



Evaluating the performance and usability of HTTP vs gRPC in communication between microservices

Najem Hamo
Simon Saberian

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Software Engineering. The thesis is equivalent to Weeks weeks of full-time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Authors:

Najem Hamo

E-mail: nahm20@student.bth.se

Simon Saberian

E-mail: sisb20@student.bth.se

University advisor:

Binish Tanveer

Department of Software Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Microservices is an architectural technique that has only gotten more popular as the need for scalable and performant internet-based applications has grown. One of the characteristics of microservices is communication through lightweight protocols like HTTP. These protocols are usually provided through frameworks that enable an abstracted form of communication and when implementing services using the Go language, the most common frameworks are gRPC and *net/http*. The aim of this thesis is to evaluate and compare the performance and usability of gRPC and HTTP frameworks in order to determine which one is better suited for microservices so that developers can be empowered to be more informed when making choices about their technology. We investigated the performance and usability by conducting two experiments. For the first one, we created two services that were implemented as identically as possible using Go but one communicated using the *net/http* framework and the other using gRPC. The services implemented methods that return small, medium, and large payload sizes and were then load-tested at varying numbers of virtual users. The second experiment was conducted by recruiting a set of participants that were tasked with completing two sets of coding tasks once using gRPC and once using HTTP. After the tasks were completed they were asked to fill out a questionnaire to measure their experience using the frameworks, the answers were then turned into a score which we could use to analyze the frameworks. The results from the performance experiment indicated that gRPC performed better in terms of throughput and latency, while HTTP performed better in scalability, and the results from the usability experiment indicated that HTTP was found to be more usable by the participants.

Keywords: HTTP, gRPC, Cloud Computing, Load testing, Micro-services

Acknowledgments

We would like to express our deepest appreciation to our advisor, Binish Tanveer, for her unwavering support, guidance, and encouragement throughout the research process. Her expertise in the field has been invaluable, and we have learned a great deal from her.

We would also like to thank Knowit company for their generous support. Their assistance was instrumental in allowing us to complete our research and achieve our goals.

We would also like to extend our thanks to the students from Blekinge Institute of Technology and the team at Knowit who participated in the experiment for this project. Their contribution has been essential in providing valuable data and insights into the usability of the gRPC and HTTP libraries.

Finally, we would like to thank our families and friends for their unwavering support and encouragement throughout our studies. Their love and understanding have been instrumental in helping us to achieve our academic goals.

Contents

Abstract	i
Acknowledgments	ii
1 Introduction	1
1.1 Background	2
1.2 Scope	3
1.3 Outline	4
2 Related Work	5
2.1 Related work	5
2.2 Similarities and Differences	7
3 Method	9
3.1 Aim	9
3.2 Research questions	9
3.3 Empirical Study	10
3.4 Performance Experiment	10
3.4.1 Test scenarios	10
3.4.2 gRPC Server Implementation	11
3.4.3 HTTP Server Implementation	11
3.4.4 ghz Measurement	11
3.4.5 K6 Measurement	13
3.4.6 Specifications	13
3.5 Usability Experiment	14
3.6 Validity and Reliability	15
3.6.1 Strengths and weaknesses	18
4 Results and Evaluation	19
4.1 Performance results	19
4.1.1 Test 1: Sending 500 000 requests for a string of length 14 bytes	21
4.1.2 Test 2: Sending 200 000 requests for a string of length 150KB	21
4.1.3 Test 3: Sending 20 000 requests for a string of length 3MB . .	22
4.2 Usability results	26
5 Discussion	28
5.0.1 Limitations	31

6	Conclusions and Future Work	33
6.1	Conclusions	33
6.2	Future Work	34
A	Usability Experiment	38
A.1	Task 1: HTTP Communication	38
A.2	Task 2: gRPC Communication	39
A.3	HTTP server and client	40
	A.3.1 Server	40
	A.3.2 Client	40
A.4	gRPC server and client	40
	A.4.1 Server	40
	A.4.2 Client	41
A.5	Survey results	41
B	Performance Experiment	46
B.1	gRPC service implementation	46
B.2	Protobuf definition	47
B.3	HTTP service implementation	47

Microservices are a relatively new architectural approach that has gained significant popularity in recent years due to their ability to facilitate scalability, flexibility, and agility in software development. However, while microservices offer many benefits, they also come with their own set of challenges, particularly in terms of communication between services. In this context, choosing the appropriate communication framework is critical to the success of a microservices architecture. In a survey conducted by the O'Reilly publishing company, over 77% of respondents reported that their company is using microservices in some capacity. [1] Microservices are a software architecture style that structures an application as a collection of small, independent services, each performing a single task. These services communicate with each other through lightweight protocols to achieve the desired functionality of the application. [2] The speed of communication has a much larger impact in a microservices architecture than in a monolith architecture. In a monolith, all communication occurs within the same process on a host, while in microservices it is common to request data from services on different hosts. Spotify has around 1,200 microservices running [3], and when you send a request to one service it is common that it has to request some data from another service, and so on. Due to the sheer amount of requests needed to be made across different hosts, a small performance benefit in communication can have a large impact. HTTP frameworks (Hyper Text Transfer Protocol) and gRPC (gRPC Remote Procedure Calls) are two of the most commonly used frameworks for communication between microservices.

HTTP has been in use for decades and is one of the most widely used communication protocols on the internet[4]. It facilitates data exchange between two machines, and its simplicity and extensive adoption make it a popular choice for microservices. HTTP uses methods, such as GET, POST, PUT, and DELETE, and commonly the JSON data format to communicate between client and server. These methods are easy to understand and widely adopted, which makes HTTP a flexible choice for building APIs. HTTP communication is usually provided through the use of a framework that abstracts and simplifies the methods. The framework we have chosen to focus on in our experiments is the *net/http* framework in the Go language which is part of the standard library and makes it a common choice when creating services using Go. On the other hand, gRPC is a remote procedure call framework created by Google in 2015 [5], designed for high performance, efficiency, and low latency. gRPC uses the Protocol Buffers data serialization format, which is faster and more compact than traditional JSON-based serialization formats. gRPC has libraries for

various programming languages, making it an excellent choice for large organizations using microservices as different teams can still integrate with the larger project while using the tools that are best for them. gRPC also supports bi-directional streaming, which enables real-time communication between services.

Despite their unique strengths and weaknesses, there still exists some ambiguity and misconception about the two types of frameworks and their applications in microservices. Therefore, this study aims to evaluate the performance and usability of HTTP vs gRPC frameworks in communication between microservices, to contribute to clarifying which technology provides the best performance and usability in this context. Specifically, we will conduct a comparative analysis of the two technologies and identify the strengths and weaknesses of each. By doing so, we hope to provide insights that will guide developers in choosing the appropriate framework for their microservices architecture.

The main problem and the gap that this study aims to fill is the lack of research specifically comparing the performance and usability of HTTP and gRPC frameworks in the context of a microservices architecture. While there have been studies comparing the frameworks in general, there is a need for a deeper understanding of how these perform and can be effectively used in the context of microservices. This study aims to provide insights that will help guide developers in making informed decisions when choosing the appropriate framework for their microservices architecture, filling the gap in the existing research.

1.1 Background

Microservices are a popular architectural approach for developing complex software systems. In a microservices architecture, an application is broken down into small, independently deployable services that communicate with each other to achieve the desired functionality of the application. While this approach offers many benefits, such as scalability, flexibility, and agility [6] [7], it also introduces challenges related to service communication. One critical factor to consider when designing a microservices architecture is the choice of communication protocol[8]. The lightweight communication protocol can significantly impact the performance of the system in terms of throughput, latency, and scalability.

Throughput refers to the amount of data that can be transmitted over a network in a given time period. In a microservices architecture, the system's overall throughput is affected by the communication protocol used between the services. A protocol that can handle a higher volume of requests can improve the system's throughput and ultimately result in better performance.[9]

Latency is the time it takes for a request to be processed from the time it is sent until a response is received. High latency can cause delays and slow down the overall performance of the system. In a microservices architecture, the choice of communication protocol can affect the latency of requests between services. A protocol that

offers low latency can help improve the overall performance of the system.

Scalability is defined by us to be the ability of a system to handle an increasing workload without experiencing a significant decrease in performance. In a microservices architecture, scalability is crucial as the system's complexity grows with the number of services. In a cloud computing context, scalability usually refers to the ability of a cloud layer to increase its capacity by expanding its quantity of consumed lower-layer services [10]. A system has to scale up when a larger demand is placed on it, and it is usually done by increasing the number of instances of the service running concurrently. This increases the costs associated with running the system but improves the system's ability to handle a large demand. If individual services are able to better handle an increasing workload, then the system would not have to scale up as often, which shows how our definition of scalability relates to the common definition of scalability in cloud computing. The communication protocol used between services can affect the system's scalability, and the protocol should be able to handle an increasing number of requests and services.

In addition to performance, usability is a critical factor that can affect the success of a microservices architecture. Usability refers to how easily users can use the system. Therefore, this thesis examines the impact of communication protocols on the usability. To be more precise, usability in microservices architecture refers to the degree to which the microservices system is designed and implemented to be easily understood, accessed, and utilized by developers. It focuses on the aspects that enhance the efficiency, effectiveness, and convenience of working with microservices[11].

Virtual users [12], also known as simulated users, are software components that mimic the behavior of real users in a system. They are used to simulate the load on a system and measure its performance under different circumstances. Virtual users are particularly useful in load testing, where they can be used to simulate a large number of users accessing a system simultaneously and to identify performance bottlenecks and areas for improvement. In our study, we used virtual users to simulate the load on the microservices architecture under test and measure its performance in terms of throughput, latency, and scalability. We used different numbers of virtual users in our tests to determine how the system would perform under different levels of load.

1.2 Scope

The study will focus on comparing the performance of HTTP and gRPC in terms of latency, throughput, and scalability, and it will also evaluate the usability of the two frameworks. The study will use a set of performance metrics to compare the two frameworks and provide insights into which framework is better suited for a microservices architecture.

The study will not cover other communication protocols or frameworks outside of HTTP and gRPC, nor will it delve into other aspects of microservices architecture

beyond communication between microservices. The study will also not cover the security implications of using either framework.

The general goal of this study is to contribute to the ongoing discussion surrounding the use of HTTP and gRPC in a microservices architecture. By defining the scope of the study, we can ensure that the research remains focused and relevant and that we can provide clear and actionable insights into the use of HTTP and gRPC in a microservices architecture.

1.3 Outline

The following list provides an overview of the structure of the rest of the thesis:

2. **Related Work:** In this section, we review the existing literature related to our research problem and the proposed solution. We examine related studies that have attempted to address similar issues and discuss the similarities and differences between these studies and our work.
3. **Methodology:** The methodology section includes a description of the aim of our study, the research questions that guided our investigation, and the empirical study we conducted to test our proposed solution. We also describe the usability and performance experiment, including the test scenarios we used, and discuss the validity and reliability of our approach.
4. **Results and Evaluation:** The results and evaluation section presents the performance and usability results of our study. We analyze and interpret the results followed by tables and diagrams.
5. **Discussion:** In this section, we discuss the conclusions to our research questions, the limitations of our work, and the possible implications and contributions of our study to society and the environment.
6. **Conclusions:** This is the section where we summarize the main findings of our study, including the contributions we have made to the research field.
- 6.2 **Future Work:** Finally, we present suggestions for future work based on the limitations and opportunities that emerged from our study. These suggestions could provide useful insights for researchers and practitioners who seek to improve the efficiency of similar processes.

The aim of this chapter is to provide an overview of existing research on communication protocols in microservices architecture, particularly focusing on the comparison between the Representational State Transfer (REST) and gRPC protocols. To achieve this goal, we conducted a literature review of relevant articles and studies. The literature review covers a wide range of topics related to microservices architecture and communication protocols, including the history and evolution of REST and gRPC, their architectural differences and implementation details, and their advantages and limitations. By synthesizing and analyzing the existing literature, this chapter aims to provide a solid foundation for our own research and to identify potential gaps and opportunities for future research.

2.1 Related work

A study conducted by M. Bolanowski et.al [13], compares the latency in communication between REST and gRPC microservices written using the .NET 5 platform. The authors wrote five different microservices and designed seven scenarios to test these services using encrypted and unencrypted communication. Tests were conducted using the JMeter application and the services were under a light load of one request per second. This is similar to our work but there are a few differences. Our services are written in the Go language, also we are not only analyzing the latency but also the throughput and scalability, and lastly, we are not using JMeter for measuring our performance metrics. The results they found were that for unencrypted communication, REST had lower latency for the following scenarios: transmission of text data, numerical data, large numerical data, and structured data in the form of tables and objects. It was found that they were both equal when sending small files, and gRPC had lower latency when sending large files. When it comes to encrypted communication it was found that gRPC had lower latency for small and large file transfers, they were equal when transmitting text data, structured data, and large numerical data, and REST had lower latency for small numerical data.

Another study conducted by L.D.S.B Weerasinghe and I Perera [14], evaluates the most popular communication protocols used in microservice architecture and measures the performance of inter-service communication in a distributed environment. The study compared REST, gRPC, and WebSocket protocols by conducting experiments to measure the response time, throughput, and time taken for inter-service communication. The results of the study showed that gRPC performed better than

the other protocols in terms of inter-service communication time, throughput, and response time. On the other hand, WebSocket was found to be unsuitable for inter-service communication in a microservice architecture. The research focused solely on the performance aspect of microservice architecture, which is critical for software scalability. The study assumed that processing logs from the ELK stack was the best method to monitor application turnaround time for performance evaluation, based on industry reference architectures. However, microservice architecture has other quality attributes, such as maintainability, security, testability, and scalability, which can be evaluated in future research. Further studies can also explore better methods to improve inter-service communication in a microservice architecture. The research provides valuable insights into the performance of communication protocols in a microservice architecture, and the findings can be used to guide software companies in selecting the appropriate communication protocol for their microservice-based applications.

A study titled "MSN: A Playground Framework for Design and Evaluation of MicroServices Based sdN Controller" [15] was conducted by researchers from various universities and research centers. The authors of the study are Sisay Tadesse Arzo, Fabrizio Granelli, Domenico Scotece, Daniel Barattini, Luca Foschini, Riccardo Bassoli, and Frank H. P. Fitzek. The study presents a playground framework called MSN that can be used for designing and evaluating microservices-based software-defined networking (sdN) controllers. The framework provides a platform for implementing and testing different microservices architectures and their performance in the context of sdN controllers. The MSN framework is based on the Docker containerization technology, which allows for easy deployment and management of microservices. The authors evaluate the MSN framework by implementing a microservices-based sdN controller using the Ryu SDN controller and evaluating its performance under different traffic scenarios. They compare the performance of the microservices-based sdN controller with a monolithic controller and a traditional SDN controller. The results show that the microservices-based sdN controller outperforms the monolithic controller and performs similarly to the traditional SDN controller in terms of traffic handling capabilities. The study highlights the benefits of using microservices architectures in the context of sdN controllers and provides a framework for designing and evaluating such architectures. The MSN framework can be used as a playground for researchers and practitioners to experiment with different microservices-based sdN controller architectures and evaluate their performance.

Research carried out by Ł. Kamiński et.al [16] showed that services using gRPC had faster response times compared to REST. With the Python language, the researchers implemented services utilizing various different protocols including REST and gRPC. These services were then tested on three different test scenarios: inserting one element, fetching one element, and fetching 100 elements from a MongoDB database. The performance metrics this study is measuring are response time and the size of all the packets sent per request. When inserting and fetching one element, gRPC had a faster mean response time of about 800 microseconds and when fetching 100 elements gRPC was faster by 8 milliseconds. Their data also found that gRPC had a higher data usage with gRPC sending 275 more bytes when inserting

one element and 471 more bytes when fetching one element. This research is directly relevant to the current study as it also explores the performance of gRPC and REST (HTTP + JSON) implementations, albeit in a different language and with different test scenarios. Our study also tries to not only explore the difference in response time between the protocols but also the difference in throughput and scalability.

2.2 Similarities and Differences

There are several studies in the literature that have explored the performance of communication frameworks in the context of microservices architecture. Our study aims to evaluate the performance and usability of HTTP and gRPC frameworks in microservices communication. In comparison to related studies, we explore additional aspects such as throughput and scalability, alongside latency measurements. While some previous studies have focused on comparing the latency in communication between REST and gRPC microservices, our study expands on these findings by analyzing latency, throughput, and scalability in microservices written in the Go language. Furthermore, we employ different tools for measuring performance metrics, deviating from the approaches used in previous studies. In this section, we will compare our study with four related works and identify their similarities and differences.

- Similarities
 - All studies focus on evaluating communication frameworks in the context of microservices architecture. They recognize the importance of selecting the appropriate framework for efficient communication between services.
- Differences
 - In a related study by M. Bolanowski et al. [13] they compared the latency in communication between REST and gRPC microservices using the .NET 5 platform. However, our study analyzes latency and considers throughput and scalability as key performance indicators. Additionally, we conduct our experiments using Go language microservices, providing insights into the performance and usability of HTTP and gRPC frameworks in a different language and implementation context.
 - Another study by L.D.S.B Weerasinghe [14] and I Perera evaluated the performance of communication protocols in a distributed environment. While they focused on comparing REST, gRPC, and WebSocket protocols, our study goes beyond performance evaluation and also assesses usability aspects. We aim to identify the strengths and weaknesses of HTTP and gRPC frameworks in terms of latency, throughput, and scalability, helping developers make informed decisions when selecting a communication framework for their microservices architecture.

- Additionally, a study by Sisay Tadesse Arzo et al. [15] focuses on designing and evaluating microservices-based software-defined networking (sdN) controllers. Although their study presents a framework for sdN controllers, our study specifically compares the performance and usability of HTTP and gRPC frameworks in microservices communication. By conducting a comparative analysis of these frameworks, we aim to provide insights into their respective strengths and weaknesses, enabling developers to make informed decisions in selecting a suitable framework for their microservices architecture.
- Lastly, a study conducted by Ł. Kamiński et al. [16] explore the performance of gRPC and REST (HTTP + JSON) implementations in the Python language. While their study evaluates response times and data usage in different scenarios, we expand on these metrics by including throughput and scalability as important factors. Moreover, we conduct our experiments using Go language microservices, providing a different implementation context and a broader understanding of the performance and usability of HTTP and gRPC frameworks in a microservices architecture.

Overall, while there may be similarities in terms of the context of evaluating communication frameworks in microservices architecture, each study has its own unique research objectives, methodologies, metrics, and scope. Our study contributes to the existing research by explicitly comparing the performance and usability of HTTP and gRPC frameworks in the context of microservices architecture, filling the gap in the literature.

3.1 Aim

The aim of this thesis is to evaluate and compare the performance and usability of gRPC and HTTP frameworks in order to determine which one is better suited for microservices. By determining this we can empower developers to make a more informed choice in which technology to use in their microservices architecture. This overall aim is broken down into several measurable and operational objectives, including identifying a set of common use cases for the frameworks, implementing these use cases using both frameworks, and testing these use cases at different loads to measure throughput, latency, and scalability.

3.2 Research questions

- RQ1: Which framework has better data transfer performance regarding throughput, latency, and scalability?
- RQ2: Which framework has better performance on average across all three characteristics?
- RQ3: Which framework provides better usability when implementing and integrating it with existing systems?

The first research question - RQ1, aims to determine which framework has better data transfer performance in terms of throughput, latency, and scalability. We will test both frameworks under varying loads to gather data on their performance in these areas. The second research question - RQ2, aims to determine which framework has better overall performance across all three characteristics. To answer this question, we will analyze the data collected from the first research question and compare the performance of each framework. The third research question - RQ3, aims to determine which framework provides better usability for developers using existing programming libraries to implement and integrate with existing systems. By answering these research questions, we aim to provide valuable insights into the performance and usability of gRPC and HTTP libraries and contribute to the development of more efficient and effective software applications.

3.3 Empirical Study

Our research will utilize two controlled experiments to compare the performance and usability of gRPC and HTTP libraries. The first experiment will answer RQ1 and RQ2 and the second one will answer RQ3. We will take into account various factors that may influence the results, and we will conduct our research ethically and responsibly. By doing so, we will be able to provide valuable insights and recommendations for developers and researchers who are interested in using or evaluating these libraries.

3.4 Performance Experiment

To further elaborate on the first experiment, we used a "one factor two treatments" experiment design where two services are created that implement three methods that are identical between them. Both services were written in Go but one is using the gRPC library and the other uses the *net/http* library. We used the load-testing tools 'k6' by Grafana Labs [17] and 'ghz' by Bojan D. [18] to test the services and measure their performance. For each test scenario, we recorded the response time per request as outputted by the load-testing tools to measure the latency of the protocols, and we measured the time taken for the test scenario to complete in order to calculate the throughput of the protocols. Measuring the scalability was done by varying the number of simultaneous connections to the service for each test scenario and analyzing how fast the performance degrades as the number of users increase, each test was run with 1, 10, 50, 100, and 400 virtual users.

3.4.1 Test scenarios

- Scenario 1: Sending 500 000 requests for the string "Hello, World!" (14 bytes)
- Scenario 2: Sending 200 000 requests for a string of size 150KB
- Scenario 3: Sending 20 000 requests for a string of size 3MB

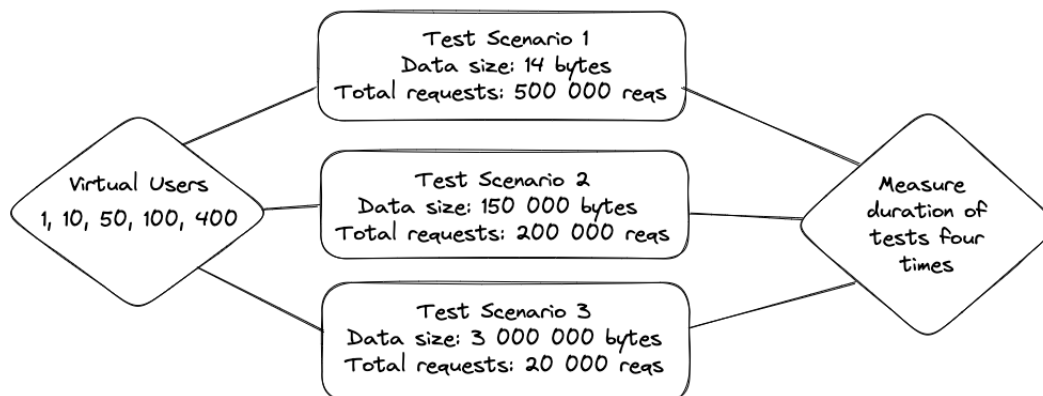


Figure 3.1: Test scenarios of the performance experiment

These scenarios aim to test the frameworks when sending messages of small, medium, and large sizes. We expect most requests in real-life scenarios are going to lie between the size of scenarios one and two which means that the data from those are able to act as a rough upper and lower bound on expected relative performance. Relative because real-world performance is impacted heavily by the specifications of the machine and network that the service is running on.

The strings for test scenarios 2 and 3 were generated using the following commands:

```
base64 -w 0 /dev/urandom | head -c 150000 > 150KB.txt
```

```
base64 -w 0 /dev/urandom | head -c 3000000 > 3MB.txt
```

These generate two files that contain 150KB and 3MB of random characters which can then be loaded into the program and be sent for each test.

3.4.2 gRPC Server Implementation

We started by defining a protobuf file that specifies our service and the methods the service will respond to. Each test scenario was defined as a separate method on the service that takes an empty request object and responds with a string. Once the protobuf file was defined it was compiled and the server was implemented according to the "Basics tutorial" for Go released by gRPC [19].

3.4.3 HTTP Server Implementation

Using the *net/http* package found in Go's standard library, we created three routes that correspond to the test scenarios and responded with the string for that test using the *WriteString* function from the *io* package found in the standard library.

3.4.4 ghz Measurement

The tests were run with the following options supplied to the *ghz* command:

```
--insecure
--skipTLS
--proto <Protobuf location>
--call Experiment.<Test number>
--total=<Total requests for test scenario>
--concurrency=<Virtual users>
--connections=<Virtual users>
--disable-template-functions
--disable-template-data
-O csv
```

Following are the descriptions of the options from the tool's documentation [20]:

- **-insecure**, Use plaintext and insecure connection.
- **-skipTLS**, Skip TLS client verification of the server's certificate chain and host name.
- **-proto**, The path to The Protocol Buffer .proto file for input.
- **-call**, A fully-qualified method name in 'package.Service/Method' or 'package.Service.Method' format.
- **-total**, The total number of requests to run.
- **-concurrency**, Number of workers to run concurrently when using const concurrency scheduler.
- **-connections**, By default we use a single gRPC connection for the whole test run, and the concurrency (-c) is achieved using goroutine workers sharing this single connection. The number of gRPC connections used can be controlled using this parameter.
- **-disable-template-functions**, Disable execution of template functions within call data and metadata. This can be useful for some performance improvements.
- **-disable-template-data**, Disable execution of templates within call data and metadata. This can be useful for some performance improvements.
- **-O**, Output type.

These options are a modified version of the options used for the *grpc_bench* project [21]. Their stated goal is "to compare the performance and resource usage of various gRPC libraries across different programming languages and technologies." [22] which is similar to the goal of our work but they only compare the performance within gRPC implementations. For our tests, we removed the *-rps* option to make sure that we don't limit the performance of gRPC and added the *-total* option to match our test cases. Since the command does not output how long the test took to complete when outputting the results as a CSV file, it was also run with the *time* command so that the time taken for the test to complete could be recorded.

3.4.5 K6 Measurement

K6 is configured using JavaScript files where you can programmatically define how the test should be run by changing a default function and setting options to match your test scenario needs. The options that we decided to use were the following:

```
discardResponseBodies: true,
noConnectionReuse: true,
scenarios: {
  contacts: {
    executor: 'per-vu-iterations',
    vus: 400,
    iterations: 1250,
  },
},
```

Following are the descriptions of the options from the tool's documentation [23]:

- **discardResponseBodies**, "Highly recommended to be set to true and then only for the requests where the response body is needed for scripting to set responseType to text or binary. Lessens the amount of memory required and the amount of [garbage collection] - reducing the load on the testing machine, and probably producing more reliable test results."
- **noConnectionReuse**, "A boolean, true or false, specifying whether k6 should disable keep-alive connections."
- **executor**, "Each [virtual user] executes an exact number of iterations. The total number of completed iterations will be vus * iterations."
- **vus**, "An integer value specifying the number of [virtual users] to run concurrently, used together with the iterations or duration options."
- **iterations**, "An integer value, specifying the total number of iterations of the default function to execute in the test run, as opposed to specifying a duration of time during which the script would run in a loop."

We chose to use the *discardResponseBodies* and *noConnectionReuse* options in order to make the behavior of k6 in line with the behavior of ghz which does not save response bodies or reuse connections. The executor, vus, and iterations were chosen to match the test case's amount of virtual users and total requests. The total time taken for the test to run is displayed at the end of each test when using k6 so there was no need to run it with the time command like with ghz.

3.4.6 Specifications

The tests were run on a personal computer with 16 GBs of 3200 MHz DDR4 memory and an Intel i5 8600K processor using a newly installed Fedora Workstation 38 operating system. We used k6 version 43 and ghz version 0.114.0. By performing our tests on a personal computer and not in a cloud environment, which would be where

microservices normally are deployed, we eliminate a number of potential sources of error that would arise such as network latency and reliability of the connection between client and server. Since both the client and server were both running on the same machine, the connections did not have to travel any physical distance, and the limitations of the network were eliminated.

3.5 Usability Experiment

For the second experiment, we designed a set of coding tasks that require the use of both gRPC and HTTP libraries to be completed by the participants. The instructions were split into two parts, one for the gRPC tasks and one for the HTTP tasks respectively. The tasks were designed to test usability with two specific usages of the libraries: implementing sending requests to a service and responding to requests. These correspond to most usages of networking libraries within services since without these functions it would not be able to do anything useful in the context of a larger system. The tasks were designed to be straightforward and able to be completed within a reasonable amount of time so that test subjects do not become fatigued or disengaged.

To recruit test subjects, we targeted a diverse group of participants, including professional developers and students, with varying levels of experience. The ten subjects were sampled from our university institution and the software consulting firm Knowit which we worked with closely throughout our study. Once the subjects were recruited, we held the experiment in person so that we could clear instructions and guidance to the test subjects, and monitor their progress throughout the completion of the tasks. Due to the small number of participants, everyone in the group was tasked with completing the HTTP and gRPC tasks. The instructions can be found in sections A.1 and A.2 of the appendix. The instructions were supplied together with an archived folder of the test environment for this experiment. Found within it were two directories, one for the gRPC environment and the other for HTTP. Each environment had a *main.go* file which contained the code for a service that the subject was tasked to communicate with, and two more files named *server.go* and *client.go* which is where the communication was to be implemented. The code for the client and server files can be found in sections A.3 and A.4 of the appendix.

After the completion of the tasks, we will ask the test subjects to fill out a questionnaire based on the System Usability Scale questionnaire [24] that we modified to fit our needs, a widely used and validated survey instrument that measures the perceived usability of a system. We will analyze the survey results to determine whether there are any significant differences in the perceived usability of gRPC and HTTP libraries.

The first question asks the participant "What is your current occupation?" so that we can record the diversity of our participants. After this, the participants were asked to rate their agreements with the following statements modified from the SUS on a five-point scale where the possible answers were "Strongly disagree", "Disagree", "Neutral", "Agree", "Strongly agree":

What is your current occupation? Developer
 Student
 Other: _____

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
1. I found the library easy to use.					
2. I think that people would learn to use this library quickly.					
3. I think that I would like to use this library frequently.					
4. I recommend improvements or changes to the library.					
5. The library had all the features or functions I expected to find.					
6. I think I would need support (an expert, guide or documentation) for working with the library.					
7. I think that the functions that I utilized from the library are well designed.					
8. I think that the library provided enough flexibility and customization options for my needs.					
9. I would recommend using this library to others.					

Figure 3.2: Modified System Usability Scale questionnaire

Once the responses were recorded, the answers were mapped to numbers with "Strongly disagree" being worth 0 and "Strongly agree" worth 4. Questions 4 and 6 are worded in a way that a "Strongly disagree" answer implies higher usability so the method for calculating the score was modified for these questions. For these the score was calculated by taking 4 minus the answer's mapped value, so an answer of "Strongly disagree" would yield $4 - 0 = 4$, and "Strongly agree" would become $4 - 4 = 0$. These scores were then summed to calculate a final score of a response.

3.6 Validity and Reliability

To ensure the validity and reliability of our load-testing and benchmarking approach, we carefully considered our chosen methods and implemented various measures to increase the accuracy of our results. We selected Ghz, a popular load-testing tool, to

measure the performance of our gRPC services. Additionally, we used K6, another widely used benchmarking tool, to supplement our findings. For both the HTTP and gRPC implementations of the test scenarios, the strings that are sent are saved in memory and not read from the file for each request. This makes the results less bound by the file system performance and more indicative of the protocol's performance.

Our data collection process was designed to minimize potential sources of bias and error. We took into account the limitations of our testing environment, such as hardware constraints, and conducted multiple tests to account for any variations in performance. By doing so, we were able to obtain a more accurate representation of the services' performance under different scenarios. To increase the validity of our research, we ensured that we investigated exactly what we intended to investigate. We established clear criteria for measuring the performance of the gRPC and HTTP services and collected data using the same conditions and parameters for each test. We also performed statistical analysis on our results to identify any significant differences between the services' performance. While our approach had several strengths, such as using established load-testing and benchmarking tools and carefully controlling for potential sources of bias, we acknowledge that there were also weaknesses. One limitation of our study is that our testing environment was not identical to a production environment, which may have affected the performance of the services. Additionally, our findings may not be generalizable to other gRPC and HTTP services, as the results may be influenced by the specific implementation and hardware used. But by carefully considering the validity and reliability of our approach, we were able to obtain accurate and meaningful results that provide insights into the performance of gRPC and HTTP services for different payload sizes.

When it comes to usability we have designed a set of coding tasks that accurately reflect typical usage scenarios of gRPC and HTTP libraries. These tasks have been carefully crafted to require the use of both libraries and to be representative of real-world usage scenarios, such as making requests for data and handling errors. Additionally, we have taken steps to minimize potential sources of bias in our study. We recruited a diverse group of participants, including both professional developers and students with varying levels of experience, to ensure that our results are applicable to a wide range of users. We also provided clear instructions and guidance to our test subjects and monitor their progress throughout the completion of the tasks to minimize the potential for errors and misunderstandings.

A potential validity threat for our usability experiment is the varying levels of familiarity with the Go language among the subjects. Go was chosen as the language for developing the microservices in the usability experiment. While detailed instructions were provided and assistance was offered throughout the experiment, the subjects' prior experience and proficiency with Go might have varied. This discrepancy in familiarity with Go could have influenced the subjects' ability to effectively implement, debug, and assess the usability of the frameworks. Consequently, it could introduce bias or inaccuracies in the data collected and potentially impact the validity of the study's findings. To mitigate this thread, we encouraged the subjects to ask questions

and seek clarification whenever needed. Additionally, we provided prompt support and guidance to address any Go-related challenges faced by the subjects during the experiment. Future studies could explore using subjects with more uniform levels of Go expertise to further enhance the validity of the research.

To measure the perceived usability of the gRPC and HTTP libraries, we will use a modified version of the System Usability Scale questionnaire, a widely used and validated survey instrument that has been shown to produce reliable and valid results. This questionnaire has been carefully chosen to provide an objective measure of the perceived usability of the two libraries, and we will analyze the survey results to determine whether there are any significant differences between the two libraries. Our approach for usability testing is designed to be both valid and reliable, using carefully crafted coding tasks and a validated survey instrument to measure the perceived usability of gRPC and HTTP libraries. We believe that our approach will provide valuable insights into the usability of these libraries and help developers make informed decisions when choosing which library to use for their projects.

3.6.1 Strengths and weaknesses

Strengths:

- **Validity and reliability:** The study carefully considered the chosen methods and implemented various measures to increase the accuracy of the results, including selecting established load-testing and benchmarking tools, using the same conditions and parameters for each test, and conducting multiple tests to account for any variations in performance.
- **Data collection process:** The study minimized potential sources of bias and error by taking into account the limitations of the testing environment and conducting multiple tests to obtain a more accurate representation of the services' performance under different scenarios.
- **Usability testing:** The study used carefully crafted coding tasks and a validated survey instrument to measure the perceived usability of gRPC and HTTP libraries, providing valuable insights into the usability of these libraries and helping developers make informed decisions when choosing which library to use for their projects.

Weaknesses:

- **Testing environment:** The study's testing environment was not identical to a production environment, which may have affected the performance of the services.
- **Generalizability:** The study's findings may not be generalizable to other gRPC and HTTP services, as the results may be influenced by the specific implementation and hardware used.
- **Participants:** Although the study recruited a diverse group of participants, including both professional developers and students with varying levels of experience, the sample size may not be large enough to provide a comprehensive understanding of the perceived usability of the libraries.

4.1 Performance results

The performance experiment we conducted using 'k6' by Grafana Labs and 'ghz' by Bojan D. was a comprehensive test that provided us with detailed results to analyze. The purpose of the experiment was to test two services that implement the three different test scenarios mentioned in 3.4.1, and we measured the response time per request as outputted by the load-testing tools. In this way, we were able to evaluate the performance of each service and compare their results under varying loads and request types. The experiment yielded a wealth of data that allows us to draw meaningful insights about the performance of the tested services. By analyzing the results, we can gain a better understanding of how each service performs in different scenarios, which can help us optimize the services for improved performance in the future.

In the following sections, we will examine the results of the experiment in detail, exploring the throughput and response time metrics for each test scenario under both gRPC and HTTP communication protocols. The results are presented in detailed tables and diagrams, providing a clear overview of the performance metrics. Here's a quick explanation of the terms used in our tables:

- **VUs** stands for Virtual Users. The VUs column indicates the number of virtual users involved in the test.
- **Avg** represents Average. In our tables, it signifies the average value of a specific metric, such as response time or throughput.
- **Min** stands for Minimum. It represents the lowest recorded value for a given metric, offering insights into the best-case scenario for latency and worst-case for throughput.
- **Max** refers to Maximum. It indicates the highest recorded value for a particular metric, highlighting the worst-case scenario for latency and best-case for throughput.

Through this analysis, we hope to gain valuable insights that can inform future optimization efforts and contribute to a better understanding of the performance characteristics of these services.

Latency - gRPC			
	Test1	Test2	Test3
VUs	Avg(Min, Max) ms		
1	0.07(0.04, 1.73)	0.18(0.13, 1.3)	2.55(1.72, 4.73)
10	0.16(0.04, 1.73)	0.84(0.15, 10.12)	14.29(2.99, 55.76)
50	0.59(0.04, 7.09)	3.98(0.16, 20.02)	69.1(3.64, 396.65)
100	1.18(0.05, 15.07)	7.82(0.17, 45.52)	136.66(5.05, 1794.18)
400	4.28(0.06, 21.67)	30.72(0.2, 203.55)	547.75(6.28, 9635.97)

Table 4.1: Latency - gRPC

Latency - HTTP			
	Test1	Test2	Test3
VUs	Avg(Min, Max) ms		
1	0.08(0.049, 11.96)	0.33(0.22, 3.56)	4.71(1.62, 13.95)
10	0.35(0.046, 14.05)	0.89(0.21, 11.25)	6.36(2.32, 22.81)
50	1.76(0.06, 22.84)	4.09(0.26, 35.67)	25.1(2.57, 162.31)
100	3.27(0.05, 30.13)	7.84(0.26, 49.7)	48.21(2.6, 295.71)
400	12.21(0.07, 80.98)	29.31(0.31, 166.69)	168.3(2.58, 786.65)

Table 4.2: Latency - HTTP

Throughput - gRPC			
	Test1	Test2	Test3
VUs	Avg(Min, Max) Req/Sec		
1	12347(11946, 12562)	5084(5074, 5096)	390(387, 394)
10	48984(47214, 50201)	11330(11173, 11561)	684(663, 693)
50	66339(62972, 67760)	11876(11855, 11891)	711(7010, 712)
100	69573(66862, 71083)	12108(12070, 12136)	712(708, 713)
400	72640(72401, 72791)	12016(11848, 12165)	701(690, 707)

Table 4.3: Throughput - gRPC

Throughput - HTTP			
	Test1	Test2	Test3
VUs	Avg(Min, Max) Req/Sec		
1	6395(6275, 6499)	2415(2391, 2424)	208(207, 208)
10	21626(21505, 21701)	9589(9372, 9823)	1530(1450, 1592)
50	25888(24665, 26417)	11406(11287, 11514)	1793(1765, 1813)
100	26730(26388, 27177)	11564(11025, 11841)	1814(1744, 1854)
400	27123(26824, 27336)	11989(11905, 12092)	1890(1855, 1910)

Table 4.4: Throughput - HTTP

4.1.1 Test 1: Sending 500 000 requests for a string of length 14 bytes

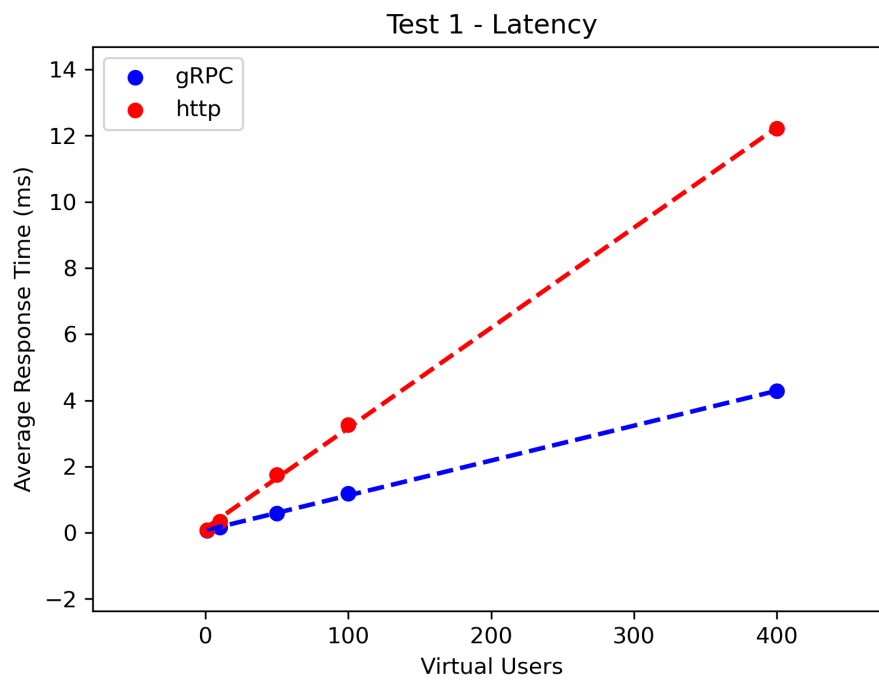
For this test scenario, we observed that the gRPC service outperformed the HTTP service in terms of both throughput and response time for all numbers of virtual users. The throughput of the gRPC service was on average **2.36** times that of the HTTP service. Similarly, the response time of the HTTP service was on average **2.12** times slower than the gRPC service. By analyzing the slopes of the trend lines in Figures 4.1(a) and 4.1(b), we can tell that the latency of gRPC increased at a slower rate than HTTP, and the throughput increased at a faster rate than HTTP as the number of virtual users increased. This indicates that gRPC had better scalability for this test.

4.1.2 Test 2: Sending 200 000 requests for a string of length 150KB

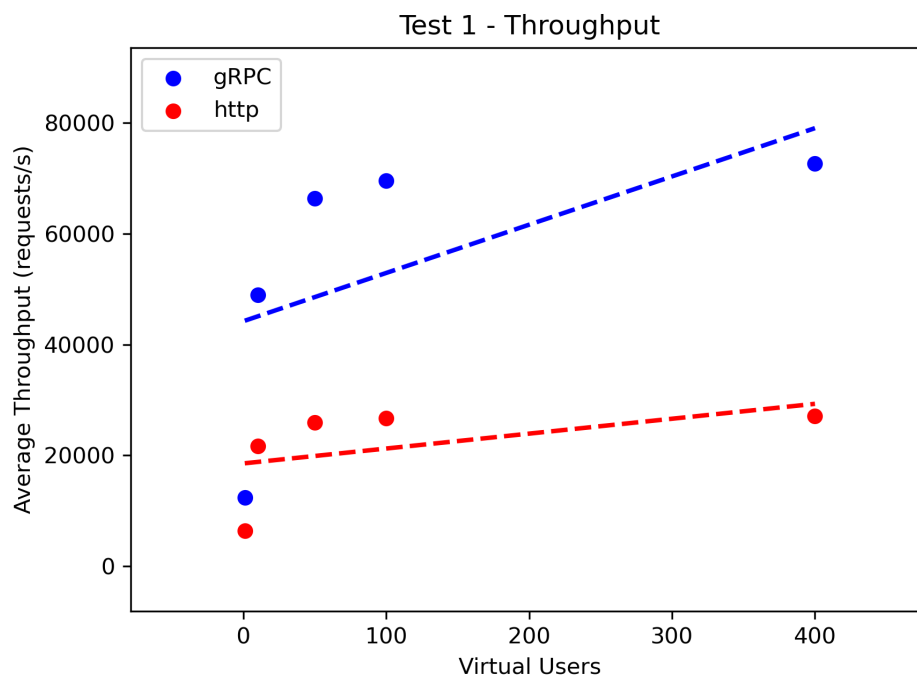
For this test scenario, the gRPC service slightly outperformed the HTTP service in terms of throughput and response time but the difference is less than Test 1. The throughput of the gRPC service was on average **1.27** times that of the HTTP service. Similarly, the response time of the HTTP service was on average **1.11** times slower than the gRPC service. The trend lines in Figures 4.2(a) and 4.2(b) indicate that the latency of gRPC and HTTP increase at a very similar rate with HTTP increasing at a slightly slower rate, and in terms of throughput, HTTP increased at a higher rate as the virtual users increased. This indicates that HTTP had slightly better scalability in this test.

4.1.3 Test 3: Sending 20 000 requests for a string of length 3MB

For this test scenario, we observed that the HTTP service outperformed the gRPC service in terms of both throughput and response time. The HTTP service's throughput was on average **2.10** times higher than the gRPC service, and the response time for the gRPC service was **2.32** times slower than the HTTP service on average. Figures 4.3(a) and 4.3(b) show that the latency of the HTTP service increased at a much slower rate than the gRPC service, and the throughput increased at a faster rate which indicates that HTTP had better performance for this test.

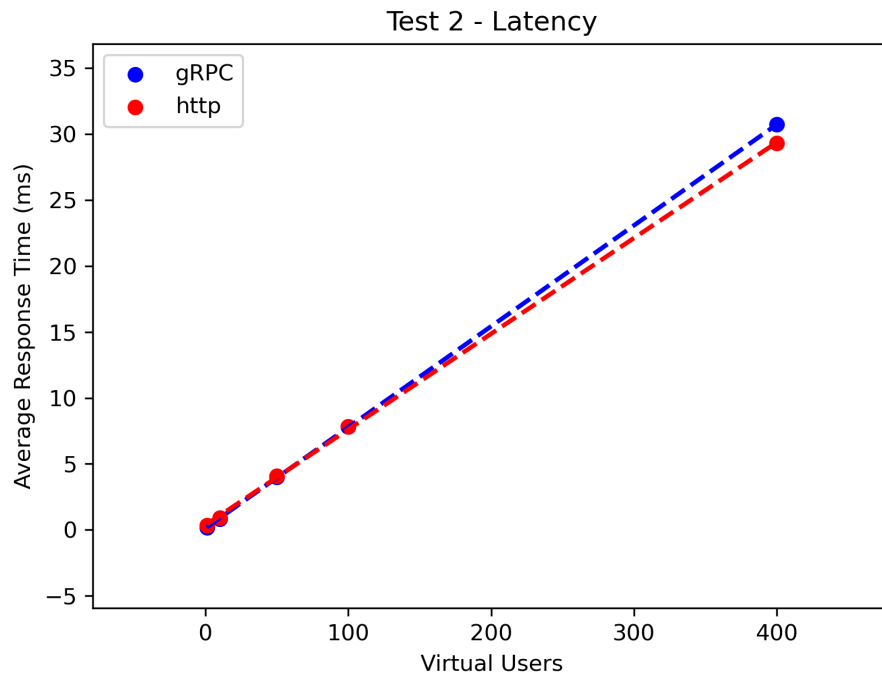


((a)) Latency

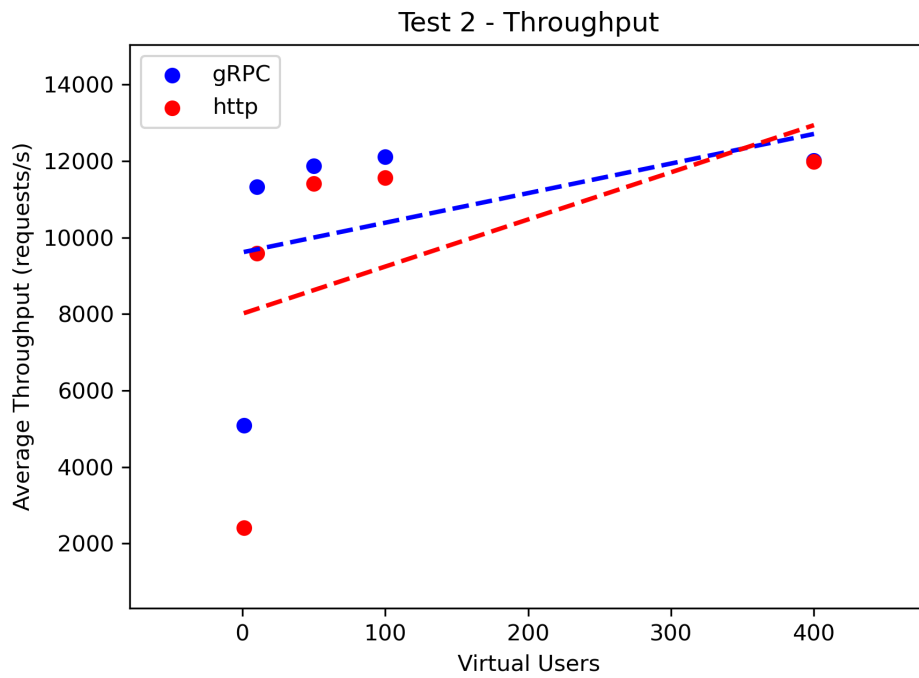


((b)) Throughput

Figure 4.1: Test 1 performance results

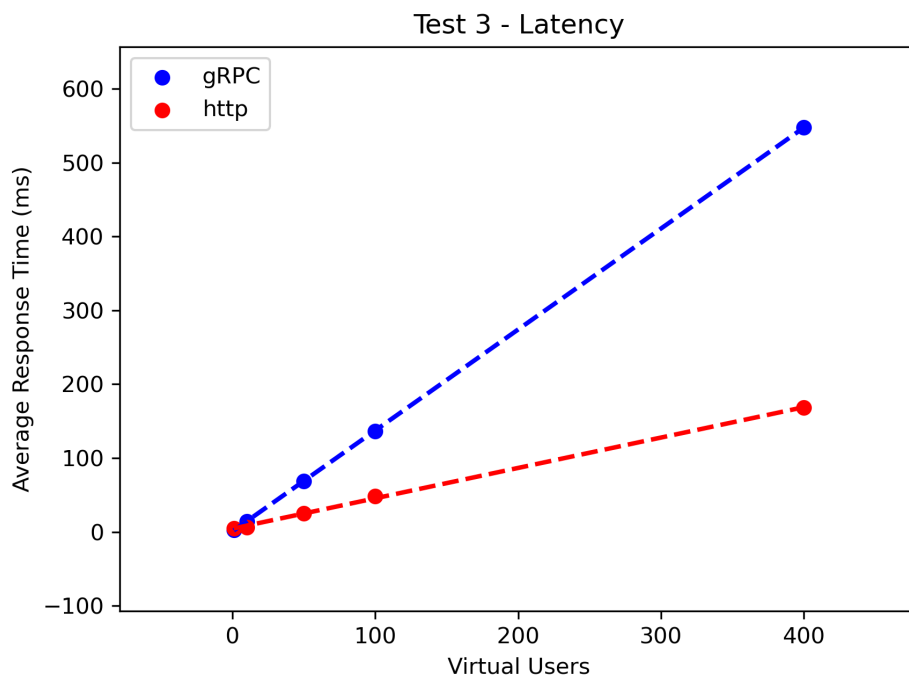


((a)) Latency

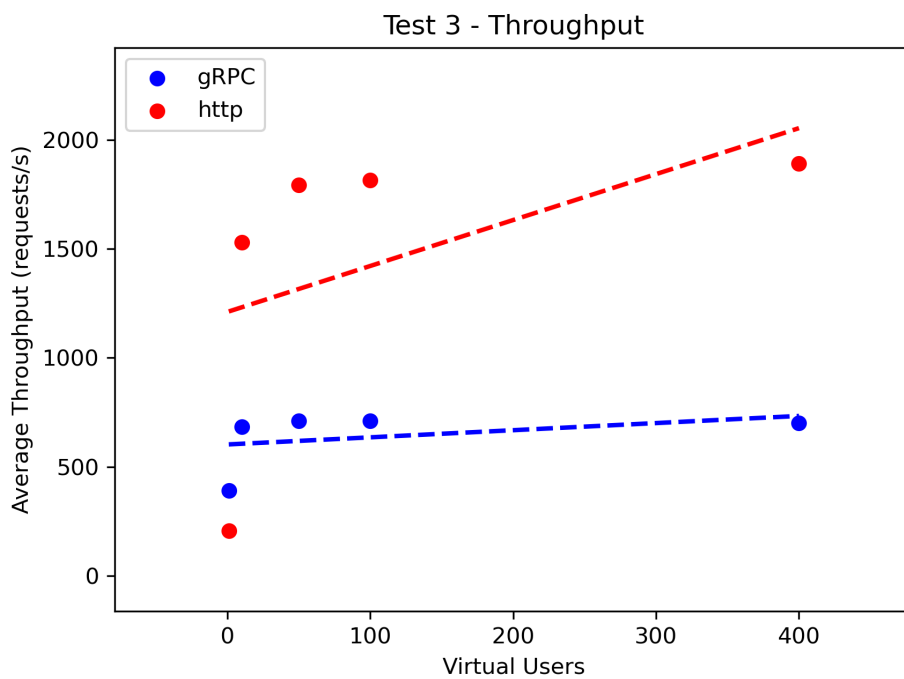


((b)) Throughput

Figure 4.2: Test 2 performance results



((a)) Latency



((b)) Throughput

Figure 4.3: Test 3 performance results

Across tests one and two, the gRPC service performed better in terms of latency and throughput while the HTTP service excelled in test three, but in terms of scalability the HTTP service outperformed the gRPC service in tests two and three. The following table aims to show which framework was observed to have better performance for each performance metric:

Test	Latency	Throughput	Scalability
1	gRPC	gRPC	gRPC
2	gRPC	gRPC	HTTP
3	HTTP	HTTP	HTTP

4.2 Usability results

In our usability experiment, a total of 10 individuals participated and completed the questionnaire. Out of these, 7 were students and 3 were developers. The following are the average scores per question in the questionnaire, and the average sum of all scores:

Question	gRPC	HTTP
Q1	0.85	2.6
Q2	1.2	2.8
Q3	1.35	2
Q4	1.1	1.9
Q5	1.9	2.4
Q6	0.4	1.2
Q7	1.7	2.5
Q8	2.2	2.4
Q9	1.5	2.4
Sum	12.2	20.2

The results from our usability experiment suggest that there are differences in user usability between gRPC and HTTP. Looking at the table, we can see that HTTP received higher scores than gRPC in every question. This indicates that, on average, users found HTTP to be a more satisfying framework to use than gRPC. It is worth noting that the difference in scores between the two frameworks is particularly noticeable in questions Q1, Q2, and Q6. These questions relate to ease of use, ease of understanding, and ease of learning, respectively. The fact that gRPC scored lower than HTTP in these questions suggests that it may have a steeper learning curve or be more difficult to understand and use for users who are not familiar with it.

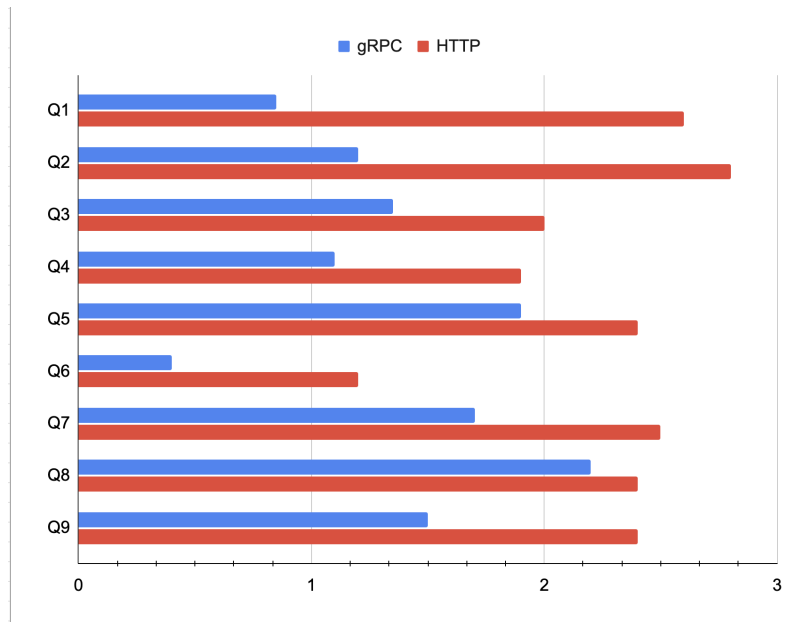


Figure 4.4: Usability average score

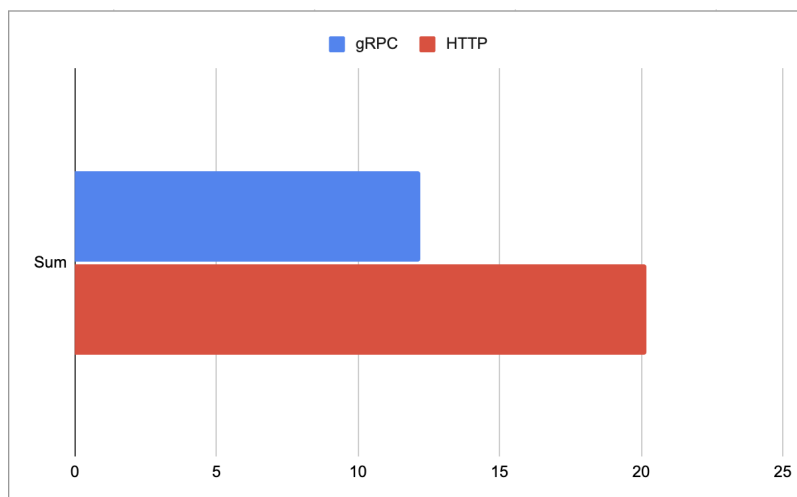


Figure 4.5: Sum of the usability average score

Analyzing these results shows that the responses for HTTP were on average **1.89** times higher than gRPC and the average sum of scores was **1.65** times higher. This gives an indication that the respondents found the HTTP library more satisfying to use than gRPC.

Overall, the usability results provide valuable insights into the user experience of gRPC and HTTP. While gRPC may have better performance in terms of throughput and latency, HTTP appears to have an advantage in terms of usability. These findings can help developers make more informed decisions about which framework to use for their specific needs and goals.

As software systems become more complex, the need for efficient and scalable communication between components is crucial. Therefore, selecting the right communication framework is important. This study aims to compare the performance and usability of gRPC and HTTP frameworks to provide developers with insights on which framework is more suitable for their communication needs. Specifically, we aim to answer the research questions of which framework has better data transfer performance and which framework provides better usability.

To reiterate, our research questions for this thesis were the following:

- **RQ1: Which framework has better data transfer performance regarding throughput, latency, and scalability?**
- **RQ2: Which framework has better performance on average across all three characteristics?**
- **RQ3: Which framework provides better usability when implementing and integrating it with existing systems?**

RQ1 sought to determine which framework had better data transfer performance in terms of throughput, latency, and scalability. Our analysis showed that gRPC outperformed HTTP in terms of throughput and latency in tests one and two. However, HTTP performed better in scalability in tests two and three. Throughput and latency are critical performance metrics for data transfer. Throughput measures the amount of data that can be transferred in a given amount of time, while latency measures the time it takes for a message to travel from the sender to the receiver. In our tests, gRPC consistently performed better than HTTP in these two metrics, indicating that it may be the better choice for data transfer when high throughput and low latency are a priority. On the other hand, scalability is essential when it comes to transferring data to large numbers of users. In our tests, HTTP performed better than gRPC in scalability in tests two and three. This indicates that HTTP may be a better choice for data transfer when scalability is a priority. In conclusion, for RQ1, gRPC has better performance in throughput and latency, while HTTP has better performance in scalability.

The difference in the results for tests one and three are quite puzzling. The fact that gRPC was much faster during test one but slower in test three was quite unexpected for us. We are not quite sure what the explanation for this could be,

but we think it could be likely that the cost of encoding the string to a format able to be sent through the connections is greater for gRPC than HTTP. Since gRPC communicates using objects instead of strings, perhaps it is performing more actions on that object than HTTP would be doing on a string alone. An alternative explanation could be that the limitation could stem from our choice of load-testing tools. Perhaps *ghz* fails to efficiently handle large responses while *k6* is able to discard them. Without a proper analysis of the source code, it is impossible to say. It could be argued however that since these tools are popular open-source tools, they would be sufficiently efficient due to it being easier to find performance flaws if the source is available. We are confident in the validity of our results due to our choice of tools, and the implementations of our services are as identical as we could make them. Our choice of options when running the tools and our testing environment reduces the number of sources of error due to things not related to the frameworks.

Moving on to RQ2, our analysis sought to determine which framework had better performance on average across all three characteristics. We found that out of the nine measurements (the three performance metrics multiplied by three tests), gRPC outperformed HTTP in five, while HTTP outperformed gRPC in four. This indicates that on average, gRPC has better performance than HTTP in data transfer. However, it is essential to note that the performance of each framework depends on the specific use case and requirements. Thus, it is crucial to evaluate the requirement of a particular system before choosing a framework for data transfer. In conclusion, gRPC has better performance on average across all three performance characteristics, but the choice of the framework should be made based on the specific requirements of the system.

The results of RQ1 and RQ2 are interesting in the way that it contrasts with the results of the work by Bolanowski et.al [13]. The results of their study showed that when sending the string "Hello, World!" gRPC had a much slower response time than HTTP, and when sending files of size 3.4 MB gRPC had a faster response time. These results seem to contradict the findings of our study which indicated that gRPC had a faster response time when sending "Hello, world!" and slower when sending 3 MB files. The reason for this could be due to methodological differences between the studies such as data collection methods or the implementation of the services themselves. Our study used the tools *ghz* and *k6* to measure the performance of our services while Bolanowski's used *JMeter*, which has an influence on the observed performance. A review of load testing tools by *k6* showed that *JMeter* has response times that are about double that of *k6* [12] which could explain the discrepancy between our results when sending large files however it doesn't explain the difference when sending the hello world string. While the review does not mention gRPC load testing tools, it could be speculated that the gRPC implementation in *JMeter* might also be slower than *ghz*, but it could be argued that if the difference between *ghz* and *JMeter*, and *k6* and *JMeter* is the same then the relative performance between HTTP and gRPC in our study should still be equal the Bolanowski's study. There are a few possible explanations for this difference: the difference in performance between *ghz* and *JMeter* could be greater

than the difference between k6 and JMeter, or the implementation of the services could vary in such a way that the performance favors gRPC in our case. The difference in performance between Go and .NET 5 could not explain this discrepancy because .NET's gRPC implementation has better performance than Go in benchmarks by the *grpc_bench* project [25].

When it comes to RQ3, and based on the results of our usability experiment, it appears that HTTP provides better usability when implementing and integrating it with existing systems compared to gRPC. Our modified System Usability Scale showed that the average score for HTTP was 1.89 times higher than gRPC, and the average sum of all scores was 1.65 times higher for HTTP. These results suggest that developers find HTTP easier to use and more satisfying than gRPC.

While the reason for this difference in usability is beyond the scope of our study, it is possible that the more established and familiar nature of HTTP may have contributed to this result. HTTP has been in use for several decades and is a well-established protocol with a large community of developers, while gRPC is a relatively new technology that is still gaining adoption in the industry.

It is important to note that these results may not generalize to all developers and use cases. The participants in our study were a small sample of professional developers and students with varying levels of experience, and our coding tasks may not have fully represented all potential use cases for gRPC and HTTP. Additionally, there may be other factors beyond usability that influence a developer's choice of framework, such as performance or compatibility with existing systems.

Our conclusions have implications for research and practice in the field of software engineering. When it comes to practice, we expect developers to be better informed on which framework is better suited for their use case. If they find themselves sending small or medium-sized messages they could find performance gains by switching to using gRPC for their inter-service communications but if they require large messages to be sent for say, large file transfers, they should consider HTTP. In the case of a small company where development velocity is important, using HTTP could bring gains due to higher usability when using it making it easier to implement features. In terms of research, we have identified some areas where gRPC has a performance advantage over HTTP and one where the opposite is true. This can help guide researchers where a closer analysis may be warranted and also map where the knowledge gaps exist for gRPC. Our work also helps uncover some knowledge gaps that have existed in this area, the usability of gRPC vs HTTP. We hope this can inspire future works to build on our results in this area and perhaps give the developers of gRPC data to help inform them about what could be improved on in future versions of gRPC.

Our work in evaluating the performance and usability of service communication in microservices has the potential to bring about positive impacts on both society and the environment. Efficient service communication can minimize the consumption of computing power and network bandwidth leading to energy savings and reduced environmental impact. A large benefit of microservice architectures is being able to scale the number of services up or down depending on the current demand, and

improved scalability of service communication of microservices can reduce the number of virtual machines needed for a given load and it could also enable reduced requirements on physical infrastructure if an organization deploys its microservices on-premises which would result in cost and energy savings. It could enhance collaboration by reducing latency leading to faster delivery of information and quicker data processing, resulting in increased efficiency and productivity among different teams and other collaborative efforts. Improved usability of communication frameworks reduces complexity and learning curve for developers, enabling them to build and integrate microservices more efficiently which leads to increased productivity and allows organizations to deliver services and features more quickly. Simplified communication frameworks allow developers to focus more on creating innovative solutions rather than dealing with the intricacies of communication protocols leading to faster innovation. It also lowers the barrier to entry, making it easier for developers of varying skill levels to adopt and utilize microservices effectively. This can lead to a broader range of developers participating in the development of innovative technologies which fosters diversity and collaboration.

Our work does not have many ethical implications. The act of empowering developers to make informed decisions about their choice of technology is ethically neutral, but it is up to the developers that implement services using our work as a background to create solutions that do not infringe on other people's privacy or rights. Our data collection in the usability experiment was handled as ethically as possible. We did not collect any personally identifiable information and compensated the participants after the experiment was completed to make sure that the participants' time and privacy were dutifully respected.

5.0.1 Limitations

There are some limitations to our study and aspects of the tools which we decided not to study in our work. Bi-directional streaming is a feature that exists for gRPC which can help alleviate long response times for large messages by only sending small chunks of it at a time, this feature was ignored in our work due to HTTP not having the same feature by default. A large part of using gRPC is also writing protobuf files to define what services exist and how they should communicate. This aspect was ignored in our usability experiment in order to focus on what we determined to be the most important feature of the frameworks which is sending and receiving requests. Another aspect we decided to ignore was the different data types that protobuf allows to be sent and only focus on strings. It is possible to send strings, ints of different sizes, floats, booleans, and bytes. These can be serialized from their base 10 representations to binary which can be really efficiently sent to the receiver but in order to ensure that the size of the packets was consistent between HTTP and gRPC we decided to only use strings that can't be serialized in the same way. In the usability experiment, some of the work was already done by us ahead of time such as creating the client's gRPC connection to the server, serving a gRPC server on a specific port that has a predefined empty method, and serving an HTTP server on a port which listens on a specific route with an empty

handler method. This left the participant to decipher how to create the client and use the required methods on it, and define what should be returned from the empty methods to send the correct data. This meant that the tasks could be smaller, easier to execute, and require the participants to learn fewer new concepts, which let us hone in on specific parts of the frameworks but it also meant that we didn't account for some aspects of their API which are required to use the frameworks. The sample size of our study was relatively small, with only 10 participants, and they were all recruited from the same institution, which may limit the generalizability of our findings. Additionally, the study only focused on perceived usability and did not measure actual performance or other aspects of the protocols, which may be important to consider in making a final decision about which protocol to use.

6.1 Conclusions

Based on the results of our performance experiment, we can conclude that gRPC outperforms HTTP in terms of throughput and response time when handling small to medium-sized payloads. Specifically, gRPC had a significantly higher average throughput and lower average response time compared to HTTP in Tests 1 and 2, where the payload sizes were relatively small (14 bytes and 150KB, respectively). This suggests that gRPC is a more efficient option for handling requests that involve small to medium-sized data. However, when it comes to handling larger payloads, the results were different. In Test 3, where the payload size was 3MB, HTTP outperformed gRPC in terms of both throughput and response time. This suggests that HTTP is a better option for handling requests that involve large data. The choice between gRPC and HTTP depends on the nature of the data being transmitted. If the payload size is small to medium, gRPC is likely to offer better performance. However, if the payload size is large, HTTP might be a more efficient option. It is important to consider these factors when deciding which protocol to use in a given scenario.

When it comes to our usability experiment and based on the results, it is clear that there are differences in the perceived usability of gRPC and HTTP libraries. The average scores for HTTP were consistently higher than those for gRPC across all questions in our modified System Usability Scale questionnaire, indicating that the respondents found the HTTP library more satisfying to use than gRPC. These findings have important implications for developers who are considering using either gRPC or HTTP for their projects. While both protocols have their strengths and weaknesses in terms of performance, our results suggest that HTTP may be the better choice when it comes to usability. Developers may want to consider this factor along with other factors, such as the specific requirements of their project and the resources available when making a decision about which protocol to use.

6.2 Future Work

While our study provides valuable insights into the performance and usability of gRPC and HTTP frameworks, there are several avenues for further research that could expand on our findings and provide even more comprehensive information for developers. Future research could build on our study by using larger and more diverse samples, as well as incorporating other measures of protocol performance and usability. Additionally, further investigation could be conducted into the factors that influence the perceived usability of gRPC and HTTP, such as the design of the APIs and the quality of documentation and support resources. By continuing to explore the strengths and weaknesses of these protocols, developers can make informed decisions about which one to use in their projects, ultimately leading to better software development outcomes. Moreover, there are specific areas that could benefit from future research. One potential area for future work is to investigate the impact of different hardware configurations and network conditions on the performance of gRPC and HTTP services. Our study was conducted on a single machine and network, and while we attempted to control for potential variations in performance, it's possible that our results may not be representative of all possible scenarios. Conducting additional tests on different hardware configurations and network conditions could provide a more complete picture of the performance of these frameworks in different environments.

Another area for future work is to investigate the impact of different types of payloads on the performance of gRPC and HTTP services. In our study, we used a fixed set of strings for our test scenarios, but real-world applications often use a wide variety of payloads, such as JSON, XML, or binary data. Investigating the performance of these frameworks with different types of payloads could help developers better understand how these frameworks perform under different conditions and make more informed decisions about which library to use for their specific application.

Furthermore, our usability study focused on a relatively small sample size of 10 participants, and while our results provide valuable insights into the perceived usability of gRPC and HTTP libraries, a larger sample size could help to validate our findings and provide even more accurate information about the usability of these libraries. Future research could involve a larger number of participants to obtain a more representative perspective on usability.

Overall, our study provides a solid foundation for future research into the performance and usability of gRPC and HTTP libraries. We hope that our findings will encourage other researchers to continue exploring these important topics, incorporating larger and more diverse samples, investigating the impact of different hardware configurations and network conditions, and considering the influence of various payload types. By expanding on our research, we can enhance our understanding of these protocols and support developers in making informed decisions for their projects.

References

- [1] M. Loukides and S. Swoyer. “Microservices Adoption in 2020.” (2020), [Online]. Available: <https://www.oreilly.com/radar/microservices-adoption-in-2020/> (visited on 2023-03-05).
- [2] A. W. S. Inc. “What are microservices? | AWS.” (2023), [Online]. Available: <https://aws.amazon.com/microservices/> (visited on 2023-03-05).
- [3] H. Krishna. “5 Microservices Examples: Amazon, Netflix, Uber, Spotify & Etsy.” (2021), [Online]. Available: <https://www.sayonetech.com/blog/5-microservices-examples-amazon-netflix-uber-spotify-and-etsy/> (visited on 2023-03-05).
- [4] MediaGenesis. “Understanding Old New Protocols.” (2017), [Online]. Available: <https://mediag.com/blog/goodbye-http-understanding-old-new-protocols/> (visited on 2023-05-10).
- [5] grpc. “About gRPC.” (2023), [Online]. Available: <https://grpc.io/about/> (visited on 2023-03-16).
- [6] N. Chaurasia. “A guide to Agility in cloud computing.” (2023), [Online]. Available: <https://www.sprintzeal.com/blog/agility-in-cloud-computing> (visited on 2023-05-10).
- [7] G. Cloud. “Advantages and Disadvantages of Cloud Computing.” (2023), [Online]. Available: <https://cloud.google.com/learn/advantages-of-cloud-computing> (visited on 2023-05-10).
- [8] Soma. “10 Things to Keep in Mind while Designing and Developing Microservices.” (2023), [Online]. Available: <https://medium.com/javarevisited/10-things-to-keep-in-mind-while-designing-and-developing-microservices-8336ac1b7387> (visited on 2023-05-10).
- [9] J. Montemagno, T. Jain, C. Brent, *et al.* “Communication in a microservice architecture.” (2022), [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture> (visited on 2023-05-10).

- [10] S. Lehrig, H. Eikerling, and S. Becker, “Scalability, elasticity, and efficiency in cloud computing,” *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures - QoSA '15*, 2015. DOI: <https://doi.org/10.1145/2737182.2737185>.
- [11] I. D. Foundation. “What is Usability?” (2023), [Online]. Available: <https://www.interaction-design.org/literature/topics/usability> (visited on 2023-05-10).
- [12] R. Lönn. “Open source load testing tool review 2020.” (2020), [Online]. Available: <https://k6.io/blog/comparing-best-open-source-load-testing-tools/> (visited on 2023-05-07).
- [13] M. Bolanowski, K. Żak, A. Paszkiewicz, *et al.*, “Efficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems,” *IOS Press*, Aug. 2022. DOI: 10.3233/FAIA220242.
- [14] L. Weerasinghe and I. Perera, “Evaluating the Inter-Service Communication on Microservice Architecture,” *IEEE*, Dec. 2022. DOI: 10.1109/ICITR57877.2022.9992918.
- [15] S. T. Arzo, D. Scotece, R. Bassoli, *et al.*, “MSN: A Playground Framework for Design and Evaluation of MicroServices-Based sdN Controller,” *Journal of Network and Systems Management*, Oct. 2021. DOI: 10.1007/s10922-021-09631-7.
- [16] Ł. Kamiński, M. Kozłowski, D. Sporysz, P. Z. Katarzyna Wolska, and R. Roszczyk, “Comparative review of selected Internet communication protocols,” *arXiv*, Dec. 2022. DOI: 10.48550/arXiv.2212.07475.
- [17] Grafana Labs. “Grafana k6.” (2023), [Online]. Available: <https://k6.io> (visited on 2023-05-10).
- [18] Bojan D. “Ghz - grpc benchmarking and load testing tool.” (2023), [Online]. Available: <https://ghz.sh> (visited on 2023-05-10).
- [19] gRPC Authors. “Basics tutorial.” (2023), [Online]. Available: <https://grpc.io/docs/languages/go/basics/> (visited on 2023-05-07).
- [20] Bojan D. “Options Reference.” (2023), [Online]. Available: <https://ghz.sh/docs/options> (visited on 2023-05-07).
- [21] Hubert Bugaj *et al.* “bench.sh.” (2023), [Online]. Available: https://github.com/LesnyRumcajs/grpc_bench/blob/master/bench.sh (visited on 2023-05-07).
- [22] Hubert Bugaj *et al.* “README.md.” (2023), [Online]. Available: https://github.com/LesnyRumcajs/grpc_bench/blob/master/README.md (visited on 2023-05-07).
- [23] Grafana Labs. “Options Reference.” (2023), [Online]. Available: <https://k6.io/docs/using-k6/k6-options/reference/> (visited on 2023-05-07).
- [24] J. Brooke, “Sus – a quick and dirty usability scale,” in Jan. 1996, pp. 189–194.

- [25] Hubert Bugaj et.al. “2022 04 23 bench results.” (2022), [Online]. Available: https://github.com/LesnyRumcajs/grpc_bench/wiki/2022-04-23-bench-results (visited on 2023-05-07).

Appendix A

Usability Experiment

These are the instructions provided to the developers and students:

A.1 Task 1: HTTP Communication

In this task, you will use the `net/http` library to communicate with a service that is running on your local machine. The task is divided into two parts: sending requests and listening to requests. In the first part, you will make a GET request, receive a JSON-encoded response, and then send a POST request back to the server with new data. In the second part, you will edit a server file to respond to a GET request with a specific JSON string and then test it using a web browser or an HTTP client. Feel free to look through the files and read the code in case you get stuck.

Part 1: Sending requests

1. Start the `http/main.go` server and enter the `http/client.go` file.
2. Make a GET request to `http://localhost:8080/getNumber` using the HTTP library in the main function. The response body will be encoded in JSON in this format: `"number: <integer>"`.
3. Create a new JSON string in the same format as above, but double the number that was returned previously.
4. Send the JSON data with a POST request to `http://localhost:8080/checkNumber` (You may have to use `bytes.NewBufferString` to convert the JSON string into a buffer).
5. If you receive the string `"Correct!"` then you are done.

Part 2: Listening to requests

1. With the `http/main.go` server still running, enter the `http/server.go` file.
2. Edit the `retrieveUser` function so that it responds with the JSON string `"id":1969, "name":"MargaretHamilton"`.
3. Run the `server.go` file.

4. Start your web browser or other HTTP client (e.g. Insomnia or Postman) and make a GET request to localhost:8080/checkUser. This will try to make a request to /retrieveUser on your machine on port 9009 so make sure your server is running and listening on the correct port.
5. If you received the string “Correct!” then you are done.

Once you are done with **Task 1**, please take a moment to complete our survey. Your participation is greatly appreciated and the survey should only take a few minutes to complete. Thank you kindly for your time and contribution! Here’s the survey link: <survey link>

A.2 Task 2: gRPC Communication

This task is similar to the previous one but this time you will communicate using gRPC, rather than HTTP. The task is divided into two parts: sending requests and listening to requests. In the first part, you will create a CheckerClient defined in protobufs, make a request to GetNumber, and send the response back to the server after doubling it. In the second part, you will edit a server file to respond to a CheckUser method with a specific user defined in protobufs and then test it using a gRPC client. Feel free to look through the files and read the code in case you get stuck.

Part 1: Sending requests

1. Start the grpc/main.go server and enter the grpc/client.go file.
2. Create a new CheckerClient that is defined in the protobufs.
3. Make a request to GetNumber using the client you just created with the background context and empty pb, and save the reply to a variable.
4. Double the number and send it with a request to CheckNumber.
5. If you receive the string “Correct!” then you are done.

Part 2: Listening to requests

1. With the grpc/main.go server still running, enter the http/server.go file.
2. Edit the RetrieveUser method so that it returns a user with an id of 1969, the name “Margaret Hamilton” and nil for the error.
3. Run the server.go file.
4. Start a gRPC client (e.g. Insomnia, Postman, Ezy, grpcurl, Grip) and call the /Checker/CheckUser method on the server at localhost:7070 using the .proto file located in the grpc/protos folder. The CheckUser method will try to make a request to RetrieveUser on localhost:9009 so make sure your server is running and listening on the correct port.

5. If you receive the string "Correct!" then you are done.

Once you are done with **Task 2**, please take a moment to complete our survey. Your participation is greatly appreciated and the survey should only take a few minutes to complete. Thank you kindly for your time and contribution! Here's the survey link: <survey link>

A.3 HTTP server and client

A.3.1 Server

```
package main

import (
    "net/http"
)

func retrieveUser(w http.ResponseWriter, req *http.Request) {

}

func main() {
    http.HandleFunc("/retrieveUser", retrieveUser)
    http.ListenAndServe(":9009", nil)
}
```

A.3.2 Client

```
package main

func main() {

}
```

A.4 gRPC server and client

A.4.1 Server

```
package main

import (
    "context"
    "google.golang.org/grpc"
```

```

        "google.golang.org/protobuf/types/known/emptypb"
        "log"
        "net"
        pb "survey/grpc/protos"
    )

    type Server struct {
        pb.UnimplementedSenderServer
    }

    func (s *Server) RetrieveUser(_ context.Context,
    _ *emptypb.Empty) (*pb.User, error) {

    }

    func main() {
        log.Println("gRPC_server_started")
        lis, _ := net.Listen("tcp", ":9009")

        s := grpc.NewServer()
        pb.RegisterSenderServer(s, &Server{})
        s.Serve(lis)
    }

```

A.4.2 Client

```

package main

import (
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
)

func main() {
    conn, _ := grpc.Dial("localhost:7070",
    grpc.WithTransportCredentials(insecure.NewCredentials()))

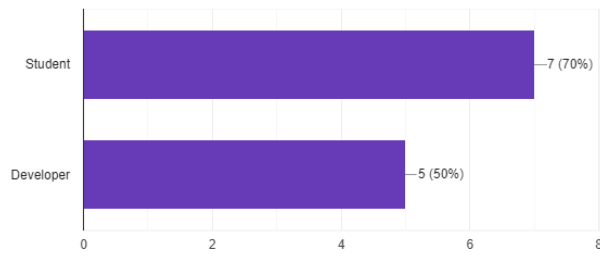
    defer func(conn *grpc.ClientConn) {
        conn.Close()
    }(conn)
}

```

A.5 Survey results

What is your current occupation?

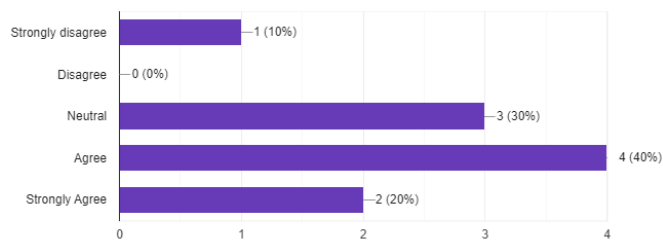
10 responses



((a)) 1

I found the library easy to use.

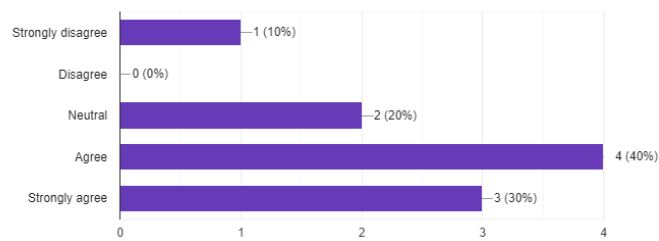
10 responses



((b)) 2

I think that people would learn to use this library quickly.

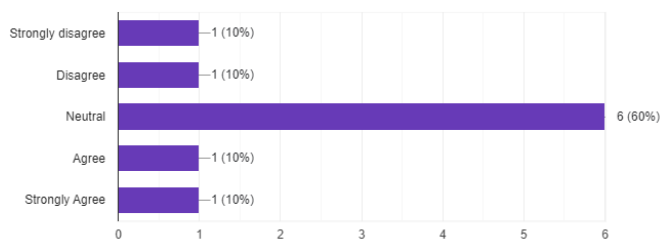
10 responses



((c)) 3

I think that I would like to use this library frequently.

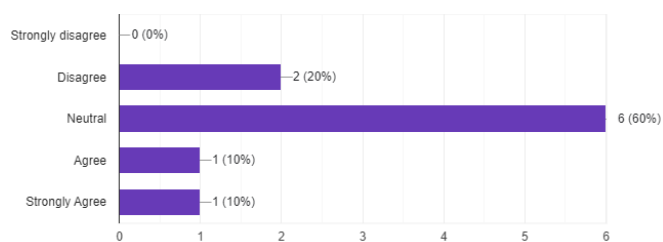
10 responses



((d)) 4

I recommend improvements or changes to the library.

10 responses

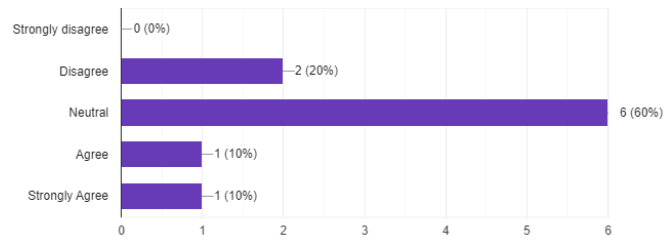


((e)) 5

Figure A.1: Results - HTTP questionnaire Part 1

I recommend improvements or changes to the library.

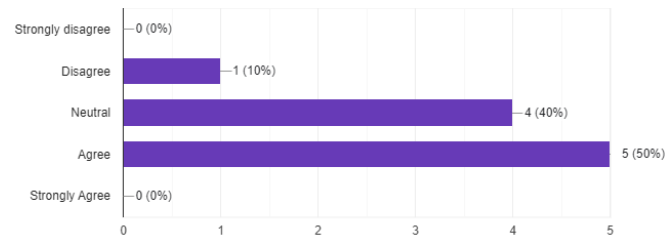
10 responses



((a)) 6

The library had all the features or functions I expected to find.

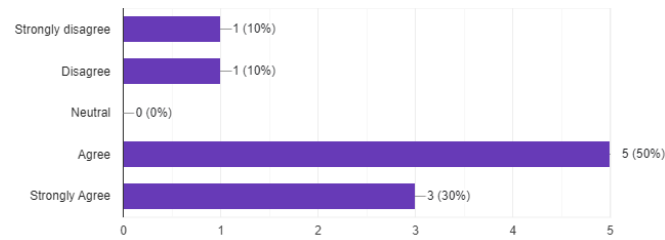
10 responses



((b)) 7

I think I would need support (an expert, guide or documentation) for working with the library.

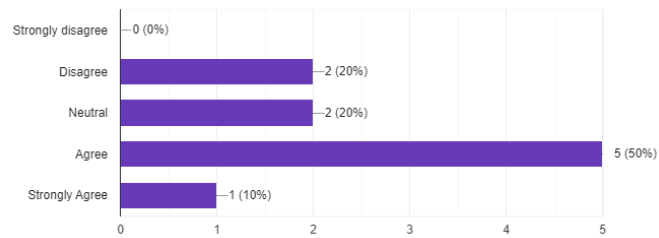
10 responses



((c)) 8

I think that the functions that I utilized from the library are well designed.

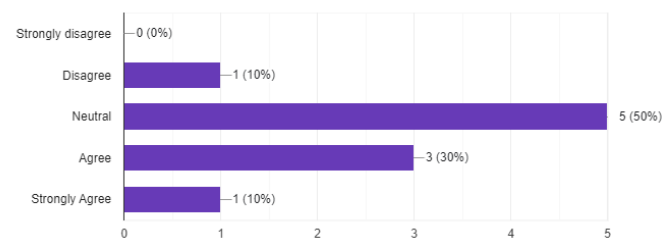
10 responses



((d)) 9

I would recommend using this library to others.

10 responses

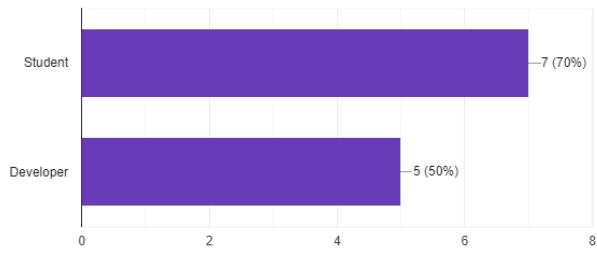


((e)) 10

Figure A.2: Results - HTTP questionnaire Part 2

What is your current occupation?

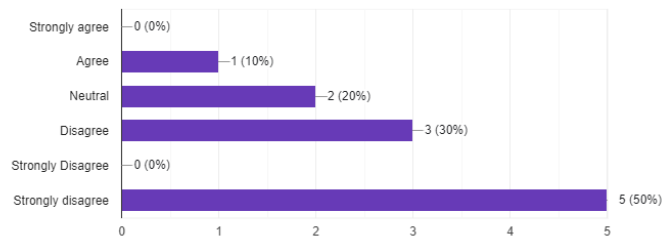
10 responses



((a)) 1

I found the library easy to use.

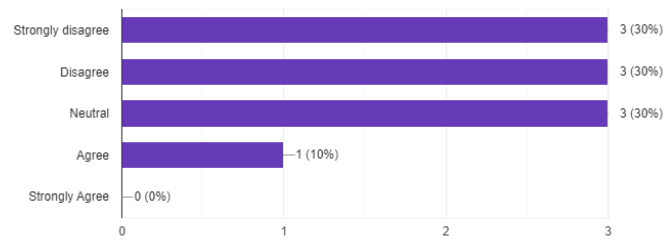
10 responses



((b)) 2

I think that people would learn to use this library quickly.

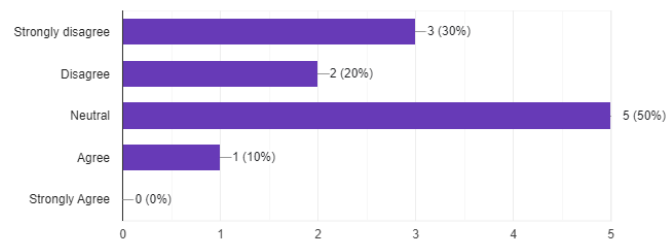
10 responses



((c)) 3

I think that I would like to use this library frequently.

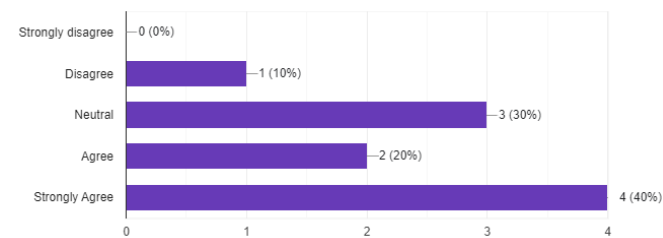
10 responses



((d)) 4

I recommend improvements or changes to the library.

10 responses

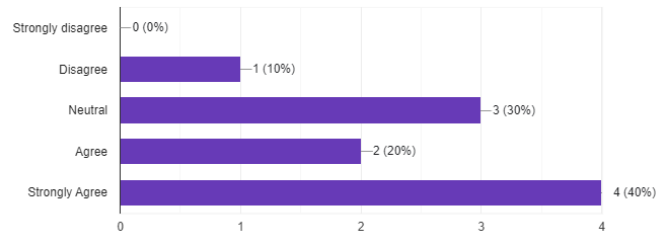


((e)) 5

Figure A.3: Results - gRPC questionnaire Part 1

I recommend improvements or changes to the library.

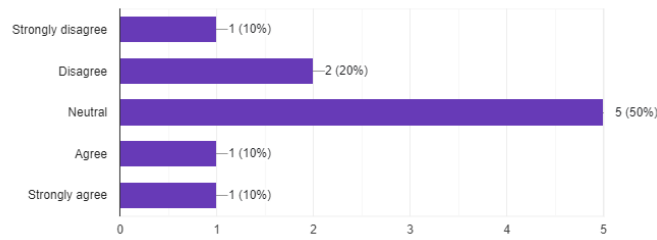
10 responses



((a)) 6

The library had all the features or functions I expected to find.

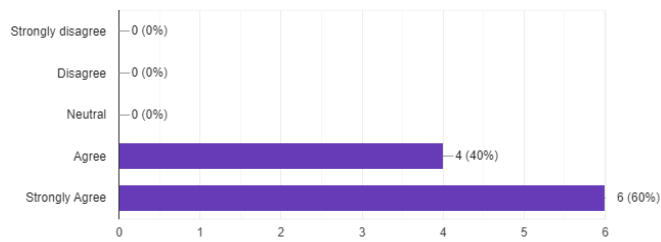
10 responses



((b)) 7

I think I would need support (an expert, guide or documentation) for working with the library.

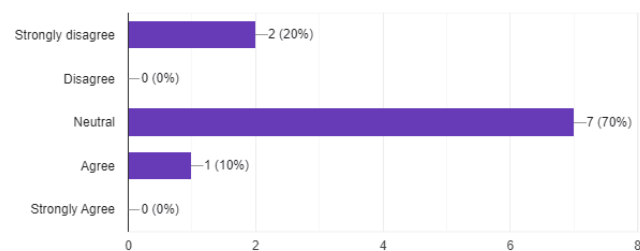
10 responses



((c)) 8

I think that the functions that I utilized from the library are well designed.

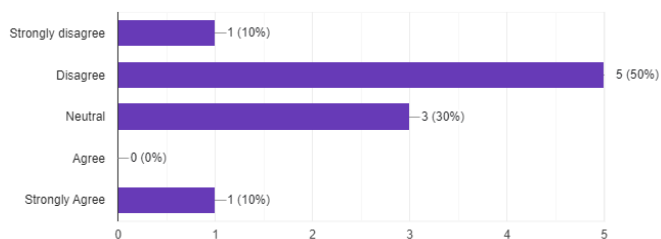
10 responses



((d)) 9

I would recommend using this library to others.

10 responses



((e)) 10

Figure A.4: Results - gRPC questionnaire Part 2

Appendix B

Performance Experiment

B.1 gRPC service implementation

```
package main

import (
    pb "Benchmarks/grpc/experiment"
    "context"
    "google.golang.org/grpc"
    "log"
    "net"
    "os"
)

type Server struct {
    pb.UnimplementedExperimentServer
}

var test1Return = pb.Test1Res{Message: "Hello, World!"}
var test2Return pb.Test1Res
var test3Return pb.Test1Res

func (s *Server) Test1(ctx context.Context, _ *pb.Empty) (*pb.Test1Res, error) {
    return &test1Return, nil
}

func (s *Server) Test2(ctx context.Context, _ *pb.Empty) (*pb.Test1Res, error) {
    return &test2Return, nil
}

func (s *Server) Test3(ctx context.Context, _ *pb.Empty) (*pb.Test1Res, error) {
    return &test3Return, nil
}

func main() {
    log.Println("gRPC server tutorial in Go")
}
```

```

dat1, _ := os.ReadFile("150k.txt")
dat2, _ := os.ReadFile("3M.txt")

listener, err := net.Listen("tcp", ":9090")
if err != nil {
panic(err)
}

log.Printf("150k len: %v", len(string(dat1)))
log.Printf("3m len: %v", len(string(dat2)))

test2Return = pb.Test1Res{Message: string(dat1)}

test3Return = pb.Test1Res{Message: string(dat2)}

s := grpc.NewServer()
pb.RegisterExperimentServer(s, &Server{})
if err := s.Serve(listener); err != nil {
log.Fatalf("failed to serve: %v", err)
}
}

```

B.2 Protobuf definition

```

syntax = "proto3";

option go_package = "./experiment";

service Experiment {
  rpc Test1 (Empty) returns (Test1Res) {}
  rpc Test2 (Empty) returns (Test1Res) {}
  rpc Test3 (Empty) returns (Test1Res) {}
}

message Empty {
}

message Test1Res {
  string message = 1;
}

```

B.3 HTTP service implementation

```

package main

```

```
import (  
    "io"  
    "log"  
    "net/http"  
    "os"  
)  
  
var test2Json string  
var test3Json string  
  
func Test1(w http.ResponseWriter, req *http.Request) {  
    io.WriteString(w, "Hello, World!")  
}  
  
func Test2(w http.ResponseWriter, req *http.Request) {  
    io.WriteString(w, test2Json)  
}  
  
func Test3(w http.ResponseWriter, req *http.Request) {  
    io.WriteString(w, test3Json)  
}  
  
func main() {  
    log.Println("HTTP Server Started...")  
  
    dat1, _ := os.ReadFile("150k.txt")  
    dat2, _ := os.ReadFile("3M.txt")  
  
    test2Json = string(dat1)  
    test3Json = string(dat2)  
  
    http.HandleFunc("/test1", Test1)  
    http.HandleFunc("/test2", Test2)  
    http.HandleFunc("/test3", Test3)  
  
    http.ListenAndServe(":8080", nil)  
}
```