapter 4.

In a third case study focusing on SOAP and REST web services for multimedia conferencing [28], it was observed that RESTful services demonstrated an approximately threefold decrease in latency compared to SOAP services. The performance difference is present in both local and distributed networks. Here, the authors argue that the large difference is due to SOAP's message processing, where the data in the message envelope has to be extracted resulting in longer request handling. In similarity with this thesis, the study conducted experiments where all components were hosted on a single machine and measured average latency. To build on this previous work, this thesis also includes measurements of the 95th percentile latency, i.e. the latency that 95% of the data transfers are below.

As the aim of this thesis is to advance prior research, two new types of tests are conducted. The first focuses on scalability, examining the saturation point of concurrent users for each protocol. The second explores the impact of security overhead on performance, specifically investigating how enabling TLS affects performance.

2.3.2 Microservice Implementation for Benchmarking

To properly assess the real user-like performance of gRPC, SOAP, and REST, it is important to consider the implementation of microservices. A recent research article put forward a novel proposal for a crowdsourced testing system. The web service was developed using Node.js. In contrast to other similar studies, the system under examination was considerably more complex. Specifically, the server side of the system comprised multiple microservices, including an event controller, a node manager, a test manager, an HTTP server, and a database server [29].

In [30], a more simplistic method was taken, utilizing a Java server and client to only handle the fundamental functionalities associated with HTTP's *GET, POST, PUT, and DELETE* methods. Payload size remained unchanged throughout the different tests. The analysis aimed to assess the response time performance of SOAP and REST.

In [7], microservices were implemented using three distinct programming languages: Java, Python, and Rust. The application simulated an inventory service to handle requests for data of four varying sizes, which inspired the request types in this thesis. Each programming language employed a specific API development framework, namely Spring for Java, FastAPI for Python, and Axum for Rust.

The implementation in [31] was also quite limited. However, in this study, the service could handle four different types of requests to evaluate the processing performance with various payload sizes and database lookups. The experiment involved two services, a server to handle client requests and a database service to deliver data to the server. The entire system was developed using Java.

The microservice implementation in this thesis is influenced by these earlier studies. It models a simplified real-world application to replicate internal business logic latency, and handles requests of diverse sizes. Further details regarding the implementation are presented in Chapter 4.

2.3.3 Benchmark suites

In [1], the benchmark suite for evaluating inter-process communication between microservices revolved around Apache JMeter. Designed for load testing functional behavior and performance measurement, Apache JMeter is an open-source software completely built with Java, and was originally developed for testing web applications [32].

The Crowdsourcing-based Testing System (CTS) proposed in [29], offers a different approach to benchmarking web services. The idea is that distributed test nodes across the globe provide a better representation of real user behavior. Through more accurate simulations, the system under test can be more reliably evaluated. In the proposed system, different metrics can be investigated and data collected through a central testing microservice. However, this kind of testing system is not publicly available. Nonetheless, the concept is an interesting alternative to tools operating within a local infrastructure - such as Locust.

The framework for load-testing web services, Locust, was used as the primary benchmarking tool in [31]. Locust provides evaluation functionality for response time, throughput, and content size in data exchange. The Locust tool is developed in Python and can simulate multiple concurrent users, which expedites real-world-like testing. Locust is used for benchmarks in this thesis and its incorporation is explained in Chapter 4.

Chapter 3

Method

This chapter outlines and explores the research method applied in this thesis. Section 3.1 delves into each essential step of the research process. Section 3.2 explains the experimental design and planned measurements, the test environment, including used hardware and software tools. Finally, Section 3.3 describes the performance analysis and evaluation of the benchmark data.

3.1 Research Process

The research process employed to address the research question of this thesis comprises four main steps. These steps are outlined in Figure 3.1. The primary method was a comparative empirical case study. First, a literature review of the domain was conducted. Secondly, microservices were implemented with the investigated communication protocols, SOAP, REST, and gRPC, followed by a configuration of the benchmarks suite. Thirdly, performance benchmarking was executed on the microservices along with data collection of performance metrics. Finally, an analysis and evaluation of performance based on the compiled data concluded the research.

3.2 Experimental design

The main performance assessment in this thesis relies on the data collected from benchmarking three distinct microservices implemented with the communication protocols SOAP, REST, and gRPC. The experimental design and the planned measurements are influenced by previous studies exploring

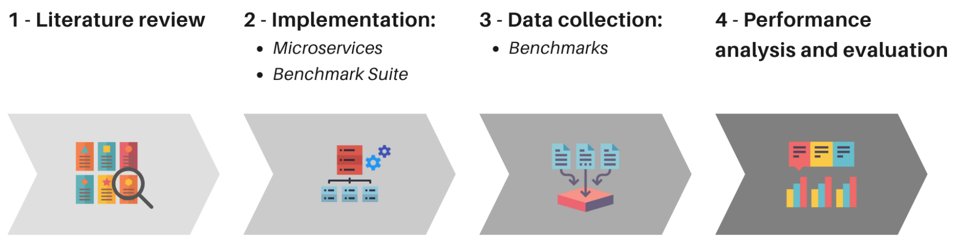


Figure 3.1: Research process overview.

similar scenarios. These related works laid the foundation for the design choices made for the microservice implementation and the benchmarking suite, elaborated upon in subsection 3.2.1. Additionally, subsection 3.2.2 provides an overview of the final testing environment.

3.2.1 Implementation and Benchmarking Design

A prior study investigated the performance of gRPC and REST protocols in microservice communication within an e-commerce context [1]. The experiment centered around a client requesting a product page, with information fetched from multiple microservices using different communication protocols. The evaluation aimed to evaluate the performance efficiency of gRPC and REST by measuring throughput, defined as the total number of requests and responses completed within a set timeframe. This study's approach served as a foundational model for the microservice design in this thesis, particularly in simulating a microservice delivering product information. However, to enhance the evaluation beyond previous work, inspiration was drawn from additional studies, such as the comparative review by Ł. Kamiński et al. [2] and another similar case study [8], by M. Bolanowski et al.. These sources influenced the experiments with their varied request types to more accurately mimic real-world scenarios and provide a more comprehensive performance assessment.

Moreover, latency data was gathered, including average and 95th percentile metrics, essential for API performance evaluation [33]. The 95th percentile represents the latency value below which 95% of the requests fall, providing valuable insight into worst-case delays. Additionally, a scalability test was devised to contribute novelty to existing research by exploring how each communication protocol performs under constantly increasing concurrent client loads, thus augmenting the scope of previous investigations. Furthermore, another test that extends prior work was conducted. This test specifically investigates the security overhead introduced by enabling TLS and its impact on performance. Evaluating the effect of TLS on performance provides valuable insight into the trade-offs between security considerations and system responsiveness. Given the reliance of real-world applications on secure communication, this investigation holds significant relevance in understanding the practical implications of implementing TLS within communication protocols.

Details of each test within the benchmark suite, along with implementation specifics, are outlined in Chapter 4. In summary, for testing throughput and latency, four request shapes were tested. They represented different payload sizes and invoked various server functions: *small, typical, large, and stress*. Each request type was simulated by 50, 100, and 200 virtual users using the load-testing tool *Locust*, targeting one microservice for 120 seconds each. Figure 3.2 provides an overview of the benchmark suite, illustrating the testing process. For instance, the initial test involved 50 virtual users sending small requests to the SOAP-based microservice for 120 seconds. Subsequently, the scalability test and security overhead test were conducted (see Chapter 4 for details).

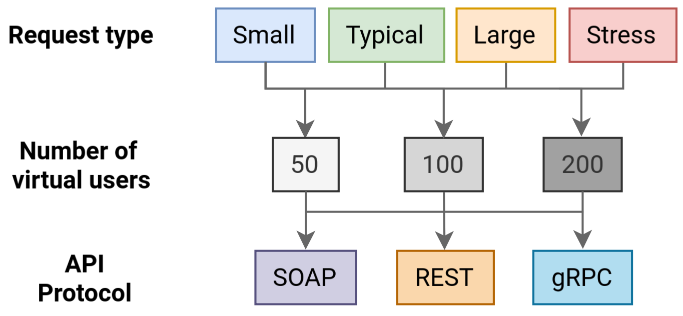


Figure 3.2: Overview of all different benchmark request shapes and the combinations of virtual users making up the benchmark suite, excluding the scalability test.

3.2.2 Test Environment

The topology of the final test environment is illustrated in Figure 3.3. The entire test environment was hosted on a single machine. A microservice was implemented for each communication protocol: SOAP, REST, and gRPC. However, all three microservices utilize an identical core service that encapsulates the provided functionality. The difference among the microservices lies in the API endpoint exposed, implemented with each respective communication protocol. Correspondingly, the load-testing framework Locust manages distinct virtual users (clients) directed towards the various endpoints. Locust was selected as the primary benchmarking tool due to its user-friendly features for spawning and controlling concurrent virtual users targeting an existing web service API. This selection eliminated the need to develop a benchmarking suite from scratch.

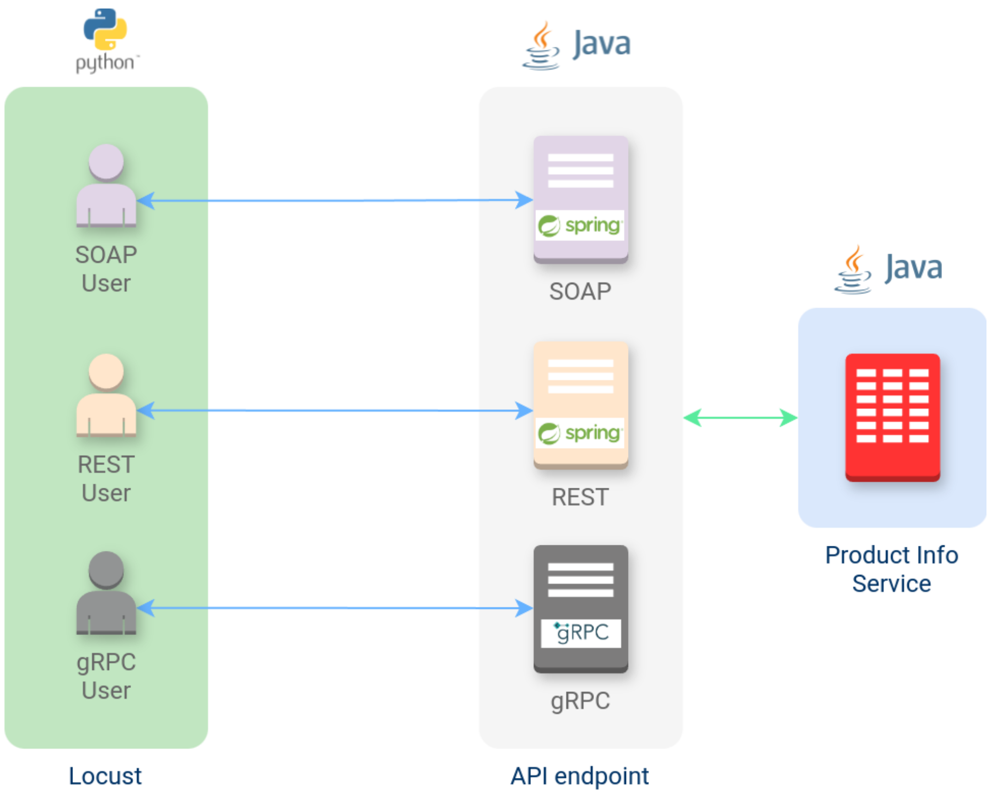


Figure 3.3: Test environment.

3.2.2.1 Hardware

The hardware utilized was a PC equipped with 32 GB DDR5 6000 MHz memory and an AMD Ryzen 7 7800X3D CPU, which provides eight cores and 16 threads.

3.2.2.2 Software

The device hosting all benchmarks ran Pop!_OS 22.04 LTS as the operating system. The Spring framework [34] was used to simplify the implementation of the microservices in Java. Locust [9] was used for benchmarking and runs on Python. Chapter 4 delves into more detail about the involved software.

3.3 Analysis and Evaluation

The core of this thesis's analysis lies in the examination of data collected from microservice benchmarking. To comprehensively assess the performance of SOAP, REST, and gRPC, the benchmarks were designed to capture key performance metrics: throughput, latency, and scalability. Each benchmark test yielded data on these metrics, which were then visualized in column charts for comparative analysis across all communication protocols.

Throughput analysis involved measuring the total requests completed within a fixed time frame of 120 seconds, across varying loads of 50, 100, and 200 concurrent virtual clients. The higher the number the better performance. By varying the shapes of the requests in size and server functionality, real-life scenarios were more accurately simulated. The variance in server functionality simulates diverse internal latencies on the server side, providing insight into how effectively the protocols manage data transformation. Simultaneously, latency measurements offered a glimpse into the efficiency of each protocol, quantifying the time taken in milliseconds for client requests to receive responses. Both the average and 95th percentile latency were measured to capture worst-case scenarios, thus offering a comprehensive understanding of responsiveness intervals.

Scalability evaluation delved into the examined limits for concurrent users of each service, offering valuable insights into their capacity to handle constantly increasing loads.

Comparing the performance of protocols in terms of throughput and latency with and without TLS implementation provides a more relevant evaluation for real-world applications. This is particularly significant as production environments typically necessitate secure communication, emphasizing the practical importance of such assessments.

Chapter 4

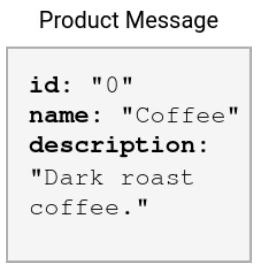
Implementation and Benchmarking

This chapter describes the implementation of each benchmarked microservice developed with SOAP, REST, and gRPC. The implementation specifics of these are presented in section 4.1. Additionally, the benchmark suite is examined closer in section 4.2. The source code for this project has been published at `https://github.com/markusnewtonh/api-bench`.

4.1 Microservices

For the purpose of conducting detailed performance analysis on the three communication protocols highlighted in this thesis - SOAP, REST, and gRPC - distinct microservices were developed for each protocol using Java and the Spring framework [34]. Leveraging Spring facilitated the seamless creation of self-contained microservices, thus streamlining the implementation process. Notably, Spring provides comprehensive user guides for both SOAP and REST [35, 36]. Developing a gRPC service did not require Spring. Java version 17 served as the foundation for all microservices.

As depicted in Figure 3.3 in Chapter 3, each microservice shared common functionality, termed as the `Product Info Service`. This service was encapsulated within a single Java file and accessed via distinct API interfaces tailored for each communication protocol. This approach aimed to minimize discrepancies among implementations, ensuring that the primary differentiation lay in the protocols themselves on the server side. The selection of a service delivering product information was deliberate, aimed at simulating real-world scenarios reflecting the data transmitted within each message, as inspired by [1]. The messages, comprising requests and responses, carried the information outlined in Figure 4.1.



Product Message

```
id: "0"
name: "Coffee"
description:
"Dark roast
coffee."
```

Figure 4.1: The product information structure of each message (requests/responses) during the benchmarking. This is an example, not data used in the actual tests.

The product info service exposed two functions:

- `echo()`: Takes a product message and returns a copy of the message to the sender.
- `modify()`: Takes a product message and returns a copy of the message containing a reversed product description.

These two functions were designed to emulate a basic request and response mechanism, alongside data transformation similar to trivial business logic, which led to varying internal latencies within the service.

4.2 Benchmark Suite

4.2.1 Benchmarks implementation

The primary tool utilized for benchmarking was the load-testing framework Locust, developed in Python, with Python version 3.10.12 employed. Locust facilitated the generation of concurrent virtual users, or clients, for each microservice. Specifically, three distinct virtual user types were implemented for Locust, corresponding to each protocol: a SOAP user, a REST user, and a gRPC user. These users were tasked with sending four different types of requests to the APIs developed with the respective protocols. Locust distributes the users evenly among its workers (threads) and sends as many requests as possible, ultimately limited by the efficiency of each protocol and the process scheduling done by the hosting operative system.

The request types varied in terms of the product data contained in each message, drawing from previous studies [8, 7] that explored communication protocol performance across different message sizes and formats. The following list explains the different shapes.

- **Small**: Sends a small request with a small product description of 16 characters. Calls the function `echo()`.
- **Typical**: Sends a request with a typical product description of 361 characters. Calls the function `modify()`.
- **Large**: Sends a request with a large product description of two million characters. Calls the function `echo()`.
- **Stress**: Sends a request with a large product description of two million characters. Calls the function `modify()`.

Given the distinct nature of each user, the implementation for each request type had to be tailored to suit the specifics of each protocol. For SOAP, the product object was generated using the SOAP-specific complex type element format. In the case of REST, the product object was transmitted as a JSON object. Conversely, for gRPC, the object was instantiated utilizing the object creation functionality provided by gRPC's official Java library. In the same manner, the respective remote procedure call was appropriately invoked for each protocol.

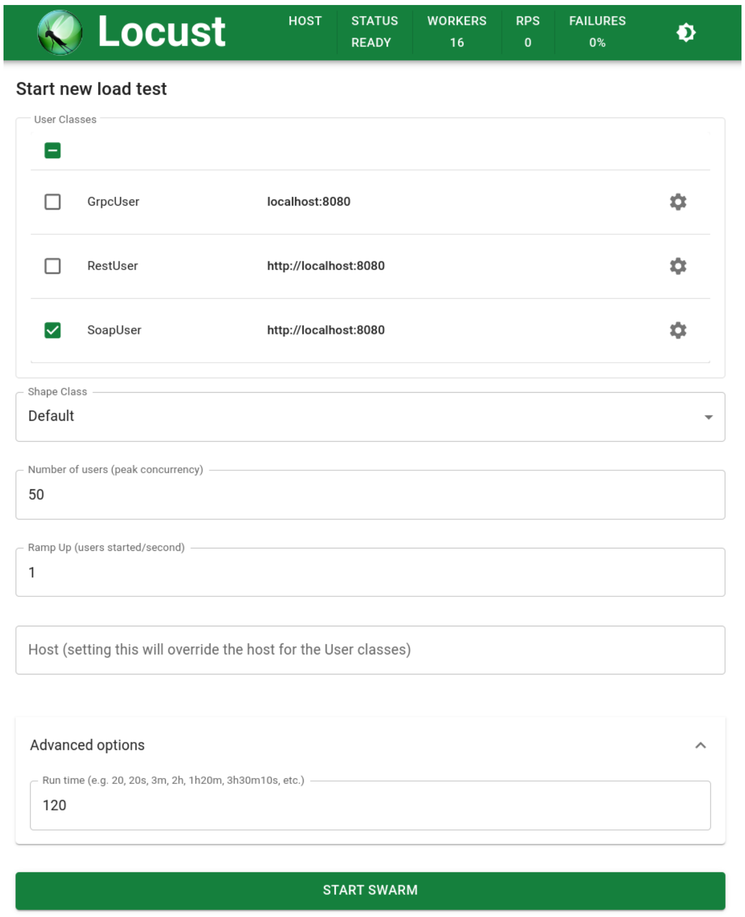
4.2.2 Benchmarks execution

For all experiments, Locust was initiated with 16 workers, one allocated to each CPU thread, to maximize system resource utilization. The benchmarking process comprised three sets of tests.

The initial benchmarks focused on evaluating the throughput and latency of each communication protocol. This involved sending one type of request (*small, typical, large, stress*) at a time, with 50, 100, and 200 virtual users over a duration of 120 seconds per test. These tests were conducted through Locust's web user interface, allowing interactive adjustment of user count and test duration. Figure 4.2 showcases the user interface. Following each test, data was extracted directly from Locust's data export functionality.

The subsequent tests, designated as a concurrency examination or *stress test* [37] and inspired by a previous study [38], were conducted individually for each microservice to assess scalability. In this setup, there was no predefined time limit. Instead, the test involved setting the user count to 10,000 and the ramp-up (the rate at which users were added per second) to 15, in contrast to 50 for the previous tests. The objective was to identify the threshold at which the service's capacity would be saturated, meaning when the rate of handled Requests Per Second (RPS) would plateau or when failures in request processing would begin to occur.

The final phase of testing involved configuring one-way TLS for both the microservice and clients across all protocols. First, TLS was implemented for each protocol. Subsequently, targeting each protocol at a time, throughput and latency were measured for 100 virtual users transmitting the *typical* message type over 120 seconds. These results were then compared to those obtained from the previous tests with identical parameters but without TLS enabled.



Start new load test

User Classes

☐	GrpcUser	localhost:8080
☐	RestUser	http://localhost:8080
✓	SoapUser	http://localhost:8080

Shape Class: Default

Number of users (peak concurrency): 50

Ramp Up (users started/second): 1

Host (setting this will override the host for the User classes)

Advanced options

Run time (e.g. 20, 20s, 3m, 2h, 1h20m, 3h30m10s, etc.): 120

START SWARM

Figure 4.2: The Locust web user interface for setting benchmark properties.

Chapter 5

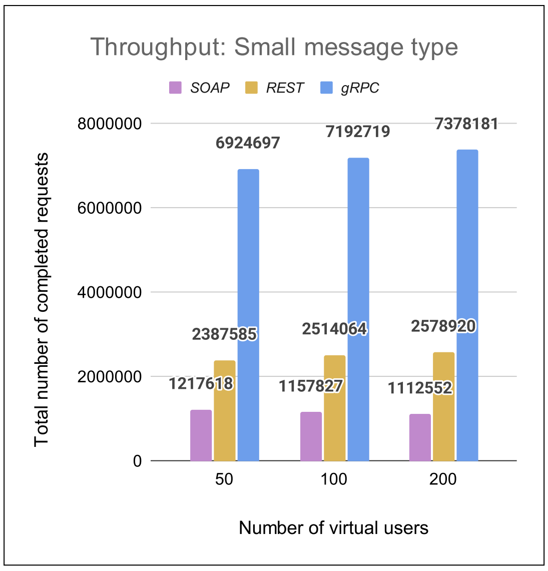
Results and Evaluation

In this chapter, the results of the benchmarking process are presented and analyzed. Section 5.1 presents the findings of the throughput benchmarks, Section 5.2 assesses the results of the latency tests, Section 5.3 presents the scalability evaluation, and Section 5.4 discusses the impact of TLS on performance. Finally, in Section 5.5 a comparison with related works and the limitations of this thesis are discussed.

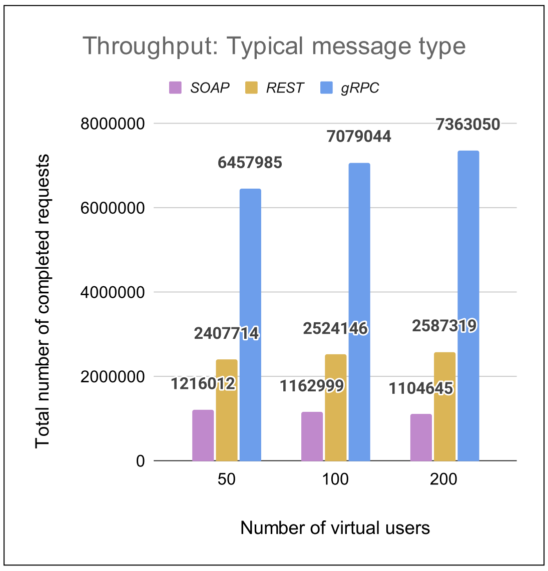
5.1 Throughput

The initial round of tests evaluated the request throughput of individual microservices across different communication protocols. The results of these benchmarks depict the total count of successful requests completed (response received) within a 120-second testing period for each message type (small, typical, large, stress). Refer to Figure 5.1 for a comprehensive representation of these findings, presented as column charts categorized by message type. Within each chart, the columns display the throughput performance of each communication protocol, further grouped based on the number of concurrent virtual users actively sending requests to the targeted microservice.

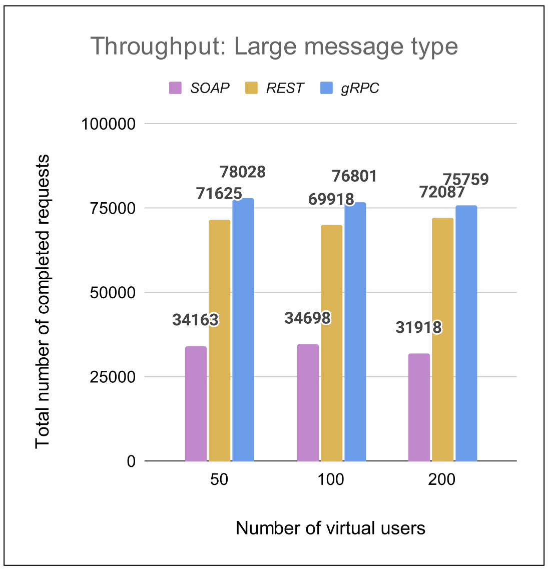
The throughput of a microservice stands as a crucial, perhaps the most vital, performance metric when assessing service efficiency. This metric evaluates both the efficiency of request transmission on the client side and the processing efficiency on the server side. Analysis of the results presented in Figure 5.1 unmistakably demonstrates that gRPC outperforms both SOAP and REST in terms of throughput. The difference in throughput for *small* and *typical* message types among all three communication protocols is notably similar. gRPC emerges as the leading performer, exhibiting an average



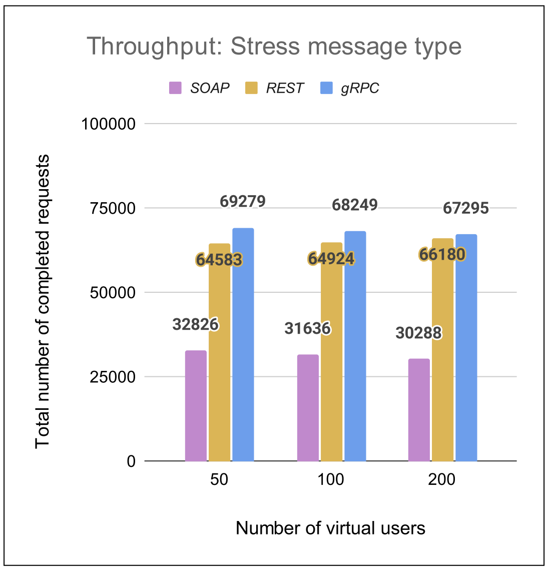
(a) Results of throughput benchmarks with *small* message type.



(b) Results of throughput benchmarks with *typical* message type.



(c) Results of throughput benchmarks with *large* message type.



(d) Results of throughput benchmarks with *stress* message type.

Figure 5.1: Throughput benchmark results. The total number of completed requests within 120 seconds across three different amounts of concurrent users. Each chart presents the results for one message type. The columns represent the performance of each protocol. The higher the number the better.

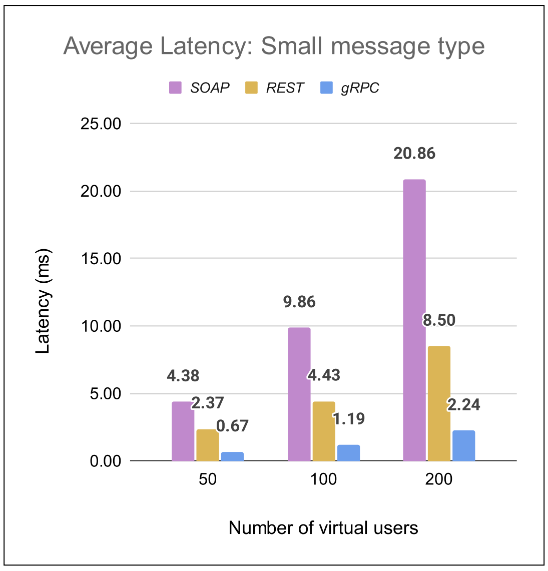
increase of **618%** to SOAP and **287%** to REST for small messages, and an average increase of **602%** to SOAP and **278%** to REST for typical messages. Despite the slight variation in size between *small* and *typical* messages and the difference in internal server latency, the results suggest that this disparity does not significantly impact throughput performance. This is evident from the marginal difference in throughput for the same protocol across both message types.

The message types *large* and *stress* exhibit no difference in size. However, there is a distinction in the server-side functionality invoked. The stress message type introduces emulated internal latency, which prolongs processing time and consequently contributes to the total latency. This effect can be observed in Figures 5.1c and 5.1d, resulting in only minor discrepancies between the throughput results for the two request types. Comparing the performance variation between the larger message types, *large* and *stress*, with the smaller types, *small* and *typical*, reveals a significant reduction in the performance advantage of the leading gRPC. When sending the *large* message type, gRPC surpasses SOAP with **229%** of its throughput and REST with **108%** of its throughput. Similarly, the *stress* type results in gRPC achieving **216%** of SOAP's throughput and **105%** of REST's throughput. These findings suggest that larger message sizes lead to diminished performance gains using gRPC compared to SOAP and REST. One possible explanation for this phenomenon is likely the serialization technique applied by gRPC, which inherently requires more time for larger messages but is tailored for the efficient transmission of smaller messages.

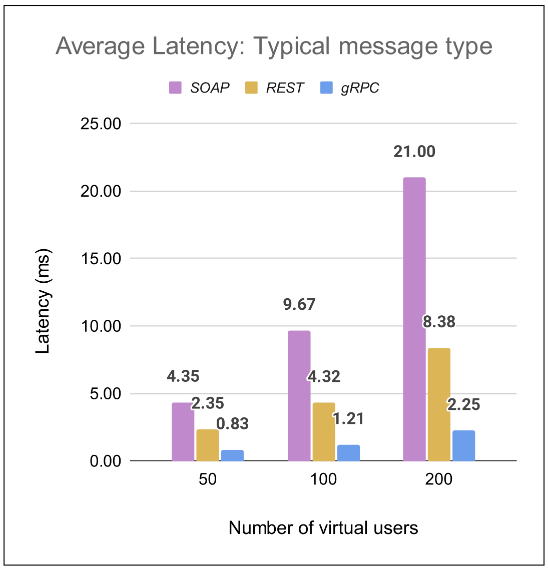
5.2 Latency

As the throughput was being assessed, latency data was simultaneously gathered for each client request. To provide a thorough understanding of latency, both average and 95th percentile latency were recorded. The 95th percentile latency represents the latency that 95% of all requests are completed within, providing insight into the worst-case delays for each protocol. Figure 5.2 illustrates the compiled results of average latency across all message types, while Figure 5.3 displays the 95th percentile latency for each message type.

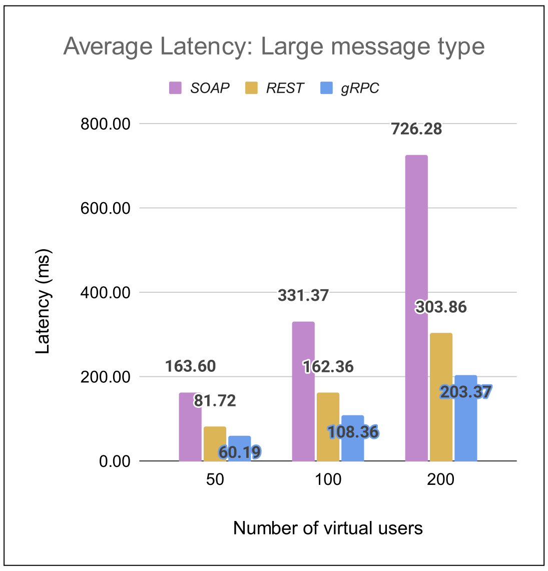
The latency results closely align with the throughput outcomes. Once again, gRPC surpasses both SOAP and REST, as anticipated given the interconnectedness of throughput and latency. A service's ability to handle requests directly impacts the number of requests it can manage within a



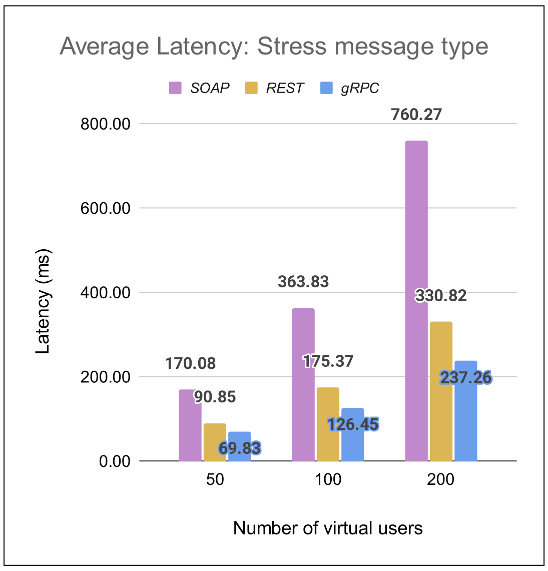
(a) Average latency of *small* message type.



(b) Average latency of *typical* message type.

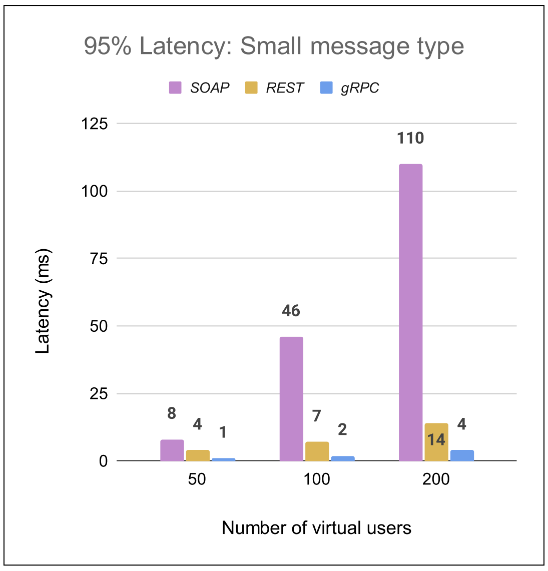


(c) Average latency of *large* message type.

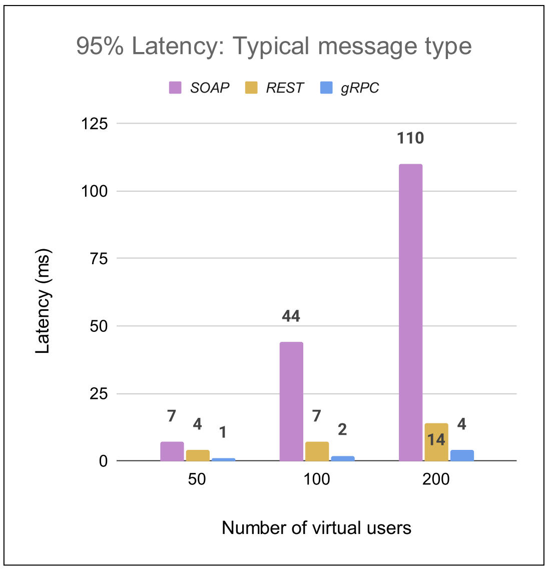


(d) Average latency of *stress* message type.

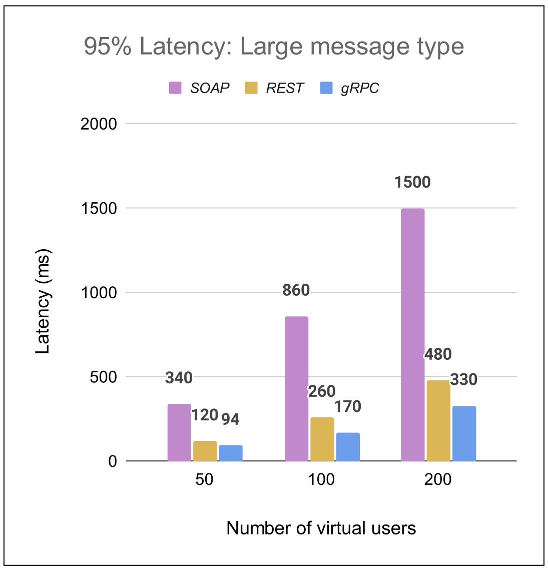
Figure 5.2: Average latency results. The average time in milliseconds for a data transfer to be completed across three different amounts of concurrent users. Each chart presents the results for one message type. Each chart presents the results for one message type. The columns represent the performance of each protocol. The lower the number the better.



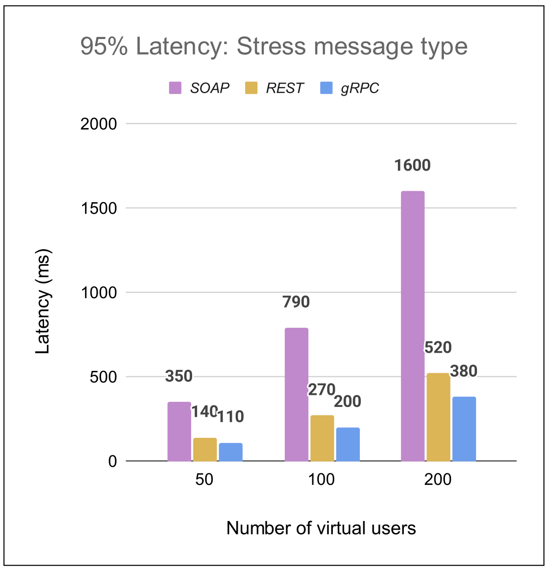
(a) 95th percentile latency of *small* message type.



(b) 95th percentile latency of *typical* message type.



(c) 95th percentile latency of *large* message type.



(d) 95th percentile latency of *stress* message type.

Figure 5.3: 95th percentile latency results. The time in milliseconds for a data transfer to be completed that 95% of the transfers fall below across three different amounts of concurrent users. The columns represent the performance of each protocol. The lower the number the better.

specified timeframe. However, the performance gap in terms of latency among the protocols exceeds the difference in throughput. As depicted in Figure 5.3a, gRPC demonstrates notably lower average latency across varying numbers of clients, amounting to **13%** of SOAP's average latency and **27%** of REST's average latency for small messages. Similarly, for typical messages, gRPC exhibits average latency at **14%** of SOAP's and **30%** of REST's. However, for larger messages, the performance advantage diminishes yet again. The average latency for gRPC, using the *large* message type across different user numbers, averages at **33%** of SOAP's average latency and **69%** of REST's average latency. For the *stress* message type, the corresponding figures are **36%** and **74%**.

In terms of the 95th percentile latency, SOAP demonstrated the poorest performance compared to gRPC, with an average worst-case latency **19.5** times that of gRPC's measurement for small messages. gRPC consistently outperformed the other protocols across all latency measurements which the charts clearly illustrate in Figure 5.3. This outcome was anticipated, partly due to gRPC's use of protocol buffers, which generally result in faster message processing compared to SOAP's XML and REST's JSON. Additionally, gRPC benefits from employing the more lightweight HTTP/2, which its counterparts do not utilize.

These results demonstrate that gRPC is significantly faster than the traditional protocols, regardless of message shape. This aligns well with the hypothesis for this thesis and highlights the performance advantage gained by the underlying technologies used by gRPC.

5.3 Scalability

The scalability benchmarks aimed to determine the saturation point of concurrent clients for each microservice. Specifically, this entailed identifying the point at which the maximum RPS were achieved, signifying that latency continued to rise while RPS remained stagnant.

For the SOAP-implemented microservice, the saturation point was reached at **210 concurrent clients**, measuring **8,905.1 RPS**. This can be seen in Figure 5.4's top and bottom charts, which were exported from Locust. The gray dotted vertical line that runs through both charts indicates where the saturation point has been reached. The saturation point for the REST microservice was reached at **360 concurrent clients**, measuring **20,032.3 RPS** as seen in Figure 5.5.

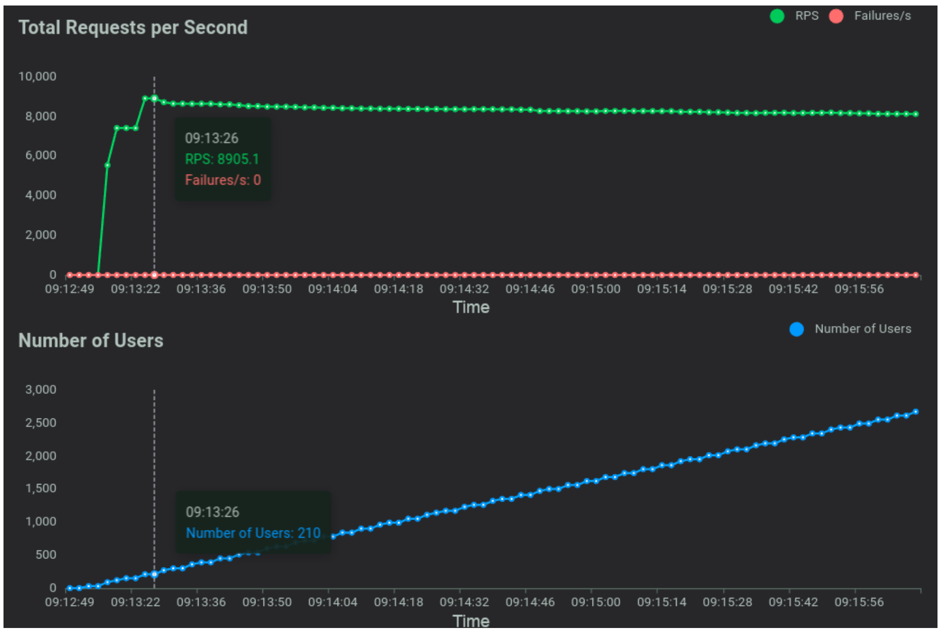


Figure 5.4: Scalability test results for the SOAP microservice. The gray dotted vertical line highlights the saturation point.

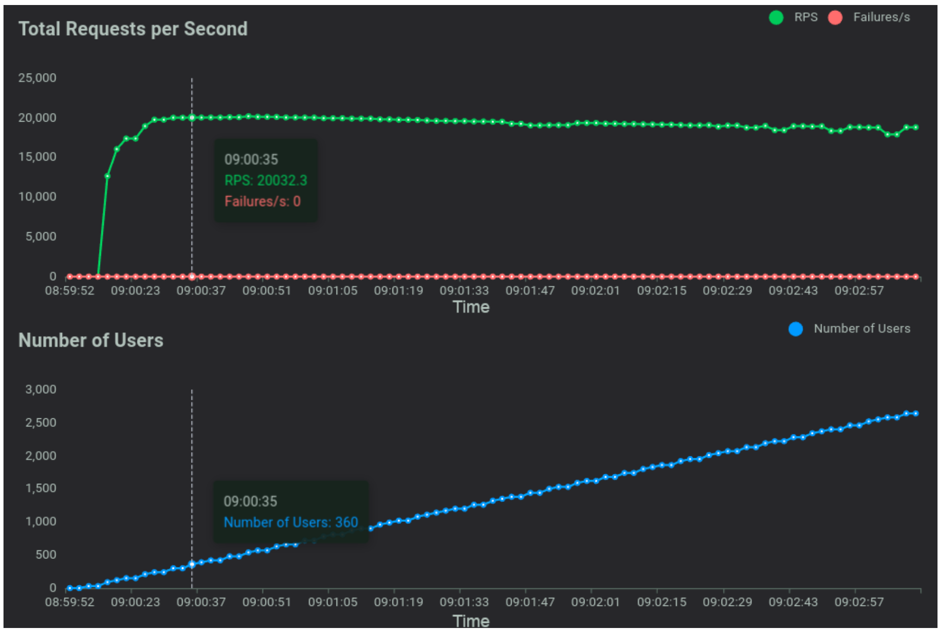


Figure 5.5: Scalability test results for the REST microservice. The gray dotted vertical line highlights the saturation point.

The gRPC microservice's saturation point was reached at **600 concurrent clients**, measuring **69,614.3 RPS**. This can be seen in Figure 5.6.

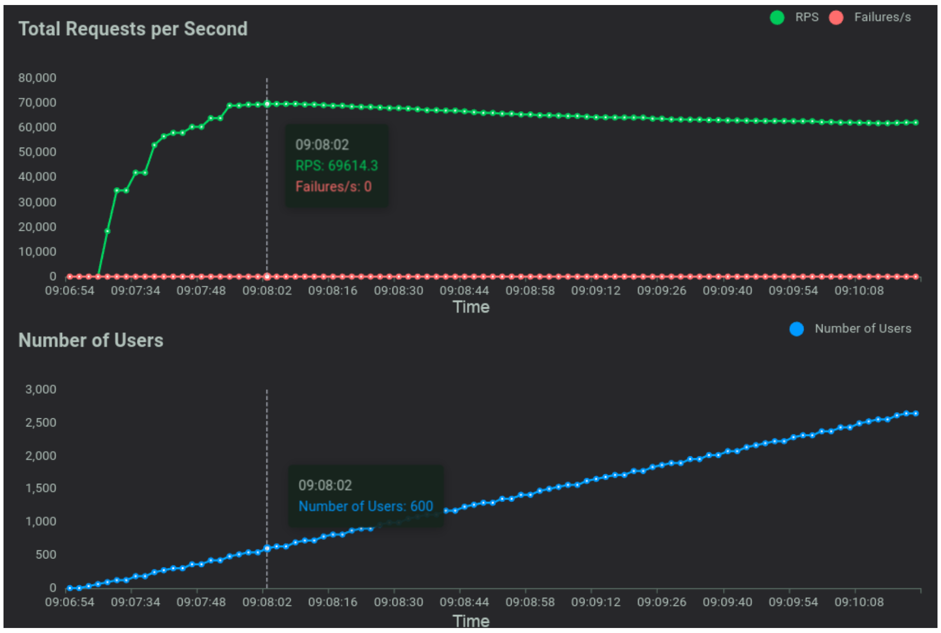


Figure 5.6: Scalability test results for the gRPC microservice. The gray dotted vertical line highlights the saturation point.

The benchmarking, which assessed throughput and latency, was conducted with varying numbers of concurrent clients, or virtual users, for each service. This, per se, illustrates the scalable performance of each communication protocol regarding its capacity to accommodate an uptick in user requests. However, these tests do not reveal the saturation point of the microservices, which is why a conclusive test incorporating a continuously escalating number of virtual users for scalability was integrated into the benchmarking suite. Table 5.1 presents the performance metrics in terms of maximum RPS and the corresponding concurrent user count for each protocol.

The saturation point marks the stage where a service's RPS ceases to rise with an increasing number of virtual users. At this juncture, the service reaches its capacity to handle parallel requests, leading to a rise in response time and subsequently, an increase in latency as user count escalates. This observation highlights that a service with a scaling RPS, which sustains higher throughput, possesses the capability to accommodate greater traffic, rendering it more scalable compared to a service with a lower user threshold and/or maximum

Table 5.1: Listing of scalability performance across the communication protocols.

Scalability Saturation Point

Protocol	**RPS**	**User Count**
SOAP	8,905.1	210
REST	20,032.3	360
gRPC	**69,614.3**	**600**

RPS.

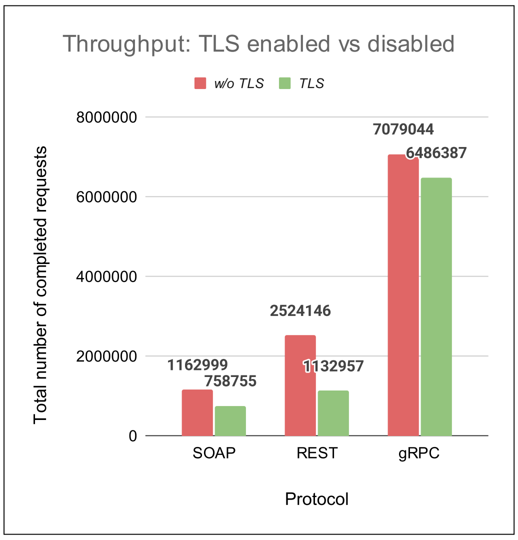
The maximum RPS attained by gRPC peaks at **69,614.3** with **600 active users**, as illustrated in Figure 5.6 and Table 5.1. This demonstrates unparalleled performance when contrasted with both SOAP and REST, which reach their limits at **210** and **360 active users** respectively. While it is essential to acknowledge the potential impact of other processes on the test machine during benchmarking, the consistent and substantial performance advantage observed with gRPC across all tests suggests superior scalability compared to the other two protocols.

These findings underscore the superior efficiency of a service implemented with gRPC. In the context of a production environment, the results imply substantial performance gains associated with adopting gRPC. For instance, an API exposed through gRPC could manage **3.48** times the RPS compared to its closest rival, REST. This aspect becomes particularly noteworthy when the service is anticipated to encounter significant and recurrent traffic loads.

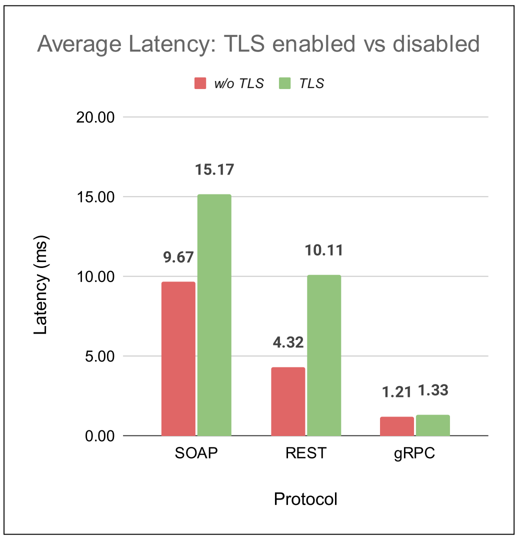
5.4 Security Overhead

The last set of tests were designed to assess the impact of TLS on throughput and latency performance, i.e. investigating the security overhead performance trade-off. In the experiments, 100 virtual users were spawned that sent the *typical* message type over a period of 120 seconds, with and without TLS enabled, focusing on one protocol at a time. The results of these tests are depicted in Figure 5.7, comparing the throughput, average latency, and 95th percentile latency between configurations with and without TLS across all protocols.

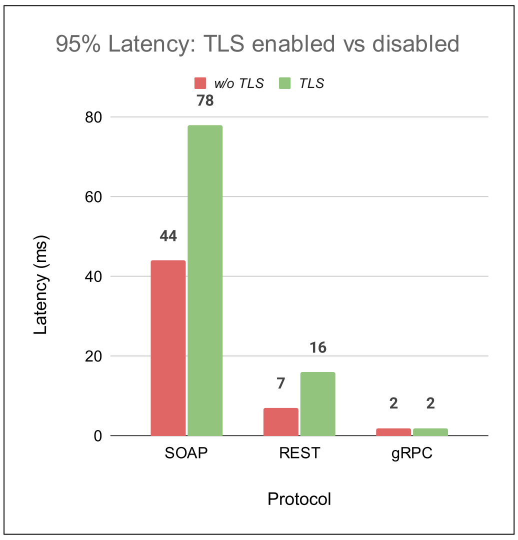
Examining Figure 5.7a to evaluate the impact of TLS on throughput, it is evident that all protocols experienced a negative effect due to the security configuration. Surprisingly, REST suffered the most from this, contrary to earlier expectations, surpassing SOAP. The reduction in completed requests for REST was substantial, with a **55%** decline in performance, while SOAP experienced a **35%** decrease. In comparison, gRPC observed a lower **8%** drop in throughput.



(a) Throughput results for each protocol with and without TLS enabled. A higher value is better.



(b) Average latency results for each protocol with and without TLS enabled. A lower value is better.



(c) 95th percentile latency results for each protocol with and without TLS enabled. A lower value is better.

Figure 5.7: The performance results of the benchmarks with TLS enabled compared to the results of the benchmarks without TLS. The benchmarks were run with 100 virtual users continuously sending *typical* requests over 120 seconds.

Despite SOAP exhibiting a lesser drop in performance compared to REST with TLS, its inherent lower throughput still makes it less appealing in terms of performance. Ultimately, gRPC emerges as the top performer, excelling in both security overhead introduced by TLS and actual throughput.

The average latency findings illustrated in Figure 5.7b offer insights into the underlying reasons for the throughput patterns observed. It is evident that TLS significantly impacts REST's latency, more than doubling its round-trip time. This correlation corroborates the throughput data, highlighting REST's struggle with security overhead compared to its counterparts. Meanwhile, gRPC maintains its superior performance, exhibiting only a marginal variance in latency with TLS integration.

Examining the 95th percentile latency results depicted in Figure 5.7c, TLS appears to have minimal influence on gRPC's worst-case delays, whereas REST encounters a substantial increase in worst-case latency—**229%** higher compared to its implementation without the security layer.

These findings hold particular relevance for developers in real-world applications due to the crucial role of security. Given the standard practice of integrating a security layer into API communications, it is important to account for the associated overhead when selecting a communication protocol based on performance. TLS stands out as one of the most prevalent security layers, underscoring its relevance in this study. The results demonstrating its impact on performance highlight the influence of security overhead on both throughput and latency. Based on the outcomes of this thesis's benchmarks, gRPC emerges as the optimal choice for a communication protocol when opting to secure data transfers with TLS. Given that TLS encrypts all transmitted data and gRPC generates smaller messages through its use of protocol buffers, it is understandable that gRPC's performance is less adversely affected compared to SOAP and REST, which require encrypting larger volumes of data.

5.5 Discussion

gRPC stands out as the top performer in all experiments conducted in this degree project. In the local test environment used, neither SOAP nor REST proved viable alternatives given the results. However, REST remains widely used in production today, and migrating to gRPC is not as straightforward as the findings of this project might suggest. As mentioned in [21], there are two main disadvantages of gRPC compared to REST. Firstly, gRPC is often considered too strongly typed in front-end applications, limiting flexibility for external parties. REST's JSON is more flexible, being text-based and easily parsed. To address this, gRPC is often implemented with a reverse proxy that transcodes HTTP JSON calls to gRPC calls, though this introduces additional complexity when migrating from REST to gRPC.

Secondly, another disadvantage of gRPC is the risk of breaking previously working communication when updating service definitions in a proto file. This necessitates regenerating code for involved parties when the proto file is altered. Although changes do not usually affect existing communication, significant changes require code regeneration to ensure proper service communication. This issue is less prevalent with REST services, as the endpoints of communicating services are not as tightly coupled to a shared service definition.

For back-end inter-process communication, gRPC is a strong choice, as supported by the results of this degree project. In this context, messages do not need to be human-readable, and the implementation of communication microservices can be more tightly coupled without affecting external parties. Many large companies, such as Docker, Netflix, and Cisco have adopted gRPC as their internal system's inter-process protocol [21]. However, the experiments in this thesis show that larger data exchanges benefit less from gRPC compared to REST than smaller data transmissions do. This is also important to consider when planning a migration.

5.5.1 Comparison with Related Work

To contextualize the findings of this thesis, comparisons with the aforementioned relevant prior research are drawn. For instance, R. Lagerström et al. conducted a performance efficiency study comparing gRPC with REST, revealing that gRPC exhibited superior throughput, akin to the results of this investigation [1]. However, their study delved into a slightly more intricate system. Notably, the disparity in throughput performance was less pronounced compared to the findings of this project, with REST even outperforming gRPC under certain conditions, particularly when the user load exceeded 200. This stands in stark contrast to this research, where gRPC consistently outperformed REST by a considerable margin across all test scenarios. Various factors could contribute to this discrepancy, such as differences in the underlying systems and hardware configurations. A noteworthy distinction regarding the systems pertains to the hosting platform of the microservices. Lagerström's research involved hosting the microservices in a cluster environment, whereas in this study, the microservices were hosted on the same machine as the benchmarking software. This is a viable reason for the performance discrepancies.

In the comparative analysis of internet communication protocols conducted by L. Kamiński et al. [2], there were also notable differences in the benchmark suite. Nevertheless, the assessment of latency bore resemblances to the tests conducted in this thesis, particularly in terms of system design, wherein both studies involved hosting clients and servers on the same machine. Kamiński et al.'s evaluation of latency for Create, Read, Update, and Delete (CRUD) requests between client and server concluded that gRPC exhibited the highest speed among various protocols, including REST and several other protocols, aligning closely with the findings presented in this thesis. Furthermore, it is worth noting that the original intention of L. Kamiński's study was to incorporate SOAP into the benchmarking process. However, due to encountered limitations with Python, this protocol was ultimately excluded. In contrast, this thesis addressed this gap by including and effectively implementing SOAP with Python (for Locust), thereby introducing a novel dimension to the previous research. This expansion not only enhances the comprehensiveness of this thesis's analysis but also underscores the commitment to addressing and overcoming challenges encountered in prior studies.

5.5.2 Method and Limitations

Exploring the performance of communication protocols entails consideration of various parameters, including system and service design, hardware configuration, programming language selection, benchmarking suite, and more. Initially, the system design in this thesis was relatively simplistic, with each protocol represented by a single microservice, as opposed to more intricate protocols involving multiple services. While this simplicity may limit the direct applicability to real-world scenarios, it also reduces potential internal or external latencies and network errors between services, thereby focusing the investigation on the protocols themselves. Additionally, running the benchmark tool and microservices on the same machine, despite contesting resources, eliminates potential errors and overhead associated with hosting clusters or multiple machines.

However, regarding the benchmarking tool, Locust utilizes different libraries for performing the transmission of requests for each protocol, which could very well introduce discrepancies in requests being sent per second depending on the underlying implementation.

Moreover, employing the same programming language, Java, for implementing the microservices facilitated the isolation of differences between protocols, avoiding the complexities introduced by diverse programming languages.

Furthermore, the scalability test design, which primarily focused on increasing the number of virtual users, enabled a straightforward comparison of protocol differences, aligning well with the scope of this study. However, it is important to note that scalability typically encompasses the ability to adapt and scale in response to increased workload on the server side. In this thesis, the microservices did not dynamically adjust the number of parallel services based on workload or distribute the load among hosting machines in a network.

Another limitation of this study is the single choice of security layer implemented to investigate the security overhead performance trade-off. There are various options for securing API communication. However, TLS is an industry standard making it appropriate to include in this thesis. The decision to use a single security protocol has its basis in delimiting the scope of the thesis.

Finally, regarding message and request design, this study encompassed a range of sizes and formats. However, there remains room for further diversification in the types of requests to uncover potential discrepancies or limitations between the protocols. Expanding the variety of requests could provide deeper insights into how each protocol handles different scenarios and workload types, enhancing the robustness of the analysis.

Chapter 6

Conclusions and Future work

In this chapter, Section 6.1 presents the concluding remarks on the thesis's work. Section 6.2 provides insights into potential future avenues of exploration. Lastly, Section 6.3 includes reflective thoughts to conclude this thesis.

6.1 Conclusions

This thesis aimed to address the research question of how the performance of gRPC concerning throughput, latency, and scalability within a local environment compares to the traditional communication protocols SOAP and REST. Through benchmark tests conducted on identical microservices implementing APIs of the three protocols with simulated clients sending various types of requests, the findings demonstrate that gRPC surpasses its counterparts across all metrics. Among the three, SOAP exhibited the poorest performance, while REST approached the performance level of gRPC particularly for larger messages. Given that gRPC's performance advantage decreases for larger messages compared to the two other protocols, developers should consider this when planning to migrate to gRPC if their services involve exchanging large data.

This thesis also contributes two new types of benchmarks exploring areas missing in previous research. The first type of test involved comparing the saturation point in terms of scalability for microservices of all protocols. This was performed by stress testing each protocol's service to find its limit of concurrent virtual clients and requests handled per second. The results of these tests show that gRPC could handle 166% more concurrent users and complete 348% more requests per second than its closest competitor REST.

Moreover, the security overhead introduced by TLS on these protocols was also investigated. Experiments evaluating TLS's impact on performance were conducted and demonstrated that gRPC is affected least by the security layer, making it the best performer even when TLS is implemented. Unexpectedly, REST's performance was hindered most by TLS configuration of all three protocols, taking SOAP's role as the expected bottom performer.

Although the benchmark suite could be expanded and diversified, the experiments executed in this study yield empirical insights for researchers and developers seeking guidance in selecting the most suitable API communication protocol for their specific microservice applications.

6.2 Future work

To expand on the research conducted in this thesis, potential future steps could involve deploying a similar system on a cloud platform or a cluster to gain deeper insights into the performance of communication protocols in a typical distributed environment. Furthermore, enhancing the complexity of the system design for similar benchmarks in subsequent studies could lead to more accurate measurements that better reflect real-world business scenarios. Given the contemporary relevance of transitioning from traditional formats like REST, further investigation into effective migration strategies or the integration of previous protocols with gRPC presents another avenue for exploration. This topic has been addressed in a blog post by the gRPC community [39]. Engaging with this community to solicit additional insights and recommendations for improving setup configurations would provide an opportunity to delve deeper into the subject matter with cutting-edge implementations. Furthermore, future work could investigate alternative security protocols to TLS, evaluating their impact on performance.

6.3 Reflection

The objective of this thesis is to offer guidance to researchers and developers in selecting an appropriate communication protocol tailored to their specific requirements. Whether applied in distributed system research or large-scale software application development, this thesis provides insights into the performance of three prevalent protocols. Regarding the ethical and sustainability implications of this research, I reckon that the overall impact is limited. Nonetheless, the findings reveal that gRPC demonstrates superior efficiency compared to its counterparts on identical local machines. It can be argued that this efficiency suggests gRPC may also be more resource-efficient, potentially leading to more sustainable energy utilization in large-scale deployments. Consequently, gRPC emerges as a more sustainable alternative in both environmental and economic contexts when compared to traditional protocols. However, it is important to note that detailed measurements of CPU usage and power consumption fall outside the scope of this thesis, precluding definitive conclusions in this regard.

In terms of sustainability, this thesis contributes to several of the United Nations (UN) Sustainable Development Goals (SDGs), specifically numbers 9, 12, and 13. The investigation of communication protocols' efficiency supports the development of sustainable infrastructure by identifying efficient protocols that could lead to reduced energy consumption in the industry. More efficient services reduce server load, resulting in less power needed by data centers, which ultimately leads to more sustainable production, contributing to SDG 9 and 12. Additionally, reduced power consumption means less greenhouse gas emissions from data centers, aligning with the climate action goals of SDG 13.

References

[1] B. Shafabakhsh, R. Lagerström, and S. Hacks, “Evaluating the Impact of Inter Process Communication in Microservice Architectures.” in *Proceedings of the 8th International Workshop on Quantitative Approaches to Software Quality co-located with 27th Asia-Pacific Software Engineering Conference (APSEC 2020), Singapore (virtual), December 1, 2020*, ser. CEUR Workshop Proceedings, vol. 2767. CEUR-WS.org, 2020, pp. 55–63. [Online]. Available: https://ceur-ws.org/Vol-2767/07-QuASoQ-2020.pdf [Pages 1, 5, 20, 22, 24, 29, and 43.]

[2] L. Kamiński, M. Kozłowski, D. Sporysz, K. Wolska, P. Zaniewski, and R. Roszczyk, “Comparative Review of Selected Internet Communication Protocols,” *Foundations of Computing and Decision Sciences*, vol. 48, no. 1, Mar. 2023, pp. 39–56, Mar. 2023. doi: 10.2478/fcds-2023-0003 [Pages 1, 20, 24, and 44.]

[3] I. Grigorik and Surma, “Introduction to HTTP/2,” 2024. [Online]. Available: https:////web.dev/articles/performance-http2 [Accessed: 2024-05-28] [Page 2.]

[4] S. G. Du, J. W. Lee, and K. Kim, “Proposal of GRPC as a New Northbound API for Application Layer Communication Efficiency in SDN,” in *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication*, ser. IMCOM ’18. Association for Computing Machinery, 2018. doi: 10.1145/3164541.3164563 pp. 1–6. [Page 2.]

[5] J. Gough, *Mastering API Architecture: design, operate, and evolve API-based systems*, first edition ed. Sebastopol, CA: Oreilly, 2023. ISBN 9781492090632 [Pages 3 and 7.]

[6] S. Gadge and V. Kotwani, “Microservice architecture: API gateway considerations,” GlobalLogic Organisations, Tech. Rep., 2017. [Page 3.]

[7] M. Johansson and O. Isabella, “Comparative Study of REST and gRPC for Microservices in Established Software Architectures,” Bachelor’s thesis, Linköping University, Department of Computer and Information Science, 2023.

[Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-195563 [Pages 5, 21, and 30.]

[8] M. Bolanowski, K. Zak, A. Paszkiewicz, M. Ganzha, M. Paprzycki, P. Sowinski, I. Lacalle, and C. E. Palau, "Eficiency of REST and gRPC Realizing Communication Tasks in Microservice-Based Ecosystems," in *New Trends in Intelligent Software Methodologies, Tools and Techniques - Proceedings of the 21st International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques, SoMeT 2022, Kitakyushu, Japan, 20-22 September, 2022*, ser. Frontiers in Artificial Intelligence and Applications, vol. 355. IOS Press, 2022. doi: 10.3233/FAIA220242 pp. 97–108. [Pages 5, 24, and 30.]

[9] "What is Locust? - Locust 2.26.0 documentation," 2024. [Online]. Available: https://docs.locust.io/en/stable/what-is-locust.html [Accessed: 2024-05-28] [Pages 5, 19, and 27.]

[10] [x]cube LABS, "Microservices Architecture: Implementing Communication Patterns and Protocols," 2023. [Online]. Available: https://www.xcubelabs.com/blog/microservices-architecture-implementing-communication-patterns-and-protocols/ [Accessed: 2024-05-28] [Page 8.]

[11] A. S§kh, "Web Application Architecture: The Latest Trends and Best Practices for 2023," 2023. [Online]. Available: https://www.peerbits.com/blog/web-application-architecture.html [Accessed: 2024-05-28] [Page 8.]

[12] S. Joshi, "API architecture: A guide to 4 major API types: Open, internal, partner, and composite," 2024. [Online]. Available: https://sendbird.com/developer/tutorials/introduction-to-apis-types-of-apis-api-architecture-and-beyond [Accessed: 2024-05-28] [Page 8.]

[13] A. Doan, A. Halevy, and Z. G. Ives, *Principles of data integration.* Waltham, MA: Morgan Kaufmann, 2012. ISBN 978-0-12-416044-6 [Page 9.]

[14] B. Reselman, "An architect's guide to APIs: SOAP, REST, GraphQL, and gRPC," 2023. [Online]. Available: https://www.redhat.com/architect/apis-soap-rest-graphql-grpc [Accessed: 2024-05-28] [Pages 9 and 11.]

[15] J. Snell, P. Kulchenko, and D. Tidwell, *Programming Web services with SOAP*, first edition. ed. Sebastopol, California: O'Reilly Associates, 2002. ISBN 0-596-55201-7 [Pages 9, 10, and 11.]

[16] W3C, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," 2018. [Online]. Available: https://www.w3.org/TR/xml/#sec-intro [Accessed: 2024-05-28] [Page 10.]

[17] F. Halili and E. Ramadani, "Web Services: A Comparison of Soap and Rest Services," vol. 12, no. 3. CCSE, 2018. doi: https://doi.org/10.5539/mas.v12n3p175 p. p175. [Pages 11 and 12.]

[18] F. Halili, M. K. Halili, and I. Ninka, "Evaluation and Comparison of Styles of Using Web Services," in *2014 Sixth International Conference on Computational Intelligence, Communication Systems and Networks*. IEEE, 2014. doi: 10.1109/CICSyN.2014.38 pp. 139–144. [Page 11.]

[19] E. Wilde and C. Pautasso, Eds., *REST: from research to practice*. New York: Springer, 2011. ISBN 9781441983022 [Page 14.]

[20] gRPC Authors, "Introduction to gRPC," 2024. [Online]. Available: https://grpc.io/docs/what-is-grpc/introduction [Accessed: 2024-05-28] [Page 14.]

[21] K. Indrasiri and D. Kuruppu, *gRPC : up and running : building cloud native applications with Go and Java for Docker and Kubernetes*, first edition. ed. Beijing: O'Reilly, 2020. ISBN 1-4920-5828-9 [Pages ix, 15, 16, 17, 42, and 43.]

[22] B. Reselman, "HTTP/2: 5 things every Enterprise Architect needs to know," 2021. [Online]. Available: https://www.redhat.com/architect/http2 [Accessed: 2024-05-28] [Page 16.]

[23] Tech School, "HTTP/2 - The secret weapon of gRPC," 2021. [Online]. Available: https://dev.to/techschoolguru/http-2-the-secret-weapon-of-grpc-32dk [Accessed: 2024-05-28] [Pages ix and 16.]

[24] M. P. . P. Team, *NET application architecture guide*, 2nd ed., ser. Patterns & practices. Redmond, Wash.: Microsoft Press, 2009. ISBN 9780735627109 [Page 17.]

[25] Digital Samba, "Exploring the Importance of Network Performance Metrics," 2024. [Online]. Available: https://www.digitalsamba.com/blog/a-deep-dive-into-the-network-performance-metrics [Accessed: 2024-05-28] [Pages xi and 18.]

[26] S. N. Mohan M S, "Scale and Load Testing of Micro-Service," *International Research Journal of Engineering and Technology (IRJET)*, vol. 09, Jul. 2022, pp. 2629–2632, Jul. 2022. [Page 19.]

[27] P. Siriwardena, *Advanced API Security OAuth 2.0 and Beyond*, 2nd ed., ser. Books for professionals by professionals. Berkeley, CA: Apress, 2020. ISBN 1-4842-2050-1 [Page 19.]

[28] F. Belqasmi, J. Singh, S. Y. Bani Melhem, and R. H. Glitho, "SOAP-Based vs. RESTful Web Services: A Case Study for Multimedia Conferencing,"

IEEE Internet Computing, vol. 16, no. 4, Jul. 2012, pp. 54–63, Jul. 2012. doi: 10.1109/MIC.2012.62 [Page 20.]

[29] X. Liu, Y.-J. Hsieh, R. Chen, and S.-M. Yuan, "Distributed Testing System for Web Service Based on Crowdsourcing," *Complexity*, vol. 2018. Hindawi, Nov. 2018, Nov. 2018. doi: 10.1155/2018/2170585 [Pages 21 and 22.]

[30] P. K. Potti, "On the Design of Web Services: SOAP vs. REST," Master's thesis, University of North Florida, 2011. [Online]. Available: https://digitalcommons.unf.edu/etd/138 [Page 21.]

[31] J. Berg and D. Mebrahtu Redi, "Benchmarking the request throughput of conventional API calls and gRPC : A Comparative Study of REST and gRPC," Bachelor's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2023. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-334990 [Pages 21 and 22.]

[32] Apache Software Foundation, "Apache JMeter - Apache JMeter™," 2024. [Online]. Available: https://jmeter.apache.org [Accessed: 2024-05-28] [Page 22.]

[33] D. Goyal and K. Lane, *API analytics for product managers : understand key API metrics that can help you grow your business*, 1st ed. Birmingham, England: Packt Publishing, 2023. ISBN 1-80324-196-9 [Page 24.]

[34] Spring, "Spring," 2024. [Online]. Available: https://spring.io/ [Accessed: 2024-05-28] [Pages 27 and 28.]

[35] "Getting Started | Producing a SOAP web service," 2024. [Online]. Available: https://spring.io/guides/gs/producing-web-service [Accessed: 2024-05-28] [Page 28.]

[36] "Getting Started | Building REST services with Spring," Apr. 2024. [Online]. Available: https://spring.io/guides/tutorials/rest [Accessed: 2024-05-28] [Page 28.]

[37] S. Pargaonkar, "A Comprehensive Review of Performance Testing Methodologies and Best Practices: Software Quality Engineering," *International Journal of Science and Research (IJSR)*, vol. 12, no. 8. International Journal of Science and Research, Nov. 2023, pp. 2008–2014, Nov. 2023. doi: 10.21275/SR23822111402 [Page 30.]

[38] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," 2019. [Online]. Available: http://arxiv.org/abs/1905.07997 [Accessed: 2024-05-28] [Page 30.]

[39] “Can gRPC replace REST and WebSockets for Web Application Communication?” 2023. [Online]. Available: https://grpc.io/blog/postman-grpcweb/ [Accessed: 2024-05-28] [Page 47.]

TRITA-EECS-EX-2024:474
Stockholm, Sweden 2024

www.kth.se