Degree Project in Technology

First cycle, 15 credits

Benchmarking the request throughput of conventional API calls and gRPC

A Comparative Study of REST and gRPC

JOHAN BERG
DANIEL MEBRAHTU REDI

Stockholm, Sweden, 2023

Benchmarking the request throughput of conventional API calls and gRPC

A Comparative Study of REST and gRPC

JOHAN BERG

DANIEL MEBRAHTU REDI

Degree Programme in Computer Engineering
Date: June 20, 2023

Supervisor: Mira Kajko-Matsson
Examiner: Johan Montelius
School of Electrical Engineering and Computer Science
Swedish title: Mätning av genomströmningen av förfrågningar hos konventionella API-anrop och gRPC
Swedish subtitle: En jämförande studie mellan REST och gRPC

Abstract

As the demand for better and faster applications increase every year, so does the demand for new communication systems between computers. Today, a common method for computers and software systems to exchange information is the use of REST APIs, but there are cases where more efficient solutions are needed. In such cases, RPC can provide a solution. There are many RPC libraries to choose from, but gRPC is the most widely used today.

gRPC is said to offer faster and more efficient communication than conventional web-based API calls. The problem investigated in this thesis is that there are few available resources demonstrating how this performance difference translates into request throughput on a server.

The purpose of the study is to benchmark the difference in request throughput for conventional API calls (REST) and gRPC. This was done with the goal of providing a basis for making better decisions regarding the choice of communication infrastructure between applications. A qualitative research method with support of quantitative data was used to evaluate the results.

REST and gRPC servers were implemented in three programming languages. A benchmarking client was implemented in order to benchmark the servers and measure request throughput. The benchmarks were conducted on a local network between two hosts.

The results indicate that gRPC performs better than REST for larger message payloads in terms of request throughput. REST initially outperforms gRPC for small payloads but falls behind as the payload size increases. The result can be beneficial for software developers and other stakeholders who strive to make informed decisions regarding communication infrastructure when developing and maintaining applications at scale.

Keywords

REST, gRPC, JSON, Protocol Buffers, API, HTTP, Benchmark, Performance, Microservices, Network Communication

Sammanfattning

Eftersom efterfrågan på bättre och snabbare applikationer ökar varje år, så ökar även behovet av nya kommunikationssystem mellan datorer. Idag är det vanligt att datorer och programvara utbyter information genom användning av APIer, men det finns fall där mer effektiva lösningar behövs. I sådana fall kan RPC erbjuda en lösning. Det finns många olika RPC-bibliotek att välja mellan, men gRPC är det mest använda idag.

gRPC sägs erbjuda snabbare och mer effektiv kommunikation än konventionella webbaserade API-anrop. Problemet som undersöks i denna avhandling är att det finns få tillgängliga resurser som visar hur denna prestandaskillnad översätts till genomströmning av förfrågningar på en server.

Syftet med studien är att mäta skillnaden i genomströmning av förfrågningar för konventionella API-anrop (REST) och gRPC. Detta gjordes med målet att ge en grund för att fatta bättre beslut om val av kommunikationsinfrastruktur mellan applikationer. En kvalitativ forskningsmetod med stöd av kvantitativa data användes för att utvärdera resultaten.

REST- och gRPC-servrar implementerades i tre programmeringsspråk. En benchmarking-klient implementerades för att mäta servrarnas prestanda och genomströmning av förfrågningar. Mätningarna genomfördes i ett lokalt nätverk mellan två datorer.

Resultaten visar att gRPC presterar bättre än REST för större meddelanden när det gäller genomströmning av förfrågningar. REST presterade initialt bättre än gRPC för små meddelanden, men faller efter när meddelandestorleken ökar. Resultatet kan vara fördelaktig för programutvecklare och andra intressenter som strävar efter att fatta informerade beslut gällande kommunikationsinfrastruktur vid utveckling och underhållning av applikationer i större skala.

Nyckelord

REST, gRPC, JSON, Protocol Buffers, API, HTTP, Benchmark, Prestanda, Mikrotjänster, Nätverkskommunikation

Acknowledgments

First and foremost, we would like to express our heartfelt gratitude to our supervisor Mira Kajko-Mattsson for her guidance, support, and continuous assistance throughout the entire thesis process. Her expertise and insightful feedback have been instrumental in shaping this thesis report.

We would also like to extend our appreciation to our examiner, Johan Montelius. His constructive input and feedback during the initial stages of the thesis, as well as ongoing guidance throughout the process, helped to focus the scope of the thesis.

Additionally, we would like to thank seminar participants and opponents who dedicated their time and shared their feedback on the thesis report and our presentation.

Last but not least, we would like to extend our thanks to our friends and family for their encouragement and patience throughout this demanding time. Their constant support has been a tremendous source of motivation.

Stockholm, June 2023
Johan Berg Daniel Mebrahtu Redi

Contents

List of Figures

List of Tables

List of Listings

List of acronyms and abbreviations

API	Application Programming Interface
Arch.	Architecture
CPU	Central Processing Unit
CRUD	Create Read Update Delete
Freq.	Frequency
gRPC	gRPC Remote Procedure Call
HTTP	Hypertext Transfer Protocol
HTTP/1.1	Hypertext Transfer Protocol version 1.1
HTTP/2	Hypertext Transfer Protocol version 2
IDL	Interface Description Language
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LAN	Local Area Network
OS	operating system
protobuf	Protocol Buffers
RAM	Random Access Memory
REST	Representational State Transfer
RPC	Remote Procedure Call
SOA	Service-Oriented Architecture
SSH	Secure Shell
URL	Uniform Resource Locator

Chapter 1

Introduction

As the demand for better and faster applications increase every year, so does the demand for new communication systems between computers. In order for a computer to send a message to another computer over the Internet, it needs to establish a connection with the target host and transfer data in accordance with agreed upon communication protocols that ensure reliable and correct transmission. In the early days of computer networking, computers communicated with one another by being connected to the same Local Area Network (LAN). As technology developed and the Internet became widely available, new communication protocols were developed to handle the increasing demand for communication systems.

Today, a common method for computers and software systems to exchange information is the use of Representational State Transfer (REST) Application Programming Interfaces (APIs). An API is a set of definitions that allow different applications to communicate with an agreed upon protocol and data format. APIs on the web enable many of today's software systems, but more complex applications encounter new challenges in terms of scalability, performance and efficiency in their communication infrastructure.

1.1 Background

When designing the communication between network-connected services, it is important to consider factors such as desired functionality, scalability and performance. Using a conventional way of implementing a web-based API is desirable in many ways. It is easy to use, read, debug, and experiment with, making it a good choice for most projects. But there are cases where more efficient solutions are needed. This is where Remote Procedure Call

(RPC) comes in handy. Generally, RPC implementations specialize in making network calls by abstracting away the communication logic while also using a more efficient way of transferring the data. There are a number of different RPC libraries to choose from, but gRPC Remote Procedure Call (gRPC) is the most widely used one. Adopting gRPC is said to bring advantages such as faster development cycles, less code duplication, lower network latency, higher request throughput, and more stable communication compared to conventional API calls [1, 2, 3]. Measurements show that this claim holds when measuring response time [4, 5, 6], but there are few resources showing how this difference translates to a difference in request throughput on a server. The resources that do exist fail to give a comprehensive and detailed view. If a server uses less computing and network resources per request, the number of server instances needed when the application is under a certain load decreases, which reduces operating costs. Therefore, today's applications that use conventional design of web-based APIs might be using more computing and network resources than necessary.

1.2 Problem

There are few benchmarks showing the difference in request throughput between conventional web-based API calls and gRPC.

1.3 Purpose

The purpose of this thesis is to benchmark the request throughput for conventional web-based API calls and gRPC calls and to evaluate the differences and performance trade-offs.

1.4 Goal

The goal of this thesis is to provide a basis for making better decisions regarding communication infrastructure between applications, such as microservices. This basis can be of use for software developers, network engineers, architects and stakeholders to use more effective communication systems. This thesis will specifically focus on comparing the request throughput of traditional API calls and gRPC, with the goal of identifying performance differences.

1.5 Research Methodology

Since this study mainly consists of gathering and analyzing performance data from handpicked technologies, a qualitative research approach based on quantitative data was used. This is explained in further detail in Chapter 3.

The data was collected with quantitative observation by measuring the maximum request throughput during benchmark tests. The measurements were analyzed by comparing the data from the two subjects: an application using JavaScript Object Notation (JSON) over Hypertext Transfer Protocol version 1.1 (HTTP/1.1) and an application using gRPC.

1.6 Target Audience

The outcome of this thesis can be of interest for software developers, software architects, and network engineers that are looking to create a new project or migrate an existing project. The results can be beneficial for software developers and architects by helping them make more informed decisions regarding communication infrastructure when building and maintaining applications at scale. Similarly, researchers and students studying distributed systems or software architecture can also make use of the results, as they provide insights into the performance trade-offs of different network communication alternatives.

1.7 Scope and Limitations

Many variables can impact the real-world performance of network calls. To keep this thesis in a reasonable scope, some limitations to the study have been put in place:

- In order to measure the performance of only the network protocols to the degree possible, the application for testing did not include any business logic, database calls, or other time-consuming tasks.
- There are many modern implementations of RPC libraries. Comparing several of them would take too much time. Therefore, this report only focuses on comparing gRPC to a conventional REST API.
- There are many ways of implementing REST and gRPC APIs, and many more ways of constructing the tests. The dimensions of the benchmarks

in this study have been kept within reasonable limits in order to see general patterns, but not to extensively find answers for every possible scenario.

- The tests were conducted on a local network. This simulates services deployed in the same cloud cluster to some degree, but calls over the Internet are also of interest to measure.

1.8 Benefits, Ethics, and Sustainability

The insights in this report can be beneficial in regards to more efficient computing resource usage. In computing, a faster program implies that less computations are used in order to achieve the same amount of work. Therefore, a faster program uses less energy, reducing the energy cost of running it. The same argument can be made about the efficiency of the network protocols used, reducing load on the network infrastructure. Today's applications are often deployed using cloud hosting providers, where services are typically billed based on the amount of resources used [7]. A reduction in resource usage can therefore be of interest from a financial and sustainability perspective.

1.9 Terminology

This section clarifies some terms that are used extensively throughout this thesis. More technical descriptions of some acronyms can be found in Section 2.2.

The word *benchmark* refers to a test designed to evaluate the performance of computer software.

The term *request throughput* is used throughout this thesis when referring to the number of requests per second handled by a server. It is usually used in the context of *maximum throughput*, the highest number of requests that a server can handle (during benchmark measurement).

There are many types of APIs. For example, a code library API concerns the set of functions and data types that it exposes to users of the library, while a web API concerns the names, methods, and data formats of the endpoints that are served by a web server. RPC libraries are not exclusively used as replacements for web APIs, but this thesis focuses on gRPC which does. Therefore, the term API mainly refers to a web API in this thesis.

A conventional web API is often called a *REST API*, or *RESTful*, if it follows the design principles of Representational State Transfer (see Section 2.2.5). If an API uses JSON and HTTP/1.1, it is not guaranteed to follow the conventions of REST. However, the term *REST* is often used loosely to describe variants of this combination. In this thesis, the term *REST* is used interchangeably between its formal definition and when referring to programs that use JSON and HTTP/1.1 (the left column in Figure 2.1).

The words *server* and *client* can refer to both the roles of physical computers, and the software that runs on them. In this thesis they mostly refer to software, but in Section 4.2 both meanings are used.

The term *serialize* means to convert the state of a data object, which can be spread out pieces of memory, to a sequence of bytes that can be persisted or transported.

1.10 Structure of the Thesis

The remainder of this thesis consists of the following chapters:

Chapter 2: REST and gRPC: This chapter provides a more in depth coverage and context of relevant concepts. Literature about the subject is also reviewed and compared to what this thesis aims to cover.

Chapter 3: Research Methodology: This chapter describes what methods, tools, and resources were used to obtain the results.

Chapter 4: Implementation and Execution of Benchmarks: This chapter describes the work that was done to achieve the results: iterations of implementation of servers and client, and how the final benchmarks were conducted to produce the results.

Chapter 5: Results: This chapter shows the results of the benchmarks.

Chapter 6: Analysis and Discussion: This chapter contains analysis of the results, a discussion about validity and sources of error in the results, and a comparison with other measurements of REST and gRPC performance.

Chapter 7: Conclusions and Future Work: This chapter discusses our conclusions drawn from the result, provides our input of our work and proposes future work.

Chapter 2

REST and gRPC

In this chapter, the problem domain is explained in more detail in Section 2.1. Then, Section 2.2 gives explanations with examples of relevant technologies and concepts for the reader. Last, Section 2.3 evaluates literature and previous work about the subject, and how this study relates to it.

2.1 Interservice Communication

When sharing resources on the World Wide Web became prevalent, Hypertext Transfer Protocol (HTTP) rose to become the dominant way of sending data on the Internet. As web sites grew in complexity and scale, incentive grew for developers to separate user-facing logic (frontend) from services that handle business logic and data (backend). One consequence of separating services is that a common language for exchanging data between them is needed. This language, or data format, is typically defined in a service's API. Due to JavaScript being the main scripting language available in all web browsers, JSON has become the standard choice when serializing data for APIs on the Web [8].

When Web applications grow in size and complexity, the pattern of splitting applications into smaller services becomes more necessary. This shift led to the rise of the pattern Service-Oriented Architecture (SOA) in the early 2000s [9], which later was built upon and became microservices.

Microservices is an architectural style for large and complex applications that involves splitting the application into a collection of loosely coupled, collaborating services, where each service is independently deployable [10]. It has also received criticism due to the fact that, by their nature, microservices are heavier and more complex to develop, test, and debug [11]. For example, an error in one service might have been caused by a different service that it is depended upon. Their increased complexity and distributed nature also introduces overhead and data consistency issues.

The extra communication between microservices can also prevent applications with throughput limitations from being able to scale. A recent article describes how a service in Amazon's Prime Video achieved a 90% cost decrease when moving from distributed microservices to a monolithic architecture [12].

Building internal communication in a cluster of services comes with new challenges regarding choice of technologies. The conventional choice is to use REST APIs with JSON encoded data over the HTTP/1.1 protocol due to its simplicity and widespread use in web development. A large, performance-critical microservice application that needs to send thousands of internal messages per second could suffer from using an inefficient method for inter-service communication, but knowing which solutions work best in such a case can be hard without proper research and testing. As microservice applications grow in size and complexity it becomes even more important to choose a good method for inter-service communication. While JSON is seen as a good choice for most projects, newer technologies provide more efficient alternatives for larger performance critical applications that need to send hundreds of thousands of messages per second. One of those alternatives is to use an RPC library.

RPC is a way for a computer program to call a function (procedure) to execute in a different program, usually on a different machine over the network. The call is coded as if it were a normal function call, without the programmer explicitly coding the details for the network call [13]. The use of RPC abstracts the logic for handling the remote call, which saves work for the programmer.

Among the many RPC libraries available, gRPC is the most widely used one. It supports all mainstream programming languages, and adopting gRPC is said to bring advantages such as faster development cycles, less code duplication, lower network latency, higher request throughput, and more stable communication due to using Hypertext Transfer Protocol version 2 (HTTP/2) [14]. There are several other RPC libraries that challenge the leading position of gRPC [15, 16, 17, 18, 19, 20], but none with the same level of widespread compatibility and support.

There are many ways of designing APIs. Knowing which solutions work best for a service that deals with a lot of traffic can be hard without proper research and testing. This thesis compares conventional API designs to gRPC in the scope of request throughput.

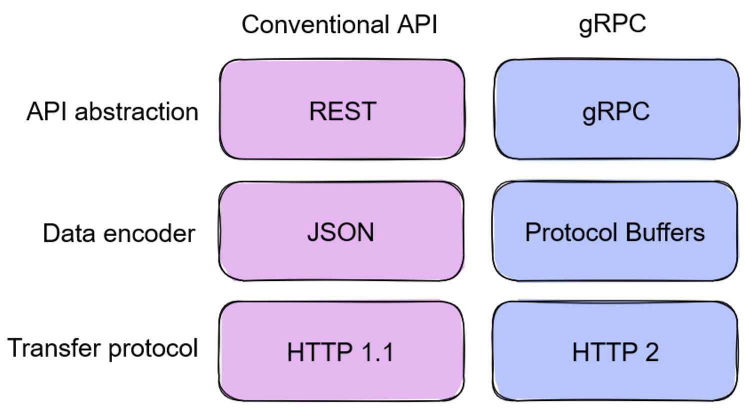

Figure 2.1: Concepts explained in Section 2.2

2.2 Explanation and Comparison of Relevant Concepts

This section explains the relevant concepts needed to read and understand this thesis. Figure 2.1 shows the relationships between technologies mentioned. This section covers these concepts from bottom to top, with comparisons along the way.

If an API uses JSON and HTTP/1.1, it is not guaranteed to follow the conventions of REST. However, the term *REST* is often used loosely to describe variants of this combination. In this thesis, the term *REST* is used interchangeably between its formal definition (see Section 2.2.5) and when referring to programs that use JSON and HTTP/1.1.

While HTTP/2 is rising in popularity, we chose to use HTTP/1.1 for the REST implementation in this study due to it usually being the default setting in most frameworks. The left column in Figure 2.1 represents a more naive approach, and the right column a more thought through implementation.

2.2.1 HTTP

Hypertext Transfer Protocol (HTTP) is an application protocol used for transmitting data over the Internet. It is known as the foundation of data communication on the World Wide Web and is used by web browsers and web servers to communicate with each other. [21]

The 1.1 version of the protocol, which was first introduced in 1997, has received updates over the years [22]. In a typical HTTP/1.1 request and response cycle, the client sends a request to the server which includes a method (such as GET, POST, PUT, or DELETE), a resource identifier (part of the Uniform Resource Locator (URL)), headers with metadata, and a message body. The server then responds with a status code indicating the success or failure of the request, more headers, and if the request was successful, a message body containing the requested resource.

HTTP/1.1 has several limitations such as inefficient use of network resources, high latency, and poor performance over slow or unreliable connections. To improve on these issues, HTTP/2 was developed to use several new features [23]. Among these features are multiplexing, which allows several requests to be sent over the same connection, server push, which allows the server to send resources that the client is likely to request in advance, and header compression, which greatly reduces the size of headers in subsequent requests.

Overall, HTTP/2 provides significant performance improvements over HTTP/1.1, making it a better choice for modern applications.

2.2.2 JSON

JavaScript Object Notation (JSON) is a lightweight, text-based file format and data-interchange format [24]. It is by far the most popular format used when accessing APIs on the Internet [8], but it is also used for other purposes, such as configuration files. It is easy for humans to read and write, and easy for computers to parse and generate. Although it is based on JavaScript (as the name suggests), it is completely language-independent and can be used in all mainstream programming languages.

The format consists of key-value objects, arrays, and values such as strings, numbers, `true`, `false`, and `null`. Listing 2.1 shows an example object represented in JSON. When sending data over the network, unnecessary whitespace is usually removed to reduce the amount of bytes sent. The example object takes up 55 bytes when whitespace is removed.

Listing 2.1 A JSON object

```json
{
    "query": "apples",
    "page_number": 2,
    "result_per_page": 50
}
```

2.2.3 Protocol Buffers

Protocol Buffers (protobuf) are a language-neutral mechanism for serializing structured data [25]. It was originally developed by Google for internal use, but later released as open-source [26]. Contrary to self-describing data formats such as JSON, protobufs involves an Interface Description Language (IDL), such as proto3 [27], to define the structure of messages. The *proto definitions* are then used by a compiler to generate code stubs for the programming language(s) of choice, which can then be used to serialize and deserialize messages. The messages are serialized to a binary format which is more compact than text formats.

Although protobufs are a great alternative for serializing data, they have some limitations. The protobuf documentation lists a few points [28], the most significant being:

- Messages have no way of being parsed correctly without the correct IDL specification.
- The entire message has to be in memory at once, meaning very large messages need a different approach.
- Messages are not compressed. Data with special-purpose compression algorithms, such as images, will produce much smaller files for data of such types.

Listing 2.2 shows an example of a protobuf definition of a message (object) that can be used in a gRPC request or response.

Listing 2.2 A protobuf message definition (proto3 syntax)

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

Listing 2.3 shows the bytes representing a `SearchRequest` object with the same values as the JSON example in Listing 2.1. The main resources used for decoding the bytes was Wireshark [29] and the Protobuf docs [30].

Listing 2.3 Hexadecimal representation of the bytes that resulted from encoding a `SearchRequest` protobuf object. Line 1: The bytes. Lines 2-5: The values contained and arrows to the bytes that hold them. Lines 7-23: Explanation of all byte values.

```
1  Bytes:      0a 06 61 70 70 6c 65 73 10 02 18 32
2                    ↑↑ ↑↑ ↑↑ ↑↑ ↑↑ ↑↑    ↑↑    ↑↑
3                     |  |  |  |  |  |     |     |
4  Characters:        a  p  p  l  e  s     |     |
5  Integers:                               2    50
6
7  0a:
8      .0001... = Field Number 1
9      .....010 = Length-delimited
10 06:
11     Value Length 6
12 61 70 70 6c 65 73:
13     The ASCII values of the letters in "apples"
14 10:
15     .0010... = Field Number 2
16     .....000 = varint (integer)
17 02:
18     Value 2
19 18:
20     .0011... = Field Number 3
21     .....000 = varint (integer)
22 32:
23     Value 50
```

To reiterate, the protobuf message shown in Listing 2.3 can be decoded to reveal that it contains a string ”apples”, and the integers 2 and 50. Despite this, the meaning of the value fields (what they represent) can not be known without the protobuf definition that was used to encode it.

Compared to the JSON message in Listing 2.1, which takes 55 bytes to send, the same message encoded with protobufs only takes 12 bytes to send.

2.2.4 API

An Application Programming Interface (API) is a set of rules that provides a way for applications to communicate with each other [31]. This means that developers can use an API to access the functionality of another system.

In the context of HTTP, the APIs are typically used to retrieve information. The application that wants to retrieve information (client) sends an API request. The request is then served by the application providing the information (server) by sending a response to the client.

2.2.5 REST

Representational State Transfer (REST) is a software architectural style that defines a set of constraints to be used when designing web services. A REST API is built around the idea of resources where each resource is identified with a unique URL. Requests to a REST API are made using HTTP verbs such as GET, POST, PUT, and DELETE. Each HTTP verb corresponds to actions such as retrieving data, creating new resources, updating existing resources, and deleting resources. As REST APIs communicate via HTTP requests, functions like creating, reading, updating, and deleting records may be used within a resource to preform standard database functions. These functions, usually referred to as Create Read Update Delete (CRUD) operations are often associated with RESTful APIs and correspond to the HTTP verbs.

2.2.6 RPC and gRPC

Remote Procedure Call (RPC) is a software communication protocol that uses the client-server model for communication between different software applications over a network. gRPC is an open-source RPC framework developed by Google [3]. It is designed to build scalable and fast APIs that can run in many environments. It provides a connectivity layer between client applications and server applications on different machines, making it easier to create distributed applications and services. gRPC combines HTTP/2 and protobufs to achieve these goals. Protobufs are used for serializing data and for generating code stubs for the client and server, which saves time and reduces the chance of errors such as version mismatches or carelessness.

One of the advantages of gRPC is its support for multiple programming languages, unlike many other RPC implementations that only work in a single programming language. gRPC has official support for C++, Go, Java, Node.js, PHP, Python, Ruby and more [33]. This advantage makes it easier to transfer shared data between services that use different programming languages [14]. Figure 2.2 shows an example of this.

Despite its benefits, gRPC has some limitations and challenges. Unlike JSON, protobuf messaging format is not human readable. The binary format used in HTTP/2 is also hard to read without proper tools. This makes using extra tools necessary for debugging or manual testing of endpoints.

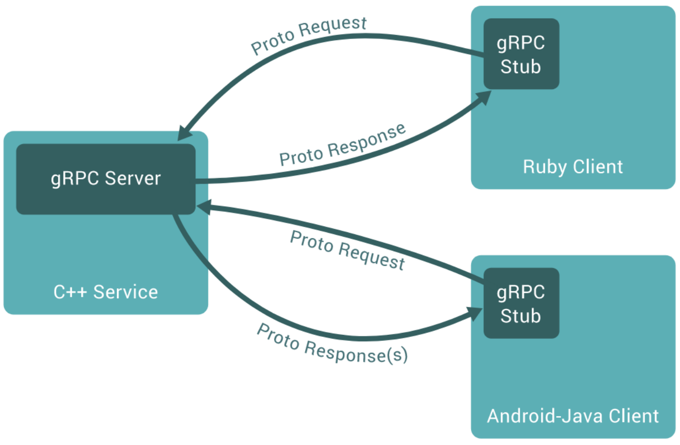

Figure 2.2: Services written in different languages using gRPC for interservice communication [32]. The server and client code stubs are generated based on a shared protobuf specification. Appears here under CC BY 4.0

2.2.7 Differences in API Design

The differences in API design varies between REST APIs and gRPC. While REST APIs are usually designed to only follow the CRUD operations on resources, gRPC methods can be designed to fit arbitrary purposes more specifically. While CRUD operations are also common operations to perform between gRPC services, the protocol also offers more advanced features due to the use of HTTP/2, such as streams.

A stream call can be set up between two services, letting one service send messages at any time, without the overhead of establishing a new connection. This enables services to achieve the Observer design pattern across an API, something that is not possible with REST APIs over HTTP/1.1. A gRPC call is made by directly calling a generated method. On the client side, developers can invoke these generated methods just as if they were local functions. As the provided pseudo-code shows, represented in Listing 2.4, the gRPC call demonstrates how the HTTP logic is abstracted away.

Listing 2.4 Pseudo-code for calling an endpoint

```
// REST API (normal HTTP call)
client.post("/api/items", request_body)

// gRPC call (HTTP logic is abstracted away)
client.items(request_body)
```

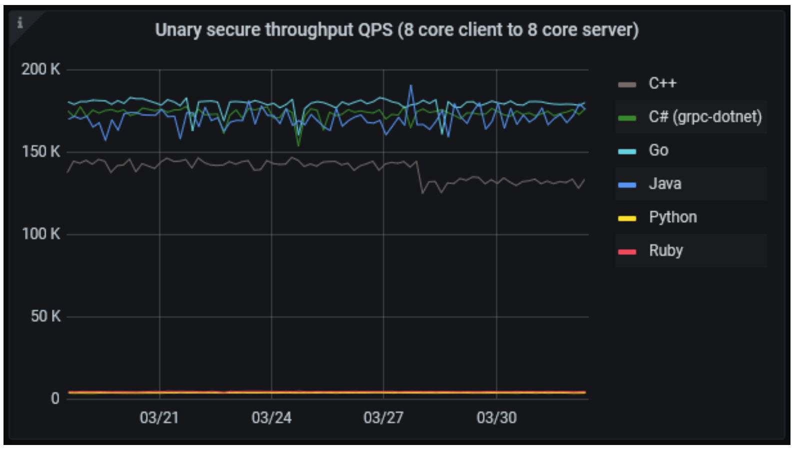

Figure 2.3: Screenshot of one of many gRPC continuous benchmarks [35]

2.3 Previous Work

This section summarizes and discusses other work in the field of benchmarking JSON REST APIs and gRPC. Several sources have measurements showing that gRPC calls have better response times than a REST API, and throughput is measured in few places. This is what this thesis aims to expand knowledge about.

2.3.1 Continuous Benchmarks of gRPC

The gRPC development team uses a continuous performance benchmarking workflow as part of the development [34]. Every few hours, performance tests are run for several languages against the master branch, and the results are reported to a dashboard for visualization, available at [35]. The benchmarks measure the response time, requests per second, and Central Processing Unit (CPU) usage of a varying set of tests. Figure 2.3 shows one of the graphs displayed on the dashboard.

Overall, the benchmarks on the dashboard show the relative performance for different language implementations, but not how gRPC performs compared to other RPC variants or other types of API calls.

2.3.2 Difference in Response Time

In a blog post from 2019 [4], Ruwan Fernando describes how using JSON REST APIs in his microservice project started to reach throughput limits. He constructed a test with C# where the same type of message was sent hundreds of times over a REST API using JSON and a gRPC endpoint, while the execution time was measured. The benchmark results showed that, for small payloads, the gRPC execution time was barely performing better than REST, while for large payloads, it performed roughly two times better.

In a blog post from 2021 [5], Recep İnanç, Software Development Engineer at Amazon, did a similar comparison in Java Spring, where the response time of many calls with small, medium and large payloads were sampled using JMeter and shown on graphs. His results showed a slight edge for REST APIs when sending small payloads, a tie when sending medium sized payloads, and a factor of two win for gRPC for large payloads. The sizes of these three categories of payloads were not shown.

In an IEEE conference paper from 2021 [36], Kumar et al. measured the time of making many sequential calls with REST, gRPC, and Apache Thrift [15] between microservices on the same machine, and applied optimizations to increase performance in such setups. The number of requests ranged from 1,000 to 100,000, and the three payload sizes were 100 kB, 500 kB, and 1,000 kB. Their results show that gRPC outperformed REST, and Thrift even more so. Many implementation details, such as programming languages and message composition were not shown.

These three sources show what difference can be expected in response time between the two technologies, but to what level the request throughput can differ is left for question.

Chapter 3

Research Methodology

This chapter describes the research methods used in this thesis. Section 3.1 provides an overview of the research strategy used. Section 3.2 lists the research phases. Section 3.3 describes the our research methods and motivates why they were appropriate for this study. Section 3.4 presents the research instruments used for data collection and evaluation of our results. Section 3.5 lists the possible validity threats to our study.

3.1 Research Strategy

In order to ensure that that we achieved valid results and optimized output within the time constraints of this thesis, an appropriate research strategy was chosen. An overview of our research strategy listing the research phases, research methods, research instruments and validity threats used is presented in Table 3.1. The rest of this chapter contains a section explaining each part.

Table 3.1: Overview of the research strategy

Research Phases	Research Methods	Research Instruments	Validity Threats
Literature study Choice of criteria Choice of environment Implementation of tools Benchmark execution Result evaluation	Qualitative study Comparative study Quantitative data	Research literature Benchmark parameters Testing hardware Benchmarking software	Credibility Transferability Reliability Objectivity

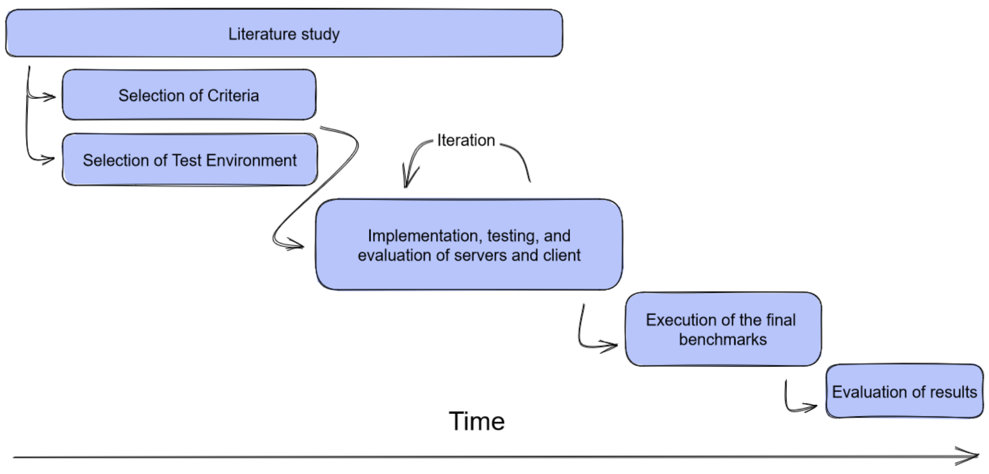
Literature study
Selection of Criteria
Selection of Test Environment
Iteration
Implementation, testing, and evaluation of servers and client
Execution of the final benchmarks
Evaluation of results
Time

Figure 3.1: Research phases

3.2 Research Phases

This section describes how the authors organized the research process model. Figure 3.1 shows a time diagram illustrating the phases of the research process.

The first and largest phase was the Literature study. In the beginning of the thesis project, the subject of this thesis was still too loosely defined. By studying the problem domain and evaluating existing work in the field, the scope and goal of the thesis got more focused during its early stages. The internet was searched to gather knowledge about the technologies under test and the underlying foundations. The search was conducted using Google, Google Scholar, and IEEE Xplore with search terms such as *grpc*, *rest vs grpc*, *grpc performance*, *grpc throughput*, *grpc benchmark*, and *grpc competitors*.

When sufficient knowledge had been gathered through studying the subject, the phase of choosing what criteria to assess and in what environment to conduct the experiment was started.

When all criteria were defined, the implementation of the testing tools started. During this phase, the literature study phase was still ongoing. The plan for what to implement was done, but developing the test programs involved consultation of relevant documentation throughout the process. This was an iterative process where the different sets of code were written, tested, and evaluated. Evaluation cycles improved the code while enhancing the reliability and robustness of the benchmark tests, and mitigated potential validity threats.

When all code for the final testing was ready, the execution of the final benchmarks were performed, and the measured results were collected. Lastly, an evaluation of the results was conducted to validate, analyze and draw conclusions from the benchmarks.

3.3 Research Methods

Since this study is not a statistical study with test subjects from a random sample in a larger population, a qualitative research approach was chosen. The benchmark tests that are conducted give quantitative data, that then undergoes qualitative comparison.

Inductive reasoning is used to draw conclusions about the software under test. Due to the limited scope of the study and selection of measurements, no generalized conclusions about other communication technologies can be made from the results. The conclusions only cover the specific conditions which were measured during the study.

3.4 Research Instruments

The research instruments used in this study were (1) *research literature*, (2) *benchmark parameters*, (3) *testing hardware*, and (4) *benchmarking software*.

The research literature laid the foundation for which benchmark parameters to use in the experiment and which technologies to use when developing the benchmarking software. The benchmark parameters selected were (1) the two *request technologies* to compare, (2) *request message size* for comparing efficiency of data encoding methods, and (3) *programming language* to see if each type of language saw similar magnitudes of performance gains across the tests. The testing hardware and benchmarking software were used to conduct the experiment. The choice and implementation of these is explained further in Chapter 4.

3.5 Validity Threats

To evaluate the validity of our study, four criteria are used: (1) *credibility*, (2) *transferability*, (3) *reliability*, and (4) *objectivity*. These criteria are used to judge the quality of qualitative research [37]. They are explained further here.

Credibility is to what degree the research can be trusted. The potential threats to the credibility of our research are: (1) An unrealistically assembled test application and workload that does not reflect real-world usage, leading to misleading or biased results. This includes choice of what metrics to measure and manipulate, the choice of the hardware and software used, and network congestion on local networks or the Internet. (2) Flaws or limitations in our implementation of the software may affect the accuracy of the results.

Transferability, which concerns to what extent findings may be applicable in other situations, may be limited by differences in server hardware, application architecture, usage of the application, or network configuration. Another aspect of transferability is how the findings can be generalized to other RPC libraries or network call protocols. Due to the small sample size (two protocols, four message sizes, and three programming languages), the findings can not be generalized with confidence to other setups. In general, results from isolated benchmarks are not always accurate indicators for real world performance, and this also applies to our research.

The *reliability* of the study, which concerns the consistency of measurement values under the same conditions, is regarded as fairly high due to the well-defined and isolated test environment. The major consistency factor is the randomness benchmarks, which could lead to varying results. Small variations in performance can be expected across CPUs of the same model, this difference is considered negligible in the scope of this study.

Objectivity is that the researchers should not allow their own biases to affect the study. There is possibility of unintentional biases in our selection of languages and frameworks used. It is also possible that different amounts of optimization effort was put into each server, leading to misleading results. There is also possibility of bias in our interpretation of the benchmark results.

Chapter 4

Implementation and Execution of Benchmarks

This chapter describes how the study was conducted. Section 4.1 gives an overview of the entire study. Section 4.2 describes and motivates the decisions regarding evaluation criteria, and shows the setup of the test environment. Section 4.3 describes the data models and messages used in the tests, and how they were shaped to fit the study. Section 4.4 describes how the six servers were implemented, and what languages and frameworks were used. Section 4.5 describes how the benchmark client was designed and implemented. Section 4.6 describes how the final benchmarks were run to produce the final results. All source code can be found at:
`https://github.com/jonaro00/rest-vs-grpc`

4.1 Overview

Initially, the evaluation criteria and test environment were specified. Then, servers were set up to implement the same API using REST (with JSON) and gRPC respectively. This was done for the programming languages Java, Python, and Rust, for a total of six servers. A client program was also created, capable of spamming requests of both types while counting the number of completed requests per second. The client runs a test against one server at a time, requesting responses with one of the four pre-defined payload sizes. The smallest response payload is an empty response, leaving just the OK response status and headers. The other three sizes of messages have response bodies with dummy data, encoded as JSON and protobufs respectively.

The servers' request throughput were measured by running the client test

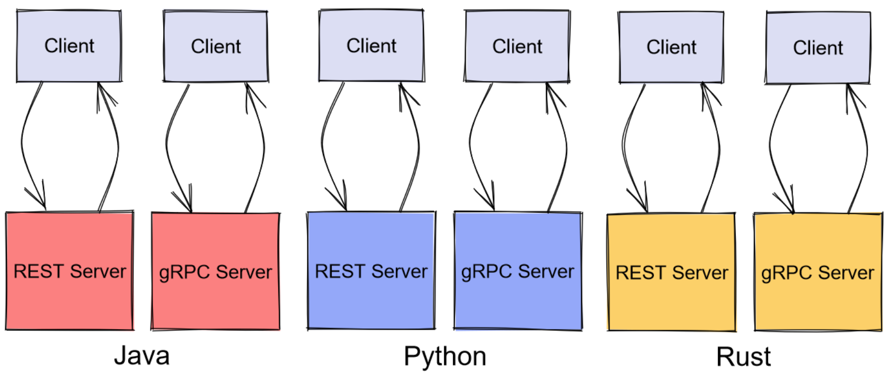

Figure 4.1: The six servers. Each server was benchmarked four times, one for each message size. The same client was used for all benchmarks, operating in either REST mode or gRPC mode

with each message size against each server, totalling in 24 tests (six servers, four message sizes). This is visualized in Figure 4.1.

4.2 Selection of Criteria and Environment

The main measurement criterion in this experiment is the number of requests that a server can handle per second (request throughput). In order to test this criterion to the fullest extent, the servers were implemented to handle each request without any state, business logic, database calls, or logging.

In order to test how the serialization formats perform when payload sizes increase and how that effects the throughput, four sizes of request payload were chosen. The smallest possible requests were used by the client, and all payload data was placed in the server response. The client used a GET request for REST, and a custom empty object with no data for the gRPC servers. The smallest payload size (”XS”) was chosen to be an empty response body. This serves as a baseline for the throughput, as the request will only consist of HTTP headers and leave serialization out of the picture. The other three sizes of messages (”S”, ”M”, and ”L”) were constructed to contain dummy data of varying types: lists, objects, strings, numbers, booleans, enums, and null. See Section 4.3 for more details on the messages.

The tests were replicated in three programming languages of different characteristics to compare the performance among them, but also to compare the magnitude of performance differences that each programming language can observe when switching between REST and gRPC. The choice of languages was based on the authors’ areas of interest and experience.

The experiment is about measuring the maximum throughput seen from the server with the CPU being the main targeted bottleneck. Because of this, the client program was not hosted on the same hardware as the server, as they would have competed for execution time.

A Raspberry Pi 400 was chosen to host the server and a desktop computer to host the client. Since the desktop has a much more powerful CPU, it will always be ready to send requests for the server to handle. A gigabit (1 Gbit/s) network switch was used, with both hosts directly connected to it. This is also the max Ethernet networking speed supported by both machines. One gigabit was assumed to be enough to not limit the benchmarks, but turned out to be a limiting factor for some of them, see Chapter 5. The test environment was mainly chosen based on what hardware the authors had available. Figure 4.2 shows the network setup. Table 4.1 lists the main specifications of the test machines.

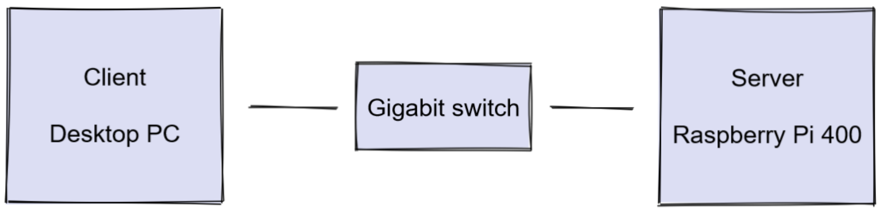

Figure 4.2: Setup of test environment network

Table 4.1: Specification of client and server

	Client	**Server**
Computer	Desktop	Raspberry Pi 400
CPU	Intel Core i5-7600K	Broadcom Cortex-A72
Freq.	3.8GHz	1.8GHz
Arch.	x86_64	aarch64
Cores	4	4
Threads	4	4
RAM	32GB DDR4-3000	4GB LPDDR4-3200
OS	Ubuntu 22.04 in WSL2 on Windows 10	Raspberry Pi OS (Debian 11)

4.3 Data Models and Message Size

For the purpose of testing the servers, a protobuf definition file was written with mockup data resembling a hypothetical data model of an inventory service of a large company. Listing 4.1 shows an excerpt. The same data model was followed when implementing the equivalent JSON-based services.

Listing 4.1 Inventory service proto definition

```
// A hypothetical Inventory service
service Inventory {
  // Empty request to check responsiveness
  rpc HeartBeat (Empty) returns (Empty);
  // Get aggregate statistics for all items
  rpc ItemsStatus (Empty) returns (ItemsStatusResponse);
  // Get summary for every item
  rpc ItemsSummary (Empty) returns (ItemsSummaryResponse);
  // Get full details of every item
  rpc ItemsFull (Empty) returns (ItemsFullResponse);
}
```

The `HeartBeat` (size ”XS”) response contains no data, and servers as a reference for the maximum performance throughput possible. The `ItemsStatus` (size ”S”) response contains an object with strings and numbers. It was shaped to reach a size of approximately 1 kB when serialized as JSON. The `ItemsSummary` (size ”M”) and `ItemsFull` (size ”L”) responses contain a list of identical objects with various data. The length of the lists were adjusted to reach sizes of approximately 50 kB and 500 kB when serialized as JSON.

Table 4.2: The four message payloads with serialized lengths and aliases

Size alias	Serialized length with JSON [bytes]	Serialized length with protobuf [bytes]
XS	0	5
S	646	556
M	49,945	22,080
L	500,820	220,410

Table 4.2 shows the length, in bytes, of the messages when serialized by the two methods, along with the alias they will be referred to as in the results.

Note that this is the size of the message payload, not the size of a network frame or the total bytes sent on wire.

4.4 Implementation of Servers

This section describes how the six servers were implemented, and what languages and frameworks were used. Section 4.4.1 is an overview of the languages and frameworks. Section 4.4.2 to Section 4.4.4 contain brief descriptions of the programming languages used and subsections with more details regarding implementation decisions that were made when implementing each test server.

4.4.1 Languages and Frameworks

In order to see how much an observed difference between conventional APIs and gRPC is affected by the underlying programming language, the experiment was conducted for three programming languages that are candidates when building a backend service: Java, Python, and Rust. Needless to say, the languages used are not solely responsible for performance. The choice of frameworks and their implementations are also a significant factor. Table 4.3 shows the versions and libraries used to build the servers.

Table 4.3: Programming languages and libraries used for testing REST and gRPC

Language & Version	**REST API framework**	**gRPC library**
Java 17.0.6	Spring	io.grpc (official)
Python 3.11.2	FastAPI	grpcio (official)
Rust 1.69.0	Axum	Tonic

Before implementing the full Inventory service, minimal functionalities of all frameworks used was first ensured by implementing a *Hello World* testing service in each language. After verifying that all frameworks functioned as expected and that they worked together with the client across the network, the implementation of the Inventory service was started.

4.4.2 Java

Java [38] is an object-oriented compiled language. It has been a popular choice for many years, and is still taught at many schools and universities. Java code is compiled to intermediate byte code, which can then be executed on multiple platforms by the Java Virtual Machine (JVM).

The two servers written in Java used Java version 17.0.6 and the Spring framework [39]. Additionally, Spring Boot — a framework that simplifies the creation of web servers [40], was used for both the REST and gRPC server. It offers a wide range of features and modules for creating REST APIs, such as Spring MVC. The project dependencies were managed using Maven, which facilitated the installation and management of the required Java libraries and frameworks.

4.4.2.1 REST Server

To enable the development of the REST application, the Spring Web dependency was added to the project. This dependency provides a set of classes and utilities that help build RESTful application using Spring MVC.

In order to create the REST API, two new packages were created: the Controller package and the Item package. The Item package represents the model which defines the dataclasses for the objects that are returned by the API. The Controller package contains the REST controller which is responsible for processing incoming requests and returning responses. For the heartbeat endpoint, modifications was made so that the response body was now an empty response body (void) instead of an empty string. The `@GetMapping` annotation was used to map the endpoints to their function for handling the API call.

4.4.2.2 gRPC Server

To generate the necessary classes for the gRPC server, a protobuf Maven plugin was used. Unlike the REST server, the models were automatically generated when the server was compiled. These generated sources were then included in the project structure and added as a dependency. A subclass of `InventoryGrpc.InventoryImplBase` was created and methods for each gRPC endpoint were overwritten and modified to return the appropriate response. The `@GRpcService` annotation allowed for the automatic registration of the annotated bean as a gRPC service. The annotation also allows the server to utilize the built-in threading mechanism.

4.4.3 Python

Python [41] is an object-oriented interpreted language, famous for having simple syntax and being beginner-friendly. Python code is parsed and executed by an interpreter, which makes the language more flexible, but slower than compiled languages.

Both servers used version 3.11.2 of the Python interpreter. A *virtual environment* was set up to isolate the versions of Python executables and dependencies. Dependencies were installed with the default `pip` tool.

4.4.3.1 REST Server

For the REST server, the FastAPI framework was used together with Pydantic for replicating the data models. In order to host and manage multiprocessing of the FastAPI instance, the Uvicorn web server was used.

To implement the API, the data models were first replicated as subclasses of `pydantic.BaseModel`. Then, the API endpoints could be easily defined by writing functions annotated with the decorator `@app.get()`.

The heartbeat endpoint was at first implemented by returning an empty string. However, this made the HTTP response consist of an empty JSON response as the body. To solve this, an instance of the `Response` class from FastAPI was instead returned, which made the endpoint correctly respond with an empty body. This change also resulted in better performance, since no JSON encoder got involved.

The non-empty responses were annotated to use `ORJSONResponse`, an optional extension to FastAPI for accelerating the encoding of the response, which resulted in a few percent better performance during testing. Another way performance was increased was to use asynchronous functions (`async def`) instead of synchronous ones.

The final part was to tweak the Uvicorn server for optimal performance. A major improvement was seen when disabling logging of requests. The amount of workers (server processes in parallel) was tweaked to yield the maximum performance on the server machine, resulting in 10 workers.

4.4.3.2 gRPC Server

The Python gRPC models were generated by issuing a command to the `grpc_tools` package on the command line. Since Python does not have a building stage, a way of automatically generating the gRPC models before executing the program was not found. The models had to be regenerated every time changes were made.

Implementing the generated Inventory service was simple. A subclass of a generated `InventoryServicer` class was defined. One method for each RPC endpoint was then added, returning the corresponding model object.

To achieve concurrency, the gRPC server from the official package uses a thread pool executor. However, this thread pool struggled to fully utilize the CPU.

4.4.4 Rust

Rust [42] is a multi-paradigm, compiled, memory-safe language rising in popularity. Code is compiled to a platform-specific standalone executable.

Both servers used version 1.69.0 of the Rust compiler, and were built with frameworks that build upon the Tokio asynchronous runtime [43]. Tokio serves as the foundation for building network applications and handles multi-threading, synchronization, scheduling facilities, and more.

4.4.4.1 REST Server

For the JSON-based server, the web framework Axum was used alongside the Serde serialization framework. The models were implemented as structs and enums, using the `#[derive()]` procedural macro to implement the `serde::Serialize` and `serde::Deserialize` traits. These traits allow the Rust data structures to be serialized in a wide variety of formats, JSON in this case. The `Clone` trait was also derived to easily create multiple copies of structs to fill vectors with the `vec![]` macro. The heartbeat endpoint was declared with an empty asynchronous function, and the endpoints with data were declared with asynchronous functions that return the JSON representation of the hardcoded responses.

4.4.4.2 gRPC Server

Unlike Java and Python, Rust does not have official gRPC or protobuf libraries. Instead, community-developed implementations Tonic and Prost were used. Using Prost required a separate installation of the official `protoc` protobuf compiler, as it does not come bundled.

Building the protobuf definitions was handled by code in a separate crate from the other code using the `tonic_build` crate. A build script was used, so that the protobufs were recompiled every time the protobuf definitions changed. Using a separate crate for the generated protobuf models meant that both the server and client implementations could depend on it, and guarantee that both parts use the same version of the protobuf schema.

In the gRPC server implementation, the generated protobuf definitions provided an `Inventory` trait for implementing a valid server that follows the specification, along with all the data models, meaning no extra work was needed to ensure that the server was following the specification.

4.5 Implementation of Client

The client was designed to spam an endpoint with requests of one type (REST or gRPC) and size (referred to as ”XS”, ”S”, ”M”, or ”L”; see Table 4.2). When the client runs, it counts how many requests were completed each second, and reports the average request count per second across the test.

The client was written in Rust and uses some of the same libraries as the Rust server. The gRPC-related code reuses the same code stubs generated by the shared protobuf crate. The Tokio asynchronous runtime is also used in the client. Tokio’s abstraction for handling parallel execution are called *tasks*, which are similar to OS threads, but more lightweight [44]. In this section, the word *thread* is used to describe parallel execution on the CPU, but refers to a Tokio task.

Early in the implementation phase, the client used a single-threaded loop for sending one request at a time. This was sufficient to verify functionality of the servers, but not to saturate them with requests. This spamming behaviour was parallelized to a multi-threaded solution, where each thread spams requests sequentially. Such a thread will be referred to as a *spammer thread*.

It was quickly discovered that the different servers performed very differently based on how many client threads were used. A ”discovery phase” was introduced, where the client exponentially increased the number of threads to find an amount with optimal throughput. After the exponential increases, a binary search-like approach continued the search for an optimal thread count. However, this approach turned out to give very inconsistent results and rarely found a ”better” amount of threads than the exponential search due to the randomness involved in benchmarking. Because of this, the binary search-like approach was dropped, leaving only the exponential search.

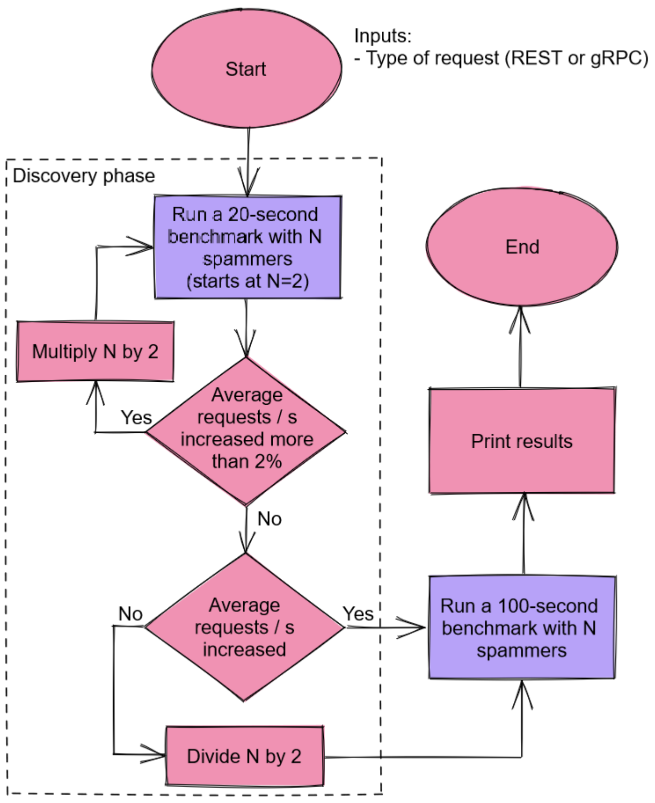
Start
Inputs:
- Type of request (REST or gRPC)
Discovery phase
Run a 20-second benchmark with N spammers (starts at N=2)
Multiply N by 2
Average requests / s increased more than 2%
Yes
No
Average requests / s increased
No
Yes
Divide N by 2
Run a 100-second benchmark with N spammers
Print results
End

Figure 4.3: Logical flow of a full test, simplified. Purple rectangles refer to one benchmark round (Figure 4.4)

Figure 4.3 shows pseudo-code for how the client performs the discovery phase and final benchmark. An example of the discovery phase can also be seen in Figure 4.6 and Figure 4.7. Each iteration of the discovery phase was set to run for 20 seconds, and the final test for 100 seconds. Every benchmark began with a 1 second ”warmup” phase before measuring anything, so that all spammer threads had time to reach maximum throughput. This made the client find a reasonable number of spammer threads to use for each server without spending too much time trying to find the ”perfect” number. The

Figure 4.4 shows pseudo-code for how a benchmark runs. It starts off by taking note of the current time and calculating when to end the benchmark. It then sets up an `mpsc` (multiple producer, single consumer) channel for receiving messages. Next, the spammer threads are spawned.

A spammer is simply an infinite loop that sends requests sequentially. If a response is successfully received, a message is passed to the main thread, containing the timestamp (whole second) relative to the benchmark start when the response was received. Spammer threads keep on running until cancelled

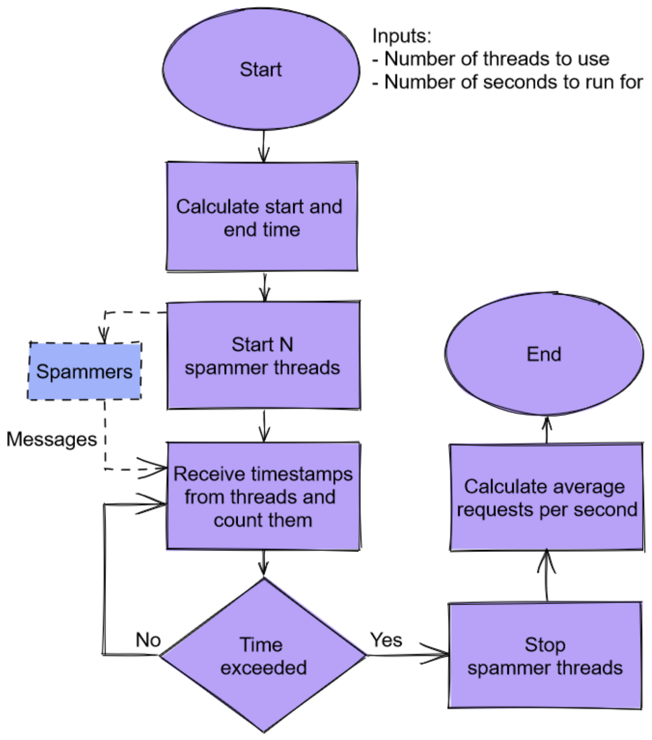
Start
Inputs:
- Number of threads to use
- Number of seconds to run for
Calculate start and end time
Start N spammer threads
Spammers
Messages
Receive timestamps from threads and count them
Time exceeded
No
Yes
Stop spammer threads
Calculate average requests per second
End

Figure 4.4: Logical flow of one benchmark round, simplified. The blue rectangle refers to several instances of spammer threads (Figure 4.5)

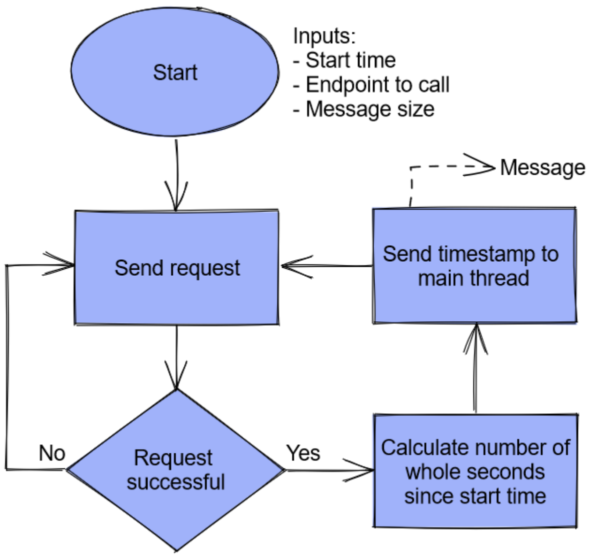

Figure 4.5: Logical flow of a spammer thread, simplified

by the main loop at the end of the test. Pseudo-code for a spammer thread is shown in Figure 4.5.

4.6 Execution of Benchmarks

Both the client and server computers were restarted. The minimal set of programs required to run the tests were opened on the client computer: A terminal for running the client software, a terminal for connecting to the server via Secure Shell (SSH), a terminal for taking notes of the results, and Windows' *Task Manager* for monitoring CPU and Network usage. The 24 benchmarks were executed one by one by starting the corresponding server program on the server, and running one benchmark for each message size, while taking notes of the results.

Each benchmark started with a discovery phase where, starting from two clients, the throughput was measured during 20 seconds. If a higher number of clients yielded better results, the number of clients were doubled, and the 20-second test was repeated. This step was repeated until the average requests per second started to flatten out. This behaviour is visualized in Figure 4.6 and Figure 4.7.

When the optimal number of clients for maximal throughput was found, the benchmark was run for 100 seconds. The benchmark produced an average number of requests per second, which is the final result for that server and message size.

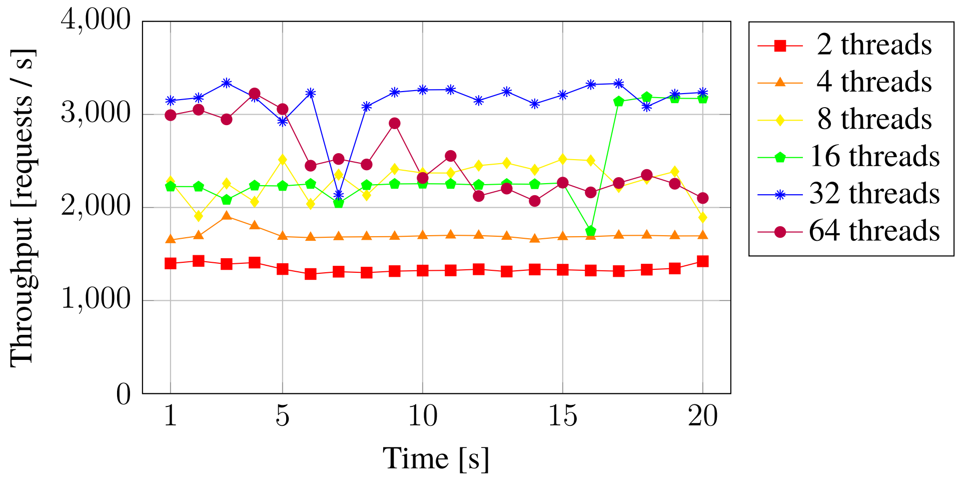

Figure 4.6: Discovery of optimal client thread count against the Java gRPC server with M-sized messages. Each line chart represents a 20-second benchmark

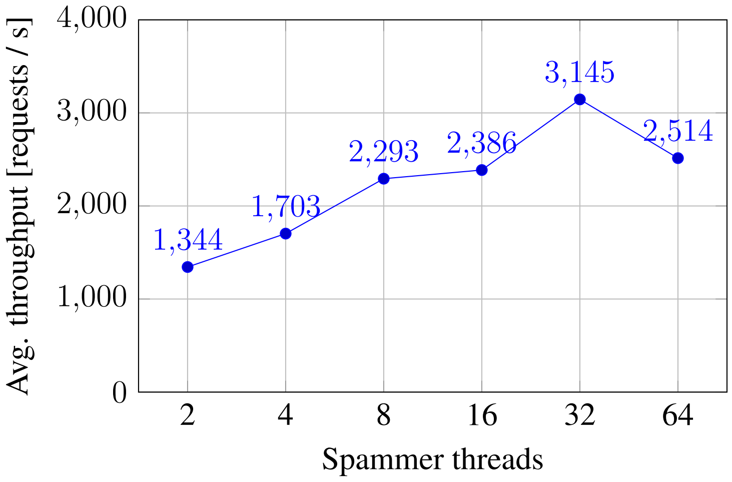

Figure 4.7: Average requests/s per thread count during discovery phase. Aggregation of Figure 4.6. During this test, 32 was a clear sweet spot, and was therefore used as the number of threads in the full 100-second test

Chapter 5

Results

This chapter presents the results from the 24 benchmarks, split into 3 charts, one for each of the programming language used on the server: Java, Python, and Rust. Each chart has two halves, one for the REST benchmarks, and one for the gRPC benchmarks. Each half of the charts have four bars, one for each message size used in the response from the server. The bars represent the average number of requests per second that the server was able to deliver when under a load from an optimal number of clients. Section 4.6 describes how the benchmarks were executed. All values are rounded to two significant numbers. The full output of all tests is included in Appendix A.

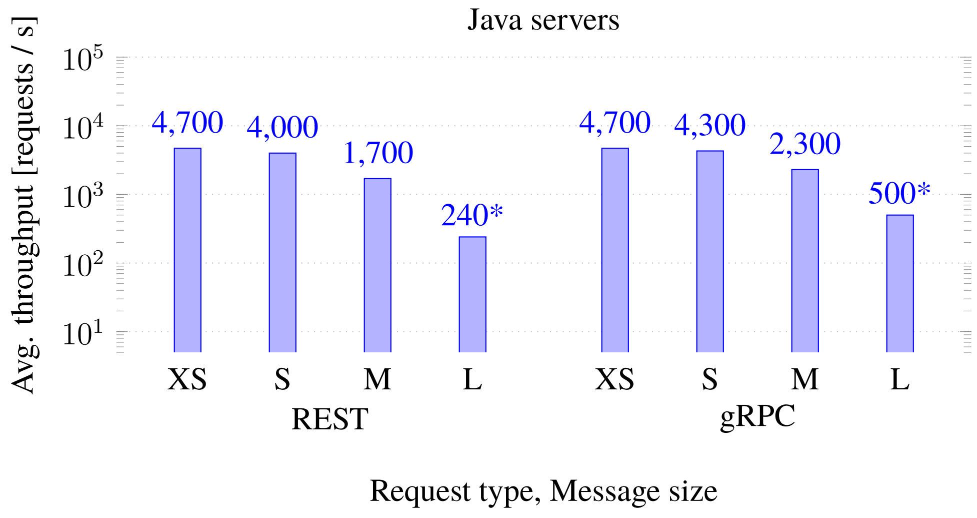

Figure 5.1: Results benchmarks against Java servers. An asterisk (*) denotes that the benchmark was capped by the 1 Gbit/s network bandwidth instead of the server's CPU

Figure 5.1 shows the results of the benchmarks against the REST server and gRPC server implemented in the Java programming language. The request throughput for both servers was measured to 4,700 requests/s for the XS message size. The REST server achieved a throughput of 4,000 requests/s for the S message size while the gRPC server reached 4,300 requests/s. For the M message size, REST recorded 1,700 requests/s, and gRPC 2,300 requests/s. Regarding the L message size, REST exhibited an average of 240 requests/s, and gRPC 500 requests/s. Notably, for both servers, the L message size benchmark was limited by the 1 Gbit/s network bandwidth.

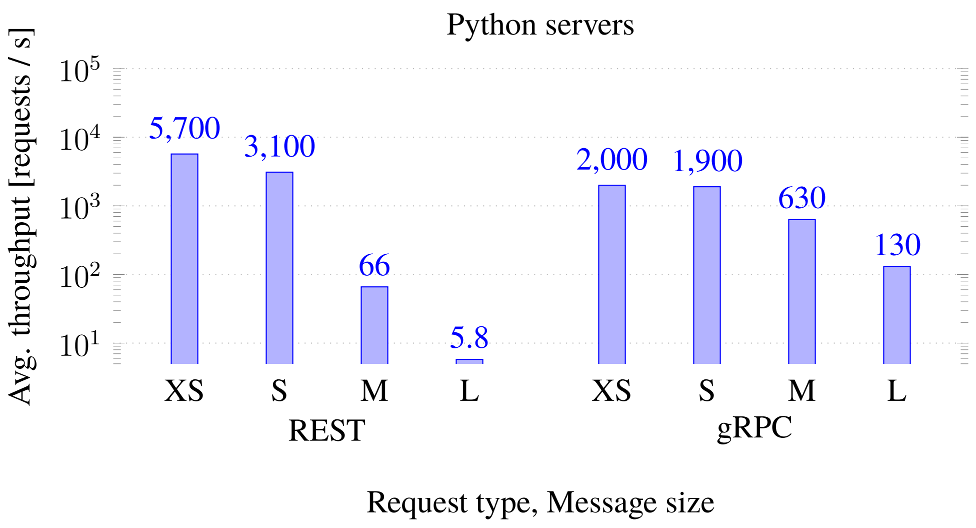

Figure 5.2: Results benchmarks against Python servers

The results of the benchmarks against the Python servers, illustrated in Figure 5.2, show that the throughput for the XS message size was measured to 5,700 requests/s for the REST server while the gRPC server reached 2,000 requests/s. For the S message size, REST saw a throughput of 3,100 requests/s, while gRPC demonstrated a throughput of 1,900 requests/s. For the M message size, REST recorded 66 requests/s while gRPC demonstrated a higher 630 requests/s. The L message size also showed a great disparity where the REST server had 5.8 requests/s while gRPC had 130 requests/s.

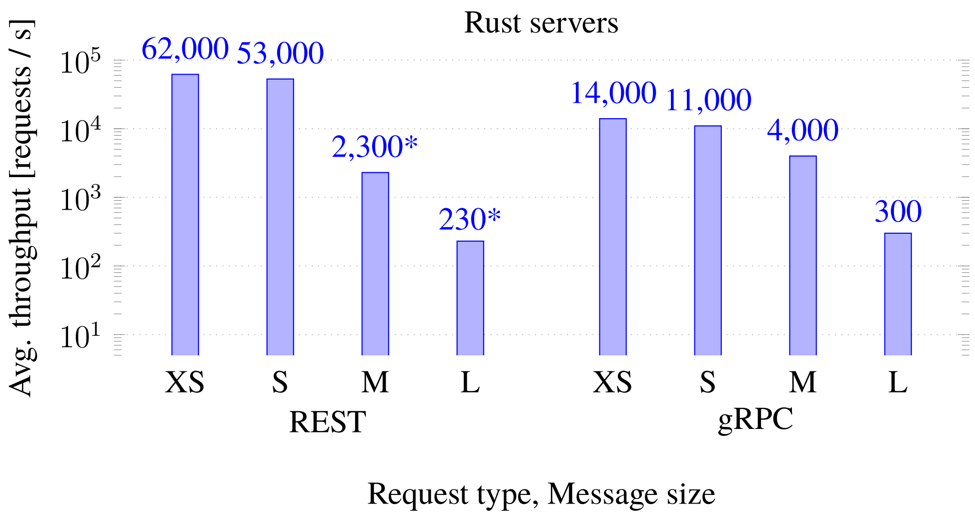

Figure 5.3: Results benchmarks against Rust servers. An asterisk (*) denotes that the benchmark was capped by the 1 Gbit/s network bandwidth instead of the server's CPU

Figure 5.3 presents the benchmarks against the Rust servers. It shows a throughput of 62,000 requests/s for the REST server and 14,000 for the gRPC server when sending XS messages. For S messages the REST server and gRPC server had a throughput of 53,000 requests/s and 11,000 requests/s respectively. For the M sized message, REST recorded 2,300 requests/s while gRPC demonstrated 4,000 requests/s. Notably, the REST server's benchmark was capped by the 1 Gbit/s network bandwidth. For the L size messages, the benchmark was once again capped for the REST server which resulted in an 230 requests/s. For the same L message size, the gRPC server had 300 requests/s.

Chapter 6

Analysis and Discussion

In this chapter, the results presented in Chapter 5 are analyzed and discussed. Section 6.1 has a basic analysis of the results. Section 6.2 contains a discussion of the findings, with comparison to expectations and other work. Section 6.3 analyzes the reliability of these results by addressing the validity threats introduced in Section 3.5.

6.1 Analysis

A trend that can be seen across all three languages is that gRPC gets better and better the bigger the response payload is. For Python and Rust, REST starts off better at small payloads, but falls behind when the payload size increases. For Java, the performance was already equal at small payloads, but REST still falls behind when the payload size increases. At medium and large message sizes, all gRPC servers outperformed their REST counterpart.

It is not certain, but fairly likely that the Rust and Java REST servers that were capped by the network bandwidth would also reach lower throughput than the corresponding gRPC server even if the bandwidth cap was raised.

In places where the throughput got capped by the network bandwidth, the throughput can be seen as the inverse of the size of the encoded payload, seen in Table 4.2. The large message took 56% less bytes of space when encoded with protobufs compared to JSON, and the Java gRPC server managed 52% more throughput than the REST server. The large messages were also 10 times larger than the medium messages, and the Rust REST server managed 10 times less throughput as a consequence.

6.2 Discussion

The fact that REST performed better for the empty messages was expected. An empty HTTP/1.1 response consists of only the status code and response body. An empty gRPC request and response cycle contains more overhead data and a small protobuf object that goes through the encoding and decoding steps. Although an empty HTTP/2 response is smaller than an equivalent HTTP/1.1 response, the added complexity of gRPC makes it slower at empty or small payloads.

Overall, the Java servers performed much worse than our expectations, especially the REST server. In the continuous gRPC benchmarks [35], Java tends to be among the best performing languages. Although those tests are run in a very different environment and without Spring Boot, it was not expected that Python would come so close to Java's performance and even surpass in the XS size REST test. We deem the most likely explanations for this unexpected behavior could be inefficiencies in the Spring Boot framework, Java and/or Spring Boot being sub-optimal to run on our server hardware, or oversights and errors in our implementation or execution of the Java project.

From the literature studies presented in Section 2.3, certain similarities can be seen in our and others' results from similar research. Ruwan Fernando performed similar benchmark tests over a REST API using JSON and a gRPC endpoint [4]. Even though the test was done in C# as the programming language, he found that for small payloads, the gRPC execution time was barely performing better than REST, but for large payloads, gRPC clearly performed better. Recep İnanç also did a similar study in Java using the Spring framework and concluded that for smaller message payloads, REST had better performance in terms of response time, but the results moved towards favoring gRPC for larger payloads [5].

In this study, we examine request throughput for REST APIs and gRPC for three different programming languages (Java, Python and Rust). The results align well with the expectations set by the previous studies and literature presented in Section 2.3. In this study, the connection is found that regardless of programming language we see a consistent trend that gRPC performed better than REST as the payload size increased.

Kumar et al. measured the time of making many sequential calls with REST and gRPC in a microservice environment [36]. The results showed that gRPC outperformed REST, indicating gRPC's superiority. Their tests only used payloads of 100 kB and above, which is above the M size messages used in our tests. While specific details regarding programming languages and message composition were not provided, their findings support the notion that gRPC can deliver better performance compared to REST in response time.

By measuring the maximum network throughput of REST APIs and gRPC across different programming languages we provide valuable insights into their scalability and performance expectations.

Overall, the results in any benchmark test should be taken with a grain of salt. The best way to check which solution works best for a particular service is to try different solutions in the same environment as the service. This is of course not always possible, which is why the results in this thesis can serve as a pointer towards what to expect.

6.3 Validity Analysis

This section analyses the four validity threats listed in Section 3.5, one subsection for each validity thread.

6.3.1 Credibility

Credibility is to what degree the research can be trusted. The fact that our benchmarking was conducted using an unrealistically assembled test application and workload that does not reflect real-world usage, and the choices of metrics, hardware and software used contributes to a lower credibility of our research. Another dimension of this threat is how the potential flaws and limitations our implementations of the client and server affected the results.

6.3.2 Transferability

Transferability concerns to what extent our findings may be applicable in other situations. As mentioned, the goal with the research study is to bring valuable insight to make informed decisions regarding communication infrastructure when developing and maintaining applications at scale. Although isolated benchmark tests cannot fully generalize to real-life values, our results serve as pointers, providing indications of performance characteristics and expectations.

6.3.3 Reliability

Reliability is the consistency of measurement values under the same conditions. There are many factors in our development and benchmarking process that could cause the results to differ under similar circumstances. The major one is how the randomness in the discovery phase benchmarks could result in varying choices of spammer threads when repeating the test against the same server. This is the biggest factor that could lead to varying results. Minor factors that can contribute to the variance are the CPU usage of other programs on the server and network usage of other programs in the same network. There is also the tiny factor of performance difference across CPUs of the same model, but this difference is considered negligible in the scope of this study.

However, the overall reliability is considered fairly high due to the isolated and well-defined test environment.

6.3.4 Objectivity

Objectivity is that the researchers should not allow their own biases to affect the study. There was a certain amount of bias in the selection of languages and frameworks used. The languages were chosen based on familiarity and previous experience. However, the study was conducted in three languages to begin with in order to alleviate this bias from distorting the results. The choice of frameworks was mostly based on perceived popularity when researching the alternatives, but bias from familiarity and ease of use was also present.

Another aspect of objectivity is that different amounts of time were spent to implement and optimize each server. For instance, the Python REST server underwent some experimentation with various parameters and addons in order to make it perform slightly better, while the Java REST server was left at the *minimum viable product* state. Giving an ”equal” amount of optimization effort for each server will always be subjective in some regards, but we acknowledge that this bias was present.

There is also a possibility of bias when interpreting the results. This has been mitigated by acknowledging the validity threats and not draw any strong conclusions due to the research being qualitative.

Chapter 7

Conclusions and Future Work

The increasing demand for better and faster applications has led to increased demand for more efficient communication systems. Today, a common method for software systems to exchange information is with the use of APIs. Many developers build APIs using JSON serialization over HTTP/1.1, but there exists more complex applications where more efficient solutions are needed. A large, performance-critical application that sends thousands of internal messages per second could suffer from using an inefficient method for inter-service communication. As complex applications face challenges of scalability, performance and efficiency in their communication infrastructure, conventional API implementations may not always fulfill these requirements. RPC libraries such as gRPC are said to offer higher performance API calls, but many still favor conventional design of APIs. Data that shows the potential benefits of different alternatives can be helpful when deciding how to build communication interfaces. The problem addressed in this thesis is that there are few benchmarks showing the difference in request throughput between conventional API calls and gRPC. The purpose of this report was to provide benchmarks of the request throughput for conventional API calls and gRPC calls in order to evaluate the differences and performance trade-offs. This was done with the goal of providing a basis for making better decisions regarding communication infrastructure between applications. The result of the report is benchmarks comparing the protocols, presented in Chapter 5. Based on the results obtained, this thesis addresses the problem by providing benchmarks that compare request throughput between conventional API calls and gRPC. The conclusions drawn from the results are listed in Section 7.1. Areas of future work and suggestions on how to improve and extend this study are suggested in Section 7.2.

7.1 Conclusions

The conclusions drawn from the research are:

- Using gRPC can increase request throughput when messages are not tiny. To what extent this applies to any particular application requires further testing.
- For straightforward or simple projects with low performance requirements, the selection of a network protocol for API calls becomes less important. In such cases, conventional REST APIs might be favored, due to their simplicity.
- When network bandwidth matters, using a more efficient data compression procedure is better. This rule applies generally, and not only to protobufs. One thing to consider is that compression can cost more CPU usage. In the case of large gRPC payloads it can be an improvement over JSON.

7.2 Future Work

Improvements to this study can be done by conducting tests in a more realistic scenario, with for example database calls and business logic. This could be performed in the form of a case study. Additionally, it could include testing of different RPC libraries, and different programming languages in the implementation of a service, to gain a more comprehensive overview of where gains in performance can be seen. Another major aspect that this report did not cover are requests with payload, such as POST requests. This would make decoding of the encoding formats a part of the picture.

There are also several other parameters that could be tuned for a more comprehensive comparison. Benchmarking more sizes of messages and messages with different compositions of data (such as many strings, many numbers, nested, flat) can highlight strengths and weaknesses of different encoding algorithms. Other aspects of the network protocols can also be adjusted. A growing number of HTTP servers support HTTP/2. Using HTTP/2 on the REST servers will result in a more competitive benchmark. Comparing gRPC streaming with *Websockets* or *Server-sent events* is also a consideration.

Since this report only covered testing on a local network with low latency, this study can be further improved by testing over a higher latency, for example over a larger geographical distance over the Internet, and comparing the results. Doing tests on cloud-based infrastructure is also of relevance, since it is a common environment for today's applications. This also enables the testing to be done on more realistic hardware.

References

[1] J. H, “gRPC vs. REST: How Does gRPC Compare with Traditional REST APIs?” Nov. 2022. [Online]. Available: https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/ [Page 2.]

[2] S. Wettasinghe, “Everything you need to know about gRPC,” Oct. 2022. [Online]. Available: https://medium.com/@swettasinghe23/everything-you-need-to-know-about-grpc-ccd3637d145f [Page 2.]

[3] “gRPC.” [Online]. Available: https://grpc.io/ [Pages 2 and 13.]

[4] R. Fernando, “Evaluating Performance of REST vs. gRPC,” Apr. 2019. [Online]. Available: https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da [Pages 2, 15, and 40.]

[5] R. İnanç, “Benchmarking — REST vs. gRPC,” Jan. 2021. [Online]. Available: https://medium.com/sahibinden-technology/benchmarking-rest-vs-grpc-5d4b34360911 [Pages 2, 16, and 40.]

[6] Sanyog Kumar Singh, “gRPC vs. REST - Performance Test using JMeter,” May 2022. [Online]. Available: https://www.dltlabs.com/blog/grpc-vs-restperformance-test-usingjmeter-532131 [Page 2.]

[7] S. Spilka, “Cloud Pricing Models - Shedding light upon pricing options.” [Online]. Available: https://www.exoscale.com/syslog/cloud-pricing-models/ [Page 4.]

[8] K. Kunwar, “The Reason why JSON is so Popular,” Jul. 2020. [Online]. Available: https://www.prokurainnovations.com/json/ [Pages 7 and 10.]

[9] “SOA Source Book - What Is SOA?” [Online]. Available: https://collaboration.opengroup.org/projects/soa-book/pages.php?action=show&ggid=1314 [Page 7.]

[10] C. Richardson, “What are microservices?” [Online]. Available: http://microservices.io/index.html [Page 7.]

[11] Ethan J. Jackson, “The Dark Side of Microservices,” Feb. 2020. [Online]. Available: https://dev.to/ethanjjackson/the-dark-side-of-microservices-3pbd [Page 7.]

[12] M. Kolny, “Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%,” Mar. 2023, section: Video Streaming. [Online]. Available: https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90 [Page 8.]

[13] “Remote procedure call,” Jan. 2023, page Version ID: 1135050297. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Remote_procedure_call&oldid=1135050297 [Page 8.]

[14] “Introduction to gRPC,” Feb. 2023, section: docs. [Online]. Available: https://grpc.io/docs/what-is-grpc/introduction/ [Pages 8 and 13.]

[15] “Apache Thrift - Home.” [Online]. Available: https://thrift.apache.org/ [Pages 8 and 16.]

[16] “fRPC Documentation.” [Online]. Available: https://frpc.io/introduction [Page 8.]

[17] “DRPC - Storj Labs.” [Online]. Available: https://storj.github.io/drpc/ [Page 8.]

[18] “Go Micro,” Apr. 2023, original-date: 2015-01-13T23:30:18Z. [Online]. Available: https://github.com/go-micro/go-micro [Page 8.]

[19] zeromicro, “go-zero.” [Online]. Available: https://go-zero.dev/ [Page 8.]

[20] “tRPC - Move Fast and Break Nothing. End-to-end typesafe APIs made easy. | tRPC,” Sep. 2022. [Online]. Available: https://trpc.io/ [Page 8.]

[21] “An overview of HTTP - HTTP | MDN,” Mar. 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview [Page 9.]

[22] “Evolution of HTTP - HTTP | MDN,” Apr. 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP [Page 9.]

[23] "Introduction to HTTP/2." [Online]. Available: https://web.dev/performance-http2/ [Page 10.]

[24] "JSON." [Online]. Available: https://www.json.org/json-en.html [Page 10.]

[25] "Protocol Buffers." [Online]. Available: https://protobuf.dev/ [Page 11.]

[26] "History." [Online]. Available: https://protobuf.dev/history/ [Page 11.]

[27] "Language Guide (proto 3)," section: programming-guides. [Online]. Available: https://protobuf.dev/programming-guides/proto3/ [Page 11.]

[28] "Overview." [Online]. Available: https://protobuf.dev/overview/ [Page 11.]

[29] "Wireshark · Go Deep." [Online]. Available: https://www.wireshark.org/ [Page 11.]

[30] "Encoding," section: programming-guides. [Online]. Available: https://protobuf.dev/programming-guides/encoding/ [Page 11.]

[31] "What is an API?" [Online]. Available: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces [Page 12.]

[32] "gRPC website," May 2023. [Online]. Available: https://github.com/grpc/grpc.io/blob/ee39b786fe68a32eb8e1b094f1b449543a2d30a2/static/img/landing-2.svg [Pages xi and 14.]

[33] "Supported languages." [Online]. Available: https://grpc.io/docs/languages/ [Page 13.]

[34] "Benchmarking," Jan. 2022, section: docs. [Online]. Available: https://grpc.io/docs/guides/benchmarking/ [Page 15.]

[35] "gRPC Performance Multi-language on GKE (@upstream/master) - Grafana." [Online]. Available: https://grafana-dot-grpc-testing.appspot.com/?orgId=1 [Pages xi, 15, and 40.]

[36] P. K. Kumar, R. Agarwal, R. Shivaprasad, D. Sitaram, and S. Kalambur, "Performance Characterization of Communication Protocols in Microservice Applications," in *2021 International Conference on Smart Applications, Communications and Networking (SmartNets)*, Sep. 2021. doi: 10.1109/SmartNets50376.2021.9555425 pp. 1–5. [Pages 16 and 40.]

[37] W. M. K. Trochim, “Qualitative Validity.” [Online]. Available: https://conjointly.com/kb/qualitative-validity/ [Page 19.]

[38] “Java | Oracle.” [Online]. Available: https://www.java.com/en/ [Page 25.]

[39] “Spring.” [Online]. Available: https://spring.io/ [Page 26.]

[40] “Spring Boot.” [Online]. Available: https://spring.io/projects/spring-boot [Page 26.]

[41] “Welcome to Python.org,” May 2023. [Online]. Available: https://www.python.org/ [Page 27.]

[42] “Rust Programming Language.” [Online]. Available: https://www.rust-lang.org/ [Page 28.]

[43] “Tokio - An asynchronous Rust runtime.” [Online]. Available: https://tokio.rs/ [Page 28.]

[44] “tokio::task - Rust.” [Online]. Available: https://docs.rs/tokio/latest/tokio/task/ [Page 29.]

Appendix A
Full Results from Benchmark

Source code and all results can be found at
https://github.com/jonaro00/rest-vs-grpc.

A.1 Java REST Server, XS messages

```
Trying with 2 clients...
[1447, 2047, 2104, 2057, 2090, 2073, 2095, 2119, 2109, 2052, 2105, 2111,
↪  2031, 2052, 2103, 2106, 2094, 2130, 2091, 2134] avg 2057.5
Trying with 4 clients...
[3053, 3085, 3069, 3098, 3094, 3128, 3074, 3091, 3109, 3152, 3162, 3122,
↪  3180, 3195, 3205, 3153, 3179, 3181, 3184, 3162] avg 3133.8
Trying with 8 clients...
[4265, 4041, 4196, 4196, 4282, 4208, 4180, 4111, 4139, 4171, 4186, 4015,
↪  4182, 4151, 4348, 4151, 4198, 4169, 4134, 4146] avg 4173.45
Trying with 16 clients...
[4250, 4380, 4282, 4227, 4329, 4327, 4285, 4370, 4403, 4391, 4385, 4400,
↪  4295, 4320, 4380, 4335, 4263, 4233, 4320, 4484] avg 4332.95
Trying with 32 clients...
[4492, 4438, 4550, 4474, 4504, 4475, 4486, 4492, 4477, 4476, 4499, 4518,
↪  4500, 4476, 4489, 4473, 4502, 4467, 4467, 4505] avg 4488
Trying with 64 clients...
[4566, 4612, 4731, 4570, 4567, 4611, 4620, 4708, 4587, 4789, 4663, 4604,
↪  4531, 4571, 4574, 4549, 4679, 4644, 4600, 4672] avg 4622.4
Trying with 128 clients...
[4625, 4673, 4805, 4718, 4645, 4653, 4647, 4760, 4715, 4681, 4647, 4678,
↪  4745, 4618, 4716, 4653, 4409, 4673, 4586, 4739] avg 4669.3
Choosing 128 clients.
Running test with 128 clients...
[4721, 4655, 4684, 4708, 4743, 4755, 4667, 4695, 4628, 4645, 4679, 4713,
↪  4662, 4705, 4651, 4690, 4730, 4712, 4752, 4719, 4643, 4700, 4727, 4643,
↪  4658, 4716, 4702, 4663, 4723, 4592, 4665, 4758, 4770, 4710, 4531, 4685,
↪  4609, 4674, 4707, 4650, 4682, 4684, 4653, 4660, 4732, 4637, 4693, 4641,
↪  4640, 4679, 4635, 4731, 4669, 4611, 4719, 4684, 4739, 4689, 4650, 4714,
↪  4644, 4678, 4814, 4721, 4562, 4716, 4663, 4710, 4734, 4745, 4690, 4709,
↪  4743, 4706, 4737, 4693, 4653, 4616, 4635, 4688, 4737, 4622, 4660, 4655,
↪  4714, 4639, 4621, 4763, 4708, 4619, 4661, 4767, 4770, 4623, 4625, 4686,
↪  4659, 4706, 4595, 4724] avg 4683.93
```

A.2 Java REST Server, S messages

```
Trying with 2 clients...
[1622, 1642, 1609, 1616, 1577, 1627, 1606, 1626, 1610, 1554, 1583, 1630,
↪  1669, 1657, 1584, 1637, 1654, 1637, 1646, 1529] avg 1615.75
Trying with 4 clients...
[2458, 2452, 2497, 2468, 2544, 2471, 2457, 2439, 2478, 2449, 2509, 2437,
↪  2492, 2504, 2486, 2464, 2444, 2457, 2444, 2436] avg 2469.3
Trying with 8 clients...
[3077, 3086, 3053, 3157, 3074, 3060, 3055, 3104, 3096, 3099, 3136, 3188,
↪  3096, 3101, 3140, 3104, 3123, 3175, 3118, 3133] avg 3108.75
Trying with 16 clients...
[3550, 3554, 3605, 3538, 3482, 3541, 3547, 3553, 3573, 3481, 3439, 3547,
↪  3521, 3546, 3573, 3593, 3540, 3560, 3506, 3551] avg 3540
Trying with 32 clients...
[3666, 3767, 3761, 3758, 3706, 3792, 3773, 3789, 3849, 3787, 3761, 3787,
↪  3762, 3812, 3769, 3819, 3842, 3752, 3783, 3782] avg 3775.85
Trying with 64 clients...
[3842, 3867, 3811, 3884, 3852, 3804, 3869, 3904, 4025, 3891, 3936, 3836,
↪  3862, 3870, 3903, 3849, 3863, 3829, 3880, 3892] avg 3873.45
Trying with 128 clients...
[3947, 4032, 3979, 3813, 3983, 3939, 3765, 3971, 3791, 4011, 3995, 3954,
↪  3921, 3922, 4063, 3985, 3972, 3958, 3992, 4035] avg 3951.4
Trying with 256 clients...
[3969, 3891, 3959, 3920, 3976, 3946, 4118, 3990, 3930, 3970, 3927, 4064,
↪  3988, 3955, 4084, 3955, 3928, 3900, 3987, 4029] avg 3974.3
Choosing 256 clients.
Running test with 256 clients...
[4092, 3925, 3971, 4005, 3946, 3996, 3995, 3981, 4020, 3890, 3909, 3929,
↪  4003, 4064, 3936, 3938, 3870, 3946, 4065, 4048, 4010, 4002, 3993, 4016,
↪  4045, 3955, 4029, 4006, 3975, 4052, 4007, 3919, 3986, 4012, 4103, 3948,
↪  3985, 4053, 4018, 4086, 3942, 3951, 3870, 3999, 4052, 4045, 4079, 4051,
↪  4019, 3924, 3983, 3995, 3969, 3965, 3955, 3935, 4032, 3954, 3973, 3943,
↪  3874, 3976, 3917, 3903, 4085, 3932, 4032, 4002, 3895, 3990, 4051, 3995,
↪  3990, 3867, 3955, 4044, 3948, 3906, 3931, 4035, 4038, 4016, 3952, 3992,
↪  3989, 3989, 4011, 3967, 4070, 3944, 3966, 4065, 4013, 3870, 4080, 4075,
↪  4033, 3972, 3890, 3877] avg 3985.02
```

A.3 Java REST Server, M messages

```
Trying with 2 clients...
[782, 782, 794, 791, 775, 789, 784, 805, 799, 775, 799, 800, 794, 792, 786,
↪  805, 805, 796, 801, 792] avg 792.3
Trying with 4 clients...
[1185, 1193, 1224, 1233, 1218, 1228, 1208, 1209, 1249, 1199, 1219, 1245,
↪  1209, 1234, 1211, 1218, 1213, 1201, 1228, 1202] avg 1216.3
Trying with 8 clients...
[1407, 1404, 1412, 1424, 1426, 1410, 1431, 1431, 1415, 1401, 1447, 1394,
↪  1394, 1447, 1404, 1420, 1439, 1418, 1403, 1422] avg 1417.45
Trying with 16 clients...
[1619, 1612, 1602, 1624, 1607, 1609, 1631, 1570, 1629, 1622, 1586, 1542,
↪  1626, 1616, 1616, 1646, 1629, 1618, 1636, 1630] avg 1613.5
Trying with 32 clients...
[1668, 1680, 1688, 1669, 1686, 1661, 1667, 1682, 1687, 1680, 1685, 1667,
↪  1689, 1697, 1680, 1674, 1697, 1656, 1687, 1693] avg 1679.65
Trying with 64 clients...
[1643, 1747, 1706, 1723, 1707, 1731, 1708, 1703, 1727, 1731, 1704, 1728,
↪  1733, 1709, 1730, 1724, 1710, 1716, 1704, 1708] avg 1714.6
Trying with 128 clients...
[1745, 1705, 1729, 1763, 1732, 1732, 1712, 1744, 1759, 1740, 1749, 1767,
↪  1749, 1716, 1738, 1737, 1752, 1751, 1754, 1758] avg 1741.6
Choosing 128 clients.
Running test with 128 clients...
```

```
[1746, 1758, 1751, 1750, 1750, 1745, 1735, 1725, 1745, 1759, 1746, 1712,
↪  1716, 1755, 1711, 1753, 1772, 1728, 1748, 1744, 1745, 1735, 1739, 1761,
↪  1761, 1721, 1741, 1751, 1746, 1726, 1763, 1720, 1734, 1723, 1750, 1745,
↪  1723, 1719, 1750, 1760, 1742, 1718, 1739, 1750, 1727, 1757, 1744, 1716,
↪  1731, 1732, 1747, 1774, 1732, 1723, 1754, 1743, 1716, 1748, 1760, 1737,
↪  1752, 1749, 1749, 1715, 1731, 1753, 1752, 1745, 1744, 1752, 1765, 1724,
↪  1749, 1762, 1747, 1725, 1761, 1766, 1760, 1728, 1736, 1737, 1762, 1731,
↪  1736, 1756, 1728, 1759, 1764, 1733, 1712, 1756, 1748, 1747, 1737, 1743,
↪  1748, 1749, 1735, 1753] avg 1742.51
```

A.4 Java REST Server, L messages — capped at 1 Gbit/s

```
Trying with 2 clients...
[136, 136, 139, 138, 139, 141, 140, 142, 139, 141, 140, 141, 141, 143, 140,
↪  141, 142, 142, 139, 139] avg 139.95
Trying with 4 clients...
[195, 204, 196, 196, 202, 208, 200, 194, 203, 199, 201, 200, 202, 198, 203,
↪  204, 205, 203, 206, 204] avg 201.15
Trying with 8 clients...
[229, 211, 218, 223, 229, 231, 220, 230, 226, 222, 227, 226, 223, 220, 229,
↪  227, 220, 223, 225, 226] avg 224.25
Trying with 16 clients...
[236, 238, 238, 234, 239, 236, 238, 234, 237, 235, 237, 232, 237, 232, 229,
↪  238, 232, 235, 239, 231] avg 235.35
Trying with 32 clients...
[240, 236, 237, 237, 237, 241, 236, 233, 240, 240, 236, 234, 236, 239, 238,
↪  231, 240, 237, 237, 239] avg 237.2
Choosing 32 clients.
Running test with 32 clients...
[237, 238, 233, 234, 242, 233, 239, 235, 240, 238, 238, 234, 241, 229, 236,
↪  243, 233, 237, 241, 232, 240, 235, 235, 240, 236, 235, 236, 240, 230,
↪  240, 238, 237, 237, 239, 235, 242, 230, 237, 242, 237, 234, 233, 237,
↪  238, 240, 233, 240, 239, 235, 240, 232, 238, 239, 234, 243, 230, 233,
↪  240, 235, 236, 239, 242, 233, 239, 235, 237, 232, 238, 241, 238, 234,
↪  236, 238, 233, 237, 237, 239, 237, 237, 234, 238, 239, 233, 239, 233,
↪  235, 236, 238, 237, 237, 238, 235, 240, 235, 239, 232, 240, 241, 232,
↪  238] avg 236.74
```

A.5 Java gRPC Server, XS messages

```
Trying with 2 clients...
[846, 2761, 2731, 2955, 2917, 2964, 2954, 2964, 3020, 3009, 2992, 2917, 3057,
↪  2802, 2850, 2835, 2836, 2804, 2807, 2823] avg 2792.2
Trying with 4 clients...
[4289, 4472, 4471, 4489, 4474, 4453, 4454, 4488, 4484, 4459, 4435, 4447,
↪  4450, 4467, 4446, 4413, 4405, 4463, 4478, 4454] avg 4449.55
Trying with 8 clients...
[5416, 5285, 5436, 5511, 5496, 5478, 5494, 5432, 5407, 5432, 5394, 5487,
↪  5493, 5529, 5550, 5520, 5543, 5576, 5514, 5508] avg 5475.05
Trying with 16 clients...
[3259, 3161, 5498, 5434, 5427, 5804, 5782, 5788, 5771, 5779, 5756, 5790,
↪  5777, 5784, 5766, 5776, 5802, 5742, 5727, 5726] avg 5467.45
Choosing 8 clients.
Running test with 8 clients...
```

```
[4857, 4863, 4900, 4901, 4882, 4888, 4904, 4874, 4909, 4912, 4851, 4871,
↪  4906, 4899, 4695, 2911, 2994, 3049, 4876, 4852, 4833, 4839, 4857, 4854,
↪  4885, 4929, 4909, 4906, 4898, 4905, 4835, 4661, 5542, 5519, 5529, 5523,
↪  5529, 5523, 5558, 5538, 5534, 5543, 5558, 5559, 5545, 5554, 5561, 5556,
↪  5547, 5587, 5545, 3612, 2822, 2955, 3228, 4849, 4706, 4229, 4722, 4842,
↪  4891, 4893, 4888, 4886, 4880, 4883, 4876, 4876, 4879, 4850, 4827, 3351,
↪  2903, 2815, 2819, 2924, 2719, 2810, 3095, 2996, 4506, 4751, 4897, 4873,
↪  4898, 4851, 4848, 4893, 4843, 4858, 4890, 4883, 4896, 4879, 4879, 4876,
↪  4888, 4870, 4894, 4851] avg 4685.05
```

A.6 Java gRPC Server, S messages

```
Trying with 2 clients...
[2526, 2542, 2545, 2523, 2548, 2537, 2522, 2543, 2575, 2550, 2553, 2538,
↪  2563, 2555, 2563, 2554, 2563, 2579, 2561, 2563] avg 2550.15
Trying with 4 clients...
[3434, 3577, 3541, 3535, 3601, 3533, 3557, 3524, 3526, 3558, 3552, 3544,
↪  3546, 3516, 3517, 3528, 3496, 3467, 3533, 3512] avg 3529.85
Trying with 8 clients...
[4389, 4243, 4369, 4340, 4351, 4361, 4375, 4359, 4347, 4347, 4364, 4367,
↪  4351, 4348, 4357, 4380, 4361, 4374, 4379, 3765] avg 4326.35
Trying with 16 clients...
[5780, 4192, 4833, 5742, 5720, 3396, 2752, 2740, 2774, 2739, 2743, 2749,
↪  2701, 2737, 4203, 5740, 5749, 5751, 5763, 5739] avg 4227.15
Choosing 8 clients.
Running test with 8 clients...
[4935, 4918, 4920, 4898, 4892, 4903, 4901, 2619, 2607, 2600, 2556, 2483,
↪  2555, 4193, 4336, 4333, 4346, 4346, 4332, 4359, 4357, 4339, 4319, 4329,
↪  4345, 4347, 4335, 4322, 4358, 4323, 4346, 4363, 4353, 4328, 4338, 4343,
↪  4327, 4379, 4362, 4350, 4322, 4298, 4288, 4330, 4316, 4315, 4314, 4316,
↪  4310, 4314, 4317, 3734, 4329, 3178, 4523, 4918, 4909, 4906, 4908, 4914,
↪  4914, 4924, 4941, 4928, 4889, 4934, 4900, 4910, 4529, 4156, 4352, 4332,
↪  4342, 4363, 4361, 4322, 4348, 4341, 4345, 4363, 4345, 4330, 4328, 4365,
↪  4376, 4354, 4340, 4337, 4363, 4358, 4344, 4324, 4352, 4344, 4335, 4360,
↪  4339, 4348, 4360, 4353] avg 4331.03
```

A.7 Java gRPC Server, M messages

```
Trying with 2 clients...
[1399, 1426, 1392, 1408, 1338, 1285, 1310, 1300, 1316, 1322, 1324, 1336,
↪  1312, 1334, 1331, 1323, 1316, 1332, 1345, 1422] avg 1343.55
Trying with 4 clients...
[1651, 1694, 1904, 1800, 1687, 1676, 1683, 1685, 1687, 1695, 1701, 1698,
↪  1687, 1657, 1684, 1687, 1699, 1700, 1694, 1695] avg 1703.2
Trying with 8 clients...
[2280, 1908, 2258, 2061, 2514, 2038, 2353, 2132, 2413, 2371, 2370, 2450,
↪  2478, 2405, 2520, 2505, 2220, 2309, 2386, 1893] avg 2293.2
Trying with 16 clients...
[2225, 2224, 2082, 2235, 2231, 2253, 2047, 2240, 2253, 2255, 2253, 2243,
↪  2252, 2250, 2264, 1747, 3139, 3186, 3173, 3171] avg 2386.15
Trying with 32 clients...
[3148, 3179, 3338, 3186, 2923, 3230, 2133, 3086, 3238, 3264, 3266, 3149,
↪  3245, 3114, 3209, 3321, 3332, 3083, 3218, 3236] avg 3144.9
Trying with 64 clients...
[2992, 3051, 2947, 3225, 3057, 2449, 2521, 2462, 2905, 2317, 2554, 2123,
↪  2203, 2070, 2267, 2162, 2263, 2350, 2255, 2101] avg 2513.7
Choosing 32 clients.
Running test with 32 clients...
```

```
[2308, 2339, 2328, 2366, 2401, 2360, 2314, 2358, 2390, 2339, 2325, 2232,
↪  2340, 2355, 2359, 2375, 2321, 2334, 2394, 2395, 2265, 2252, 2309, 2224,
↪  2280, 2270, 2345, 2307, 2377, 2405, 2360, 2390, 2365, 2386, 2299, 2284,
↪  2267, 2393, 2348, 2398, 2244, 2292, 2377, 2281, 2392, 2319, 2339, 2223,
↪  2233, 2247, 2334, 2280, 2323, 2363, 2384, 2354, 2256, 2334, 2334, 2397,
↪  2340, 2292, 2331, 2355, 2373, 2299, 2404, 2328, 2345, 2350, 2363, 2389,
↪  2267, 2303, 2364, 2260, 2284, 2380, 2356, 2341, 2369, 2221, 2281, 2392,
↪  2268, 2303, 2374, 2351, 2281, 2362, 2262, 2271, 2372, 2243, 2353, 2356,
↪  2181, 2392, 2296, 2384] avg 2327.99
```

A.8 Java gRPC Server, L messages — capped at 1 Gbit/s

```
Trying with 2 clients...
[288, 293, 294, 294, 286, 294, 295, 294, 293, 287, 293, 292, 293, 293, 288,
↪  295, 295, 296, 294, 289] avg 292.3
Trying with 4 clients...
[441, 477, 510, 456, 470, 507, 471, 456, 460, 449, 487, 485, 473, 501, 501,
↪  492, 460, 477, 478, 477] avg 476.4
Trying with 8 clients...
[489, 477, 526, 525, 494, 528, 528, 521, 467, 486, 495, 508, 464, 449, 385,
↪  510, 528, 528, 522, 487] avg 495.85
Trying with 16 clients...
[392, 510, 526, 509, 520, 525, 504, 432, 481, 479, 410, 383, 488, 525, 469,
↪  403, 390, 400, 446, 473] avg 463.25
Choosing 8 clients.
Running test with 8 clients...
[464, 508, 492, 487, 508, 521, 477, 501, 480, 517, 526, 500, 449, 499, 491,
↪  521, 524, 526, 516, 487, 497, 512, 475, 512, 512, 492, 510, 502, 525,
↪  520, 517, 467, 496, 491, 511, 493, 484, 509, 478, 459, 498, 518, 448,
↪  470, 460, 489, 494, 524, 452, 507, 505, 530, 526, 520, 508, 519, 504,
↪  467, 500, 497, 496, 485, 514, 492, 461, 515, 528, 494, 471, 493, 461,
↪  504, 493, 445, 502, 475, 436, 529, 500, 492, 501, 487, 463, 509, 527,
↪  528, 526, 525, 519, 506, 497, 487, 462, 499, 522, 489, 429, 488, 525,
↪  524] avg 496.91
```

A.9 Python REST Server, XS messages

```
Trying with 2 clients...
[3056, 3137, 3111, 3178, 3130, 3175, 3167, 3171, 3158, 3154, 3135, 3141,
↪  3184, 3127, 3158, 3179, 3175, 3177, 3175, 3139] avg 3151.35
Trying with 4 clients...
[3960, 4039, 4045, 4049, 4046, 4051, 4048, 4048, 4041, 4052, 4043, 4040,
↪  4045, 4050, 4050, 4055, 4050, 4050, 4044, 4034] avg 4042
Trying with 8 clients...
[5274, 5163, 5272, 5265, 5269, 5264, 5262, 5271, 5264, 5267, 5270, 5269,
↪  5273, 5271, 5272, 5270, 5276, 5268, 5259, 5265] avg 5263.2
Trying with 16 clients...
[4998, 4988, 5017, 4869, 4661, 4663, 4664, 4658, 4662, 4669, 4658, 4658,
↪  4672, 4739, 4736, 4734, 4731, 4737, 4726, 4719] avg 4747.95
Choosing 8 clients.
Running test with 8 clients...
[5731, 5739, 5735, 5736, 5737, 5723, 5718, 5712, 5728, 5733, 5728, 5697,
↪  5711, 5716, 5727, 5735, 5740, 5737, 5712, 5720, 5730, 5703, 5719, 5725,
↪  5714, 5715, 5706, 5705, 5726, 5741, 5739, 5714, 5720, 5693, 5736, 5745,
↪  5724, 5721, 5702, 5700, 5695, 5719, 5736, 5728, 5715, 5717, 5712, 5723,
↪  5723, 5729, 5714, 5689, 5737, 5734, 5731, 5729, 5716, 5715, 5743, 5755,
↪  5745, 5719, 5695, 5718, 5725, 5739, 5718, 5752, 5719, 5700, 5724, 5705,
↪  5741, 5728, 5730, 5738, 5715, 5703, 5736, 5740, 5729, 5740, 5722, 5708,
↪  5732, 5751, 5717, 5701, 5727, 5742, 5712, 5734, 5719, 5736, 5736, 5700,
↪  5717, 5698, 5744, 5739] avg 5723.47
```

A.10 Python REST Server, S messages

```
Trying with 2 clients...
[1672, 1652, 1670, 1662, 1672, 1676, 1680, 1676, 1670, 1682, 1680, 1666,
↪  1670, 1678, 1656, 1666, 1678, 1662, 1676, 1668] avg 1670.6
Trying with 4 clients...
[2479, 2546, 2550, 2548, 2546, 2548, 2544, 2548, 2543, 2549, 2544, 2552,
↪  2548, 2552, 2544, 2536, 2544, 2552, 2544, 2548] avg 2543.25
Trying with 8 clients...
[2686, 2631, 2686, 2700, 2697, 2705, 2698, 2707, 2697, 2701, 2705, 2704,
↪  2698, 2706, 2693, 2700, 2699, 2699, 2695, 2699] avg 2695.3
Trying with 16 clients...
[2996, 2953, 2897, 2970, 2933, 2912, 2868, 2922, 2924, 2902, 2841, 2914,
↪  2937, 2924, 2915, 2892, 2950, 2928, 2903, 2938] avg 2920.95
Trying with 32 clients...
[3206, 3168, 3179, 3227, 3143, 3154, 3166, 3193, 3142, 3183, 3131, 3088,
↪  3101, 3133, 3167, 3195, 3112, 3155, 3146, 3108] avg 3154.85
Trying with 64 clients...
[3477, 3446, 3454, 3467, 3477, 3431, 3451, 3429, 3464, 3459, 3459, 3486,
↪  3433, 3445, 3461, 3434, 3441, 3445, 3469, 3447] avg 3453.75
Trying with 128 clients...
[3325, 3381, 3396, 3359, 3413, 3360, 3395, 3377, 3407, 3388, 3368, 3399,
↪  3402, 3358, 3396, 3385, 3370, 3413, 3382, 3374] avg 3382.4
Choosing 64 clients.
Running test with 64 clients...
[3093, 3119, 3119, 3113, 3120, 3101, 3119, 3073, 3115, 3109, 3122, 3108,
↪  3114, 3114, 3115, 3081, 3075, 3070, 3066, 3072, 3059, 3068, 3069, 3082,
↪  3095, 3124, 3140, 3109, 3093, 3118, 3125, 3119, 3105, 3134, 3107, 3113,
↪  3105, 3086, 3092, 3104, 3121, 3121, 3096, 3114, 3123, 3129, 3124, 3114,
↪  3115, 3120, 3105, 3115, 3156, 3159, 3114, 3094, 3106, 3123, 3096, 3072,
↪  3070, 3074, 3065, 3081, 3062, 3086, 3089, 3093, 3108, 3113, 3098, 3103,
↪  3117, 3109, 3111, 3103, 3113, 3111, 3108, 3096, 3110, 3121, 3080, 3086,
↪  3062, 3058, 3078, 3078, 3061, 3059, 3065, 3060, 3091, 3063, 3105, 3116,
↪  3070, 3079, 3081, 3079] avg 3098.94
```

A.11 Python REST Server, M messages

```
Trying with 2 clients...
[46, 43, 45, 45, 44, 46, 43, 45, 44, 45, 44, 45, 44, 44, 44, 46, 44, 45, 44,
↪  44] avg 44.5
Trying with 4 clients...
[65, 65, 66, 64, 67, 67, 65, 68, 64, 67, 66, 67, 64, 67, 66, 65, 68, 65, 66,
↪  65] avg 65.85
Trying with 8 clients...
[86, 83, 87, 85, 86, 85, 88, 84, 86, 84, 86, 85, 88, 84, 86, 88, 82, 87, 86,
↪  83] avg 85.45
Trying with 16 clients...
[87, 84, 86, 84, 88, 85, 90, 86, 83, 89, 86, 85, 87, 82, 87, 89, 80, 89, 89,
↪  84] avg 86
Choosing 16 clients.
Running test with 16 clients...
[66, 69, 65, 60, 67, 73, 68, 65, 61, 66, 73, 68, 59, 66, 67, 68, 72, 60, 67,
↪  67, 67, 65, 67, 68, 66, 67, 63, 70, 67, 67, 66, 59, 74, 67, 67, 67, 59,
↪  73, 68, 67, 58, 68, 66, 74, 66, 60, 67, 67, 73, 60, 67, 66, 68, 71, 61,
↪  67, 67, 67, 66, 67, 66, 68, 65, 67, 67, 67, 67, 66, 66, 68, 67, 66, 60,
↪  72, 68, 67, 66, 59, 67, 73, 67, 67, 59, 67, 73, 67, 59, 67, 68, 72, 67,
↪  60, 67, 67, 73, 59, 68, 66, 68, 65] avg 66.45
```

A.12 Python REST Server, L messages

```
Trying with 2 clients...
[6, 6, 6, 6, 4, 6, 6, 6, 6, 6, 6, 5, 5, 6, 6, 6, 6, 6, 5, 6] avg 5.75
Trying with 4 clients...
[12, 12, 11, 10, 11, 12, 11, 12, 10, 10, 12, 12, 12, 9, 11, 12, 12, 11, 10,
↪  12] avg 11.2
Trying with 8 clients...
[10, 11, 12, 12, 10, 12, 11, 10, 12, 12, 10, 12, 10, 12, 11, 10, 11, 11, 13,
↪  11] avg 11.15
Choosing 4 clients.
Running test with 4 clients...
[7, 6, 5, 6, 6, 6, 6, 6, 6, 5, 6, 5, 6, 6, 6, 5, 6, 6, 6, 6, 6, 5, 6, 5, 6,
↪  6, 6, 6, 6, 5, 6, 6, 6, 5, 6, 5, 6, 6, 6, 6, 6, 6, 5, 6, 5, 6, 6, 6, 5,
↪  6, 6, 6, 6, 5, 5, 6, 6, 6, 6, 6, 5, 6, 6, 5, 6, 6, 5, 6, 6, 6, 6,
↪  6, 5, 6, 5, 6, 6, 6, 6, 5, 6, 6, 6, 5, 6, 5, 6, 6, 6, 6, 6, 6, 5, 6, 6,
↪  5, 6, 6] avg 5.77
```

A.13 Python gRPC Server, XS messages

```
Trying with 2 clients...
[2189, 2216, 2204, 2189, 2147, 2132, 2143, 2047, 2050, 2055, 2057, 2061,
↪  2059, 2059, 2052, 2062, 2058, 2056, 2058, 2052] avg 2097.3
Trying with 4 clients...
[1873, 1859, 1920, 1967, 1972, 1911, 1963, 1938, 1973, 1939, 1887, 1949,
↪  1966, 1932, 1831, 1969, 1903, 1983, 1962, 1971] avg 1933.4
Choosing 2 clients.
Running test with 2 clients...
[2055, 1923, 2030, 2054, 2014, 2053, 2056, 2103, 2052, 2045, 2055, 2067,
↪  2060, 2047, 2056, 1993, 1913, 2040, 2083, 2088, 2042, 2017, 1949, 1912,
↪  2013, 1966, 1928, 2008, 2086, 2052, 2056, 2003, 2077, 2074, 2073, 2060,
↪  2070, 1903, 1884, 1961, 2032, 2052, 2060, 2002, 2075, 2054, 2089, 2044,
↪  1928, 1874, 1887, 1889, 1974, 2056, 2068, 2081, 2052, 2064, 2060, 2049,
↪  2059, 2045, 2073, 2093, 1936, 2051, 2070, 2054, 2050, 2059, 2002, 1877,
↪  1873, 1915, 1912, 1911, 1912, 1908, 2059, 2062, 2007, 2033, 1999, 1949,
↪  1972, 1904, 1992, 2085, 2001, 2018, 2034, 2003, 2082, 1953, 1849, 1909,
↪  1920, 1897, 1903, 1914] avg 2006.56
```

A.14 Python gRPC Server, S messages

```
Trying with 2 clients...
[1900, 1951, 1830, 1791, 1834, 1821, 1840, 1856, 1823, 1804, 1818, 1944,
↪  1881, 1919, 1862, 1878, 1841, 1823, 1829, 1828] avg 1853.65
Trying with 4 clients...
[1698, 1730, 1789, 1789, 1782, 1649, 1741, 1774, 1787, 1778, 1736, 1798,
↪  1772, 1783, 1781, 1749, 1782, 1781, 1801, 1800] avg 1765
Choosing 2 clients.
Running test with 2 clients...
[1902, 2013, 1869, 1991, 1995, 1996, 1866, 1871, 1853, 1927, 1921, 1814,
↪  1807, 1809, 1800, 1813, 1819, 1816, 1822, 1834, 1826, 1823, 1828, 1829,
↪  1882, 2005, 1995, 1967, 1864, 1835, 1806, 1827, 1930, 1902, 1983, 1992,
↪  1991, 2005, 2005, 1962, 1974, 1975, 1969, 1820, 1876, 1994, 1870, 1873,
↪  1799, 1850, 1886, 1941, 2006, 2019, 1912, 1828, 1823, 1823, 1831, 1818,
↪  1808, 1822, 1833, 1842, 1823, 1822, 1832, 1827, 1830, 1839, 1973, 1852,
↪  1827, 1906, 1818, 1810, 1815, 1813, 1823, 1821, 1815, 1813, 1824, 1829,
↪  1827, 1829, 1835, 1825, 1825, 1810, 1807, 1817, 1832, 1830, 1978, 2006,
↪  1815, 1844, 1865, 1831] avg 1872.75
```

A.15 Python gRPC Server, M messages

```
Trying with 2 clients...
[644, 654, 649, 647, 637, 629, 653, 638, 656, 642, 628, 652, 648, 648, 647,
↪ 626, 646, 653, 642, 643] avg 644.1
Trying with 4 clients...
[675, 677, 672, 673, 476, 531, 673, 676, 676, 668, 642, 667, 666, 667, 661,
↪ 588, 618, 667, 666, 666] avg 645.25
Choosing 4 clients.
Running test with 4 clients...
[541, 666, 664, 665, 664, 609, 662, 658, 661, 666, 636, 455, 497, 664, 662,
↪ 665, 666, 620, 638, 663, 663, 666, 664, 640, 664, 663, 658, 661, 476,
↪ 446, 456, 661, 663, 664, 666, 646, 629, 657, 657, 662, 664, 461, 657,
↪ 663, 668, 668, 663, 613, 663, 665, 665, 661, 520, 670, 666, 662, 664,
↪ 621, 432, 592, 670, 667, 662, 663, 540, 662, 662, 664, 663, 622, 587,
↪ 663, 662, 658, 665, 523, 632, 662, 664, 662, 660, 524, 665, 663, 662,
↪ 665, 605, 627, 666, 662, 661, 659, 634, 657, 664, 667, 662, 487, 648,
↪ 663] avg 633.76
```

A.16 Python gRPC Server, L messages

```
Trying with 2 clients...
[129, 132, 129, 132, 126, 129, 128, 133, 131, 118, 130, 132, 127, 126, 116,
↪ 132, 132, 131, 127, 119] avg 127.95
Trying with 4 clients...
[130, 130, 132, 132, 123, 115, 131, 134, 132, 132, 125, 131, 135, 130, 128,
↪ 108, 130, 133, 132, 130] avg 128.65
Choosing 4 clients.
Running test with 4 clients...
[111, 130, 130, 131, 132, 105, 126, 130, 128, 131, 129, 102, 130, 131, 129,
↪ 132, 118, 109, 131, 129, 131, 130, 124, 131, 130, 130, 131, 119, 132,
↪ 131, 133, 132, 126, 104, 131, 131, 131, 131, 115, 111, 132, 131, 131,
↪ 131, 105, 132, 129, 130, 130, 132, 121, 133, 128, 131, 130, 116, 130,
↪ 130, 131, 130, 116, 132, 132, 129, 129, 122, 124, 129, 129, 129, 133,
↪ 125, 134, 130, 132, 130, 116, 131, 135, 130, 132, 115, 132, 131, 129,
↪ 130, 126, 131, 131, 129, 133, 133, 125, 129, 132, 131, 131, 127, 130,
↪ 132] avg 127.47
```

A.17 Rust REST Server, XS messages

```
Trying with 2 clients...
[6107, 6220, 6115, 6383, 5619, 6441, 6200, 6161, 6328, 6091, 6252, 6595,
↪ 6230, 6171, 6189, 6575, 6295, 6248, 6479, 6228] avg 6246.35
Trying with 4 clients...
[11267, 11331, 11246, 11195, 10913, 11216, 11212, 11482, 11177, 11064, 11173,
↪ 11064, 11328, 11116, 11398, 10716, 11021, 11313, 11133, 11265] avg
↪ 11181.5
Trying with 8 clients...
[21018, 19110, 19602, 19298, 19126, 19584, 19474, 19201, 19038, 18717, 19325,
↪ 19337, 19108, 17893, 18894, 19321, 19048, 18759, 19176, 18926] avg
↪ 19197.75
Trying with 16 clients...
[30976, 28584, 28687, 28551, 28590, 28602, 28600, 28588, 28550, 28577, 28615,
↪ 28571, 28514, 28577, 28431, 28337, 28526, 28616, 28604, 28588] avg
↪ 28684.2
Trying with 32 clients...
[47557, 46273, 45808, 45789, 45237, 46032, 45659, 45971, 44913, 45129, 45837,
↪ 45665, 45367, 45709, 45324, 46170, 45650, 45689, 46219, 46228] avg
↪ 45811.3
Trying with 64 clients...
```

```
[59114, 57978, 60337, 58239, 56322, 57670, 57659, 57820, 57402, 55502, 59420,
↪  58399, 56260, 57711, 58782, 61693, 57190, 59044, 57003, 56732] avg
↪  58013.85
Trying with 128 clients...
[65039, 64341, 66358, 62373, 65709, 66261, 65833, 66342, 63755, 63732, 64035,
↪  66622, 65451, 67159, 66948, 65816, 64590, 65902, 63557, 63618] avg
↪  65172.05
Trying with 256 clients...
[63522, 63922, 64065, 63554, 65225, 63586, 65389, 64243, 63578, 64503, 62964,
↪  63663, 63826, 65628, 65321, 64989, 65360, 65538, 65245, 64613] avg
↪  64436.7
Choosing 128 clients.
Running test with 128 clients...
[63300, 60927, 65508, 61139, 62694, 65039, 62290, 65502, 64430, 61606, 61257,
↪  59209, 59202, 61023, 62689, 61861, 62248, 62412, 61548, 59306, 60196,
↪  61153, 66405, 62885, 61220, 61261, 61087, 60631, 63404, 62984, 63555,
↪  62735, 60650, 62156, 61316, 60187, 61202, 65255, 60535, 65925, 61597,
↪  62671, 62667, 60422, 61084, 61694, 60475, 62093, 60970, 62481, 63094,
↪  62336, 60482, 60847, 61973, 62764, 62093, 61445, 60137, 59684, 61158,
↪  60625, 62826, 60871, 61750, 61260, 60465, 63517, 65165, 62227, 63250,
↪  63543, 60740, 63319, 64185, 61202, 63735, 62032, 63481, 60986, 63661,
↪  62145, 61334, 64439, 62285, 62559, 62373, 63815, 62761, 64190, 62719,
↪  61638, 60690, 61660, 61731, 60405, 60578, 59712, 64326, 63557] avg
↪  62118.56
```

A.18 Rust REST Server, S messages

```
Trying with 2 clients...
[5237, 5244, 5311, 5238, 5201, 5238, 5226, 5279, 5257, 5250, 5253, 5236,
↪  5286, 5252, 5266, 5180, 5244, 5196, 5223, 5264] avg 5244.05
Trying with 4 clients...
[10152, 10174, 10156, 10172, 10019, 10179, 10165, 10163, 10188, 10054, 10178,
↪  10173, 10190, 10194, 10075, 10165, 10194, 10157, 10181, 10061] avg
↪  10149.5
Trying with 8 clients...
[18994, 19293, 19278, 19226, 19253, 19146, 19168, 19310, 19283, 19296, 19106,
↪  18689, 18938, 19238, 19277, 19318, 18873, 18975, 18924, 18845] avg
↪  19121.5
Trying with 16 clients...
[29738, 26955, 26845, 25659, 28971, 30240, 27496, 26909, 26708, 27122, 27162,
↪  27231, 27240, 27248, 26662, 26641, 26860, 27192, 27198, 27160] avg
↪  27361.85
Trying with 32 clients...
[42836, 43336, 43539, 43821, 42815, 40643, 43912, 44453, 44202, 44403, 44608,
↪  42548, 44310, 43940, 44900, 43341, 42475, 42518, 44316, 44461] avg
↪  43568.85
Trying with 64 clients...
[54864, 51081, 53044, 53424, 53247, 53457, 52837, 52814, 52681, 53454, 53004,
↪  53209, 52140, 53941, 53415, 52675, 50204, 51531, 53699, 54120] avg
↪  52942.05
Trying with 128 clients...
[52887, 53098, 53649, 53939, 54056, 52212, 53917, 53794, 52826, 54759, 52690,
↪  54740, 54121, 52895, 54193, 54732, 54614, 53778, 54207, 53974] avg
↪  53754.05
Choosing 128 clients.
Running test with 128 clients...
```

```
[53836, 53012, 53444, 52366, 52706, 52790, 52957, 53595, 52142, 53310, 51742,
↪  52194, 52817, 53176, 52586, 52014, 53706, 53471, 52941, 53678, 52510,
↪  51808, 53667, 53683, 52849, 53231, 52750, 53159, 53896, 53217, 52421,
↪  52430, 53799, 53039, 53659, 53905, 53236, 53162, 53624, 53766, 53032,
↪  51260, 52454, 54107, 53352, 52888, 52803, 52787, 53468, 53254, 53500,
↪  53284, 52168, 52640, 53044, 51522, 53232, 53391, 52468, 53040, 53277,
↪  52459, 53663, 52833, 55163, 53078, 53234, 52894, 53176, 53737, 51563,
↪  52596, 52906, 54293, 53563, 53409, 52537, 52800, 53219, 52226, 53618,
↪  52553, 53319, 53852, 53540, 53216, 53556, 54150, 53514, 53242, 53644,
↪  53280, 53243, 53840, 52767, 52407, 52528, 52419, 52996, 53875] avg
↪  53081.73
```

A.19 Rust REST Server, M messages — capped at 1 Gbit/s

```
Trying with 2 clients...
[1198, 1453, 1468, 1465, 1469, 1474, 1473, 1475, 1477, 1476, 1248, 1158,
↪  1153, 1154, 1155, 1157, 1157, 1155, 1160, 1404] avg 1316.45
Trying with 4 clients...
[1951, 2004, 2008, 2009, 2005, 2010, 1996, 2006, 2042, 2005, 2012, 2028,
↪  2024, 2024, 2012, 2012, 2020, 2000, 2004, 1981] avg 2007.65
Trying with 8 clients...
[2310, 2310, 2325, 2316, 2327, 2312, 2312, 2326, 2326, 2319, 2314, 2317,
↪  2324, 2322, 2330, 2322, 2315, 2317, 2318, 2319] avg 2319.05
Trying with 16 clients...
[2349, 2349, 2351, 2348, 2349, 2352, 2348, 2348, 2351, 2348, 2351, 2348,
↪  2350, 2349, 2351, 2347, 2352, 2349, 2346, 2351] avg 2349.35
Choosing 16 clients.
Running test with 16 clients...
[2349, 2351, 2346, 2351, 2350, 2349, 2349, 2349, 2351, 2349, 2351, 2349,
↪  2345, 2351, 2351, 2348, 2350, 2351, 2349, 2348, 2349, 2349, 2349, 2350,
↪  2349, 2350, 2350, 2350, 2349, 2349, 2348, 2351, 2349, 2348, 2351, 2348,
↪  2350, 2351, 2349, 2349, 2349, 2351, 2347, 2350, 2350, 2348, 2351, 2348,
↪  2351, 2348, 2352, 2346, 2350, 2350, 2350, 2350, 2346, 2351, 2350, 2348,
↪  2349, 2350, 2349, 2351, 2349, 2350, 2348, 2350, 2348, 2348, 2352, 2348,
↪  2349, 2350, 2351, 2348, 2349, 2351, 2348, 2349, 2350, 2348, 2351, 2350,
↪  2348, 2348, 2350, 2351, 2350, 2347, 2350, 2350, 2350, 2350, 2348, 2351,
↪  2348, 2348, 2352, 2348] avg 2349.36
```

A.20 Rust REST Server, L messages — capped at 1 Gbit/s

```
Trying with 2 clients...
[110, 110, 110, 111, 111, 110, 111, 111, 111, 110, 111, 110, 111, 110, 111,
↪  111, 110, 111, 111, 110] avg 110.55
Trying with 4 clients...
[176, 182, 189, 189, 193, 186, 189, 189, 186, 187, 182, 185, 185, 185, 193,
↪  181, 186, 190, 192, 188] avg 186.65
Trying with 8 clients...
[224, 227, 222, 225, 226, 228, 226, 228, 229, 225, 224, 227, 223, 225, 226,
↪  228, 226, 223, 223, 227] avg 225.6
Trying with 16 clients...
[223, 232, 229, 231, 230, 224, 231, 223, 232, 228, 227, 228, 224, 228, 228,
↪  225, 228, 228, 226, 227] avg 227.6
Choosing 16 clients.
Running test with 16 clients...
```

```
[230, 226, 233, 232, 231, 230, 232, 233, 235, 231, 231, 232, 230, 232, 230,
↪  232, 233, 231, 233, 231, 232, 229, 234, 230, 232, 231, 234, 228, 233,
↪  227, 231, 227, 229, 231, 229, 229, 232, 228, 231, 227, 225, 228, 228,
↪  228, 228, 229, 224, 229, 230, 231, 231, 226, 232, 226, 232, 226, 230,
↪  230, 229, 232, 226, 223, 231, 222, 231, 226, 232, 226, 234, 231, 231,
↪  232, 227, 231, 225, 231, 231, 229, 232, 227, 231, 228, 230, 230, 230,
↪  227, 231, 227, 229, 228, 229, 224, 228, 229, 229, 225, 229, 230, 222,
↪  228] avg 229.48
```

A.21 Rust gRPC Server, XS messages

```
Trying with 2 clients...
[5111, 5108, 5111, 5114, 5013, 5114, 5122, 5118, 5111, 5060, 5099, 5085,
↪  5129, 5133, 5032, 5113, 5138, 5119, 5112, 5059] avg 5100.05
Trying with 4 clients...
[6307, 6589, 6809, 6853, 6578, 6550, 6392, 6497, 6130, 6803, 6492, 6169,
↪  6382, 6622, 6714, 5889, 6804, 6859, 6793, 6796] avg 6551.4
Trying with 8 clients...
[9398, 9357, 9365, 9358, 9381, 9335, 9401, 9357, 9373, 9380, 9338, 9342,
↪  9324, 9355, 9316, 9288, 9298, 9331, 9352, 9314] avg 9348.15
Trying with 16 clients...
[10164, 10709, 10814, 10705, 10777, 10706, 10645, 10690, 10682, 10722, 10734,
↪  10766, 10735, 10688, 10759, 10751, 10761, 10666, 10665, 10767] avg
↪  10695.3
Trying with 32 clients...
[11460, 10938, 11371, 11388, 11339, 11520, 11408, 11412, 11353, 11386, 11373,
↪  11401, 11393, 11425, 11429, 11344, 11470, 11474, 11332, 11318] avg
↪  11376.7
Trying with 64 clients...
[11821, 11627, 11891, 11720, 11824, 11651, 11806, 11865, 11749, 11633, 11736,
↪  11819, 11705, 11716, 11748, 11780, 11766, 11655, 11809, 11850] avg
↪  11758.55
Trying with 128 clients...
[11882, 12473, 12619, 12471, 12025, 12467, 12283, 12561, 12221, 12353, 12572,
↪  12296, 12450, 12217, 12352, 12399, 12189, 12438, 12357, 12366] avg
↪  12349.55
Trying with 256 clients...
[13189, 12754, 13328, 13422, 13205, 13178, 13374, 13166, 13125, 13244, 13362,
↪  12996, 13250, 13307, 13207, 13154, 13054, 13162, 13218, 12909] avg
↪  13180.2
Trying with 512 clients...
[13882, 13891, 14009, 13962, 13838, 13304, 14144, 14091, 13662, 13880, 14092,
↪  13969, 13977, 13746, 13915, 13873, 13903, 14025, 13857, 13866] avg
↪  13894.3
Trying with 1024 clients...
[13180, 13743, 13439, 14225, 13913, 13377, 14404, 13703, 13672, 14218, 13933,
↪  13277, 13959, 13754, 13922, 12996, 14457, 13752, 14135, 13254] avg
↪  13765.65
Choosing 512 clients.
Running test with 512 clients...
[13680, 13917, 14118, 14137, 14008, 13621, 13711, 13526, 13801, 14018, 13829,
↪  13959, 13488, 13917, 13653, 13847, 13904, 13620, 13935, 13443, 13876,
↪  13514, 13842, 13367, 13637, 14109, 13653, 13777, 13666, 13614, 13660,
↪  13663, 13629, 13481, 13703, 13706, 13782, 13727, 13464, 13424, 13709,
↪  13522, 13714, 13717, 13260, 13765, 13431, 13819, 13313, 13753, 13107,
↪  13785, 13710, 13467, 13253, 13587, 13895, 13214, 13243, 13843, 13562,
↪  13632, 13506, 13383, 13348, 13212, 13711, 13660, 13479, 13485, 13547,
↪  13423, 13830, 13269, 13721, 13632, 13575, 13316, 13703, 13539, 13480,
↪  13665, 13476, 13299, 13496, 13626, 13276, 13498, 13333, 13305, 13216,
↪  13866, 13211, 13348, 13366, 13452, 13654, 13288, 13420, 13603] avg
↪  13599.44
```

A.22 Rust gRPC Server, S messages

```
Trying with 2 clients...
[4535, 4362, 3871, 3953, 4437, 4143, 4383, 4420, 4378, 4096, 4301, 4057,
↪  4144, 4437, 4174, 4447, 4343, 4521, 4223, 4177] avg 4270.1
Trying with 4 clients...
[5637, 5672, 5735, 5778, 5737, 5832, 5638, 5666, 5844, 5737, 5635, 5823,
↪  5686, 5780, 5656, 5634, 5649, 5667, 5604, 5647] avg 5702.85
Trying with 8 clients...
[6968, 7000, 7029, 6966, 7025, 6994, 6951, 6956, 6960, 6954, 6964, 6966,
↪  6986, 6942, 6969, 7005, 6989, 6921, 6983, 6916] avg 6972.2
Trying with 16 clients...
[7373, 7508, 7605, 7481, 7601, 7663, 7564, 7507, 7655, 7518, 7563, 7532,
↪  7570, 7608, 7604, 7776, 7617, 7517, 7605, 7612] avg 7573.95
Trying with 32 clients...
[7736, 7737, 8073, 8128, 7877, 7994, 7947, 8021, 8034, 7958, 7999, 8190,
↪  7970, 7998, 7944, 8015, 8066, 8247, 8071, 8032] avg 8001.85
Trying with 64 clients...
[8410, 8391, 8428, 8329, 8400, 8472, 8450, 8622, 8312, 8400, 8564, 8402,
↪  8479, 8460, 8498, 8534, 8535, 8358, 8435, 8441] avg 8446
Trying with 128 clients...
[8830, 9207, 9371, 9163, 9088, 9244, 9259, 9095, 9259, 9085, 9178, 9435,
↪  9540, 9339, 9173, 9373, 9310, 9333, 9424, 9362] avg 9253.4
Trying with 256 clients...
[9698, 8994, 9407, 9645, 9854, 9380, 9794, 9596, 9405, 9523, 9300, 9815,
↪  9605, 9551, 9405, 9433, 9239, 9040, 9258, 9586] avg 9476.4
Trying with 512 clients...
[9868, 9876, 9825, 9771, 10779, 10495, 9875, 9831, 10003, 10018, 9210, 10116,
↪  9848, 9875, 10117, 10514, 10052, 10048, 9923, 10138] avg 10009.1
Trying with 1024 clients...
[10236, 10748, 11133, 11769, 11523, 11184, 10936, 11269, 11185, 10237, 11729,
↪  10372, 10751, 11207, 10613, 11716, 11442, 10539, 11246, 11766] avg
↪  11080.05
Trying with 2048 clients...
[9764, 10420, 11756, 11134, 12347, 11588, 11188, 11482, 11104, 10098, 12473,
↪  11024, 10655, 11303, 12847, 10853, 10999, 10930, 10691, 11159] avg
↪  11190.75
Choosing 2048 clients.
Running test with 2048 clients...
[12032, 11263, 11138, 12411, 10590, 12364, 10943, 11928, 11325, 10062, 10307,
↪  10907, 11889, 11309, 9970, 11609, 11130, 10266, 10692, 11266, 11752,
↪  10680, 11434, 12720, 11407, 10107, 12087, 12117, 10949, 11088, 11052,
↪  10732, 10615, 11414, 12441, 11646, 10617, 10294, 10667, 10875, 11411,
↪  10669, 12297, 11119, 11116, 11457, 10923, 12143, 10153, 11275, 10451,
↪  12244, 10368, 11726, 10878, 10266, 11375, 11898, 12363, 10747, 10921,
↪  11483, 11464, 11024, 10490, 10701, 11605, 12016, 10151, 11547, 11697,
↪  11441, 9926, 9518, 11202, 10963, 10521, 11251, 11899, 10528, 11506,
↪  11108, 10921, 10439, 12625, 9682, 11982, 11254, 11262, 12250, 10352,
↪  10923, 10000, 11455, 11002, 11708, 11242, 11437, 10177, 9994] avg
↪  11146.41
```

A.23 Rust gRPC Server, M messages

```
Trying with 2 clients...
[1885, 1882, 1849, 1882, 1867, 1853, 1913, 1914, 1897, 1894, 1889, 1885,
↪  1882, 1840, 1891, 1890, 1895, 1893, 1911, 1892] avg 1885.2
Trying with 4 clients...
[2292, 2386, 2403, 2382, 2351, 2328, 2408, 2418, 2428, 2408, 2352, 2439,
↪  2351, 2382, 2402, 2338, 2390, 2389, 2368, 2408] avg 2381.15
Trying with 8 clients...
[3006, 3018, 2999, 3006, 3000, 2978, 3009, 2998, 2975, 3003, 3030, 3005,
↪  3018, 2989, 2998, 3005, 2999, 3012, 2987, 2959] avg 2999.7
Trying with 16 clients...
[3179, 3277, 3332, 3346, 3320, 3277, 3262, 3256, 3323, 3289, 3281, 3305,
↪  3280, 3255, 3281, 3249, 3269, 3257, 3299, 3279] avg 3280.8
```

```
Trying with 32 clients...
[3513, 3315, 3564, 3553, 3412, 3491, 3529, 3494, 3488, 3556, 3466, 3525,
↪  3437, 3449, 3456, 3558, 3433, 3545, 3516, 3526] avg 3491.3
Trying with 64 clients...
[3682, 3620, 3852, 3694, 3548, 3654, 3504, 3602, 3701, 3612, 3501, 3593,
↪  3766, 3692, 3756, 3467, 3647, 3661, 3690, 3840] avg 3654.1
Trying with 128 clients...
[3792, 3719, 3859, 4110, 4125, 3939, 4201, 4090, 3903, 3883, 3695, 3975,
↪  3898, 4166, 3882, 4034, 3947, 3797, 3974, 3794] avg 3939.15
Trying with 256 clients...
[3966, 3784, 4356, 4105, 3880, 3994, 4126, 3945, 4030, 4326, 3947, 3784,
↪  3960, 4223, 4269, 3920, 4016, 3904, 3941, 4117] avg 4029.65
Trying with 512 clients...
[3745, 3585, 3821, 3600, 3826, 3468, 3899, 3807, 3874, 3862, 3700, 4162,
↪  3796, 4035, 3668, 3870, 3637, 3642, 3861, 3851] avg 3785.45
Choosing 256 clients.
Running test with 256 clients...
[3931, 3914, 4314, 3728, 4119, 3725, 4040, 3896, 3858, 3948, 4002, 3638,
↪  4138, 3932, 4065, 3715, 4005, 4110, 3765, 4167, 4219, 3810, 3950, 3968,
↪  4193, 4097, 3757, 3721, 3772, 3667, 4044, 4045, 3778, 4138, 3879, 3718,
↪  3725, 3841, 3933, 3971, 4086, 3835, 3993, 3989, 4207, 4117, 3901, 4093,
↪  3858, 4169, 4182, 4162, 4030, 3637, 4058, 4039, 4044, 4006, 3914, 3896,
↪  3780, 3718, 3943, 3964, 3981, 3812, 3954, 3802, 3739, 3803, 4122, 4008,
↪  3754, 4071, 3644, 4094, 4240, 4053, 3806, 3852, 3784, 3907, 4369, 4020,
↪  3991, 4240, 3742, 3971, 4045, 4315, 4201, 4417, 4199, 3841, 4041, 3922,
↪  3991, 3805, 4042, 4096] avg 3965.31
```

A.24 Rust gRPC Server, L messages

```
Trying with 2 clients...
[175, 175, 174, 175, 175, 174, 175, 174, 175, 175, 174, 174, 175, 174, 174,
↪  175, 174, 174, 174, 175] avg 174.5
Trying with 4 clients...
[247, 243, 247, 252, 258, 251, 253, 256, 249, 252, 252, 248, 251, 245, 253,
↪  250, 258, 246, 251, 258] avg 251
Trying with 8 clients...
[305, 299, 308, 303, 307, 304, 304, 306, 302, 300, 306, 297, 302, 303, 299,
↪  303, 300, 303, 300, 300] avg 302.55
Trying with 16 clients...
[291, 289, 291, 318, 295, 303, 301, 294, 322, 303, 285, 288, 324, 304, 313,
↪  314, 309, 299, 303, 291] avg 301.85
Choosing 8 clients.
Running test with 8 clients...
[305, 305, 303, 297, 310, 305, 298, 306, 307, 303, 306, 299, 304, 302, 302,
↪  302, 307, 304, 300, 299, 302, 300, 298, 302, 297, 293, 304, 299, 297,
↪  301, 301, 307, 303, 302, 305, 306, 299, 298, 305, 302, 306, 307, 295,
↪  300, 304, 300, 304, 301, 306, 305, 297, 295, 305, 314, 295, 302, 300,
↪  299, 309, 303, 300, 298, 302, 301, 297, 298, 302, 304, 301, 305, 297,
↪  299, 302, 304, 301, 298, 296, 302, 302, 302, 301, 304, 306, 297, 302,
↪  306, 305, 305, 300, 307, 305, 297, 292, 296, 300, 303, 297, 299, 296,
↪  308] avg 301.69
```

TRITA-EECS-EX- 2023:437