Microservices Benchmark: REST vs. gRPC

A Project Report Submitted by

Harsh Parashar, Aryan Kumar, Prateek Singhal

for the requirements for the major project of

SDE

Indian Institute of Technology Jodhpur

Computer Science and Engineering

November, 2024

Abstract

This report presents a detailed benchmarking study comparing the performance of REST and gRPC communication protocols in microservices architectures. With the increasing adoption of microservices in diverse applications, the efficiency of service communication plays a critical role in overall system performance. The study evaluates the performance of REST and gRPC across three key tasks: Simple Request-Response, Streaming Data, and Large Payload Handling, using metrics such as latency, throughput, and resource utilization i.e. Network, Memory and CPU. The benchmarking study compares these protocols across three key microservice interactions: Simple Request-Response, Streaming Data, and Large Payload Handling. Each service is implemented using both REST and gRPC, and evaluated using Docker containerization within Google Cloud Platform (GCP) to replicate real-world deployment conditions. Performance metrics such as latency, throughput, and resource utilization (CPU, memory, and network usage) are monitored with Prometheus and visualized through Grafana dashboards. The services are subjected to varying loads with Locust to simulate real-world traffic. Results derived from these experiments demonstrate that gRPC consistently outperforms REST in terms of lower latency, higher throughput, and more efficient resource utilization, particularly in high-throughput and streaming data scenarios. This report provides actionable insights based on these real-world experiments, offering developers and system architects data-driven guidance for choosing the most suitable communication protocol based on project requirements and performance constraints.

Contents

List of Figures

List of Tables

Microservices Benchmark: REST vs. gRPC

1 Introduction and background

Microservices architecture has become the preferred design pattern for building scalable, maintainable, and resilient software systems. This approach [1] breaks down complex monolithic applications into a collection of loosely coupled, independently deployable services, each responsible for a specific business function. The ability to scale and manage each microservice independently offers significant advantages, but it also presents unique challenges, especially in terms of communication between the services.

Effective communication is essential for the smooth operation of microservice-based systems. As each service operates in isolation, interactions between them occur through well-defined communication protocols. These protocols dictate how data is exchanged and influence factors such as latency, throughput, and overall system performance. Among the various communication protocols available, Representational State Transfer (REST) and gRPC (Google Remote Procedure Call) are two of the most widely adopted. REST has been the dominant protocol for web services, leveraging the simplicity of HTTP/1.1 and textual formats like JSON for data exchange. On the other hand, gRPC, developed by Google, uses HTTP/2 for more efficient multiplexing, along with protocol buffers for data serialization, offering the potential for improved performance in high-throughput and low-latency scenarios.

While REST continues to be widely used due to its simplicity and ease of integration with a variety of systems, gRPC is increasingly being chosen for performance-sensitive applications, particularly in environments requiring real-time data exchange, such as IoT, gaming, and cloud-native systems. As microservices grow in complexity and scale, selecting the right communication protocol becomes crucial for maintaining system responsiveness and optimizing resource utilization.

This report explores the performance differences between REST and gRPC through a series of experiments with real-world microservices implementations deployed in Google Cloud Platform (GCP). By focusing on key performance metrics such as latency, throughput, and resource utilization (CPU, memory, and network usage)

2 System Design and Architecture

2.1 Microservice Design

The project is structured into three primary microservices, each implemented with both REST and gRPC:

- **Simple Request-Response Service**: Facilitates lightweight interactions where small data is transferred between services.
- **Streaming Data Service**: Enables continuous data transmission over time, simulating scenarios with long-lasting connections.
- **Large Payload Service**: Handles communication involving large data transfers between microservices.

Each microservice is containerized using Docker to simulate real-world, distributed systems. The services are designed with separate endpoints and provide Prometheus metrics for detailed performance monitoring.

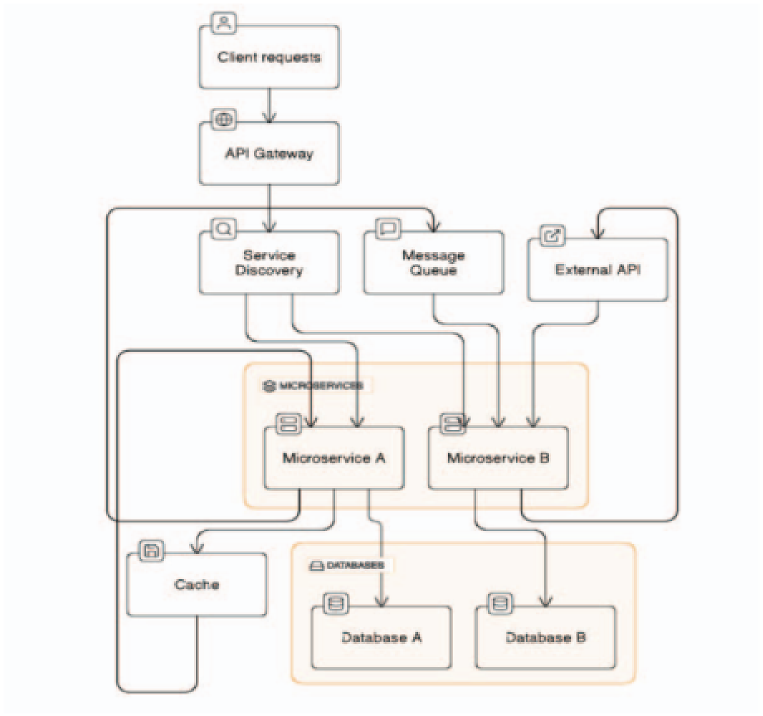

Figure 2.1: Microservice Architecture

2.2 Communication Protocols

- **REST**: Based on HTTP/1.1, REST is widely adopted due to its simplicity and compatibility with web services. It uses standard HTTP methods (GET, POST, PUT, DELETE) for communication.

- **gRPC**: A modern RPC framework developed by Google, gRPC uses HTTP/2 and protocol buffers for serialization. It supports streaming data and is designed for high-performance scenarios, especially in polyglot environments.

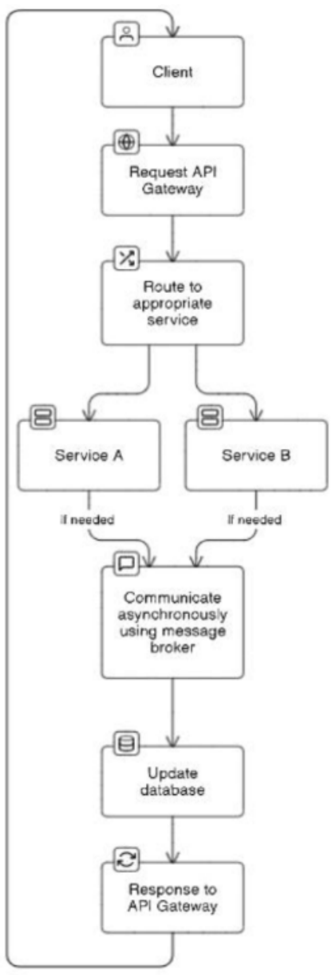

Figure 2.2: Microservice Architecture - REST

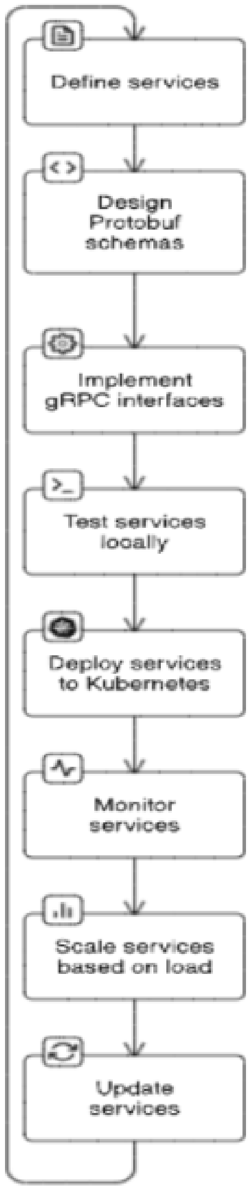

Figure 2.3: Microservice Architecture - gRPC

3 Development Stack

The project utilizes the following tools and technologies:

- **Languages**: Python (for REST) and gRPC with Protocol Buffers.

- **Containerization**: Docker and Docker Compose to replicate distributed microservices.

- **Monitoring Tools**: Prometheus for scraping metrics and Grafana for visualizing latency, throughput, CPU, memory, and network utilization.
- **Testing Tools**: Locust for load testing to simulate real-world workloads.

The deployment of the services was carried out using **Google Cloud Platform (GCP)** Virtual Machines (VMs) to replicate real-world cloud environments. The GCP VMs ensured a scalable and distributed architecture for testing the performance of REST and gRPC protocols in different microservices configurations.

4 Problem definition and Objective

Microservices architecture has become the preferred design pattern for building scalable, maintainable, and resilient software systems. However, while it offers advantages such as independent service scalability, it also introduces unique challenges, particularly in terms of communication between the services. As each service operates independently, interactions between them rely heavily on communication protocols, which play a critical role in determining system performance. Among the widely adopted protocols, REST and gRPC are the most common, but selecting the appropriate one is crucial. REST, although simple and widely integrated, may not perform optimally in high-throughput or low-latency scenarios, which gRPC is designed to address. The challenge lies in understanding how these protocols impact key performance metrics such as latency, throughput, and resource utilization, making it essential to evaluate the trade-offs between them in real-world microservices deployments.

5 Methodology

This section outlines the methodology followed for evaluating the performance of REST and gRPC protocols in microservices-based architectures. The experiments were designed to assess key performance metrics, such as latency, throughput, and resource utilization, across different types of services in a real-world cloud environment. The methodology includes the experimental setup, the performance metrics, and the testing tools used.

5.1 Experimental Setup

The experimental setup was designed to replicate a distributed microservices environment using Docker containers, with each container representing a separate service. The microservices were deployed on **Google Cloud Platform (GCP)** Virtual Machines (VMs) to ensure a realistic and scalable environment for performance testing.

CPU	2 vCPU (1 core)
Architecture	x86/64
Memory	8GB RAM
Storage	20GB Persistent
Operating System	Debian 12 (Bookworm)

We implemented three types of services:

- **Simple Request-Response Service**: Handles lightweight request-response interactions with small data transfers.
- **Streaming Data Service**: Enables continuous data transmission, simulating long-lasting connections.
- **Large Payload Service**: Handles communication involving large data transfers between services.

Each service was implemented using both **REST** (using HTTP/1.1) and **gRPC** (using HTTP/2 and Protocol Buffers), with corresponding endpoints exposed for communication.

Prometheus was used to scrape performance metrics, and Grafana visualized these metrics for analysis. Resource utilization metrics such as CPU, memory, and network usage were monitored along with key performance metrics like latency and throughput.

5.2 Latency Analysis

Latency is a crucial factor in microservices communication, as it directly affects the responsiveness of the system. To analyze latency, we measured the time taken for a request to be sent from a client service to the server service and back. We focused on the following scenarios:

- **Simple Request-Response**: This scenario evaluates the basic latency of small data transfers between services.

- **Streaming Data**: In this scenario, continuous data transmission was analyzed to evaluate the latency under sustained workloads.
- **Handling Large Payloads**: This scenario involved testing how latency behaves under the pressure of large data transfers.

We measured latency at various percentiles (e.g., 50th, 95th, and 99th) to evaluate the worst-case, average, and best-case performance under varying conditions.

5.3 Throughput Analysis

Throughput measures the number of requests processed by the system within a given time frame. This is an important metric to understand the system's ability to handle a large volume of concurrent requests.

- **Simple Request-Response**: We measured throughput under low and high load conditions to evaluate the protocols' efficiency in handling numerous simultaneous requests.
- **Streaming Data**: The throughput was evaluated for continuous data streams, assessing how both protocols handle long-lasting connections under increasing workloads.
- **Handling Large Payloads**: The system's throughput was evaluated when transmitting large data payloads between services, to gauge how well each protocol manages large volumes of data.

We recorded the number of requests processed per second, and throughput was analyzed for different service types under various load conditions.

5.4 Load Testing

To simulate real-world usage scenarios, load testing was conducted using **Locust**, a scalable load testing tool. Load tests were run to simulate varying user loads and request rates to assess how both REST and gRPC services perform under stress.

The following tests were performed:

- **REST Load Testing**: Simulated a user load of up to 100 concurrent users and request rates of 10 requests per second.
- **gRPC Load Testing**: Similarly, a user load of 100 concurrent users and request rates of 10 requests per second were simulated for the gRPC services.

These tests were designed to measure performance under high concurrency, and the results were analyzed for latency, throughput, and resource utilization.

5.5 Monitoring and Data Collection

Prometheus was used to collect key performance data such as latency, throughput, and resource utilization from the microservices. Grafana dashboards were created to visualize these metrics, enabling real-time monitoring of the system's performance during testing.

The following metrics were collected:

- **Latency**: Request-response time for both REST and gRPC services.
- **Throughput**: Number of requests processed per second.
- **Resource Utilization**: CPU, memory, and network usage.

These metrics were continuously recorded and analyzed to identify performance bottlenecks and provide insights into the optimal communication protocol based on the specific needs of the system.

6 Results Analysis

Workload	Total	Scaling (x)	Time (mins)	REST Max CPU (%)	gRPC Max CPU (%)
Light	500	5	2	56.9	26.7
Medium	1000	10	2	57.4	27.5
Heavy	5000	50	2	60.8	31.8

Table 6.1: CPU usage data for different workloads (Light, Medium, Heavy) for REST and gRPC protocols.

Test Scenario	Workload	REST Latency (ms)	gRPC Latency (ms)	REST Throughput (ms)	gRPC Throughput (ms)
Simple Request Response	Light	0.246	1.75	121	229
	Medium	0.681	1.75	116	205
	Heavy	6.19	1.75	94	175
Handling Large Payloads	Light	1.66	1.75	345	687
	Medium	3.55	1.75	343	647
	Heavy	10	1.75	309	536
Streaming Data	Light	0.103	1.75	173	341
	Medium	0.208	1.75	183	324
	Heavy	1.4	1.75	134	261

Table 6.2: Latency and Throughput for REST and gRPC under different test scenarios and workloads.

The performance data collected from the experiments was analyzed and compared to identify trends and insights. The analysis focused on the following key factors:

- Comparing latency and throughput for REST and gRPC under varying workloads (Light, Medium, and Heavy).
- Identifying which protocol offers better resource utilization in terms of CPU usage across different scenarios.
- Analyzing how each protocol performs in scenarios involving simple requests, large payloads, and streaming data.

The experiments revealed the following key observations:

- **Latency and Throughput:** In general, gRPC outperforms REST in terms of throughput but has a higher latency, particularly in heavy workloads. REST tends to show lower latency, especially in light workloads, while gRPC is better suited for handling large amounts of data, as seen in the throughput data.
- **CPU Utilization:** REST shows higher CPU utilization than gRPC across all workload intensities, with the gap widening as the workload becomes heavier. gRPC's efficient use of resources allows it to perform better under high load scenarios.
- **Memory and Network Utilization:** The memory and network usage patterns followed the same trend as CPU utilization, where gRPC exhibited more efficient use of resources in heavy loads, whereas REST showed a higher demand for resources.
- **Performance Trends:** The performance of both protocols varies depending on the workload type. While gRPC shows superior performance in high-load and continuous data stream scenarios, REST provides a more efficient solution for lightweight, less resource-intensive use cases.

These results provide valuable insights for developers and system architects in choosing the most appropriate protocol based on specific use case requirements, balancing performance, resource utilization, and scalability.

7 Service Configuration and URLs

Our services are up and running on the following configurations:

7.1 Service Machines and IPs

Service	**IP Address**
gRPC Machine	34.57.71.117
REST Machine	34.30.3.222

7.2 REST URLs

URL	**Link**
Login URL	`http://34.30.3.222:3000/login`
Targets URL	`http://34.30.3.222:9090/targets`

7.3 gRPC URLs

URL	**Link**
Login URL	`http://34.57.71.117:3000/login`
Targets URL	`http://34.57.71.117:9090/targets`

References

[1] S. A. S. K. Ritu, Ashima Kukkar, “A comparative analysis of communication efficiency: Rest vs. grpc in microservice-based ecosystems,” in *2024 International Conference on Emerging Innovations and Advanced Computing (INNOCOMP)*. IEEE, 2024, pp. 979–984.