Empirical Comparison of Current M2M Protocols

Ajitaa Jagannathan

Master of Science
School of Informatics
University of Edinburgh
2022

Abstract

Since the beginning of digital computing, machine-to-machine(M2M) communication has been a crucial component of enterprise architecture. Phone notifications available in smart watch, is one such example of M2M communication that almost everyone would have experienced. Several protocols, also referred to as M2M protocols, specify the semantics and syntax of messages transmitted across the network for this M2M communication to take place. The fact that there are so many choices for M2M protocols to choose for M2M communication to take place makes it difficult to choose which protocol to utilise for a given use-case. Unfortunately, there are some cases where programmers pick the incorrect M2M protocol for a specific use-case thereby degrading the application performance. To ensure the system achieves maximum performance, a protocol's advantages, disadvantages and performance in different settings should be considered before adopting it in a system. This study addresses this problem and provides an empirical comparison of three M2M protocols– REST[1], SOAP[2], gRPC[3] by developing the servers for each protocol in two programming languages– Java and Go, thereby adding an additional layer of comparison. The comparison documents the performance of the protocols in different scenarios such as: message size variation, server-side database interaction, concurrency variation, and monitors the network level packet exchange. Apart from these, the soft facts and characteristics are documented and compared for each protocol. From this comparison, developers will be able to fully comprehend the three protocols and its behaviour in different scenarios enabling them to choose the protocol that is most appropriate for a given use case.

[1]REST- Representational State Transfer
[2]SOAP- Simple Object Access Protocol
[3]gRPC- Google Remote Procedure Call

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ajitaa Jagannathan*)

Acknowledgements

I want to express my gratitude to my supervisor, Dr. Michael Glienecke, for igniting my interest in M2M Communication Protocols, guiding me through my dissertation, and patiently answering all of my questions while being incredibly understanding. He offered me a tremendous amount of assistance, and I am very appreciative of that. Additionally, I want to express my gratitude to my loving family, who motivated and supported me ceaselessly from miles away and my friends, who helped me with my effort by being supportive and encouraging.

Table of Contents

Chapter 1

Introduction

The process through which machines communicate with one another in a service based architecture is referred to as machine-to-machine (M2M) communication. An example of M2M communication would be the communication between a smart watch and a smart phone, wherein data like daily activities, heart rate, step count are synced from the watch to the phone, and notifications like text messages, alerts, are sent to the smart watch from the smart phone. The information provider typically receives a request from the requester, processes it, and then returns a response thereby providing the requester or client with some service. This kind of network communication is used in a variety of applications, and it requires precise set of standards and specifications. M2M protocols play this important role by defining the semantics and syntax of messages delivered over networks. M2M communication can be carried out through a variety of M2M protocols.

For M2M communication to take place there are endpoints on the information or service provider, sometimes referred to as the server, that are developed in line with the specifications and guidelines of the M2M protocols. An URI-Uniform Resource Identifier is essentially what an endpoint is, and its usage is implied by the access method. The server or the information provider can be invoked for a particular service by the consumer through the appropriate endpoint and access method. When developed accordingly, an endpoint can be used to build an application architecture that allows interaction between many different types of consumers. IoT (Internet of Things) devices, mobile phones, and computers can all use the same endpoint. When creating an endpoint for a specific use-case, the importance of choosing the most optimal M2M protocol is sometimes overlooked. Before making a decision, one must assess the protocol's advantages and disadvantages as well as understand how it would perform

in different scenarios. Thus, this project seeks to address this issue by offering a thorough comparison of the M2M communication protocols REST (Representational State Transfer), SOAP (Simple Object Access Protocol), and gRPC (Google Remote Procedure Call). This comparison considers the characteristics of each protocol as well as how well it performs in diverse settings. It also features a programming language component in the comparison taking into account Java and Go. The upcoming sections in this chapter detail the aim, motivation, results achieved and the structure of the overall dissertation report.

1.1 Aim

This project aims to provide an empirical comparison of REST, SOAP, and gRPC protocols. This comparison documents the features of each of the protocols and their performance in different settings. Both Java and Go were used in the development of the protocol based servers for this comparison. The goal of this comparison is to provide developers with a thorough understanding of each protocol and assist them in determining which one is most suitable for a particular use-case, rather than identifying which of the three protocols is the best. Since each protocol also takes into account the programming language dimension, the comparison broadens its scope in order to assist developers in selecting the programming language that best suits their needs in addition to the protocol.

1.2 Motivation

Endpoints or interfaces have become more important than ever before as a result of the necessity for digital-first strategy — businesses striving toward organizing in a digital environment[1]. The endpoints are developed following the specification of an M2M protocol. Each protocol has its own set of features, security options, advantages and disadvantages. Multiple research and comparison has been done in the past to compare M2M protocols but, only one or two settings are taken into account for the comparison. This empirical comparison is done taking into account five test settings, namely:

- Message Size Variation
- Server-Side Database Interaction

[1] https://www.redhat.com/en/blog/role-apis-increasingly-digital-world

- Concurrency in Request Load
- Network Level Packet Exchange
- Soft Facts and Characteristics

This would enable a software developer to gain deep insights on the working of the protocols in different settings and choose the most appropriate protocol for a use case.

This project proposal does not require the reader to have prior knowledge on M2M protocols. The reader will gain an understanding of what they are and how and why their performance varies in different test scenarios.

1.3 Results Achieved

For the empirical comparison of current M2M protocols, servers were developed in Java and Go for each of the chosen protocols- REST, gRPC, and SOAP.

For any M2M protocol, there are three different sizes in messages being transmitted namely:

- Small transaction style messages like bookings. Where short requests of a few bytes are sent and short responses are received,
- Medium requests with medium answers within 1Mb size,
- Large requests which are comprised of several Mbs of data.

The comparison has documented the behaviour of the protocols for variation in message size ranging from few kilobytes(kb) to few megabytes(Mb).

The servers developed conforming to a protocol's standards might need to interact with a database by performing a selection query or insertion query. The performance of the server for each of the protocols when a database interaction is involved is recorded in both Java, and Go.

In terms of real world scenarios where an endpoint receives thousands of requests at a time the servers have been tested for concurrent requests upto ten thousand and how the response time varies with respect to concurrency. This is done in both Java and Go to understand how each protocol fares in each language in such a scenario.

The servers developed in accordance to each of the three protocols have different standards for exchanging the payloads, and setting headers. The payloads may be

exchanged in JSON, XML, Binary format, etc and the headers for each protocol vary in size. All of these factors cause difference in network load during communication. Thus, the network load caused by each protocol and the order in which packets are exchanged by each protocol is studied using Wireshark and RawCap network tools.

Lastly, the soft facts for the protocols and their individual characteristics are documented and compared with each other.

1.4 Structure of the Dissertation Report

The structure of the dissertation report is as shown in Figure1.1. The next chapter

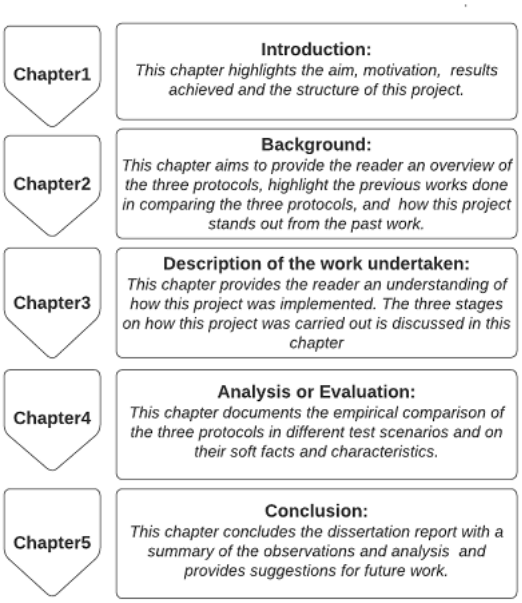

Figure 1.1: Structure of the Dissertation Report

provides the reader sufficient background information in this domain. The reader will be able to understand what each protocol is and how they function. Previous work is also highlighted to the reader, to understand how the dissertation stands out from the previous work. Chapter 3 explains the three stages in which the project was completed. The test definition and development phase is stage 1, the test execution phase is stage 2, and the results documentation and comparison phase is stage 3. In the fourth chapter the test scenarios are described and the results produced are documented and compared. This section is what would help a software developer choose the most optimal protocol for a given use-case. The dissertation's conclusion is presented in the last and final chapter, which also discusses potential directions for further research on this comparison.

Chapter 2

Background

This chapter gives the reader a thorough overview of the project that was undertaken. Readers will have a comprehensive understanding of what M2M communication protocols are, how each of the selected protocols works, and how the comparison was made. Aside from this, the reader will also be introduced to literature review– earlier attempts to compare M2M protocols, and how this study is unique and makes a significant contribution to the M2M communication field.

Data exchange is the process of taking data structured under a source schema and transforming it into a target schema, so that the target data is an accurate representation of the source data. Data exchange allows data to be shared between different computer programs[5]. Data exchange was the driving source for the development of M2M communication. The earliest form of data exchange was physical, where data would be downloaded to magnetic tapes and transported to a different system. As years passed information was transferred digitally via network wires and phone lines using all-purpose protocols like Telnet, SMTP, FTP, and http[1]. Based on http are M2M protocols like REST, gRPC and SOAP which are taken into account this comparison. An endpoint implemented on a server following the standards of the M2M protocol chosen, when programmed accordingly, can be used by multiple physical devices or clients – mobile phones, IoT devices, and computers. There are several M2M protocol options to choose from for an endpoint. In order to gain deep insights into choosing the ideal protocol for a particular use-case, this comparison compares and contrasts the performance and features of the three selected protocols under various scenarios.

[1] https://www.redhat.com/architect/apis-soap-rest-graphql-grpc

2.1 REST

A one-tiered software application that combines the user interface and data access code into a single program from a single platform is referred to as a monolithic application[2]. In contrast to the traditional monolithic design, there is an SOA (service-oriented architecture) architectural approach which is used in software engineering[3]. The flexibility, scalability, and reliability of these services, gives an organization more flexibility when developing business modules or services. Applications are loosely coupled as a result; REST, SOAP, gRPC are all one of many possible communication protocols in a SOA-system.

Representational State Transfer is the abbreviation for REST. Roy Fielding used this phrase for the first time in his doctoral dissertation [6]. It is an M2M protocol for developing stateless, reliable online APIs. RESTful is a colloquial term used to denote a web API– an application programming interface for the Web[4] that adheres to the REST specifications and transfers data through the http protocol. REST is being used throughout the IT industry.

A REST request includes an http method, an endpoint(URI[5]) which can have optional parameters as well separated by the '&' symbol, headers, and an optional body based on the http method[6]. The endpoint provides access to a resource and the http method defines what is to be done to the resource. The various http methods are:

- GET to Retrieve a resource,
- POST to Create a resource,
- PUT to Update a resource,
- DELETE to Delete a resource,
- PATCH to make partial changes to an existing resource[7].

http transmits UTF-8 messages. The headers are key-value pairs, the payload body of a message can be in any pre-defined format and the header field 'Content-Type' indicates how to interpret the payload body. Possible values for Content-Type would

[2]https://en.wikipedia.org/wiki/Monolithic_application
[3]https://collaboration.opengroup.org/projects/soa-book/pages.php?action=show&ggid=1314
[4]https://www.w3schools.com/js/js_api_intro.asp
[5]URI- Uniform Resource Identifier
[6]http Method GET usually has an empty body
[7]https://www.rfc-editor.org/rfc/rfc5789

be application/json for the case of JSON and text/xml for the case of XML. Usually, the request and response bodies in REST use JSON—JavaScript Object Notation type, which allows data objects to be represented as attribute-value pairs and arrays (or other serializable values) as shown in Figure2.1. Status codes are issued by the server along


```
{
    "message": "You have been successfully allocated a team",
    "teamDetails": {
        "member1": "Alice",
        "member2": "Derek",
        "member1id": 113517,
        "member2id": 113520,
        "courseCode": "CS31245",
        "groupId": 879253
    }
}
```


Figure 2.1: Example of JSON formatted text

with the response for a request that the client makes to indicate the result for the request. Figure2.2 explains what each status code that begins with a particular prefix implies,

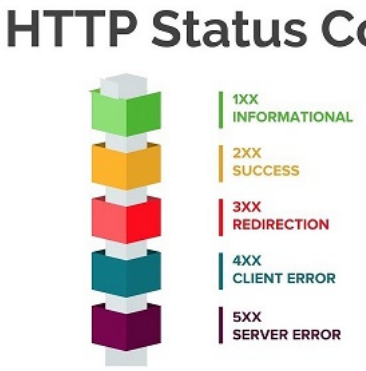

Figure 2.2: http Status codes and what they mean[10]

for instance, response code 200 means that processing was successful.

2.2 SOAP

SOAP abbreviated as Simple Object Access Protocol evolved from XML-RPC and uses the XML format for sending and receiving data. XML-RPC is a remote procedure

call (RPC) protocol which uses XML to encode its calls and http as its transport mechanism[15].

Extensible Markup Language (XML)[8] is a markup language and format for storing and transmitting data in a structured way. It mainly comprises of angular brackets which represent a tag, as shown in Figure2.3. It was first made available in 2000 and like

Figure 2.3: Example of XML formatted data

REST, SOAP is also based on the http protocol. The client and server are referred to as soap nodes, and an XML-based soap message is sent back and forth between them. The XML message's enclosing element designating it as a SOAP message is called the SOAP Envelope. The message body is contained in the SOAP Body and header details are contained in the SOAP header.

In order for the client to use a SOAP service's method and learn more about it, a unique file within SOAP called the WSDL (Web Services Description Language) file can be generated to provide information about the service and how it operates, including the service's name, the methods it offers, and the parameters and return values of each method. The schema information representing the relationships between the attributes and elements of the XML objects being exchanged by the service calls are present online, inline or as a dedicated file which is linked inline in the XSD (Xml Schema Definition). It describes the arguments and their types, fields, and any limitations on those fields (such as a maximum length or a regex pattern), among other things. Using these XML schema informations the code for the parameters is auto generated, which saves development time and effort. SOAP is not only supported in http but can be also transmitted through different methods and protocols like SMTP, JMS or message queues. When a SOAP URI is accessed over the http protocol, the server responds with response codes as shown in Figure2.2.

[8]https://www.w3.org/TR/xml/

2.3 gRPC

Google Remote Procedure Call, commonly known as gRPC, is the most recent RPC technique of machine-to-machine communication. Unlike SOAP, gRPC is significantly more recent; it was introduced in 2015, approximately 15 years after SOAP. Like REST and SOAP, gRPC is likewise built on the http protocol, but it uses http/2.0, which preserves everything of http/1.1's high-level semantics, including methods, status codes, header fields, and URIs. What is novel about this approach is how the data is packaged and sent between the client and the server[8]. Compared to http/1.1, http/2.0 is both faster and more reliable. While http/1.1 loads a single request for every TCP connection, http/2.0 decreases network delay by using multiplexing, allowing for the asynchronous delivery of all requests via a single connection between the client and web server. Another difference noted in http/2.0 is that it does not support chunked transfer encoding that http/1.1 supports[16]. There are two modes of gRPC communications, namely:(1)Unary RPCs where the client sends a single request to the server and gets a single response back, just like a normal function call, (2) Streaming RPCs where there can be Client, Server or Bidirectional mode of streaming. The data transmitted in gRPC is binary and much more faster than http/1.1.

Similar to SOAP, gRPC has a special file known as the .proto file which defines the contract of gRPC services and messages. From this file, gRPC frameworks generate a service base class, messages, and a complete client. By sharing the .proto file between the server and client, messages and client code can be generated from end to end which results in time being saved in the development phase.

2.4 Literature Review for the Comparison of Current M2M Protocols

Research attempts for comparing these M2M protocols are limited in number for gRPC, however for REST vs SOAP there are multiple literature works. The research works studied for this comparison are detailed in descending order of relevance.

- In[12], Kumar et al compares and tries to improve the performance of REST, gRPC, and Thrift using shared memory between the services, using Unix Domain Sockets along with appropriate NUMA[9] tuning. In this study, two settings were

[9]NUMA: Non-Uniform Memory Access

built: (1) Localhost Baselining, in which the client and server applications were both hosted on the same host, and (2) Docker containers on the same host using the network stack. In the first half of the study, Kumar et al. make an effort to show how REST, gRPC, and Thrift respond differently when the number of calls is kept constant at 10,000 while the payload varies from 100kB to 1000kB. They then try implementing shared memory between the services, using Unix Domain Sockets along with appropriate NUMA tuning to boost performance, and record the results. As a result, this study does not focus on comparing the protocols thoroughly(only documents how the protocols perform for varied message size),and the reader is not provided with a comprehensive comparison of the protocols.

- In[21], by creating applications in .NET 5.0 platform with identical functionality in each of the three protocols, REST, GraphQL, and gRPC are compared. This research work is published in Polish language, and Google translate is used to attempt to understand what has been done. Measurements and comparisons were made of parameters including the function's processing time, the number of transactions per second, and the amount of data processed. In this study, the protocols are compared solely based on how the database size and operations affects an application rather than on other factors like concurrency, significant message size variation, and soft facts like usability, learnability, etc.

- The work done by Bolanowski et al in [2] is similar to the first test scenario in our project– comparing the performance of the REST and gRPC when the message size varies. However, the maximum size of message taken into consideration is only 3.4Mb, whereas in our comparison upto 25Mb is taken into account. This project also does not look at concurrency, or database interaction or the network layer for REST and gRPC.

- The work done by Potti et al in [20] aims to compare the working of REST and SOAP services when they interact with the database. Functionally similar services are developed in REST and SOAP to compare the response time and throughput. Similar to this are the works done by Tihomirovs et al in [23], Pavan et al in [11], and Kumari et al in [13] which compare REST and SOAP web services for functionally similar applications developed in REST and SOAP.

- In [22], Soni et al compare REST and SOAP web services primarily on soft facts

like architecture, security, reliability, efficiency and development and architectural style. They consider one hard fact– response time for database interaction. Similarly in [9]Halili et al compare the soft facts of REST and SOAP.

- In [18] Mulligan et al, an attempt is made to evaluate both implementations of REST and SOAP with an emphasis on their performance with regard to both efficiency and scalability taking into consideration factors such as latency, network packet bytes transmitted, and performance during concurrency situations.

- It is seen that Castillo et al in[3] also compare REST and SOAP in the client server setup, as well as master-slave setup, which are both two different setup styles incorporated in the study.

- Including gRPC with REST and SOAP, Chamas et al in [4], focuses on the energy consumption of protocols like REST, SOAP, Socket, and gRPC in mobile phones. This is evaluated when algorithms of various complexities, various input sizes and types, are executed on mobile devices. The authors put forward four research questions out of which only one focuses on the communication protocols and if they influence energy consumption in mobile devices. Different sorting algorithms like bubble sort, selection sort, heap sort was implemented and run using the three protocols to document how each protocol performed for energy consumption. Similarly REST and SOAP are compared in Android mobile phones by Bloebaum et al in [1]. Apart from this, in research works like[17] Malik et al aim to compare REST and SOAP in indoor actuator networks.

One important finding from the previous works above is that each comparison focuses on a particular scenario or maximum two but, not more and gRPC is not taken into consideration by most of the previous literature. However, the empirical comparison undertaken takes into consideration four scenarios and soft facts making it stand out from previous literature work, as highlighted in the next section.

2.5 Implementation of the Comparison and How it Stands Out From Previous Work

In the implementation of the empirical comparison of current M2M protocols four test scenarios detailed below and multiple soft facts are taken into consideration. The

development of the server is done in two programming languages– Java and Go taking into account a new dimension for the comparison– programming language. Each of the four test scenarios and soft facts are highlighted along with why the implementation of the comparison stands out from previous research work:

- Message size variation and how each protocol functions when the payload size varies is documented. This is done in [12],[2],[4] but the maximum size of request is within 5Mb, whereas in the comparison performed it is documented for upto 25Mb of data transmission in REST, SOAP and gRPC.
- Server side database interaction and how the protocol servers perform are recorded. This is seen only in few of the previous works such as [21],[20], [23], [11], [13][22].
- Variation in the server's request load, and how each protocol functions when the volume of requests it receives varies is documented for 10000 requests. Concurrency is not taken into account in most of the previous work except in [18] which considers upto 100 concurrent requests.
- The network load, and packet exchange is monitored and compared for each protocols. In [18], the size of bytes transmitted are taken into consideration for REST and SOAP but, not on what packets are transmitted.
- The comparison is done taking into account two programming languages– Java, Go and also compares and contrasts soft facts for each protocol. Most of the studies only take into account the hard facts, but the characteristics and how each protocol varies from the other are omitted except in [22] and [9] which compares the soft facts for REST and SOAP.

Thus, this comparison has collated and compared the performance of the protocols in multiple test scenarios and has also taken into consideration the soft facts of the protocols. Previous work may have done one or two of the above tests in their research but not all of the above tests. Hence, this empirical comparison would provide software developers a deep insight of the performance of REST, SOAP and gRPC in various scenarios and assist them in selecting the most suitable protocol and programming language for a given use case. The next chapter will highlight how the above work has been undertaken.

Chapter 3

Description of the work undertaken

This section details the work undertaken for the project. By developing a server and client for each of the M2M communication protocols in Java and Go, an empirical comparison of the current M2M communication protocols has been conducted. The work done for this comparison may be broken down into three stages as shown in Figure3.1: (1) defining test scenarios and the development process, (2) testing the protocol implementations, and (3) documentation and comparison.

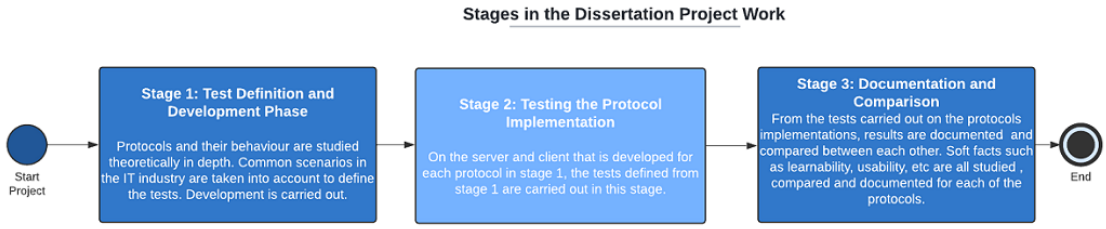

Figure 3.1: Stages in the Dissertation Project

The sections that follow in this chapter go into great detail on the work done in each stage, the challenges and problems encountered in each stage, and the solutions for the challenges.

3.1 Stage 1: Test Definition and Development Phase

This was the first phase in the project work of comparing the current M2M communication protocols.The iterative and incremental development approach was used in this phase to construct the server and client code for the protocols, define tests, and

conduct in-depth protocol research. The incremental development model divides the final application into completely functional pieces that are referred to as increments. In each iteration an increment is developed and added to the application. This is also referred to as the Agile development model[1]. Figure3.2 shows how stage1 was done using the iterative and incremental development approach where an initial study was done to get the background of these protocols, after which tests were defined for which the server and client code was developed. Following this, further study was done based on ideas generated by me and my supervisor. From this study the cycle repeated itself until we reached the time limit to proceed to stage 2 of executing the tests.

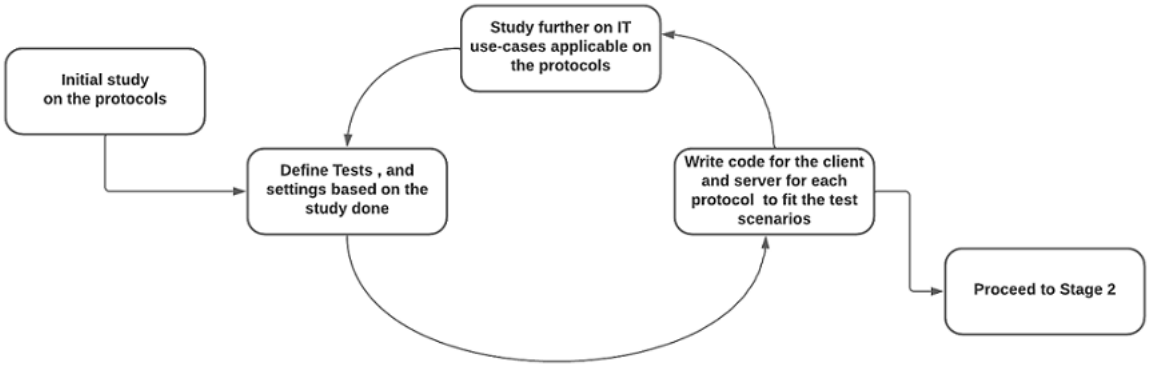

Figure 3.2: Iterative and Incremental Flow Diagram for Stage 1

3.1.1 Implementation in this Phase

As discussed in previous sections, the code for the client and server for each of the protocols is developed in Java as well as Go. In this stage, the set up of the environment was done on a Windows PC.

For Java, which is a general-purpose programming language designed to enable programmers to write once and run anywhere (WORA)[14], the SDK(Software Development Kit) version 16.0.2 along with Eclipse version 2022-06 (4.24.0) IDE is used. Spring Boot framework and Maven3.8.6 for managing dependencies is setup for the project. For Go, version go1.18.3 along with Visual Studio version 1.69.2 IDE is used.

After both programming languages and their IDEs were set up, extensive research was done on REST, SOAP, and gRPC and how they should be implemented in Java and Go. After reading[7], it was clear how the protocols' performance might be evaluated.

[1]https://medium.com/@ashutoshagrawal1010/agile-methodology-incremental-and-iterative-way-of-development-a6614116ae68

This book provided insight on how to use techniques like RED– Request rate, Errors, Duration for a given test are measured and compared. From this learning, different test scenarios were identified that could be run on the protocol's server implementation. Post this, the development of the client and server for the protocols was done in Java and Go to facilitate the running of tests. This process of research and development was iteratively done until it was time for Stage 2 which focused on the running and reporting of these tests.

The test settings and scenarios identified as a result of three iterations of this phase are:

- Message Size Variation and the Performance of the Protocols: This scenario was targeted at recording how the protocols perform when they received requests of sizes varying from few hundred kbs to 25mb. Generally, requests to an endpoint can be classified to three types based on their size namely: (1)Short requests which are of a few bytes or minimal Kbs, these requests are for short transactions like booking tickets, updating fields in an account, etc (2)Medium sized requests which are of a few Kilobytes in size within 1Mb and (3)Large sized requests which are of a few Mb, which may involve large files like uploading photos, pdfs, etc. This test scenario aims to test REST, SOAP and gRPC when message sizes are varied and record how each protocol performs in terms of response time.

- Server-Side Database(db) Interaction: This scenario was targeted to monitor how the protocols perform when the server needs to interact with the database. This is a very common use-case in the IT industry, as almost all applications interact with databases to insert and retrieve data. When the server needs to interact with the database, how each protocol performs and behaves is recorded.

- Concurrency and Load on the Server: In this scenario, the endpoints of each protocol are tested to see how they behave when the server is under concurrent load, that is, when it is receiving anything between 10 and 10,000 requests at a time. How the protocols behave in both the programming languages is taken into account and reported.

- Network Load by the Protocol Communication: In this test scenario, the communication between the server and client is monitored in the network level for the three protocols. The sequence of packets exchanged, and the total bytes transmitted are carefully recorded and compared.

- Programming Language Environment & its Performance: This scenario is to observe the memory statistics and run-time environment of Java and Go for the protocols, and record how the language's environment performs when the server of a certain protocol receives requests. It is targeted to aid in choosing the right programming language for a use case where memory of the programming language plays a role as cost in the application.

As stated above this phase was carried out in three iterations, where in each iteration a test scenario was found. In each of the iterations, once the test scenario was found the development for that scenario was carried out in Java and Go. Going by the order of test scenarios found in the three iterations, the development done in each iteration is documented in Figure3.3.

Iteration	Test Scenario	Development Done in Java and Go
1	*Message Size Variation*	Endpoint to accept short sized message of a few hundred bytes with the request payload containing a mix of integer and string data types.
		Endpoint to accept medium sized message of 1 Mb with the request payload containing a mix of integer and string data types.
		Endpoint to accept and process large sized message of upto 25Mb with the request payload containing a file (.pdf/.png/.jpeg/etc)
		Client developed to invoke large sized message endpoint of gRPC server
2	*Server-Side Database Interaction*	Endpoint to accept short sized message and interact with PostgresSQL database to perform selection and insertion operations.
		Endpoint to accept short sized message and perform processing without database interaction.
3	*Concurrency, Network Monitoring, Environment Monitoring*	Endpoints configured to handle concurrency. Endpoints integrated to Prometheus and Grafana.

Figure 3.3: Development done in each iteration

3.1.2 Challenges and Solutions

The initial setup for Java and Go had quite a few challenges– Maven setup, and GoLang integration with Visual Studio. A few forums where answers to problems were discovered from discussions with other developers who had same setup challenges include Stackoverflow[2], Java's Oracle Community[3], and the Go developer community[4]. Finding

[2]https://stackoverflow.com/
[3]https://community.oracle.com/tech/developers/categories/13287-java
[4]https://go.dev/help

test cases was first a little tricky, but after reading [7], various performance measurement techniques became evident. Through the use of these techniques, test scenarios were found by consulting the supervisor, earlier literary works, and those discussed in section 2.1.4. The development process during the three iterations was the most challenging part in the whole project. The REST protocol was well documented, and a strategy could be easily identified in Java and Go from the documentation and developer forums. Implementation required some effort and extensive self-learning due to the lack of easily available documentation for gRPC in Java SpringBoot. The JAXB dependency, sometimes referred to as the Java Architecture for XML Binding, enables Java developers to quickly incorporate XML data and processing activities in Java programs (JAXB). For SOAP in Java, this dependency is required. But as JAXB became depreciated after Java 11, SOAP had to be developed in Java 11. Due to the extremely low number of resources available for Go, it took a great deal of trial and errors to get the SOAP server to function.

3.2 Stage 2: Test the Protocol Implementations

This project phase combined the TDD- Test Driven Development with Agile to understand the test, review the test, develop the tests in the relevant clients and execute the tests, shown in Figure3.4.

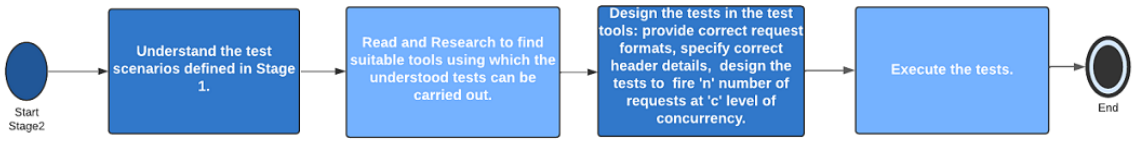

Figure 3.4: Individual phases of Stage2

The test scenarios and server configurations for each scenario were thoroughly understood, and extensive web reading was done to identify appropriate tools and software to test the selected scenarios on the protocols. Tools like Postman, Apache Bench, GHZ, Wireshark, RawCap, Prometheus, and Grafana were discovered from the research done. Reading[25] gave thorough information on endpoint testing using Postman which is a software tool that is the client to invoke an endpoint in the server. Upon invoking the server through Postman, we can see the response code, body, status, and time sent by the server. Figure3.5 shows a sample request sent to a REST endpoint using Postman.

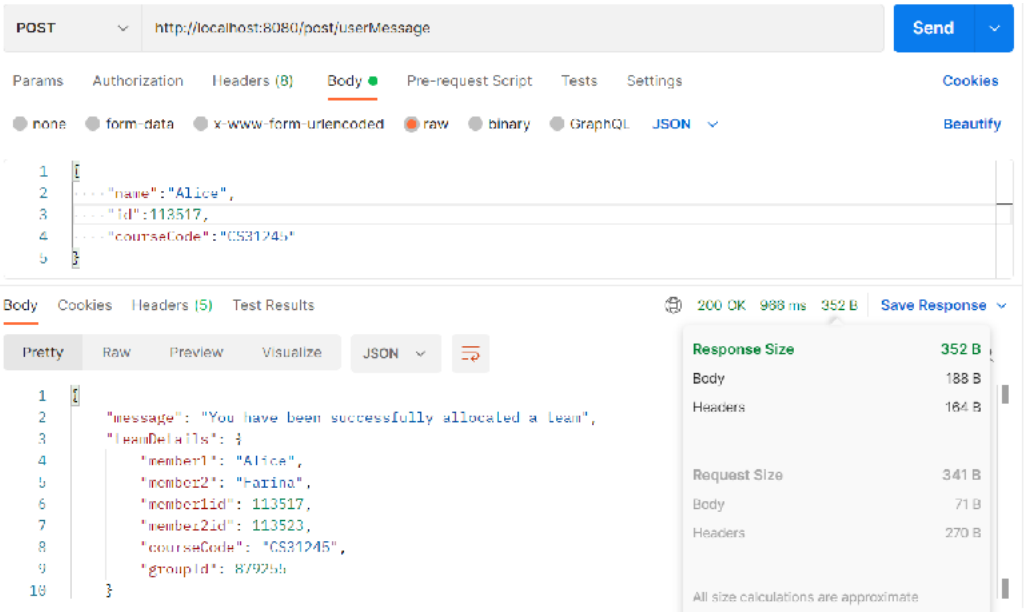

Figure 3.5: REST request sent to server using Postman Tool

To perform volume varying tests on the server Apache Bench which reports the results of tests is shown on Figure3.6 is used for SOAP and REST, and GHZ for gRPC. They are both command line computer programs used for benchmarking the http protocols. It helps in load testing the endpoint by sending 'n' number of requests where the concurrency 'c' level can be specified. This tool helps in testing the protocols when concurrency levels vary – test scenario defined in iteration3. For recording the network

```
Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /post/userMessage
Document Length:        187 bytes

Concurrency Level:      10
Time taken for tests:   1.403 seconds
Complete requests:      1000
Failed requests:        661
   (Connect: 0, Receive: 0, Length: 661, Exceptions: 0)
Total transferred:      292483 bytes
Total body sent:        222000
HTML transferred:       187483 bytes
Requests per second:    712.92 [#/sec] (mean)
Time per request:       14.027 [ms] (mean)
Time per request:       1.403 [ms] (mean, across all concurrent requests)
Transfer rate:          203.63 [Kbytes/sec] received
                        154.56 kb/s sent
                        358.19 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.5      1       5
Processing:     4   13   6.8     11     141
Waiting:        3   12   6.8     10     138
Total:          5   14   6.8     12     141
```

Figure 3.6: Apache Bench invoking REST 1000 times with concurrency level 10

level data exchange, Wireshark and RawCap were setup and used. For obtaining metrics from the programming language's environment we use Prometheus– "an open-source

systems monitoring and alerting toolkit" [5]. It helps in recording real-time metrics in a time series database (allowing for high dimensionality) with flexible queries[24]. This enables us to monitor metrics of the environment of Java, and Go. To visualise the metrics from Prometheus, we use Grafana– an open source, cross-platform online application for interactive visualisation and analytics. When connected to supported data sources, it offers charts, graphs, and alerts for the web. The metrics of Java, and Go are recorded using Prometheus and visualised via a Grafana interface that is interactive and visually appealing. Figure3.7 displays a Grafana interface with Prometheus metrics incorporated into it, including the metrics of Java Heap committed and Java Heap used.

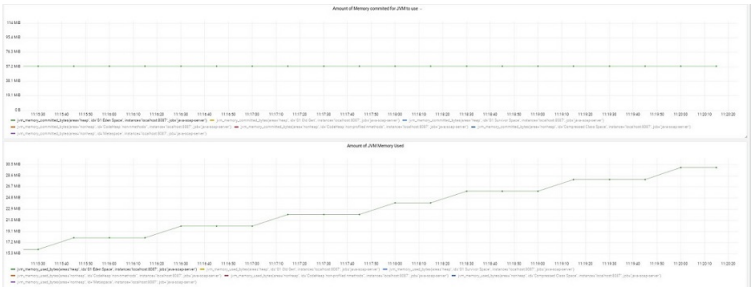

Figure 3.7: Grafana Interface with integrated Prometheus Metrics

3.2.1 Implementation in this Phase

Postman, Apache Bench, GHZ, Wireshark, RawCap, Prometheus and Grafana were setup and used to send requests to test message size variation, server side db interaction, and concurrency. Apache Bench is used for REST, SOAP and GHZ for gRPC to simulate 10000 requests on the server with concurrency varying from 0 to 10000 with an increase of 500. The RED methodology of performance measurement: request rate, errors and duration of the test was recorded. For the network level data transmission capture, RawCap with Wireshark was setup and packets were analysed. To record the performance of the environments and metrics of Java and Go, Prometheus and Grafana were used.

3.2.2 Challenges Faced and Solutions Adopted

Postman, Apache Bench, GHZ, WireShark, RawCap Prometheus and Grafana were downloaded, and installed on the system. Minor setup issues like the application not

[5]https://prometheus.io/docs/introduction/overview/

opening up, the Grafana and Prometheus integration configuration issues were all found and resolved using the documentation available. The GHZ tool for testing gRPC was found after extensive search. Capturing the local packets on Wireshark was not working initially, after which research was done and RawCap[6] a network sniffing tool was found to help in monitoring the packets.

3.3 Stage 3: Documentation and Comparison

This stage is the last and final stage in the project undertaken. All test results and test scenarios from Stages 2 and 1 of the project are documented during this stage. To identify trends in performance and behaviour in the examined test scenarios, the findings from the tests for each protocol are documented and compared to one another. This will present to a software developer how the protocols function in the test scenarios in turn allowing them to weigh the pros and cons of each protocol for a scenario to choose the most appropriate protocol for a use case. Apart from the previously specified test scenarios, each protocol's soft facts, characteristics and features supported are noted and compared in the comparison. The result of this stage is the empirical comparison that is provided in the next chapter of the report.

3.3.1 Implementation in this Phase

The implementation in this phase mainly focused on documenting the results obtained and performing research to include soft facts, features supported and characteristics of each protocol. A lot of figures, charts and graphs have been drawn and plotted using Google Sheets and Rawgraphs[7] software.

3.3.2 Challenges and Solutions

Given the time limitations, it is extremely difficult to take into account all the qualities and soft facts; as a result, the most interesting and practical information that would help a developer choose a protocol for a use case is taken into consideration.

[6] https://www.netresec.com/?page=RawCap
[7] https://www.rawgraphs.io/

Chapter 4

Analysis or Evaluation

This chapter presents the results obtained and the critical analysis done on the protocols for different test scenarios. Three servers are built in Java and Go following each of the protocol standards. Client requests are made, and servers answer in accordance with these standards. The client and server are both present on the same system, with the specifications mentioned in Table4.1. For SOAP and REST, Postman acts as a client for

Feature	**Device Specification**
Processor	11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
RAM	16.0 GB (15.8 GB usable)
System Type	64-bit operating system, x64-based processor
Operating System	Windows 11 Version 21H2

Table 4.1: System Specifications on which the Protocols are Evaluated

single requests, and Apache Bench for concurrent or a sequence of 'n' requests. For gRPC, a client program is developed in Java and Go to invoke the respective gRPC servers. For a sequence of requests or for concurrent requests GHZ is a command line tool that helps in invoking gRPC server. In the upcoming sections of this chapter, the comparison will be made on various test scenarios, on the network level and upon the soft facts and characteristics of each protocol.

4.1 Comparison Based on Message Size Variation

The performance of the protocols is highlighted in this section for varying sizes of message payloads transmitted between the client and server. Small messages have

payloads that are only a few bytes in size, medium messages have payloads that range from a few kilobytes to one megabyte, and large communications have payloads that are several Megabytes in size. A functionally similar application with three endpoints, each accepting messages of a specific size range, is constructed for each protocol in Java and Go. To each endpoint accepting a message of certain size is passed the same parameters in the request body regardless of the protocol. A series of 1000 requests are passed for the small and medium messages using Apache Bench for REST and SOAP and GHZ for gRPC. For the large message type 1 message is sent using Postman for REST and SOAP and the developed client program for gRPC in Java and Go. The comparison is recorded based on the RED Methodology[7]– reporting the Request rate, Errors and Duration for each of the tests. The initial paragraphs would discuss the duration or the total response time for each of the tests and finally the errors and request rate per second would be discussed.

Firstly, the protocols are tested for short message communication. The request is sized at an approx of 200 bytes with the request body containing fields populated with String(mixture of alpha numeric characters) and Integer(numerical) data types in this scenario. The total response time for processing 1000 requests of short message size through REST, gRPC, and SOAP is documented in Table4.2 and compared using the line graph shown in Figure4.1. From the table it is clear that REST performs the best

Protocol	**Test Duration in Java [seconds]**	**Test Duration in Go[seconds]**
REST	0.909	0.343
gRPC	2.17	0.392
SOAP	4.139	0.357

Table 4.2: Total Test Duration for 1000 Short Messages Serviced

in Java and Go when compared to gRPC and SOAP for the case of short messages of payload size of few hundred bytes. This could be due to the JSON mode of transmission which is faster than XML, and the caching ability which is not present in gRPC and SOAP[1]. With gRPC as choice 2 and SOAP as choice 3 in Java and the exact opposite in Go, the second and third protocols of choice are dependent on the language. In terms of language, Go performs faster than Java. This is due to Go's superior multithreading that

[1]https://stackify.com/soap-vs-rest/

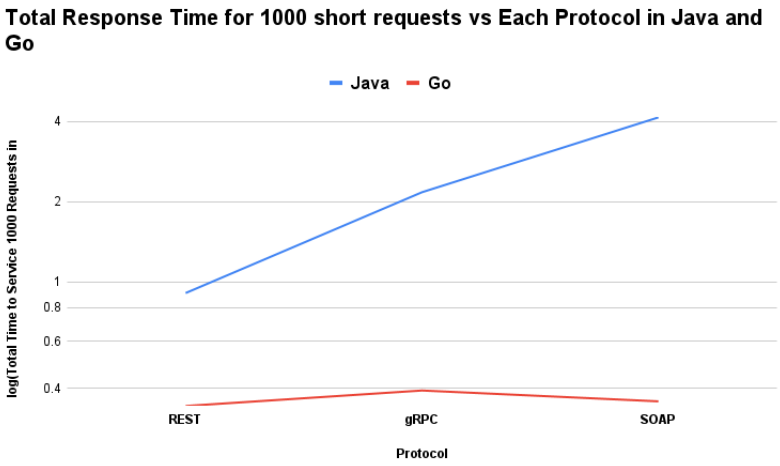

Figure 4.1: log(Total Response Time for 1000 short requests) vs Each Protocol in Java and Go

speeds up the processing[2].

In the case of medium message transactions in REST, SOAP and gRPC the message was sized at 1Mb, and the performance of the protocols when 1000 requests are sent to the servers are recorded in Table4.3 and compared in Figure4.2. The request payload fields have a mix of String and Integer data types sizing to 1Mb. For the test of medium

Protocol	**Test Duration in Java [seconds]**	**Test Duration in Go[seconds]**
REST	9.688	6.164
gRPC	1.95	0.475
SOAP	20.438	30.158

Table 4.3: Total Test Duration for 1000 Medium Messages Serviced

messages, it is seen that the rankings of the protocol in terms of response time is the same for both Java and Go in the order of gRPC, REST and SOAP. gRPC is the fastest, and this could be accounted to the http/2.0 protocol that supports binary transmission of data using protobuf. Binary transmission is usually light on the network and easy to parse, hence it responds fastest compared to REST and SOAP[3]. REST is ranked second, followed by SOAP, and this is due to the fact that SOAP is heavy weight, verbose, and textually dense to parse compared to REST. Thus, for endpoints that expect medium

[2]https://www.ideamotive.co/blog/go-vs-java-similarities-differences-and-business-applications
[3]https://blog.restcase.com/http2-benefits-for-rest-apis/

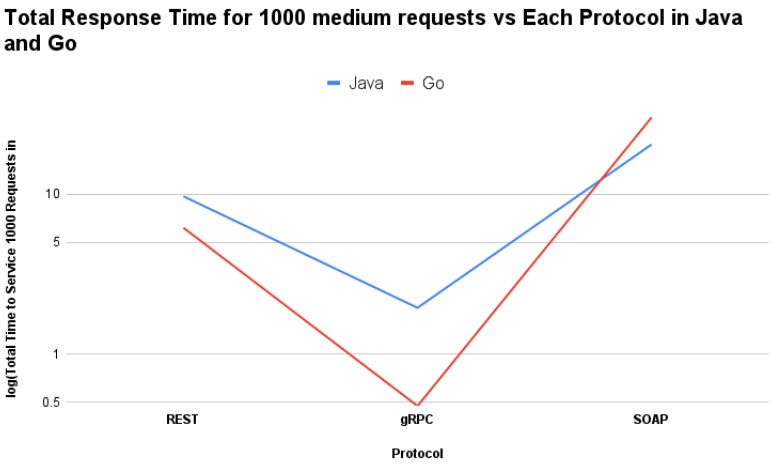

Figure 4.2: log(Total Response Time for 1000 medium messages) vs Each Protocol in Java and Go

sized messages the protocol of choice would be gRPC, followed by REST and SOAP. In terms of programming language, the performance of gRPC, and REST are better in Go, but for SOAP Java is more suited. This could be because Go has very few packages for SOAP servers and further research must be done to support SOAP better in Go[4].

Finally, in the case of large messages where endpoints are expected to receive several Mb of data, the protocol to be implemented must be chosen carefully as the network tends to face high load in such a scenario. A file of 25Mb is transferred from the client to the server using REST, gRPC and SOAP protocols in this experimental setup. This setup is different from the previous two scenarios because only 1 request is sent to the server, instead 1000 requests.

The results in terms of response time for large sized messages is reported in Table4.4 and compared in Figure4.3. The ranking of each protocol in both the languages is

Protocol	**Test Duration in Java [seconds]**	**Test Duration in Go[seconds]**
REST	0.969	0.28
gRPC	0.322	0.162
SOAP	1.05	0.937

Table 4.4: Total Test Duration for 1 Large Message Serviced

[4]https://pkg.go.dev/github.com/globusdigital/soap

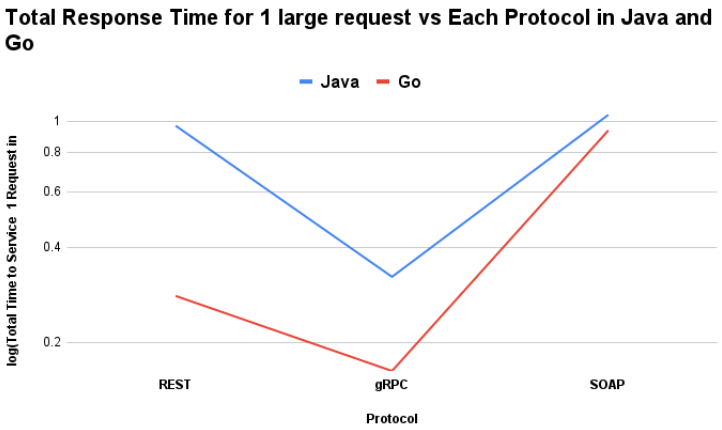

Figure 4.3: log(Total Response Time for 1 large request) vs Each Protocol in Java and Go

the same as the previous scenario of medium sized messages with gRPC ranked first, followed by REST and SOAP. gRPC outperforms REST by 3 times and SOAP by 3.2 times in Java and 1.75 times faster than REST, and 5.7 times faster than SOAP in Go. In terms of language, Go performs better than Java for all the protocols. The speed of gRPC could be accounted to the client side streaming feature that it supports- where the client sends a stream of messages to the server not just one single message. Thus the large file is read and written at the same time– the client reads few bytes and sends it as a stream to the server where the server writes it, and this process is repeated till the file is read completely. This feature of client side streaming is not supported by REST and SOAP where the server expects only a single message, thus, the client has to read the whole file and then send the whole file's bytes with base 64 encoding to the server for processing. Another reason why gRPC is much faster than SOAP and REST would be due to the fact that the file data is sent as binary by http/2.0 whereas in SOAP and REST it has to be base64 encoded. When given, the use case of large message transmission upto a few megabytes gRPC would be the first choice opted for, independent of the language. For the purpose of streaming larger sized files like videos in applications like Youtube, Vimeo other http protocols like DASH[5], and RTMPS[6] are used.

According to the RED methodology, the duration of the tests have been discussed in the above paragraphs, the Errors are found to be 0 for all the protocols in both the languages for the given scenarios. This may not be the case in a real world scenario

[5]DASH-Dynamic Adaptive Streaming over http
[6]RTMPS- Secure form of Real-Time Messaging Protocol

as, many more public networks maybe involved. The request rate per second for each protocol shares a positive correlation with the test duration and is recorded in Figure4.4.

	Request Rate in Java [per second]			Request Rate in Go [per second]		
Protocol	*Short Message - 200 bytes*	*Medium Message- 1Mb*	*Large Message- 25Mb*	*Short Message - 200 bytes*	*Medium Message - 1Mb*	*Large Message- 25Mb*
REST	1100	103.22	1.03	2915	162	3.57
SOAP	241	48.93	0.95	2802	33.16	1.06
gRPC	460	512	3.1	2551	2105	6.17

Figure 4.4: Request rate per second for each protocol considering message size variation

4.2 Comparison Based on Server-side Database(DB) Interaction

In a number of use case scenarios, the developer must develop a server that communicates with the database. The server may need to read data from the database, write data to the database, or in certain situations do both. The objective of this analysis is to contrast how different protocols function when they interact with databases and when they do not. In this case, PostgresSQL is the database configured, and the messages that are sent to the server's endpoint are short in size containing both String and Integer data types. The server reads data from one database table, which has about 100 rows, and inserts data into another table, which contains more than 10,000 rows. Thus performing both read and write operations. 1000 requests are sent to the server, and RED-recommended metrics are logged. Another endpoint which accepts similar payload without db interaction is also developed so that the results may be compared.

According to the RED metrics the Request rate per second for each protocol when database interaction is involved and not is compared in Figure4.5. There are no errors when the test is performed for all the protocols in both the languages, for both endpoints. The total test duration shares a positive correlation with the request rate, and the results for total test duration are shown in Figure4.6 and reported in Table4.5.

From the graph in Figure4.6 it is clearly seen that when there is database interaction the response time is higher than when not. This is because interacting with the database is computationally expensive as operations such as maintaining a connection pool, accessing the data from the table and inserting the data to the table are involved. The

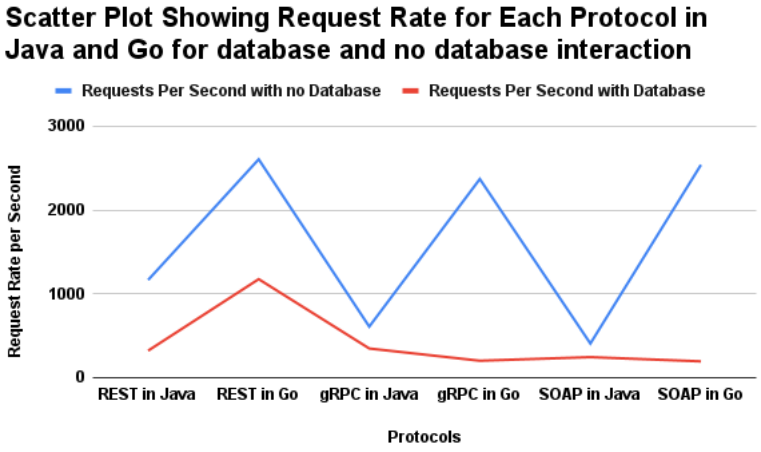

Figure 4.5: Request Rate per Second for Each Protocol in Java & Go when Database is integrated and Not Integrated

Protocol	Test Duration in Java with Db[seconds]	Test Duration in Java without Db[seconds]	Test Duration in Go with Db[seconds]	Test Duration in Go without Db[seconds]
REST	3.097	0.858	3.171	0.384
gRPC	2.873	1.642	4.94	0.422
SOAP	4.068	2.456	5.14	0.394

Table 4.5: Total Test Duration for the Protocol Servers when Databases are involved and Not

total response time for the protocols vary in both the languages, and this is due to the fact that each language has its own database interaction management algorithms implemented.

When there is database interaction involved Java appears to perform better than Go for each of the protocols with gRPC ranked first followed by REST and SOAP. Whereas in Go, REST ranks first followed by gRPC and SOAP in the last. SOAP is uniformly ranked last in both the languages and this could be due to SOAP's payload being textually intense, and heavy on the network, making the parsing process take longer than in REST and gRPC. From this test, the protocol of choice for a server with database interaction would be gRPC in Java and REST in Go. The next section performs further investigation when database interaction is involved including the concurrency

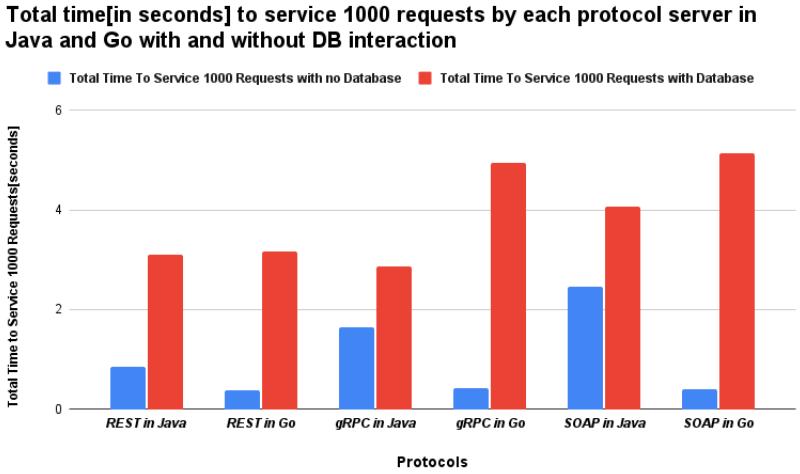

Figure 4.6: Bar graph showing the response time for each protocol with and without database interaction, when developed in Java and Go.

perspective.

In the case of no database interaction Go appears to surpass Java by being twice as fast for REST, 3.8 times fast for gRPC, and 6.2 times fast for SOAP. In both Java and Go, REST is the protocol of first preference as it has caching, clustering and load balancing mechanisms[19]. For gRPC and SOAP, the rankings are 2, 3 in Java and 3, 2 in Go. In both scenarios for database interaction, no errors were reported during the test execution.

4.3 Comparison Based on Request Load Concurrency

If an endpoint is consumed by many clients potentially thousands of clients at a time there would be an increase in load on the server, and a probable delay in response. In such a situation, the right protocol choice plays a key role. This is because each protocol has its own style and standards for parsing, transmitting and compressing the requests and providing the response. They may also have their own set of unique features accounting for the variation in response time, one such feature being the streaming feature offered by gRPC that enables the client and server to communicate through streams, thereby making reading and writing happen simultaneously and in-turn producing the response very quickly in certain situations. However, REST and SOAP do not support this feature causing an increased response time in some scenarios where streaming possibly reduces the response time.

This scenario test aims to compare the protocols on how they vary in terms of response time when the concurrency load is at a certain level. 10000 requests are sent to each server in total, for different levels of concurrency. The levels of concurrency varies from 10 to 500 with 50 increment in each test[10,50,100,150,200..500] and then 1000 to 10000 with an increase of 1000 in every test. The comparison is done involving three dimensions– where the dimension of focus is concurrency, followed by programming languages and presence/absence of database integration in the server. Thus, the protocols are compared in Java and Go for database and no database integration for varying levels of concurrency.

First, we run the tests on Java where we develop three servers each for REST, gRPC and SOAP. In each server are two endpoints one which interacts with the database and another which doesn't. The results for response time when there is varying level of concurrency on a server with no DB interaction in Java is shown in Figure4.7. From

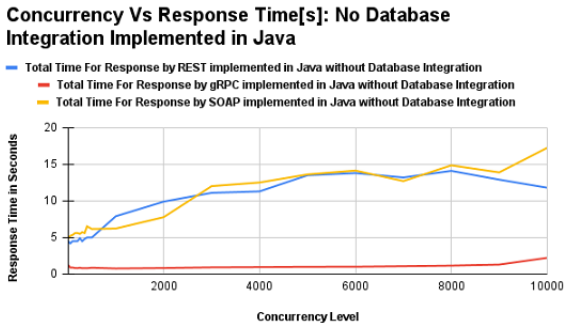

Figure 4.7: Line Graph Comparing the Response Time against Concurrency for Each Protocol when Implemented in Java without DB Interaction

the graph it is clearly seen that for all the protocols as concurrency increases for a fixed number of requests, the response time also increases. Out of the three protocols, gRPC has the fastest total response time with a range of 0.772-2.22 seconds. REST and SOAP fall within ranges of 4.19-13.1 seconds and 4.8 - 17.2 seconds respectively. Thus, in such a given scenario gRPC would be the protocol of choice followed by REST and SOAP. In the case of database interaction, the results are shown in Figure4.8 and it is clearly seen that the ranking in terms of response time is gRPC, REST, and SOAP. The rankings for both database and no database interaction is the same in Java, with gRPC topping the list and being the first protocol of choice for applications that would expect to see high levels of concurrency. This is followed by REST and then SOAP. In both the scenarios there are no failures.

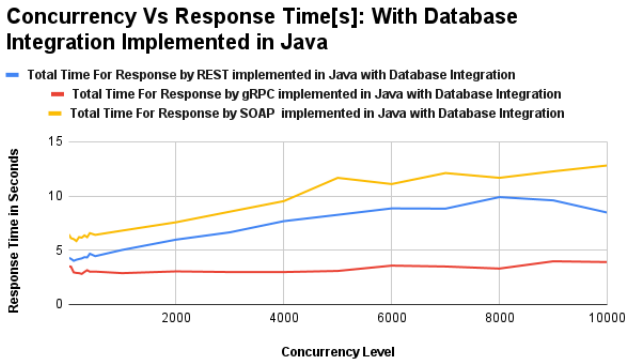

Figure 4.8: Line Graph Comparing the Response Time against Concurrency for Each Protocol when Implemented in Java with DB Interaction

For the protocol servers developed in Go, gRPC leads the list for the case of no database interaction, and REST and SOAP share almost similar values with a difference in response time of no more than 0.5 seconds at specific levels of concurrency, with REST performing better at such levels. No failures are observed in this scenario.The results are documented and compared in Figure4.9.

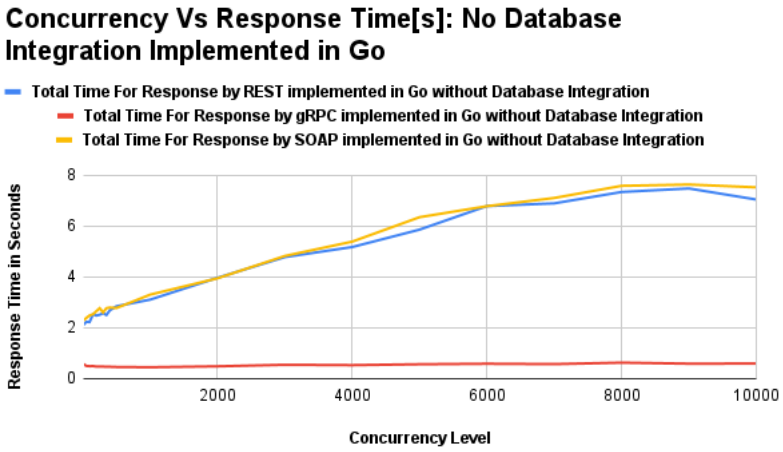

Figure 4.9: Line Graph Comparing the Response Time against Concurrency for Each Protocol when Implemented in Go without DB Interaction

For protocol servers developed in Go with db interaction, requests were serviced only upto a threshold of 200 concurrency level. This was due to the fact that PostgreSQL by default can not handle multiple clients beyond a certain threshold at a time, and Go's threadpool has to be manually configured to help in addressing this issue[7]. Post this

[7]http://go-database-sql.org/connection-pool.html

configuration change, it is seen that initially upto a concurrency level of 250 gRPC was the slowest. However, post this as concurrency was increased gRPC was seen to perform the fastest compared to REST and SOAP by being upto 20times faster than REST and 25 times faster than SOAP in certain levels. SOAP on the other hand, performs faster than REST upto the concurrency level of 6000 beyond which REST goes on to become upto 1.2 times faster than SOAP, this could be accounted due to the XML payload of SOAP causing high network load and increased parse time. The results of this are shown in the line graph in Figure4.10, where it is seen that the best performing protocol in terms of fastest response time accepting high concurrency load is gRPC followed by SOAP when concurrency is less than 6000, and REST if concurrency is higher than 6000.

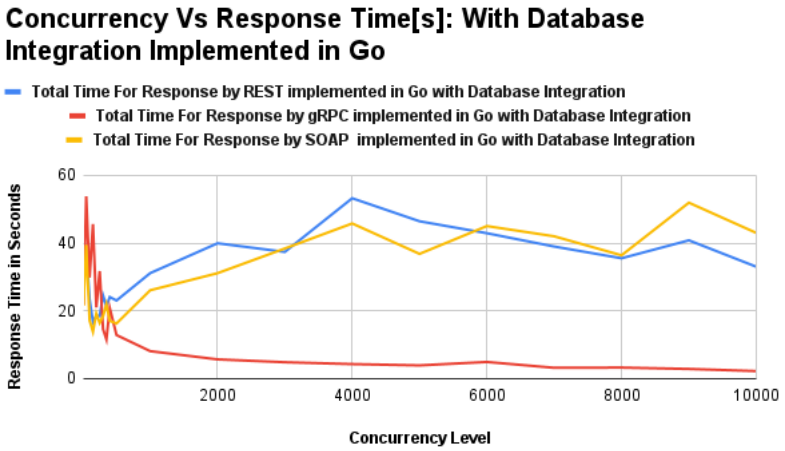

Figure 4.10: Line Graph Comparing the Response Time against Concurrency for Each Protocol when Implemented in Go with DB Interaction

Overall, it can be observed that gRPC is the clear winner in the use-case of an endpoint that receives a lot of concurrent load, followed by REST when a database is not involved. If a database is involved, the second and third protocol of choice would be REST followed by Go. For a server in a high concurrency situation without database interaction, Go would be the preferred language, whereas Java for a server in a high concurrency situation with database interaction.

4.4 Comparison Based on Network Level Packet Exchange

Each protocol produces varying load on the network. This section analyses the network load for each of the protocols when it is implemented in Java. The client for REST and SOAP is Postman, and for gRPC the client is a programmed Java client. The size of the message exchanged is short and the request body is within few hundred bytes. To monitor the network load caused by each protocol, Wireshark and RawCap network monitoring and sniffer tools are used to capture the packets exchanged. http/2.0 wire format is more efficient due to multiplexing, compression and binary protocol transmission instead of http/1.1 which is textual[8]. All http protocols are layered over TCP - Transmission Control Protocol applicable in the transport layer of the network stack. Between applications operating on hosts connecting through an IP network, the TCP protocol ensures reliable, ordered, and error-checked transmission of bytes of data[9]. The client establishes a connection with the server through a three way handshake connection: the client sends a SYN flag to server performing the active open, in response to this the server replies with a SYN,ACK flag indicates that the server has acknowledged the receipt of SYN from the client and is okay to proceed with the connection, finally the client sends an ACK back to the server in response to the SYN,ACK flag. After this handshake, data is sent between the client and server and ACKs are exchanged between each other acknowledging the receipt of data to the sender. There is another flag known as the PSH flag which is sent by the sender to the receiver indicating that the data can be pushed to the application directly, to which the receiver responds with an ACK.[10]. Post this data exchange, keep alive flags are sent between the client and server to ensure the connection is not broken, or RST,ACK maybe sent to abort the connection. The following paragraphs will detail the network exchange seen in http/1.1 Protocols– REST, and SOAP followed by http/2.0 Protocol– gRPC.

In the case of REST, gRPC and SOAP the request body fields are populated with the same values to ensure the request body is of the same size across the protocols. This setup enables us to measure how the network load varies for each protocol independent of the request body sent– for each protocol the headers are differently sized, the request

[8]https://blog.restcase.com/http2-benefits-for-rest-apis/

[9]https://en.wikipedia.org/wiki/Transmission$_{C}ontrol_{P}rotocol$

[10]https://packetlife.net/blog/2011/mar/2/tcp-flags-psh-and-urg/

body for gRPC and REST is in JSON but for SOAP it is XML. For the case of gRPC based on http/2.0 protocol, additional packets such as Magic, Settings and Ping are transmitted. This may cause difference in the number of packets and bytes exchanged between the client and server for each of the protocols. From Wireshark it is observed that the SOAP has the maximum amount of bytes exchanged followed by REST and gRPC.

In SOAP, the total number of bytes exchanged including the overhead of the protocol headers for Ethernet, IP and TCP, is 1737 bytes with a tcp three way handshake, 1 client packet (request) of 705 bytes, 1 server packet (response) of 808 bytes, and TCP flags of sizes varying between 40 to 52 bytes. The exchange captured on Wireshark for SOAP is shown in Figure4.11. The XML payload, which is extremely verbose with XML tags and an XML Envelope results in long parse and serialisation times, making the protocol heavyweight on the network. After the exchange is done, TCP Keep Alive packets are sent back and forth between the server and client to keep the connection active.

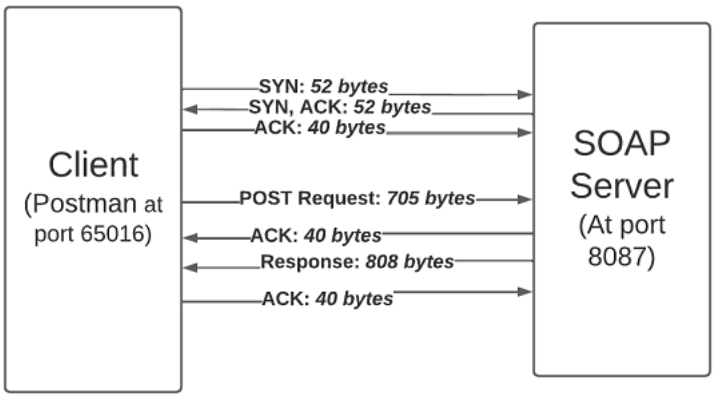

Figure 4.11: Network Level Packet Exchange for SOAP as seen on Wireshark

In the case of REST, the total bytes exchanged are 1090 including the overhead of the protocol headers for Ethernet, IP and TCP which is 1.5 times less than the total bytes exchanged by SOAP. This might be explained by the fact that JSON packaged data is more compact than XML, as well as by the fact that REST does not require additional headers and envelope like SOAP. The 1090 bytes of data transmitted can be broken down into the following components: 1 request packet of length 381 bytes, 2 chunked server responses with a total of 445 bytes, and additional TCP flags ranging in size from 40 to 52 bytes. The chunked server responses is what makes the REST communication different from SOAP, and is highlighted in Figure4.12. The PSH, ACK message sent by the server to the client contains the response data and indicates that it may be pushed to

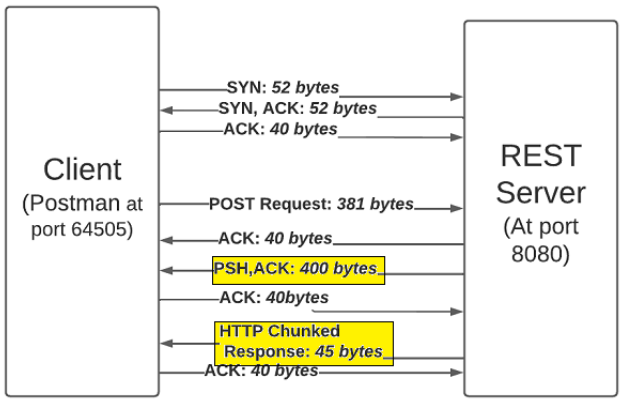

Figure 4.12: Network Level Packet Exchange for REST as seen on Wireshark

the client's application before the server actually sends the http response with response status. This is due to the chunked transfer encoding feature allowed in http/1.1, which allows content to be sent in chunks and explicitly signals when it has finished so that the connection is accessible for the subsequent http request/response[11]. Following this exchange, it can be seen that the client and server exchange a few TCP Keep Alive packets to retain the connection, much like SOAP. Thus in http/1.1, when comparing REST and SOAP in terms of network load, REST is the winner.

gRPC is based on the http/2.0 protocol and shares a few similarities with the http/1.1 protocols of REST and SOAP with the three way handshake of exchanging SYN, SYN-ACK, ACK messages before starting the connection. Every http/2.0 connection starts with the client sending a Magic Message that mimics an http/1.1 message after the three-way handshake in gRPC. This allows http/1.1 servers to reject the connection and respond with an http/1.1 response, informing the client to switch back to http/1.1, indicating that the server does not support http/2.0. Although it's formally designated as the Connection Preface, people commonly call it the "Magic" message[12]. Following the Magic Message, the server and client exchange a packet known as the SETTINGS frame. The SETTINGS frame communicates configuration data that determines endpoint communication, such as preferences and peer behaviour restrictions. Following the end of the SETTINGS frame exchanges, the request is delivered to the server. Additionally, ping packets are transferred back and forth between the server and client to determine the shortest round-trip time and see if the idle connection is still functional. The messages specific to gRPC are highlighted in Figure4.13. In contrast to REST, the

[11]https://en.wikipedia.org/wiki/Chunked_transfer_encoding
[12]https://www.rfc-editor.org/rfc/rfc7540#section-3.5

server responds with headers and response status before the data once the server's processing is complete. Another way that gRPC differs from REST and SOAP is that the connection is terminated once the client receives the response and a few pings have been sent and received. Figure4.13 shows the gRPC network interaction as captured on Wireshark. The total number of bytes including the overhead of the protocol headers for Ethernet, IP and TCP exchanged in this communication is 1547 including the 3 way handshake, magic message, settings frames, request, response and pings.

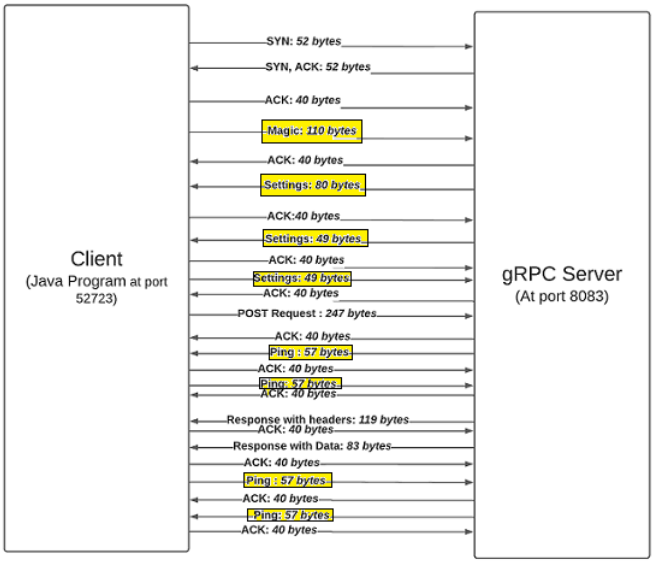

Figure 4.13: Network Level Packet Exchange as seen on Wireshark with gRPC specific packets highlighted

The overhead of the Ethernet, IP, and TCP protocol headers is included in the bytes recorded for the aforementioned three protocols. When only the TCP data segment is considered, the values provided by Wireshark reveal that gRPC has the least bytes (525), followed by REST (706), and SOAP (1433), as seen in Figure4.14.The http/2.0 binary protocol, on which gRPC is based, is the most lightweight protocol on the network when compared to REST and SOAP because it is easier to parse, has less overhead, and is binary. When only TCP data segment is taken into consideration gRPC is better than REST and SOAP. However when Ethernet, IP and TCP are considered gRPC comes second. This could be due to the multiple pings, and settings, magic frames exchanged by gRPC which does not happen in REST or SOAP. In contrast, SOAP is the most resource-intensive when compared to REST and gRPC. This could be explained by the XML tags which are very verbose, and intense to parse.

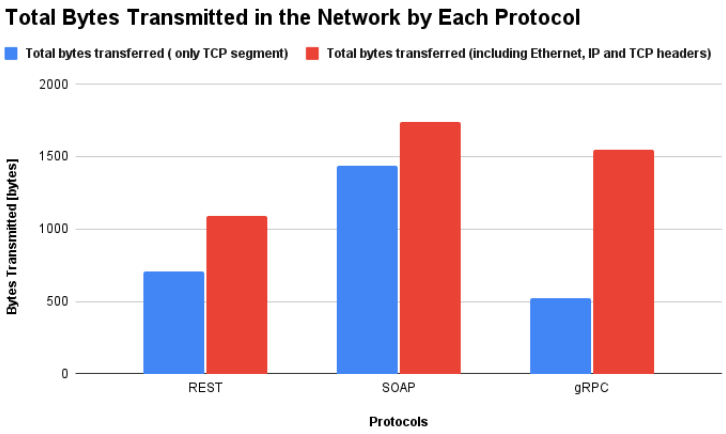

Figure 4.14: Bytes Transmitted in the Network by Each Protocol

4.5 Comparison between Java and Go

For REST, gRPC and SOAP the comparison is done between the Java and Go in terms of how the runtime environment functions when these programs are executed. A series of 1000 requests are sent to the server which is not connected to the database to observe the statistics. The memory metrics in Java and Go for the protocols are language dependent more than being protocol dependent that is– the question becomes simply Java Vs Go instead of REST in Java Vs SOAP in Java Vs gRPC in Java and same for Go. Hence, a brief analysis is done to document the findings. The total amount of memory allocated for the heaps by the system are 7.6-14Mb for Go and 158Mb for Java. This shows that Go is much more light weight compared to Java. The number of bytes of heap in use and idle for Go vary between 4-9Mb and 2.8-7.8Mb. For the case of Java the number of bytes in heap that is in use vary between 27.6 - 70.7 Mb, and idle heap bytes vary between 84-130.4Mb. This shows that Go uses at least 28% of the heap that is allocated whereas Java uses only 11% of the total heap allocated. Go appears to be more memory-efficient than Java, which needed a lot of memory allocation to implement the protocols. This might be a major factor in Go's apparent speed advantage over Java when server-side database interaction is not there.

4.6 Comparison on Soft Facts

This section aims to compare and contrast the soft facts, and characteristics of each protocol. This comparison is recorded in Table4.6 and Table4.7.

Feature	REST	SOAP	gRPC
Code Generation	Third Party Tools like Swagger	Built in WSDL for code generation	Built in protoc for code generation
API Contract	Loose/ Optional-Open API	WSDL contract can be generated on the fly	Strict and requires .proto contract
Security	TLS/SSL	TLS/SSL and provides web services security through XML digital signature, encryption, and security tokens	TLS/SSL
Streaming	No	No	Allows for bidirectional and unidirectional streaming
Payload and its Learnability	JSON is easy to learn and lightweight: data is passed as key value pairs	XML is tougher than JSON but, more powerful as well and helps reduce software risk.	JSON is easy and lightweight, but in order to communicate the proto file is required to generate the code, this is both an advantage(code is generated if proto is available) and a disadvantage(difficult to compile)
Development Community in Java and Go	https://dev.to/t/rest	https://dev.to/t/soap	https://grpc.io /community/

Table 4.6: Table Comparing the Soft Facts and Features of Each Protocol

Feature	REST	SOAP	gRPC
Message Formatting Type	JSON	XML	Protocol Buffers with JSON or XML to serialize data
Format for Parsing	Textual	Textual	Binary. Have lower overhead to parse, and is lighter on the network.
Browser Support	Yes	Yes	No. However, a workaround for this would be to build a gRPC web proxy that acts as a middleware between the browser and the backend gRPC server. The proxy could communicate with the browser using http 1.1 and communicate with the backend gRPC server using http 2.0.
Network Load and Speed	Lighter and Faster than SOAP when only TCP data segment is considered. When Ethernet, IP and TCP are considered REST is the lightest compared to gRPC and SOAP.	Due to the XML format used for data exchange and the additional XML tags that are carried over the network, network load is likewise very high. Therefore, as compared to REST and gRPC, it is the slowest.	When compared to REST and SOAP, the binary format used for data interchange speeds up data transmission over networks. When only TCP data segments are taken into account, they are the lightest; however, when Ethernet and IP data segments are taken into account, they are the second lightest due to the inclusion of additional packets like Settings, Ping, and Magic.

Table 4.7: Table Comparing the Soft Facts and Features of Each Protocol

Chapter 5

Conclusion

The goal of this chapter is to offer some final thoughts, observations, and suggestions for future work. As previously stated, this empirical comparison seeks to give software developers a reading guide to assist them in reading and comprehending how each protocol functions in various scenarios and different settings, as a result, help them choose the best protocol for a use-case.

5.1 Observations and Analysis

For the case of message size variation analysed in section4.1 , given a use-case where a developer needs to choose a protocol where the messages transmitted are short in size, REST performs the best. REST surpasses gRPC in this case because it does not transmit the extra packets that gRPC does, such as Magic, Settings, and Ping, and because the payload itself is small and light on the network with JSON being easy to serialize. However, gRPC triumphs in the case of medium and big messages. This is due to the fact that gRPC is based on the http/2.0 protocol, which is quicker than http/1.1 protocols as it uses header compression to reduce overhead and a binary protocol (Protobuf) rather than a text-based one to make the network load lighter. Even though, the payload itself is heavy and there are additional frames in gRPC, binary transmission clearly beats textual transmission for network intensive messages making gRPC a clear winner. The fact that XML, which is textually dense and very challenging to interpret, is used for data exchange makes SOAP the slowest performing protocol. As a result, responses from SOAP take longer to process. In terms of language, Go performs better than Java.

When databases are involved as seen in section4.2, it becomes clear that Java manages the connection pool and database connection parameters better than Go from

a programming language perspective by default. It can be seen that gRPC for Java and REST for Go are ranked top after 1000 requests are sent to the server for processing. However, it was found that gRPC outperformed the other protocols in both the languages when concurrency was introduced in the server-side database interaction as discussed in section4.3 . REST is the best option when there is no database interaction, however when concurrent requests arrive, gRPC is the best choice. Due to binary data transmission, simple parsing style, header compression, and minimal network traffic, gRPC is a clear winner in concurrency situations and speeds up the processing of multiple requests compared to textual mode of transmission.

When considering the network load the request-response exchanged by the protocol servers developed in Java were monitored in Wireshark[Figures[4.12, 4.13, 4.11]] detailed in section4.4. When only the TCP data segment is taken into consideration, gRPC serves to be the most lightweight protocol. However, if IP, Ethernet and TCP is taken into consideration REST tops the list. It is seen that, gRPC as such is a light protocol due to header compression and binary transmission. However, when IP, Ethernet, and TCP are taken into account, this protocol has a higher load due to the extra packets that are exchanged between the client and server, such as settings, ping, and magic packets, whereas REST does not. SOAP on the other hand is the heaviest due to the textually verbose XML format being transmitted, making it difficult to parse and heavy on the network. Finally, each protocol's characteristics and soft facts are presented and contrasted in section4.6. This offers a thorough understanding of the softer elements and features.

It follows that in the aforementioned scenarios, gRPC and REST perform superior to SOAP. REST is preferred in some situations, but gRPC is first in others. With the exception of server-side database interaction, Go is faster than Java from a programming language perspective.

5.2 Suggestions for Future Work

Other than REST, SOAP, and gRPC, there are a plethora of M2M communication protocols, some of which include Thrift, GraphQL, MQTT, etc. Future research ideas include comparing other protocols by testing them in the aforementioned scenarios, and documenting results along with soft facts and characteristics. In the programming language perspective, to give developers more alternatives outside of Java and Go, other programming languages like C#, Python, etc might be considered.

Bibliography

[1] Trude H Bloebaum and Frank T Johnsen. Exploring soap and rest communication on the android platform. In *MILCOM 2015-2015 IEEE Military Communications Conference*, pages 599–604. IEEE, 2015.

[2] Marek Bolanowski, Kamil Żak, Andrzej Paszkiewicz, Maria Ganzha, Marcin Paprzycki, Piotr Sowiński, Ignacio Lacalle, and Carlos E Palau. Eficiency of rest and grpc realizing communication tasks in microservice-based ecosystems. *arXiv preprint arXiv:2208.00682*, 2022.

[3] Pedro A Castillo, Jose Luis Bernier, Maribel Garcia Arenas, JJ Merelo, and Pablo Garcia-Sanchez. Soap vs rest: Comparing a master-slave ga implementation. *arXiv preprint arXiv:1105.4978*, 2011.

[4] Carolina Luiza Chamas, Daniel Cordeiro, and Marcelo Medeiros Eler. Comparing rest, soap, socket and grpc in computation offloading of mobile applications: An energy cost analysis. In *2017 IEEE 9th Latin-American Conference on Communications (LATINCOM)*, pages 1–6, 2017.

[5] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration*. Elsevier Science, 2014.

[6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures (Ph.D.)*. Chapter 5: Representational State Transfer (REST). University of California, 2000.

[7] Brendan Gregg. *Systems performance: Enterprise and the cloud*. Addison-Wesley, 2021.

[8] Ilya Grigorik. *Chapter 12: HTTP 2.0*. O'Reilly Media, Inc, 2013.

[9] Festim Halili and Erenis Ramadani. Web services: a comparison of soap and rest services. *Modern Applied Science*, 12(3):175, 2018.

[10] Jose I Santa Cruz G. Api calls and http status codes. `https://itnext.io/api-calls-and-http-status-codes-e0240f78f585`, 2019.

[11] Pavan Kumar, Sanjay Ahuja, Karthikeyan Umapathy, and Zornitza Prodanoff. Comparing performance of web service interaction styles: Soap vs. rest. *Journal of Information Systems Applied Research*, 6(1):4, 2013.

[12] Prajwal Kiran Kumar, Radhika Agarwal, Rahul Shivaprasad, Dinkar Sitaram, and Subramaniam Kalambur. Performance characterization of communication protocols in microservice applications. In *2021 International Conference on Smart Applications, Communications and Networking (SmartNets)*, pages 1–5, 2021.

[13] Smita Kumari and Santanu Kumar Rath. Performance comparison of soap and rest based web services for enterprise application integration. In *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1656–1660, 2015.

[14] Nick Langley. ”write once, run anywhere?”. *Computer Weekly*, ”Write once, run anywhere?”, May 2002.

[15] Simon St Laurent, Joe Johnston, and Edd Dumbill. *Programming Web Services with XML-RPC*. O’Reilly, 2001.

[16] Stephen Ludin and Javier Garza. *Learning HTTP/2: A practical guide for beginners*. O’Reilly, 2017.

[17] Sehrish Malik and Do-Hyeun Kim. A comparison of restful vs. soap web services in actuator networks. In *2017 ninth international conference on ubiquitous and future networks (ICUFN)*, pages 753–755. IEEE, 2017.

[18] Gavin Mulligan and Denis Gračanin. A comparison of soap and rest implementations of a service based interaction independence middleware framework. In *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pages 1423–1432. IEEE, 2009.

[19] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs.” big”’web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814, 2008.

[20] Pavan Kumar Potti, Sanjay Ahuja, Karthikeyan Umapathy, and Zornitza Prodanoff. Comparing performance of web service interaction styles: Soap vs. rest. In *Proceedings of the conference on information systems applied research issn*, volume 2167, page 1508, 2012.

[21] Mariusz Śliwa and Beata Pańczyk. Performance comparison of programming interfaces on the example of rest api, graphql and grpc. *Journal of Computer Sciences Institute*, 21:356–361, 2021.

[22] Anshu Soni and Virender Ranga. Api features individualizing of web services: Rest and soap. *International Journal of Innovative Technology and Exploring Engineering*, 8(9):664–671, 2019.

[23] Juris Tihomirovs and Jānis Grabis. Comparison of soap and rest based web services using software evaluation metrics. *Information technology and management science*, 19(1):92–97, 2016.

[24] James Turnbull, Tian Shi, Yuan Zhang, and Li Xiao. *Monitoring with prometheus*. 2019.

[25] Dave Westerveld. *API testing and development with Postman A Practical Guide to creating, testing, and managing apis for Automated Software Testing*. Packt Publishing, Limited, 2021.