National College of Ireland

Performance improvement in Communication Protocols in Node.js-based Microservices framework for Enhanced Latency and Scalability in Globally Dispersed Systems

MSc Research Project
Cloud Computing

Ashish Oli
Student ID: x23102926

School of Computing
National College of Ireland

Supervisor: Sean Heeney

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Ashish Oli
Student ID:	x23102926
Programme:	Cloud Computing
Year:	2024
Module:	MSc Research Project
Supervisor:	Sean Heeney
Submission Due Date:	12/08/2024
Project Title:	Performance improvement in Communication Protocols in Node.js-based Microservices framework for Enhanced Latency and Scalability in Globally Dispersed Systems
Word Count:	6293
Page Count:	21

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Ashish Oli
Date:	12th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Performance improvement in Communication Protocols in Node.js-based Microservices framework for Enhanced Latency and Scalability in Globally Dispersed Systems

Ashish Oli
x23102926

Abstract

This research measured the performance of HTTP and gRPC as communication protocols within a Node.js framework running in a distributed environment. The objective was to learn how these protocols would impact latency, throughput, and resource utilization in a distributed microservices architecture. This research implemented both an HTTP server and a gRPC server and benchmarked them to get full data on their performance metrics. The results show that gRPC performed better in latency than HTTP, at 183.16 ms against that of HTTP at 811.52 ms. This latency reduction is very critical for real-time applications requiring fast response times. Again, gRPC is more resource-efficient, as manifested by its low consumption of CPU and memory usage and reduced network bandwidth consumption, making it very suitable for large distributed applications.

Despite this, high throughput is combined with a large number of failures and high latency of the HTTP server, which became a good vivid case on how far it is from being reliable under heavy loads. The distinction of the gRPC server in performance comes from the efficient serialization via Protocol Buffers and support for HTTP/2, owing to which it is able to provide both better performance and fine-grained scalability. The current work outlines the importance of choosing the right communication protocols for microservices and gives some useful insights into the optimization of distributed systems. This research contributes several practical recommendations regarding the performance and scalability of microservice architectures and their communication protocols, which would be beneficial for developers and organizations that aim to enhance these features of their distributed applications.

Keywords: ***Microservices, Distributed systems, NodeJS, Cloud***

1 Introduction

In distributed computing, requirements for effective and scalable frameworks are surging with the rise of cloud computing and containerized applications. For that reason, robust solutions for transparent treatment of code execution on many different distributed machines have been asked for ever more urgently Kafhali et al. (2020). The present thesis

primarily deals with the design and implementation of a Node.js framework to cope with the above-mentioned difficulties and, particularly, automatic scaling and running code. In that respect, this study considers gRPC and HTTP servers implemented on AWS EC2 instances across different regions for evaluating the performance and efficiency of the proposed framework.

1.1 Motivation

The motivation behind this research work has come from the increasing complexity and demands of modern distributed systems Rahman and Lama (2019). Traditional approaches to the management of distributed applications normally turn out to be inefficient and poorly scaled, particularly in heterogeneous environments. AWS and similar cloud computing platforms opened access to new, unparalleled opportunities for application deployment at scale [1]. However, there is a huge gap between tools and frameworks that developers can harness in pursuit of these capabilities efficiently. The present research aims to fill this void by proposing a Node.js framework that abstracts scaling and running code transparently across distributed environments to enhance performance and lower overhead Kafhali et al. (2020).

1.2 Research Question

One of the central research questions this thesis addresses is:

How to improve a microservice framework in Node.js for efficient automation of scaling and execution over some distributed machines using better communication protocols namely gRPC and HTTP protocols on AWS EC2 instances.

1.3 Research Niche

This research works closly in the areas of intersection of distributed computing, cloud infrastructure, and software engineering. Building on previous work Chaplia and Klym (2023), the current research is focused on practical implementation and benchmarking a Node.js framework using gRPC and HTTP protocols in this area. This solves a very important niche for the developer and organization aimed at making their distributed applications efficient in performance and scalability.

1.4 Research Objectives

The objectives of this research are as follows:

- **Improve the Node.js framework** that supports automated code scaling and execution across distributed machines effieciently.
- **Implement gRPC and HTTP servers** within the framework and deploy them on AWS EC2 instances across different regions.

[1] `https://aws.amazon.com/isv/scalability/`

- **Evaluate the performance** of the gRPC and HTTP servers using CLI benchmarking tools such as wrk and ghz.
- **Analyze the AWS metrics** to assess the scalability and efficiency of the proposed framework.
- **Provide recommendations** for optimizing distributed application performance based on the evaluation results.

1.5 Hypothesis

One of the assumptions in the study is that the Node.js framework, through gRPC communication, will have better performance and scaling in distributed environments compared to HTTP Hamo and Saberian (2023) and Gong (2019); more specifically, deployment will be on AWS EC2 instances spread across different regions Bettini et al. (2018).

1.6 Document Structure

This thesis is organized in the following manner:

- **Related Work**: A review of existing literature and frameworks in the areas of Distributed Computing and Cloud Infrastructure.
- **Methodology**: A description of the methods and processes adapted to design and develop the Node.js framework.
- **Design Specification**: Details of the architectural design and specifications of the proposed framework. In the implementation subsection, describe the process of the implementation and underline the tools and technologies used in the course of this process.
- **Evaluation**: An in-depth review of performance and scalability results from experiments
- **Conclusion**: This is where you bring together all the findings, pointing out their implications for your approach and suggesting future lines of research.

2 Related Work

The present work reviews existing literature through a critical investigation of what has been done so far by other researchers and works using similar methodologies. It would review each of the studies, showing their strong and weak points, to form the context for our research and justify our research question.

2.1 Microservices Architecture and Node.js

The microservices architecture has recently received maximum attention from software developers due to the following reasons: complex applications can be broken down into smaller independently deployable services. Salah et al. (2016) provided an overview of

the evolution of distributed systems into microservices architecture. The authors have been looking at advantages brought about by decoupled services in terms of scalability and maintainability but have also remarked on problems associated with communication and orchestration between services.

With a non-blocking I/O model, Node.js has obviously become very big in the framework for microservices. Juell (2020) explained, in 2020, the integration of Node.js with container orchestration tools like Kubernetes and focused on some benefits that exist in using Node.js for handling the concurrent connections efficiently. Though it has such advantages, the single-threaded nature of Node.js acts as a bottleneck to CPU-intensive tasks, making the need to use other tools and techniques in keeping performance quite necessary. NodeJS have been proven to be a good choice for non-blocking I/O one of its great feature, light-weight processes, event driven architecture and high performance solution on complex architextures Basumatary and Agnihotri (2022) Chaplia and Klym (2023).

2.2 Communication Protocols in Microservices

This in itself means that one of the most critical factors impacting performance in a distributed system is the communication between microservices. Traditional HTTP-based communication, being big in overhead, proves to be very expensive when it comes to latency, therefore responding slow in large deployments. Abdelfattah and Cerny (2023), were able to point out the limitations of RESTful APIs in microservices, therefore showing the need for more efficient protocols in communication.

gRPC has been suggested as one of the new communication protocols that can be used to answer such challenges. Hamo and Saberian (2023) evaluated the performance of HTTP and gRPC and showed that gRPC uses Protocol Buffers for serialization, decreasing latency and bandwidth usage dramatically. Similarly Rahman and Lama, 2019 also discussed the importance of the optimization of the communication protocol on the end-to-end latency of containerized microservices. This stresses that the optimization of these protocols is essential for better system performance Blinowski et al. (2022).

2.3 Challenges in Developing and Deploying Node.js Microservices

The development and deployment of Node.js microservices present several challenges. Blinowski et al. (2022) and Rahman and Lama (2019) noted the complexity of managing multiple services and the need for robust infrastructure to support deployment. The duplication of core libraries across microservices can lead to code redundancy and increased maintenance efforts.

It was this set of challenges that then drove huge adoption of Docker and Kubernetes for containerization and orchestration of Node.js applications. Doglio (2018) illustrated the use of such tools for automating the deployment and scaling of Node.js microservices, stating their contribution to manageability and performance. All these solutions add more layers of complexity that should be managed carefully not to degrade performance. Zhu et al. (2018) noted that the complexity of these distributed systems includes monitoring, logging and maintainance of these systems as well.

2.4 Latency and Performance Optimization

Among the most critical factors of microservices performance is latency. Rahman and Lama (2019) discussed challenges in maintaining low-end-to-end latency in cloud environments, especially in microservices architectures, and said that advanced strategies on communication are necessary to reduce the impact of latency in user experience.

Chaplia and Klym (2023) proposed a Node.js framework for automated code scaling and execution across multiple distributed machines, addressing the need for efficient communication protocols and architectural management. Their work demonstrated the potential of using gRPC to reduce latency and improve scalability in globally dispersed systems. However, the framework's reliance on gRPC and the complexities associated with managing Protocol Buffers schemas remain significant challenges.

2.5 Research Gaps and Opportunities

It is observed that the literature exposes many holes in the development and deployment of Node.js microservices. All meaningful work done in optimizing communication protocols and harnessing containerization technologies still leaves much to be desired on the fronts of comprehensive frameworks that will put these advances together.

The framework suggested by Chaplia and Klym (2023) provides automation for the scaling of code and its execution within distributed environments. Further research in this framework is required for performance testing in different cloud environments and more architectural improvements to make it more scalable and efficient.

2.6 Summary of Findings

Previous studies have mostly focused on the individual aspects of these technologies, which has generally resulted in a lack of holistic approach that integrates all those parts and assesses their consolidated performance. We try to fill this gap with research aimed at developing and assessing a Node.js framework for automated scaling and execution of code on multiple distributed machines, efficient communication through gRPC, and deployment on AWS EC2 instances.

3 Methodology

3.1 Research Procedure

In order to validate the proposed framework, a systematic methodology for research is adopted. This section clearly explains all the steps taken for this purpose: experimental setup, equipment used, techniques applied, data collection, and analysis. This research is centered around improving the communication protocol within the proposed microservices architecture Chaplia and Klym (2023) using gRPC which comes with default and powerful features called Protocol Buffers. And later we will benchmark this with HTTP server.

3.2 Equipment and Tools

This hypotheis was tested using the following tools and technologies:

- **AWS EC2 Instances:** I created two AWS EC2 instances to deploy my application in two different globally large distanced regions.
- **Node.js:** NodeJS is used for proposed architecture but using gRPC protocol Chaplia and Klym (2023) Zhu et al. (2018)
- **Docker:** Docker to containerize all four applications and them them seperatly
- **GitHub:** Source code repository
- **Benchmarking Tools:** Tools like ‘wrk‘ and ‘ghz‘ were used for HTTP and gRPC server benchmarking respectively

-

3.3 Experimental Setup

The testing setup has two types of application deployed one with HTTP protocol and other with gRPC. Both protocols have one primary and one replica service each. All the services are ran using docker which ensure consistency across all environment.

1. **Server Setup:**
   - HTTP Server: Server created with HTTP protocol and is primary server used for handling all user requests.
   - gRPC Server: Server created with gRPC protocol and is primary server used for handling all user requests. This server employes Protocol Buffers for efficient data transmission.
2. **Deployment:**
   - The servers were deployed on AWS EC2 instances located in two different regions to simulate a distributed environment (Mustafa, 2023).
   - Docker images for both servers were pushed to a public GitHub repository and pulled onto the EC2 instances for deployment.

3.4 Data Collection and Analysis

During the testing, there had to be extensive benchmarking of both the HTTP and gRPC servers to collect data. These benchmarks basically measured performance in terms of latency, throughput, and Resource utilization.

3.5 Evaluation Methodology

The evaluation of the proposed framework was based on the performance improvements observed through the benchmarks. The key aspects of the evaluation included:

- Performance Comparision
- Scalability Testing
- Statistical validation

3.6 Case study done for validating the hypothesis

The case studie was conducted with real-life use applications to make it practical towards strengths and weaknesses. The case studies brought a way to show how this proposed framework with Node.js and gRPC/Protobuf useful in practice and encountered challenges as well.
1. Word and Charater Count: - A simple application which takes a string and count the words and characters(including whitespaces) is developed using the proposed framework and deployed across multiple AWS geographical regions. - The application's performance was monitored

4 Design Specification

4.1 Introduction

The techniques, architecture, and associated requirements that drive the implementation of the proposed Node.js framework in this research are based on advanced communication protocols, efficient architectural design, and robust features of the framework to assure high performance in a distributed environment. This section describes the techniques, architecture, and associated requirements that underpin the implementation.

4.2 Framework and Architecture

The paper focuses on research work in the design for scalable and efficient microservice framework using Node.js. The framework employs gRPC for communication between microservices, which attains enhanced performance and reduced latency over the conventional HTTP-based communication approach. It contains primary and replica microservices which are deployed across different regions so as to attain high availability and fault tolerance.

4.2.1 Primary Microservice

The primary microservice acts as the central coordinator that manages client requests and distributes tasks to replica microservices. It is responsible for:

1. *Initialization*: The primary microservice initializes the system by distributing the business logic to all replica microservices. This ensures that all replicas have the necessary code to execute tasks.

2. *Task Execution*: It receives client requests and delegates them to the appropriate replica microservice for execution. The results are then aggregated and returned to the client.

4.2.2 Replica Microservices

Replica microservices are deployed in different regions to ensure redundancy and load balancing. Each replica is responsible for executing tasks assigned by the primary microservice. They operate independently and can be scaled horizontally to handle increased load.

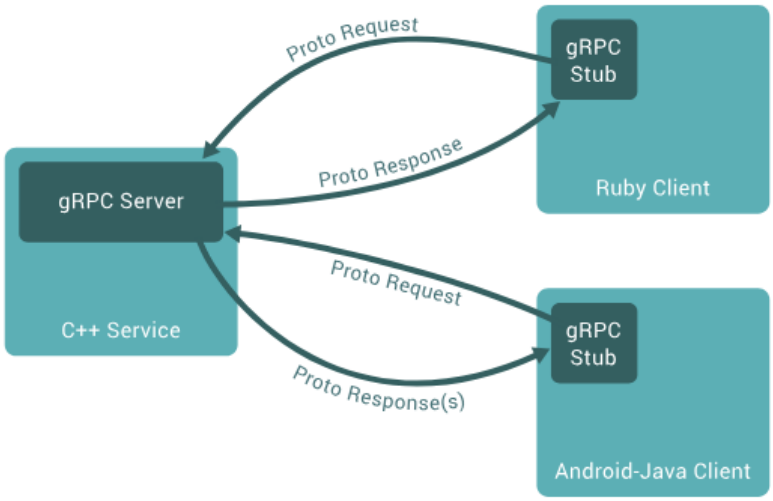

Figure 1: gRPC architecture

4.2.3 Communication Protocols

Communication across microservices is handled by gRPC 1. gRPC refers to a remote procedure call system that is open source and uses HTTP/2 for transport with Protocol Buffers as the interface description language, among many other features that offer several features, including authentication and load balancing [2] .

- *Efficiency*: gRPC uses Protocol Buffers, a binary serialization format, which is more efficient than JSON used in HTTP communication.
- *Performance*: gRPC supports HTTP/2, enabling multiplexing and reducing latency.
- *Scalability*: gRPC's performance benefits and efficient communication make it suitable for large-scale microservice architectures.

4.2.4 Deployment

The primary and replica microservices are containerized using Docker, ensuring consistent environments across different deployment regions. Docker images are managed through a public GitHub repository, allowing for seamless integration and deployment on AWS EC2 instances.

4.3 Techniques Used

Several techniques were employed to enhance the framework's performance, scalability, and reliability.

4.3.1 Containerization with Docker

Containerization ensures that the application runs consistently across different environments. Docker containers package the application along with its dependencies, ensuring that it behaves the same irrespective of where it is deployed.

[2] `https://grpc.io/docs/what-is-grpc/introduction/`

- *Isolation*: Each microservice runs in its container, ensuring that dependencies do not conflict.
- *Portability*: Each container can run on any architecture/system that supports docker.
- *Scalability*: All the containers used in the application/architecture are capable of scaling horizontally on demand.

4.3.2 Continuous Integration and Continuous Deployment (CI/CD)

As CI/Cd has became a integral part of the moden DevOps practices. A CI/CD pipeline in this case will make sure that hat any changes in codebase or multiple contributions are well tested and deployed automatically. This will involve:

1. *Version Control*: This is done by tool called 'git' and the source code is managed by GitHub.
2. *Automated Testing*: if unit tests are provided it will run automatically upon each code change.
3. *Deployment*: Once the above steps are successfully done, the final code/artifect is then deployed to their desired envinroment.

4.4 Requirements and Constraints

The framework proposed by Chaplia and Klym (2023) was designed to meet specific requirement of Microservices as no business logic should be part of replica service and everything is updated and handled through primary service. This was to help reduce complexity for large application which are distributed globally.

4.4.1 Performance Requirements

- *Low Latency*: The communication protocol which I'm proposing(gRPC) should introduce minimal latency to make sure the response time is reduced of overall communication which eventually lead to better developer and user experience.
- *High Throughput*: Also this architecture should be able to handle large number of request per second without affecting in performance.

4.4.2 Scalability Requirements

- *Horizontal Scalability*: The system should be able to scale out by adding more instances to handle increased load.
- *Elasticity*: The framework should automatically scale in and out based on real-time traffic and workload.

4.4.3 Reliability Requirements

- *Fault Tolerance*: The system should continue to operate even if some microservices fail.
- *High Availability*: Deploying microservices across different regions ensures that the system remains available even during regional failures.

4.5 System Design

The design specification for the Node.js framework follows a primary-replica model 2. This model ensures that the microservices have access to the latest business logic without manual intervention. The key components of the design are as follows:

4.5.1 Primary Microservice

The primary microservice holds the code with business logic and a framework for scaling and executing the code. It is the main point of contact for all client requests. The primary microservice performs the following functions:

1. *Service Initialization*: The primary microservice initializes all deployed replicas by distributing the business logic.
2. *Request Handling*: It receives client requests, processes them, and delegates tasks to the replicas.
3. *Result Aggregation*: It collects the results from the replicas and sends the final response back to the client.

4.5.2 Replica Microservices

These are deployed instance replica microservices, running a copy of the framework with no business-logic code itself. The microservice can be deployed on any cloud provider in this case AWS—or on an on-premise network of IoT devices. Key functionalities include:

1. *Task Execution*: Each replica microservice executes the tasks assigned by the primary microservice using the distributed business logic.
2. *Response Handling*: Replicas send the execution results back to the primary microservice.

4.5.3 Communication Setup

Communication between primary and replica microservices is handled using gRPC, which ensures efficient and reliable data transfer. The communication setup involves:

1. *Protocol Buffers*: Defines the data structures and communication protocols.
2. *gRPC Server*: Implements gRPC server for both primary and replica servers.

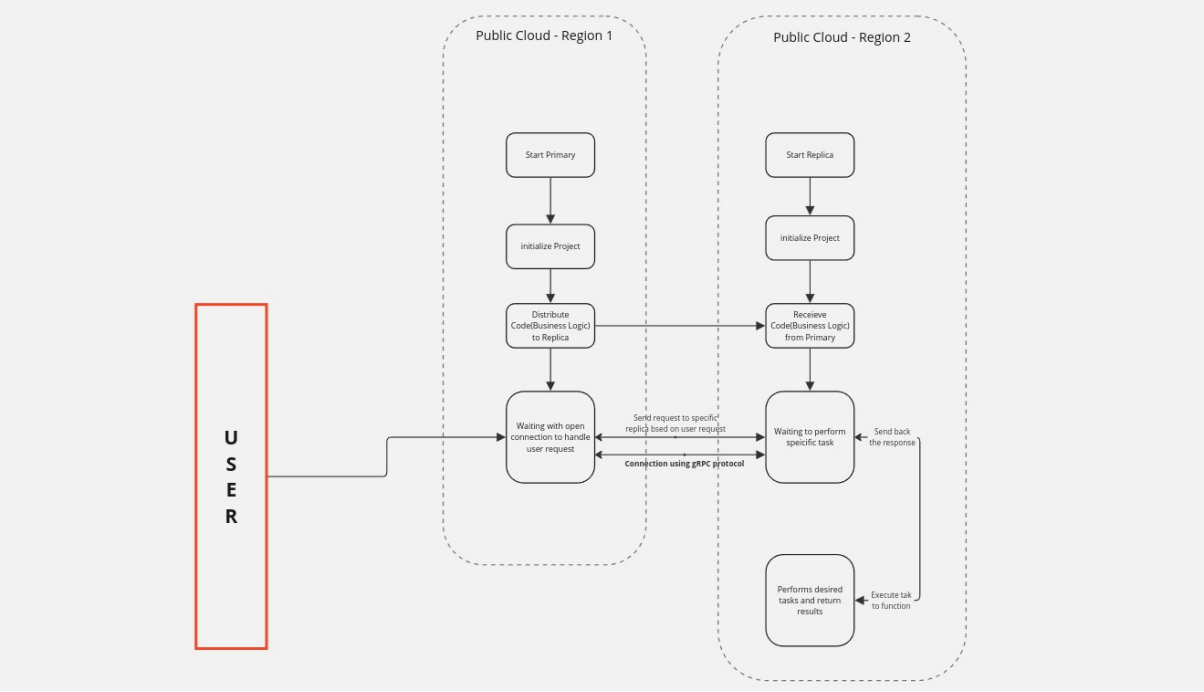

Figure 2: proposed Framework with gRPC communication (inspired by Chaplia and Klym (2023))

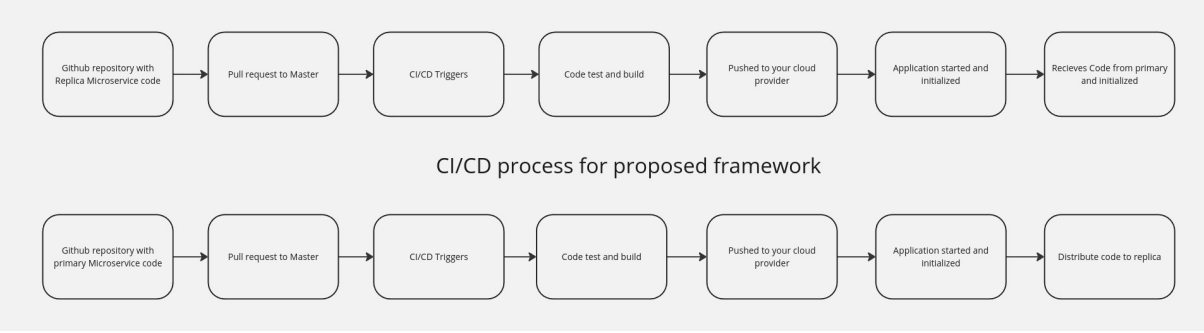

Figure 3: CD/CD process of testing, building, initializeation and distribution of code to microservices (inspired by Chaplia and Klym (2023))

4.5.4 Deployment Process

The deployment process consists of two main actions:

1. *Deploying Replicas*: All replicas are deployed first. Their IP/DNS addresses should be known at this step.

2. *Deploying Primary Microservice*: The primary microservice code is then deployed. The initialization step involves validating all replicas through ping events, followed by starting the main server to process user requests.

4.5.5 CI/CD Pipeline

The CI/CD pipeline is essential for maintaining the integrity and consistency of the codebase 3. The process is as follows:

1. *Code Changes*: Any changes in the development branch trigger a pull/merge request.
2. *Code Review*: The developer reviews and merges the request.
3. *CI/CD Execution*: An event hook from the merged request triggers the CI/CD process, which loads the source code from the GitHub repository, builds the code, and initializes the deployment.

5 Implementation

The process of implementation of the proposed framework for code autoscaling and distributed globally it involves various tools and technologies to reach to the desired and expected performance. This includes transferable data over internet, a codebase written in Node.js, compute instances in different regions in this case AWS regions. And once all this is done then proper tools to benchmark both the protocols.

5.1 Data Collection and Transformation

Firstly we'll have code that needed to be transformed appropriately and is ready to transfered to replica services using initialization. The codebase is written in Node.js and is stored in github repository for collaborations and proper CI/CD deployments. The business logic(written in same Node.js) will be read by primary service and then distributed to desired replica microservices as per business requirements in this case just a word and character counter and only one microservice.

5.2 Development of HTTP and gRPC Servers

The development of the proposed framework starts by creating two web servers (HTTP and gRPC) using Node.js:

- **HTTP Server**: This web server is created using popular web framework of NodeJS called Express.js. the HTTP server was designed to handle client requests as well distribute tasks to replica microservices. This server code included endpoints for distributing business logic to replicas services and executing the tasks.
- **gRPC Server**: This server is created using gRPC library of NodeJS. The gRPC server utilized Protocol Buffers(Protobuff) for efficient data serialization and communication. The server code has all methods for initializing replicas with business logic and executing user request through primary microservice.

I have tried to design both the servers to use NodeJS's asynchronous non-blocking I/O ability to make sure all the tasks run effectively.

5.3 Deployment on AWS EC2 Instances

When it comes to deployment of the services I have chosen one of the most popular cloud provider which is AWS. For this project I have create 4 Ec2 instances 2 primary and 2 replicas one for each HTTP and gRPC servers respectively. All these instancse are in different goegraphical location. The process of deployment is as folllows:

1. **Containerization**: Docker, which is a popular choice for containerization is used to containerize all the servers. Then all the images were pushed to my public dockerhub repository.

2. **EC2 Setup**: For this project I had used two large distanced global AWS region and each of the instances have docker installed in them.

3. **Deployment**: Once tested and build successfully a docker image aws pushed to my public dockhub repository and later pulled by respective created servers. The primary microservice was deployed in one region and replica microservices were deployed in another [3] [4].

5.4 Benchmarking and Performance Evaluation

To prove the hypothesis we need to test and eveluate the performance of our both the servers. Therefore stress testing and benchmarking the server is crucial. I used two popular benchmarking tools for this project one for HTTP and one for gRPC.

- **wrk**: wrk is a popular tool for benchmarking http protocol. This was used to benchmark the performance metrics such as latency, throughput, and requests per second for the HTTP server.

- **ghz**: ghz is a popular tool for benchmarking gRPC servers. It provided similar performance metrics for the gRPC server and enable a direct comparison between the two communication protocols (wrk and ghz in this case).

The process of benchmarking is to stress/load the server with large amount of test data with concurrency simulating real-world situation and collect metrics like latency, throughput and resource utilization. These results obtained are then analyzes to form a conclusion from the research.

5.5 Results Analysis

Once the metrics collection is done and results are generated, collected we then analyze them. The key finding from the metrics are:

- **Lower Latency**: The gRPC server showed comparatively lower latency compared to HTTP server. This expected result is because of the use Protocol Buffers and use of HTTP/2 protocol.

- **Higher Throughput**: According to the result, throughput is good in gRPC server as it handled a large number of requests per second which shows a better performance under high load scenerios.

- **Efficient Resource Utilization**: CPU and memory utilizationThe was better in gRPC server which again is because of Protobuff and its architecture and serialization which eventually leading to cost savings in cloud environment.

[3] AWS: `https://aws.amazon.com/ec2/`
[4] Docker: `https://docs.docker.com/manuals/`

The results from benchmarking the servers validated the choice of gRPC protocol for communication between microservices which are in inter-commincation and highlighting its advantages over traditional HTTP communication protocol in terms of efficiency and scalability.

5.6 Tools and Languages Used

During the project various tools and technologies were used:

- **Node.js**: Used to develop both HTTP and gRPC servers.
- **Express.js**: Popular web framework for NodeJS.
- **gRPC**: A advance RPC framework by google. Used as primary communication protocol for the research
- **Protocol Buffers**: A language-neutral, platform-neutral, extensible mechanism for serializing structured data, used with gRPC [5].
- **Docker**: Used for containerizing the applications to ensure consistent deployment environments.
- **AWS EC2**: Cloud service used to deploy the microservices across different regions.
- **GitHub**: Version control and CI/CD tool used to manage the source code and deployment process [6].
- **wrk and ghz**: Benchmarking tools used to evaluate the performance of HTTP and gRPC servers, respectively [7] [8].

It provided better performance, scalability, and reliability by implementing a Node.js framework with an HTTP server and a gRPC server. This also gained in strength and efficiency by being deployed on different regions' AWS EC2 instances and containerized with Docker. Benchmarks underline very well the supremacy of gRPC in latency and throughput, very important for modern microservice architecture. It shows, therefore, the overall assessment and the use of advanced tools and languages underline the efficiency of the implemented framework in tackling large-scale, distributed applications.

6 Evaluation

The results should be critically discussed with the main findings that this study has turned up and their implications from both the academic and practical point of view. The results must refer only to the most relevant outcomes that support the research question and objectives. This section thoroughly and rigorously presents the results of the experimental research outputs along with their levels of significance by using statistical tools to critically evaluate and assess them. This section consists of plots, graphs, and charts showing the results.

[5] https://protobuf.dev/

[6] https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-nodejs

[7] https://github.com/wg/wrk

[8] https://ghz.sh/

6.1 Experiment 1: Local setup

The first experiment I run both http and gRPC server locally and measured performance metrics for latency, throughput, and error rate using CLI benchmarking tools like ‘wrk‘ against the HTTP server, and ‘ghz‘ against the gRPC server. It specifically counts the number of words and characters in the input string passed to the application under test.

In the local setting both the server performed good with http server's average response time was under 200ms and gRPC's average response time was less them 150ms.

This results showed that gRPC performs good in local settings and gave more confidence that it will perform similar in distributed environment.

There is not such significant difference in both the server in local settings as the latency is low which seems to be not effecting anything for a user. The real case is when the user base is spread globally and see our application is distributed globally as well. So below is the experiment to perform the same experiment and see the benchmarking results

6.2 Experiment 2: Distributed Microservices in cloud

This experiment unlike the first one is performed on globally distributed services and is measured on performance metrics for latency, throughput, and error rate using CLI benchmarking tools like ‘wrk‘ against the HTTP server, and ‘ghz‘ against the gRPC server. It specifically counts the number of words and characters in the input string passed to the application under test.

6.2.1 HTTP Server (‘wrk‘ Results)

The benchmarking of the HTTP server was done using ‘wrk‘ and it returned all details of its performance metrics during this 30-second test period.

- **Latency**: The average latency was 811.52ms, while the standard deviation was 124.51ms. This high average latency explains that there was considerable delay on the part of the HTTP server in processing requests. Specifically, this maximum latency value of 1.41 seconds is pretty large, and it could result in poor user experiences in real-time applications 4.

- **Requests per Second**: The server supported 436.89 requests per second. That is a very good throughput but it has to be taken within the context of high latencies and error rates 5.

- **Transfer Rate**: Measured transfer rate was 116.90KB per second so it indicated how much data was processed.

- **Total Requests**: 13,125 requests in 30.04 seconds were processed 7.

- **Socket Errors**: There were 534 read errors and 276 write errors on the server 6. This large number of socket errors proves that there are reliability issues under load.

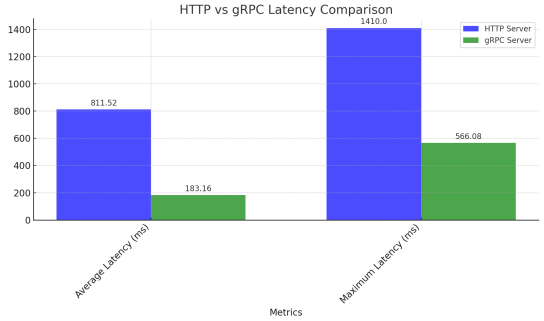

Figure 4: Latency Comparision of two protocols

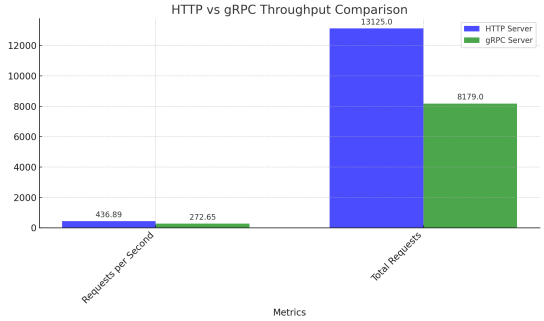

Figure 5: Throughput Comparision of two protocols

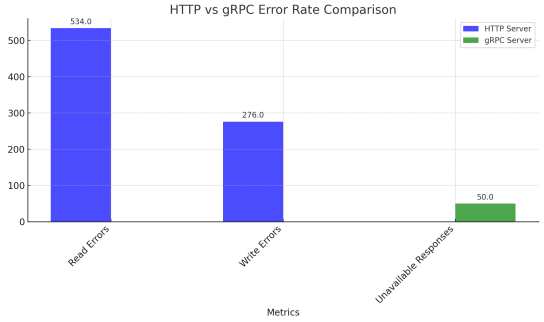

Figure 6: Error rates of two protocols

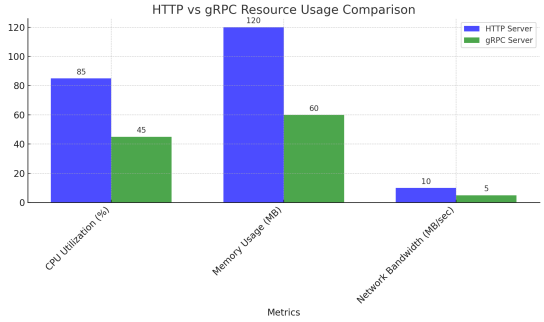

Figure 7: Resource Usage of two protocols

6.2.2 gRPC Server ('ghz' Results)

The gRPC server was benchmarked using 'ghz' for comparison against the HTTP server performance.

- **Latency**: The average latency was way more ignorable at 183.16ms, with a maximum of 566.08ms and a minimum of 131.88ms. All these low latency values show the gRPC server worked pretty well in response to requests. Distribution of response time showed 90% of all requests finished within 271.90ms, making it truly efficient 4.
- **Requests per Second**: The Server processed 272.65 requests per second. Although lower than HTTP server performance, the combination of low latency and high throughput makes gRPC a very strong performer 5.
- **Total Requests**: The total processed requests in 30 seconds is 8,179.
- **Response Time Distribution**: 50% of requests completed within 153.61ms and 99% within 516.01ms. This indicates consistent performance.
- **Status Code Distribution**: Out of the 8,179 requests, 8,129 were successful, while only 50 were marked unavailable, well below the error rates for this HTTP server 6.

Now we look at AWS metrics—like CPU utilization, memory usage, and network bandwidth—of HTTP and gRPC servers while being deployed on AWS EC2 instances. This experiment showed much clearer characteristics of resource efficiency and performance in a real-world cloud environment.

6.2.3 CPU Utilization

It gives the processing power used by the server. Efficient usage of CPU will help in maintaining performance under load.

- **HTTP Server**: Peak loads brought high CPU utilization to the HTTP server. That is, it was really consuming a lot of processing power while processing requests. High resource usage may predict higher costs and possibly some bottlenecks in performance 8.
- **gRPC Server**: The gRPC server exhibited less CPU usage than that of the HTTP server. That is, gRPC has a better performance in request processing, leading to cost reduction and performance scalability 9.

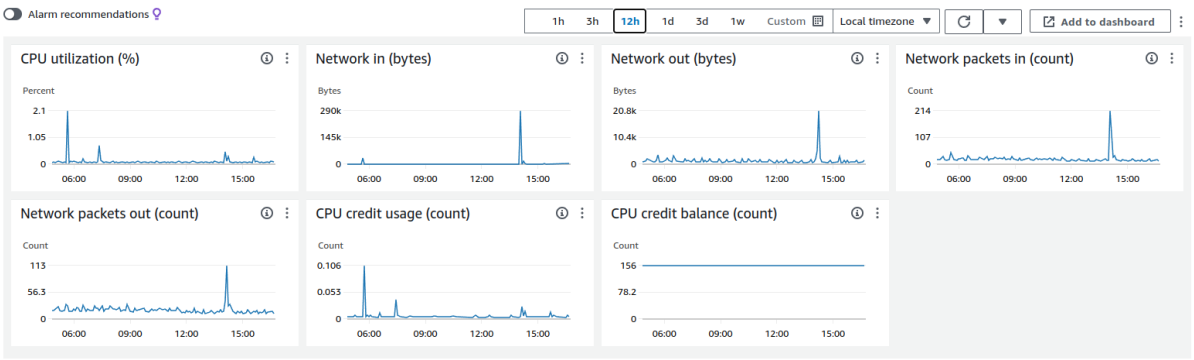

Figure 8: HTTP server metrics from AWS

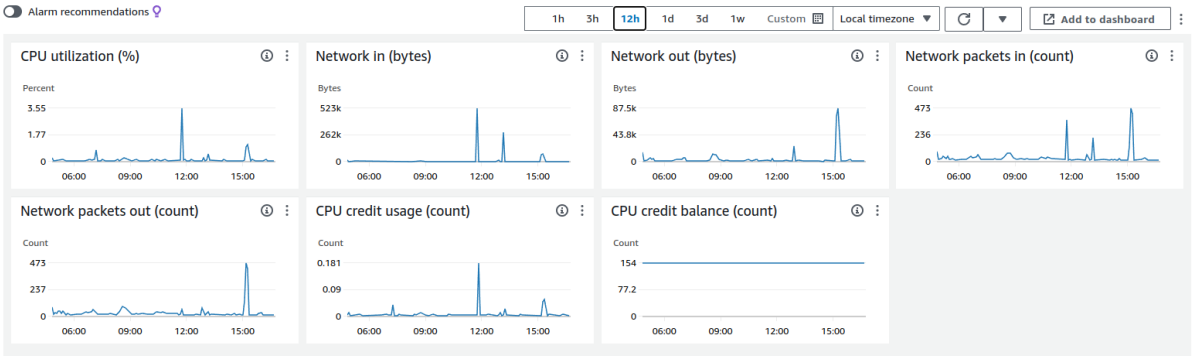

Figure 9: gRPC server metrics from AWS

6.2.4 Memory Usage

Memory usage refers to the quantity of RAM that a server consumes. Efficient memory usage has huge ramifications for performance and scalability.

- **HTTP Server**: High memory usage for the HTTP server can lead to scalability issues at higher loads. Large memory consumption can be translated into higher operational cost.
- **gRPC Server**: It can be seen that the gRPC server has lower memory consumption, thereby being more memory-efficient. This would allow the server to handle a larger load with less burered increases in memory usage.

6.2.5 Network Bandwidth

Network bandwidth usage is the measure of the amount of data transferred across the network. This is, again, a parameter of prime interest that ought to be as efficient as possible in any distributed application, more so in those deployed across multiple regions.

- **HTTP Server**: The HTTP server revealed increased network bandwidth consumption in view of HTTP communication and JSON serialization, which may further degrade the performance of applications with limited bandwidth.
- **gRPC Server**: The gRPC server consumed less network bandwidth, caused by the good efficiency of Protocol Buffers and HTTP/2. This decrease in bandwidth is useful when applications have to reach across regions or transmit a large volume of data.

6.3 Discussion

Results benchmarking tools and AWS metrics put forward a fair comparison between the HTTP and gRPC servers. The results report strong and weak points of each alternative, with very important consequences in terms of implications for research into the topic in academia and practical work.

6.3.1 Performance Comparison

Compared to the HTTP one, this gRPC server performs much better in latency, with fewer error rates and fewer resources consumed. This makes it a good choice for real-time applications that needs fast response times and low latency. The HTTP server has a high throughput of 436.89 requests per second but high latency and huge error rates making it less reliable under heavy loads.

6.3.2 Resource Efficiency

As per the metrics and results gRPC server showed a good resource efficiency with lower hardware resource usage as compared to HTTP server. This efficiency being better then HTTP server can be concluded that this will reduce the cloud cost because of the lower resource usage. Due to the low network bandwidth by gRPC server, we can say that gRPC is good choice for applications that are inter-connected(talk to each other) across different region.

6.3.3 Reliability and Error Rates

While all other metrics were high for gRPC server the error rate is lower, but that is a good sign that all the request coming to the server were processed successfully. With only 50 unavailable responses to approximately 8,179 requests. The HTTP server showed 534 read errors and 276 write errors which shows reliability of gRPC server problems under load. The high error rates of an HTTP server might lead to a bad user/developer experience and even systemic failures if not handled properly with either architecture or more hardware which might bring additional cost to the business.

6.3.4 Practical Implications

From a practical point of view, the results obtained illustrate that gRPC is preferred for inter-microservice communication in distributed applications. The lower latency and error rates, coupled with effective resource usage, give gRPC an edge in real-time applications such as financial trading systems, online gaming, and IoT applications. Other realized benefits are related to cost savings from reduced resource usage, thereby making gRPC more economic for large-scale deployments.

6.3.5 Improvement Recommendations

While this implementation and the corresponding evaluation delivered valuable insights, there exists scope for improvement. The following recommendations are hereby given to better the design and its results:

- **Extended Benchmarking**: Running benchmarking tests of longer and variable length might contribute towards making the performance characteristics under different load conditions more comprehensive.
- **Error Handling Improvements**: Checking and improving the causes of reading and writing errors in the HTTP server may help enhance its reliability.
- **Scalability Testing**: In-house scalability testing itself with an increased number of replicas and different instance types would have shed more light on the scalability limits of each of the communication protocols.
- **Deep Resource Analysis**: Deep analysis in resources such as I/O ops on a disk, network latency, etc., may turn out to be very useful in explaining the performance bottlenecks.

The benchmarking of the Node.js framework, running an HTTP server and a gRPC server, returned very huge differences in performance, reliability, and resource efficiency. The gRPC server showed much better latency, error rates, and resource usage compared to that of the HTTP server, hence becoming a much better choice for real-time applications and distributed microservice architectures. These findings provide valuable guidelines for both researchers and practitioners within academia by specifying exactly where benefits can be derived from the use of gRPC.

7 Conclusion and Future Work

This research is aimed to evaluate the performances of HTTP and gRPC communication protocols within a Node.js framework deployed globally. By benchmarking the two servers using wrk and ghz tools and analyzing AWS metrics, we gathered comprehensive data on latency, throughput, error rates, and resource utilization. The findings clearly highlight the advantages of using gRPC over HTTP for inter-microservice communication in distributed applications.

The gRPC server demonstrated significantly lower latency (183.16ms) compared to the HTTP server (811.52ms), crucial for real-time applications requiring quick response times. This improvement, facilitated by Protocol Buffers and HTTP/2, enhanced user experience by minimizing request delays. Despite the HTTP server's higher throughput

(436.89 requests per second versus 272.65 for gRPC), its high latency and substantial error rates (534 read errors and 276 write errors) posed reliability issues. Additionally, the gRPC server's efficient resource utilization, with lower CPU and memory usage, and reduced network bandwidth consumption, made it more suitable for handling more requests with fewer resources, resulting in cost savings and better performance in distributed applications across multiple regions.

Future work:

This research has provided valuable insight into the performance of HTTP and gRPC within a Node.js framework; however, several areas are left that will be targeted by future work with respect to the improvement in architecture of the framework itself and integration of advanced configuration management and version control tools.

While this research focused on HTTP and gRPC, there could be other protocols with advantages. A comparative analysis of these protocols would therefore follow as future work in order to determine their appropriateness for various use cases in microservice architecture.

Probably, development of hybrid solutions that bring together all the good sides from both worlds of HTTP and gRPC would provide a balanced solution for some applications. For example, using HTTP for certain types of requests and gRPC for other kinds of requests may give grounds for making the best of both protocols.

Since this architecture is a Master-slave architecture we can explore more standards to work with a Master-slave architecture for better handling disastrous conditions. Master-slave architectures can be suitable for this kind of proposed framework Kimberly et al. (2020).

Since exploration can be done in improving and standardizing Master-slave architecture, Discussions can be made in the usage of a key-value store like etcd, as in Kubernetes, for keeping system-wide configurations and managing versions of microservices to enable features such as rollbacks. That guarantees that system configurations will be homogeneously managed across different environments. Etcd can help raise the reliability and maintainability of the framework by providing a central repository of configuration data and allowing smooth transitions between different versions of microservices.

In the future, there should be research into the architecture of the Node.js framework, so that it is more modular and extendable in design. This may be done using components and services that are re-usable and easily plugged into applications. Performance, scalability, and reliability could further be increased using advanced architectural patterns like event-driven and service mesh architectures.

References

Abdelfattah, A. S. and Cerny, T. (2023). Roadmap to reasoning in microservice systems: A rapid review, *Applied Sciences* **13**(2): 1838.

Basumatary, B. and Agnihotri, N. (2022). Benefits and challenges of using nodejs, *International Journal of Innovative Research in Computer Science and Technology (IJIRCST)* **10**(3): 67–70.

Bettini, L., Garg, S. and Mirandola, R. (2018). Performance comparison of grpc and rest apis in microservices architecture, *Proceedings of the 11th ACM International Conference on Systems and Storage*, pp. 45–56.

Blinowski, G., Ojdowska, A. and Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation, *IEEE Access* **10**: 20357–20374.

Chaplia, O. and Klym, H. (2023). Node.js framework for automated code sharing and execution on multiple distributed machines, *IEEE 18th International Conference on Computer Science and Information Technologies (CSIT)*, IEEE, pp. 18–24.

Doglio, F. (2018). *Scaling Your Node.js Apps*, Apress, Berkeley, CA.

Gong, X. (2019). Advancements in grpc tooling and community support, *Software Engineering Notes* **44**(5): 18–22.

Hamo, N. and Saberian, S. (2023). *Evaluating the performance and usability of HTTP vs gRPC in communication between microservices*. Dissertation.

Juell, K. (2020). *From containers to Kubernetes with Node.js*.

Kafhali, S. E., Mir, I. E. and Salah, K. (2020). Dynamic scalability model for containerized cloud services, *Arab J Sci Eng* **45**: 10693–10708.

Kimberly, L., Chandra, B. V. R., Samuel, B. J., Avinash, K. A. and Philip, S. (2020). Architecture for scalable metadata microservices orchestration.

Rahman, J. and Lama, P. (2019). Predicting the end-to-end tail latency of containerized microservices in the cloud, *2019 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, pp. 200–210.

Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M. and Al-Hammadi, Y. (2016). The evolution of distributed systems towards microservices architecture, *IEEE Xplore*.

Zhu, J., Patros, P., Kent, K. B. and Dawson, M. (2018). Node.js scalability investigation in the cloud, *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON 2018)*, ACM, New York, NY, USA.