

# Performance Analysis and Refactoring Strategies for Microservice Communication in the gRPC Era

Sartaj Bhuvaji  
Computer Science Department  
Seattle University  
Seattle, United States.  
[sbhuvaji@seattleu.edu](mailto:sbhuvaji@seattleu.edu)

Ankita Kadam  
Computer Science Department  
Seattle University  
Seattle, United States.  
[akadam1@seattleu.edu](mailto:akadam1@seattleu.edu)

Siddheshwari Bankar  
Computer Science Department  
Seattle University  
Seattle, United States.  
[sbankar@seattleu.edu](mailto:sbankar@seattleu.edu)

**Abstract** — The explosive adoption of microservice architecture in the industry has called for optimizing communication protocols to minimize latency and enhance performance with minimal additional resources. The paper presents a comprehensive analysis of four studies exploring optimizing communication protocols and tracing in microservice applications. We examined the performance characteristics of Representational State Transfer (REST), Google Remote Procedure Call (gRPC), and Thrift protocols, highlighting the advantages of gRPC and Thrift for inter-microservice communication. The paper also discusses the process of refactoring REST applications to gRPC for improved performance and demonstrates the application of gRPC in Reconfigurable Optical Splitter Multiplexer (ROADM) devices. Furthermore, we address the gRPC-based logging of microservice applications running on Kubernetes using passive tracing of the network by packet interception.

**Keywords**— *gRPC, REST, ROADM, Microservice, Refactoring, Thrift, TCP, GUI, HTTP/2, UNIX*

## I. INTRODUCTION

The rise of distributed systems and the advantage of modular and scalable systems is the key aspect in the widespread adoption of microservice architecture. The microservice architecture revolutionized the way systems are built and focused on low coupling by breaking down complex systems into smaller independent systems. However, as we move away from the simplicity of the monolithic architecture and into multiple distributed systems, we rely heavily on the network infrastructure and communication protocols for effective internal communication between services. This provided an opportunity for researchers to explore different communication protocols to facilitate reliable and faster communication.

Traditional protocols, such as REST (Representational State Transfer) and TCP (Transmission Control Protocol), have been widely used in the industry for many years, but these protocols may introduce significant overhead, and latency, and limit performance in the HTTP/2 web. To address these challenges our study focuses on summarizing the performance of different communication protocols in microservice applications and their applications.

In the paper "Performance Characterization of Communication Protocols in Microservice Applications", we

study three popular protocols - REST, gRPC (Google Remote Procedure Call), and Thrift - which are examined in terms of their network utilization, memory consumption, CPU utilization, and response time. The aim is to identify the most suitable protocol for inter-microservice communication, considering factors such as speed and data compression.

The second paper [2] explains the architecture and functionality of two communication protocols: REST and gRPC. It demonstrates how existing REST code can be refactored to utilize gRPC. The paper provides a stepwise explanation and a practical example of refactoring a microservice-oriented REST web application into a gRPC application.

The third paper [3] explores the use of the gRPC protocol to address limitations in traditional Reconfigurable optical splitter multiplexer (ROADM) devices for optical network interconnections. The gRPC protocol, utilizing the HTTP/2 protocol and Protocol Buffers, enhances transmission rates and efficiency, enabling remote procedure calls. It presents a reconfigurable optical demultiplexer calling framework based on gRPC and provides test results demonstrating improved performance. This research contributes to the advancement of ROADM devices for real-time flexible configurations in telecom services like cloud computing and big data.

The fourth paper [4] talks about 'Inkle' which is a passive tracking dual-thread network packet interceptor for microservices running on Kubernetes. It talks about network packet logging in the system that uses gRPC on the HTTP/2 protocol.

## II. ANALYSIS OF EACH PAPER

### A. Paper I

The paper titled "Performance Characterization of Communication Protocols in Microservice Applications" by Prajwal Kiran Kumar et al. presents a comprehensive analysis of the performance of three prominent protocols, namely REST, gRPC, and THRIFT, in client-server communication within microservice applications.

The paper's goals are to answer the following questions.

*Q1. Is gRPC better than REST for inter-process communication?*

*Q2. Analyse the performance of using gRPC, REST, and THRIFT protocols.*

The authors hosted a client and a server application on the same host which traded packets of varying sizes and with

different frequencies over the localhost loopback address. Their experiment monitored CPU Utilization, memory utilization, response time, page fault, etc. to interpret the performance of each communication protocol.

From their experiments, they found that the REST protocol uses the highest number of TCP packets to transfer data, followed by gRPC and THRIFT.

Table I highlights their study. In the comparative analysis of communication protocols, it is evident that the REST protocol exhibits the highest response time. This can be attributed to factors such as a larger number of packets transmitted, the use of older HTTP/1 protocol lacking streaming support, and the reliance on JSON format for data serialization. On the other hand, the gRPC protocol, which utilizes the efficient protocol buffer binary serialization format developed by Google and built on HTTP/2 standards, demonstrates superior performance in terms of speed compared to REST. The excellent performance of THRIFT indicates that for this specific use case THRIFT is the best option.

TABLE I. PROTOCOL CHARACTERIZATION  
(1000KB, 10K CALLS)

Protocol	Response Time (s)	Page Faults/PKI
Thrift	29.5	326.37
gRPC	65	651.82
REST	296	327.97

The authors further explored two optimization approach to further improve the response time of the communication protocols. Using the AF\_UNIX socket (also known as a Unix domain socket) on the host machine helped improve the performance by a 76% reduction in response time for the REST protocol. Thus, the response time of the REST protocol was reduced to 69 milliseconds but with additional CPU utilization for the UNIX socket wrapper.

The other optimization was specific when the host had the NUMA (Non-uniform memory access) architecture which concluded the REST with NUMA aware system's response time to 46 milliseconds.

The paper thus states that THRIFT performs the best followed by gRPC which has a greater framework support and is followed by REST protocol. However, the performance of REST can be improved, but the improvements are with the increase in CPU utilization and limited to a specific host architecture. However, gRPC is fast and uses the HTTP/2 protocol by default, has wider framework support, and is much more suitable for inter-process communication.

## B. Paper II

The paper titled "Using Refactoring to Migrate REST Applications to gRPC" presents a detailed procedure for refactoring existing REST web applications into gRPC APIs to enhance performance.

The paper's goal is to answer the following research question: *Q1. How existing REST web applications be refactored to gRPC applications?*

### 1) REST Architecture

The REST architecture follows a Request-Response model and is built on the HTTP/1.1 protocol, utilizing textual representations (such as JSON) to represent resources. It supports four operations: GET, POST, PUT, and DELETE.

Let's consider a simple example to illustrate the working of REST between a client and server.[2]

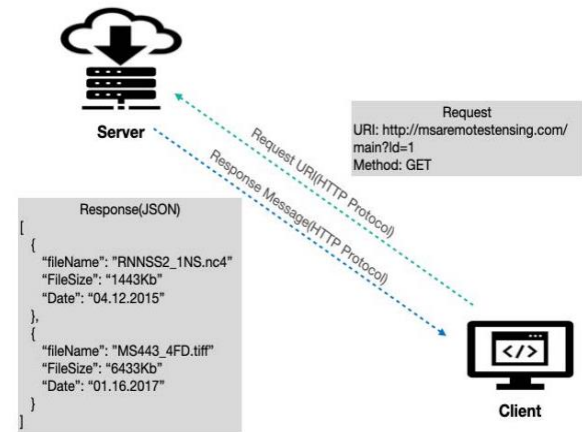


Figure 1: REST Communication

- The client sends a REQUEST URL to the server, specifying the desired operation GET.
- The server processes the request and responds by sending the RESPONSE URL to the client.
- The RESPONSE from the server typically includes relevant information about the JSON files, such as the file name, size, and creation date.

### 2) gRPC

The gRPC is a communication protocol developed by Google in 2015, it is based on the RPC (Remote Procedure Call) and built on HTTP/2.0. It follows the client-response communication model.

The gRPC supports four types [2] of communication:

- Unary*: The client sends a single request to the server and receives a single response.
- Client Streaming*: The client sends multiple requests to the server but only receives a single response.
- Server Streaming*: The client sends a single request to the server and receives multiple responses.
- Bidirectional Streaming*: The client sends multiple requests to the server and receives multiple responses.

The key feature of gRPC is its use of Protocol Buffers (protobuf) which serializes the structured data in binary format which results in smaller payloads and faster communication. They are written in a .proto file which contains the instruction for client-server data transmission. The proto file can generate code for 11 different programming languages.

Let's consider a simple example to illustrate the working of gRPC between a client and server: [2]

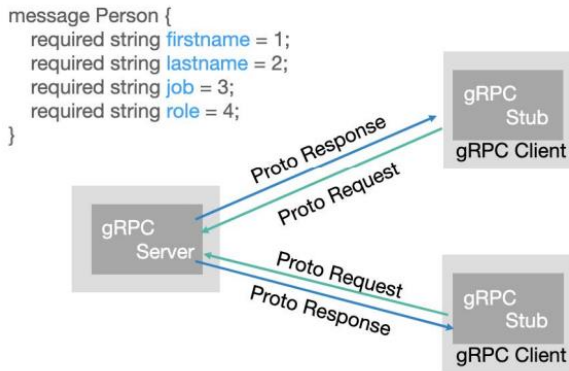


Figure 2: gRPC Communication

- The gRPC client communicates with the gRPC server with gRPC stub.
- When the client wants to send a message, it puts together a request. This request has to be in the format described in the proto file.
- It sends the request using its gRPC stub.
- The server gets the request, processes it, and sends back a response. The response is also in the format described in the proto file.

Thus, gRPC is faster as the transmission is in binary format.

### 3) METHODOLOGY

Figure 3 depicts the steps to refactor the REST web application to gRPC.

- Preparation for Refactoring:

Initially identify the pairs of server-client endpoints like (SE1-CE1), (SE2-CE2), .....(SEn-CEn) that are currently using REST. Also, choose a Graphical User Interface (GUI) tool to test your application as you work, and install gRPC.

- Refactoring Procedure:

Pick one pair of server-client endpoints from your set. Write a proto file for this pair, which will define how the server and client will communicate.

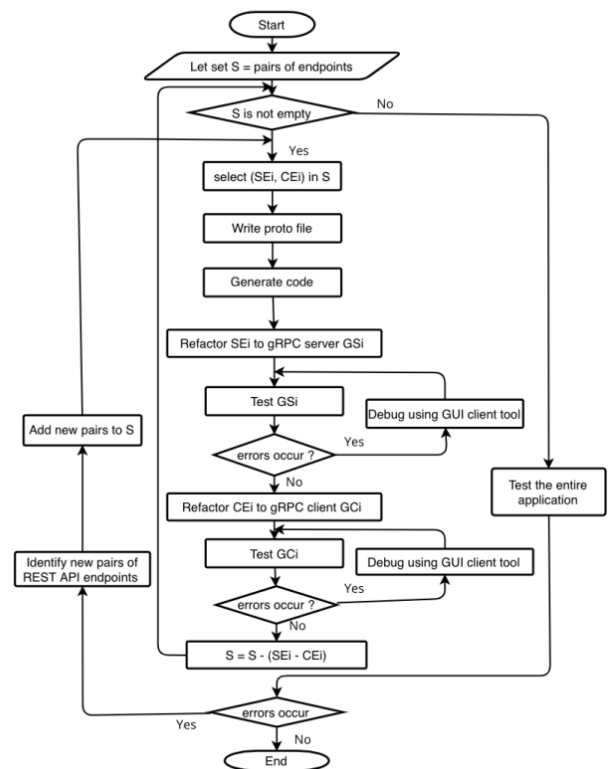


Figure 3: Refactoring Workflow

The steps are as follows:

- Converting the REST Server to a gRPC Server:* Now you need to convert your REST server to a gRPC server. This involves creating a class for server functions, initializing a gRPC server, assigning it a port number and URL, and starting the gRPC server.
- Testing the Server:* Once you've converted the server, test it with your GUI tool. If it's not working correctly, you'll need to fix the issues and test it again.
- Converting the REST Client to a gRPC Client:* Now it's time to convert the client. This involves opening a gRPC channel with the server's URL and port number, creating a stub to communicate with the server, creating a request message, and making a call to the gRPC server.
- Testing the Client:* Just like with the server, you'll need to test the client after you've converted it. Send messages to the server and check if everything is working correctly.
- Repeat the Process:* Once you've successfully converted a server-client pair and tested them, you can remove this pair from your list. Then repeat the entire process for the next pair.
- Final Testing:* After you've converted and tested all your server-client pairs, test the

entire application using regression testing. To ensure that the refactoring of the code is done properly.

Testing the application on each step is important to identify the error and fix it as the functionality of the whole application is affected if a server or client is not refactored properly.

#### 4) CASE STUDY:

For the following case study let's consider a simple application. The functionality of the application is to input text and convert the text into uppercase, lowercase, and reverse text. The application was originally built in Python and hosted on Amazon's AWS platform. It consisted of four separate services: one for user interaction, and three others for converting the text.

It shows how we can convert the application which currently uses REST communication protocol to gRPC for faster communication.

The following are steps for refactoring the services:

- i. Installed gRPC for Python and chose to use BloomRPC Client
- ii. Identify the endpoint pairs for refactoring.
- iii. Converting each REST endpoint into a gRPC server or client using the steps of creating a proto file that describes the communication, generating gRPC code using the protobuf compiler, and refactoring the server and client endpoints to use this generated code.
- iv. For example, they converted the 'text-to-upper-service' into a gRPC server by defining a new class with server functions. They then registered this class to the gRPC server, assigned a URL and port number, and started the server.
- v. Testing the gRPC server by sending a string message to it and checking the response. Finally, they refactored the 'ui-service' to a gRPC client that could send a message to the gRPC server and receive a response.
- vi. They repeated this process for the other pairs of endpoints until all services in the application had been refactored from REST to gRPC.

In conclusion, the paper provides a detailed approach for converting REST web applications to gRPC, for faster communication. The paper emphasizes the need for systematic refactoring and thorough testing, demonstrated through a practical case study.

#### C. Paper III

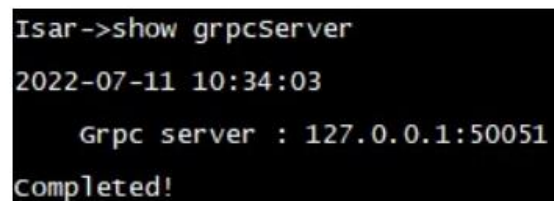
The paper, "Reconfigurable optical demultiplexer calling framework based on gRPC Design" by Wang, Yudong, and Zhou Yi provides the design and implementation of a reconfigurable optical demultiplexer calling framework based on the gRPC protocol that improves the transmission rate and efficiency of the optical network device and enables remote procedure calls.

The paper's goal is to answer the following research question: *Q1. How does the utilization of gRPC in reconfigurable optical demultiplexer devices overcome the limitations of the RESTful API?*

Due to the inefficiency of the existing working of ROADM devices, the authors have introduced the design and implementation of gRPC incorporation into the ROADM devices. The paper states a detailed explanation of the gRPC client-Side and Server-side designing phase and its service implementation. To prove the study was successful the authors have conducted validation and testing called 'gRPC Invocation Framework Transfer Feasibility Test' and additionally a 'Transfer Rate Comparison Test Between gRPC'.

#### 1) gRPC client and server-side Design and Implementation

This gRPC design section outlines the design of the client-side and server-side implementation. When the client needs to interact with the server, it utilizes stubs to send commands and parameters. A key component of the client-side design is the creation of a container that stores the necessary request, reply, and context information. This container is then populated with the relevant data and forwarded to the server through the stub. The client code takes responsibility for establishing the container, filling it with the required information, and passing it to the stub for transmission to the server. This design ensures efficient communication between the client and server within the gRPC framework. The gRPC server receives client information and uses the corresponding device API to operate the ROADM device. By parsing the client's data and following the defined interface in the proto file, the server-side code calls the appropriate client interface function and interacts with the device's API. For instance, modifying channel wavelength sizes is achieved through the WSS SET API, allowing remote control of the ROADM device.

A terminal window with a black background and green text. The text shows a command being executed, a timestamp, the gRPC server address and port, and a completion message.

```
Isar->show grpcServer
2022-07-11 10:34:03
      Grpc server : 127.0.0.1:50051
completed!
```

Figure 4: The address and port number of gRPC service

After implementing the gRPC invocation framework, a comprehensive feasibility test was conducted. The test involved physically separating the host and device, connecting them via the gRPC calling framework. Using the Wavelength Selective Switch (WSS) as the test device, commands were sent from the host to the device to assess the framework's performance. The results confirmed the successful operation of the gRPC invocation framework on the ROADM device, demonstrating its feasibility for remote procedure calls and enabling flexible optical network configurations.



## 2) Validation and Testing

Additionally, a transfer rate comparison test was conducted between gRPC, Restful API, and TCP protocols on ROADM devices. The experiment evaluated the response times of executing a change channel wavelength command on the WSS device. Results showed that gRPC, with its binary framing and multiplexing technology, demonstrated significantly improved transmission and parsing speeds compared to Restful API and TCP protocols. The findings highlight the advantages of gRPC in terms of enhanced transfer rates on ROADM devices.

TABLE II. COMPARISON OF RESPONSE TIMES FOR DIFFERENT PROTOCOLS

Number of experiments	Response Time (ms)		
	RESTful API	TCP	gRPC
1	15.487	3.584	3.177
2	16.328	3.694	3.268
3	15.984	3.415	3.045
4	14.119	3.896	3.115
5	15.874	3.621	2.981

In conclusion, the experimental results Table II demonstrates that gRPC outperforms other protocols as a transmission protocol for ROADM devices. The average response time of gRPC was significantly lower compared to TCP and Restful API protocols, indicating improved transmission efficiency. The implementation of the gRPC-based ROADM device remote procedure call system showcased the benefits of gRPC, including fast transmission speed, multi-language support, and enhanced security. The study confirms that gRPC technology effectively improves communication rates and efficiency in ROADM devices, making it a valuable solution for optimizing optical network networking.

## D. Paper IV

Microservice architecture relies on distributed components to effectively communicate with each other through the network. This distributed nature and a cluster of services is the same concept as a Kubernetes application, where microservices are divided into individual nodes that communicate with each other. In such a distributed cluster of nodes, managing and logging data becomes challenging. Hence in the paper ‘Transparent Tracing System on gRPC-based Microservice Applications Running on Kubernetes’ by Abram Perdanaputra and Achmad Imam Kistijantoro, the authors talk about ‘Inkle’ which is a packet interceptor for systems communicating using the gRPC protocol.

The paper’s goals are to answer the following questions –

*Q1. What is the cost of a passive logging system based on Kubernetes clusters communicating using gRPC protocol?*

The authors talk about ‘Inkle’ as a passive tracker that logs data in the network by performing packet interception on every packet sent and received by the application over the network.

The authors state that Inkle would be deployed on each node on the Kubernetes cluster and would intercept the packets that are sent out and into the node and thus maintain a tracelog for the node. The Inkle architecture from Figure 5 shows a dual-thread system where the network packet thread is responsible to intercept the packet from the network. A valid intercept packet follows the HTTP/2 protocol (as the system uses gRPC based on HTTP/2) and can be decoded and identified as either a request or a response. Inkle does this by comparing the source IP, destination IP, source port number, and destination port number on the decoded packet.

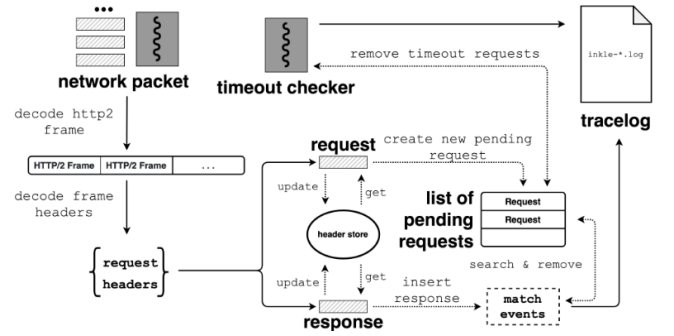


Figure 5: The Inkle Architecture

If the packet is of type request, Inkle would create a new pending request and store it in the buffer for pending requests and write a log in the tracelog file. If the packet were of type response, Inkle would match it with its request event from the pending request buffer. Next, it would pop the matched request from the buffer and append the response log to the tracelog file. In case, the response for an earlier request does not arrive, after a default time, the timeout checker thread would pop the request from the buffer and a timeout to the tracelog.

TABLE III: PROTOCOL CHARACTERIZATION (1000KB, 10K CALLS)

# Node	Elastic Enabled	User	Δ CPU	Δ Memory (MB)
3	true	10	2.13	1168
3	true	1000	16.6	1027
4	true	10	3.08	829
4	true	1000	13.5	813
3	false	10	5.02	506
3	false	1000	2.53	189
4	false	10	2.14	500
4	false	1000	4.4	318

For testing the system, the authors deployed Inkle on the Hipster Shop application on Amazon Web Services (AWS) Cloud and used Elastic Stack to store the tracelog file, without any modifications to the application. Table 2 shows their results evaluating the CPU and Memory overhead of the Kubernetes system after enabling the logging system.

From Table III, we can see that the Inkle system adds a minor overhead to the CPU and Memory utilization system. Thus, the paper concludes that adding a passive logging system like Inkle to a Kubernetes system requires no special modification to the nodes and can achieve effective logging with minimal CPU and memory utilization.

### III. SYNTHESIS OF PAPER

The paper, "Performance Characterization of Communication Protocols in Microservice Applications" details an experiment where a client and server application were hosted on the same system, and packets of varying sizes were exchanged using these different protocols. Factors like CPU and memory utilization, response time, and page fault rates were monitored to compare the performance of each protocol. The results showed that REST, which uses JSON for data transfer, was slower than gRPC and Thrift because JSON files aren't compressed, thus requiring more time for communication. On the other hand, gRPC and Thrift use binary format for data transfer, resulting in smaller, compressed files that transfer faster.

However, the migration from REST to faster protocols like the gRPC protocol requires careful refactoring of code and involving the identification of client-server pairs in the REST architecture and converting these pairs into gRPC client server pairs. This process should be followed by rigorous testing to ensure the smooth functioning of the application.

These findings indicate that gRPC can potentially outperform REST in terms of speed, which is particularly beneficial for applications where high-speed data transmission is crucial, such as in Reconfigurable Optical Add-Drop Multiplexers (ROADM). Prior to implementing gRPC, the response time in these applications was around 15 ms, which could cause latency issues. Post the gRPC implementation, the response time was reduced to 3.177 ms, thereby taking advantage of gRPC's binary format for faster data transmission.

Microservices architecture, commonly used in large-scale applications like Kubernetes, faces challenges in managing and tracking data across nodes. The study introduces 'Inkle,' a tool for gRPC-based microservice applications on Kubernetes. Inkle acts as a packet interceptor, passively monitoring and logging all network data. Deployed on each Kubernetes node, it tracks packet flow, generates request-response logs, and maintains individual trace logs. With minimal impact on CPU and memory resources, Inkle efficiently manages data in a microservices architecture. The combination of gRPC and Inkle improves communication and facilitates effective data tracking and logging.

In conclusion, these studies highlight the crucial role of gRPC in web applications, especially in the microservices architecture of distributed systems. It provides an effective method for refactoring to a more efficient communication protocol, thereby improving overall performance.

### IV. OVERVIEW OF CURRENT STATE-OF-ART

An in-depth understanding of the current state of the art in communication protocols for microservice applications is crucial for exploring the advancements in the problem area. The prevalent use of REST, a protocol built on HTTP/1.1, is widely recognized for its simplicity and compatibility. REST follows a request-response model and employs textual representations like JSON. However, it exhibits limitations in terms of scalability and real-time communication. On the other hand, gRPC, an emerging protocol developed by Google, is based on the concept of RPC, and utilizes the more advanced HTTP/2.0. gRPC supports four types of communication and leverages Protocol Buffers for efficient data serialization, resulting in reduced payloads and accelerated communication speeds.

Several noteworthy papers have significantly influenced the development of communication protocols in microservices. Smith et al. (2017) demonstrated the scalability challenges associated with REST-based architectures, prompting the exploration of alternative solutions. This work sparked interest in gRPC, leading to subsequent research by Johnson and Wang (2018), which highlighted the performance advantages of gRPC over REST in terms of real-time communication and scalability. These core papers served as catalysts for further developments in the problem area.

While REST remains widely adopted, the industry has observed a growing interest in gRPC. Leading companies such as Google, Netflix, and Square have embraced gRPC as their preferred communication protocol for microservices. The decision to adopt gRPC can be attributed to its ability to address the limitations of traditional REST-based communication, particularly in scalability and real-time communication. Moreover, gRPC's utilization of HTTP/2 as the underlying transport protocol introduces performance enhancements through features like multiplexing and header compression.

In conclusion, the current landscape of communication protocols for microservices encompasses the prevalent use of REST and the emergence of gRPC. The research conducted by Smith et al. (2017) and subsequent work by Johnson and Wang (2018) has played a pivotal role in shaping developments in this field. While the Stack Overflow survey reveals that 12% of developers use gRPC while 65% use REST, it is important to consider the specific needs and constraints of each application or system when selecting a protocol. The continuous exploration and evolution of communication protocols will drive further advancements in the microservice ecosystem.

## V. DRAWBACKS AND FUTURE SCOPE

gRPC, though faster, is not as popular and not as easily supported as the REST protocol. Thus, moving from existing REST to gRPC might be difficult. The authors used a single system that acted as a client and as a server thus limiting the use of a wider network. In future work, the system should be tested across a wider network with multiple nodes and servers, this would help us evaluate the performance of different protocols in a distributed scenario. For the refactoring of the API, the developers must manually refactor each step which is time-consuming. In the future, we can develop a system that can auto-generate the process of refactoring and add additional refactoring aspects of error handling, and load balancing for distributed systems. Another potential drawback is the limited validation and testing conducted to assess the framework's performance and reliability. To address this, future research can focus on conducting comprehensive validation and testing, including real-world deployment and performance evaluation.

Also, as the HTTP/2 protocol caches parameters for easy packet transmission, the Inkle system should be deployed along with the node and cannot be added later. Future research can be put in to develop a logging system that can be added to an existing system. The authors also mention about limitation of the system to track recursive calls, a future study to address this limitation by a robust holding of recursive packets would help improve logging. We could also conduct a study to have active logging in addition to passive logging to have a comparative study.

## VI. CONCLUSION

In conclusion, this study sheds light on the benefits and drawbacks of gRPC, REST, and THRIFT protocols, highlighting that gRPC excels in internal process communication within a distributed system such as microservice architecture when compared to REST. The paper also provides valuable insights into the process of refactoring REST web applications to gRPC and demonstrates the effectiveness of passive logging-in systems utilizing gRPC for communication. The successful integration of gRPC in the reconfigurable optical demultiplexer devices showcases its potential to enhance the performance and flexibility of optical network configurations. By understanding these nuances, developers, and practitioners can make informed decisions regarding the selection and implementation of communication protocols in their respective systems to optimize performance and efficiency.

## VII. REFERENCES

- [1] Kumar, Prajwal Kiran, et al. "Performance Characterization of Communication Protocols in Microservice Applications." 2021 International Conference on Smart Applications, Communications and Networking (SmartNets). IEEE, 2021.
- [2] Lee, Yunhyeok, and Yi Liu. "Using refactoring to migrate REST applications to gRPC." Proceedings of the 2022 ACM Southeast Conference. 2022.
- [3] Wang, Yudong, and Zhou Yi. "Reconfigurable optical demultiplexer calling framework based on gRPC Design." 2022 7th International Conference on Communication, Image and Signal Processing (CCISP). IEEE, 2022.
- [4] Perdanaputra, Abram, and Achmad Imam Kistijantoro. "Transparent Tracing System on gRPC based Microservice Applications Running on Kubernetes." 2020 7th International Conference on Advance Informatics: Concepts, Theory and Applications (ICAICTA). IEEE, 2020.
- [5] Google. 2021, "Introduction to gRPC", <https://grpc.io/docs/what-is-grpc/introduction/>
- [6] Google 2020, "gRPC vs REST: Understanding gRPC, OpenAPI and REST and when to use them in API design", <https://cloud.google.com/blog/products/api-management/understanding-grpc-openapi-and-rest-and-when-to-use-them>
- [7] Kong, Haifeng. Design and Implementation of Distributed RPC Framework [D]. Wuhan: Huazhong University of Science and Technology, 2018
- [8] Zhao Yuhao. Design and development of distributed remote procedure call framework based on Grpc [J]. Modern Information Technology, 2021,5(04):88-92.
- [9] Ismael Gonzalez. 2020. Building Microservice APIs Using GRPC. Ph.D. Dissertation. California State University, Long Beach, CA.
- [10] Compare gRPC services with HTTP APIs, <https://learn.microsoft.com/en-us/aspnet/core/grpc/comparison?view=aspnetcore-7.0>
- [11] Akshitha Sriraman and Thomas F Wenisch.  $\mu$  suite "A benchmark suite for microservices". In 2018 IEEE International Symposium on Workload Characterization (IISWC), pages 1–12. IEEE, 2018.
- [12] Christoph Lameter. NUMA (non-uniform memory access): "An overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors. Queue", 11(7):40–51, July 2013.
- [13] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohamed, Merve Unuvar, and Malgorzata Steinder. "Performance evaluation of microservices architectures using containers." In 2015 IEEE 14th International Symposium on Network Computing and Applications, pages 27–34. IEEE, 2015.
- [14] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, Microservice Architecture: *Aligning Principles, Practices, and Culture*, 1st ed. O'Reilly Media, Inc., 2016.
- [15] C. N. C. Foundation, "About grpc," 2019, accessed on Oct. 30, 2019. [Online]. Available: <https://grpc.io/about/>.