DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2020

Research on Interprocess Communication in Microservices Architecture

Master thesis performed in collaboration with IBM

BENYAMIN SHAFABAKHSH

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Research on Interprocess Communication in Microservices Architecture

BENYAMIN SHAFABAKHSH

Master in ICT and Innovation - Cloud Computing and Services
Date: June 9, 2020
Supervisor: Professor Robert Lagerström
Examiner: Professor Mihhail Matskin
School of Electrical Engineering and Computer Science
Host company: IBM Sweden
Swedish title: Forskning om kommunikation mellan processer i mikroservicearkitektur

Abstract

With the substantial growth of cloud computing over the past decade, microservices has gained significant popularity in the industry as a new architectural pattern. It promises a cloud-native architecture that breaks large applications into a collection of small, independent, and distributed packages. Since microservices-based applications are distributed, one of the key challenges when designing an application is the choice of mechanism by which services communicate with each other. There are several approaches for implementing Interprocess communication(IPC) in microservices, and each comes with different advantages and trade-offs. While theoretical and informal comparison exists between them, this thesis has taken an experimental approach to compare and contrast common forms of IPC communications. In this thesis, IPC methods have been categorized into Synchronous and Asynchronous categories. The Synchronous type consists of REST API and Google gRPC, while the Asynchronous type is using a message broker known as RabbitMQ. Further, a collection of microservices for an e-commerce scenario has been designed and developed using all the three IPC methods. A load test has been executed against each model to obtain quantitative data related to Performance Efficiency, and Availability of every method. Developing the same set of functionalities using different IPC methods has offered a qualitative data related to Scalability, and Complexity of each IPC model. The evaluation of the experiment indicates that, although there is no universal IPC solution that can be applied in all cases, Asynchronous IPC patterns shall be the preferred option when designing the system. Nevertheless, the findings of this work also suggest there exist scenarios where Synchronous patterns can be more suitable.

Keywords: Microservices, Interprocess Communication, Inter-Service Communication, Software Architecture, Cloud Computing, Distributed Systems, gRPC, RabbitMQ.

Sammanfattning

Med den kraftiga tillväxten av molntjänster under det senaste decenniet har mikrotjänster fått en betydande popularitet i branschen som ett nytt arkitektoniskt mönster. Det erbjuder en moln-baserad arkitektur som delar stora applikationer i en samling små, oberoende och distribuerade paket. Eftersom mikroservicebaserade applikationer distribueras och körs på olika maskiner, är en av de viktigaste utmaningarna när man utformar en applikation valet av mekanism med vilken tjänster kommunicerar med varandra. Det finns flera metoder för att implementera Interprocess-kommunikation (IPC) i mikrotjänster och var och en har olika fördelar och nackdelar. Medan det finns teoretisk och informell jämförelse mellan dem, har denna avhandling tagit ett experimentellt synsätt för att jämföra och kontrastera vanliga former av IPC-kommunikation. I denna avhandling har IPC-metoder kategoriserats i synkrona och asynkrona kategorier. Den synkrona typen består av REST API och Google gRPC, medan asynkron typ använder en meddelandemäklare känd som RabbitMQ. Dessutom har en samling mikroservice för ett e-handelsscenario utformats och utvecklats med alla de tre olika IPC-metoderna. Ett lasttest har utförts mot varje modell för att erhålla kvantitativa data relaterade till prestandaeffektivitet, och tillgänglighet för varje metod. Att utveckla samma uppsättning funktionaliteter med olika IPC-metoder har erbjudit en kvalitativ data relaterad till skalbarhet och komplexitet för varje IPC-modell. Utvärderingen av experimentet indikerar att även om det inte finns någon universell IPC-lösning som kan tillämpas i alla fall, ska asynkrona IPC-mönster vara det föredragna alternativet vid utformningen av systemet. Ändå tyder resultaten från detta arbete också på att det finns scenarier där synkrona mönster är mer lämpliga.

Keywords: Microservices, Interprocess Communication, Inter-Service Communication, Software Architecture, Cloud Computing, Distributed Systems, gRPC, RabbitMQ.

Acknowledgements

Getting back into academia after a few years has been challenging, and I consider the decision to be one of the significant milestones of my life so far.
First of all, I want to thank my industry supervisor Magnus Johansson for giving me the invaluable opportunity to work on this topic. I am proud of working with you and I am grateful for your support.
A big thanks to my academic supervisor, Professor Robert Lagerström, for his support on my work. The feedback that I have received from you has improved my thesis significantly.
I thank my examiner, Professor Mihhail Matskin, for accepting my thesis title. I also want to thank him for all the great work he does to support the Cloud Computing students.
Amuz Tamrakar, our casual conversation related to microservices architecture over lunch breaks in Berlin has profoundly influenced me to develop interest in this domain, and I thank you for that.
Chris Richardson, the author of "Microservices Pattern" book and founder of microservices.io website whose work has been a great help for the community and an extraordinary resource for me to get this thesis done.
Last but not least, I want to thank myself. I want to thank myself, for believing in myself and working hard.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud Computing[1] is becoming more popular than ever and more businesses are considering to migrate and run their infrastructure in the cloud. To fully leverage the potential that cloud can offer, companies require a shift in the way they design and architect their software applications. By merely taking the lift-and-shift approach[2] which is to migrate the application that is currently running on-premises to a virtual machine in the cloud, there are very minor benefits that can be gained. This approach could even result in higher operation costs in the long term. It is important for the new application to be design with cloud-first strategy[1] in mind, as well as re-architecting legacy applications prior to moving to the cloud. One of the major changes in software architecture is to decompose a large monolithic application to a set of smaller, fine-grained, and independent services that works together in a distributed network[3].
Over the past few years, microservices architecture has earned enormous attention and gained popularity from the industry. Microservices architecture has helped large organization such as Amazon and Netflix to serve millions of request per minutes.[2] While it is difficult to formally define microservices architecture, there are common characteristics that can be associated with it such as:

- Loosely coupled and independently deployable components.
- Communication between services takes place at the process level over

[1] https://www.iofficecorp.com/blog/cloud-first-strategy-2020. Why You Need To Adopt A Cloud-First Strategy In 2020

[2] https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience Why Netflix, and Amazon Migrated to Microservices

the network instead of function or class level.

- Each service can have its own stack of programming language, frameworks and database model.

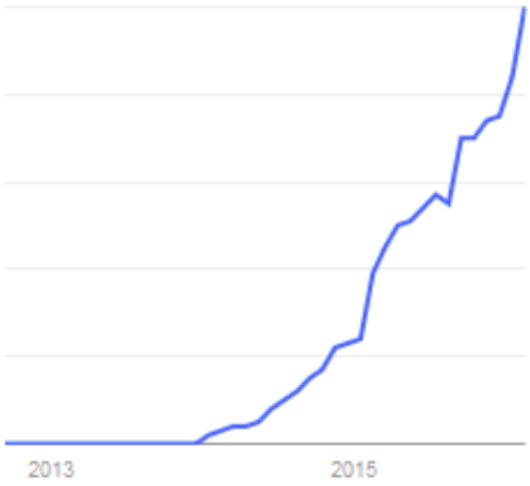

Figure 1.1: Trend graph for microservices[4]

1.1 Motivation

One of the key advantages in microservices is that it decomposes business capabilities to a small set of independent components which then can result to have a more agile delivery of new features and bug fixes into production[5]. Further, in a traditional multi-tier architecture the entire stack of applications is bounded to few technologies such as Oracle DB as the Database solution for the entire system, and Java as the programming language to process the business logic of that application. This approach has an obvious drawback as it forces all the components to follow the same stack even if there is a better alternative to handle a particular feature or a better data modeling to store the data. Conversely, in microservices architecture, each service can have its unique stack of technologies based on the functional requirements of that component and the development team's preferences without having to be dependent on other component's technology stack. Further, microservice enables horizontal scaling[6]; in monolithic architecture, because the entire system is bundled together, the only way to scale the application infrastructure is to scale everything either up or down. In microservice architecture, however, since services

are deployed and running independently from each other it is possible to scale specific components of an application independently.
Despite the growth and importance of microservices in industry, there has not been sufficient research on microservices, partly due to lacking a benchmark system that reflects the characteristics of industrial microservice systems[7].
Designing software based on microservices involves answering questions and overcoming technical challenges that often does not exist in a monolithic architecture. Some of these challenges are:

- Interprocess communication(IPC): It is related to the mechanism that microservices communicate with each other over the network.
- Service discovery: the process of finding each microservices network address in the cluster.
- Decomposition strategy: the process of breaking a large monolithic system into smaller components.
- Managing ACID Transactions: Handling ACID[3] transactions in microservices-based system is different from monolithic as a single transaction spans over multiple services that are distributed.

Each of the above-mentioned challenges can be a wide domain for itself with different techniques and tools to address each.
This thesis work is solely focusing on Interprocess communication(IPC) between services. IPC is one of the important challenges of microservices architecture that does not exist in a monolithic system[8]. In monolithic-based systems, components can call each other at the language-level while in microservices each component is running on its own process on possibly a different machine from other services, and that is where IPC plays a crucial role in microservices-based system.

The choice of IPC mechanism is a critical architectural decision as it can affect application performance and availability, and there are different IPC methods to choose from[8].

[3]https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.4.0/product-overview/acid.html

1.2 Problem Description

As of today, there are no concrete explanations or any standardized approach that can help to decide the right IPC method when designing microservices-based applications. Due to this reason, there is an abundant confusion around the question of when to use which method and what are the trade-offs for choosing that method. It becomes particularly more challenging to decide as there is no right or wrong option, but rather more or less applicable based on the technical and functional requirements.

1.3 Objective

The objective of this thesis work is to conduct research on **Interprocess communication (IPC)** in microservice architecture, and measure its impact based on the following non-functional requirements:

- **Performance efficiency**: This is measured by comparing each method throughput and latency. Throughput is the total number of requests that application is capable of processing within a specific period, while latency is the time it takes for each request to be processed.
- **Availability**: It is the property that the system is available for legitimate use. This non-functional requirement is often associated with system failure and the consequences of such an event[9].
- **Scalability**: This attribute associates with the ability of the system to function with comparable response as the number of requests to the system continue to increase.
- **Complexity**: This quality attribute describe a set of characteristics of software which focus on the ease of maintaining the software code and its integration with other components.

The motive behind selecting the above quality attributes for this research is that the choice of IPC method directly impacts the chosen non-functional requirements in a microservices-based system, while other non-functional requirements such as Security, and Maintainability can span over few other areas and goes beyond IPC. Being able to measure these qualities in the system are critical in order to a achieve an efficient management of any software system[10]. Moreover, the chosen quality attributes are among the top priorities for most

modern applications. For instance, availability is essential in order to achieve business continuity without any disruption, and such quality is highly rated by executives within the industry[11][12].

1.4 Goal

The goal of this thesis work is to compare IPC methods in microservices architecture from a non-functional requirements standpoint. To achieve this goal, an experimental microservices-based system that employs different IPC methods will be developed. The system will undergo several test scenarios that allow a comparison between each method. When developing microservice systems, the outcome of this work can be used as a guideline for selecting the right IPC method for the right use-case.

1.5 Research Question

The research question that has been derived from problem description, objective, and goal is formulated as:

How does the choice of IPC method impact the non-functional requirements of a microservices-based system?

To break-down the research question and align it with non-functional requirements that were described in the Objective section, this work aim towards answering the following sub-questions:

1. From a Performance Efficiency standpoint, what are the implications for utilizing different mechanisms for implementing IPC communication in microservices architecture?

2. How does the IPC method choice impacts the Availability of the system?

3. Which communication mechanism offers better Scalability for the system as the number of requests to the system increases rapidly?

4. Which of the communication mechanism has higher Complexity and requires more effort to be developed and maintained?

1.6 Methodology

The principal aim of this thesis work is to compare and contrast different IPC methods for microservices architecture and gain a clear understanding about each method's strengths and trade-offs from several non-functional requirements aspect. To facilitate in achieving this objective and be able to answer the research question, the author has adopted "Design science research methodology(DSRM)"[13] as the methodology for the thesis work.

> *"DSRM consistent with prior literature, it provides a nominal process model for doing Design Science research, and it provides a mental model for presenting and evaluating Design Science Research in Information System."* [13]

The Design Science methodology consist of the following activities:

- Problem Identification and motivation: In this stage, the author defines the specific research problem and justify the value of a solution.[13] This step is addressed in the first chapter of the document.
- Definitions of the objectives for a solution: In this stage, the author infers the objectives of a solution from the problem description and existing knowledge of what is possible. This step is manifested in the second and third chapters of the document, which describes the theoretical background and related work.
- Design and Development: This step is about creating the artifact needed to solve the problem which has been addressed in the fourth chapter of the document. For this step a proof of concept microservices application for an e-commerce scenario using different IPC methods have been designed an developed. This step is demonstrated during the fourth chapter of the document.
- Demonstration: This step is to demonstrate how the produced artifact can be utilized to solve the given problem. This step is manifested in the fifth chapter of the thesis.
- Evaluation and Communication: This final step aims to reiterate the problem and its importance as well as the usage of a produced artifact to solve the problem and its novelty. This activity has been covered in chapters six and seven.

1.7 Ethics and Sustainability

There are no direct or indirect ethical concerns related to this project and this work complies with the IEEE code of ethics[4]. No personal data has been obtained or used throughout the work. The data that has been produced during the testing phase contains no sensitive information that can pose any danger to an individual or any organization. Further, the author has adequately referenced any previous work that has been utilized in this work.
In regards to sustainability, microservices architecture aims to improve the way software systems run by enabling the specific components that are needed to be scale instead of having to scale the entire system. This model can result in better cost efficiency and energy consumption. Moreover, microservices offer better reliability and fail-over management than traditional architecture, which can be helpful in mission-critical systems such as in the health care industry.

1.8 Delimitation

Although there are other factors that can affect Performance efficiency, Availability, Scalability, and Complexity of a software system, this research only considers the effects of IPC on the mentioned quality factors. Moreover, measuring the difference in error rate between the IPC methods was not a part of this study. Therefore during the testing experiment the error rate difference between IPC methods has not been recorded.

[4]https://www.ieee.org/about/corporate/governance/p7-8.html

Chapter 2

Theoretical Background

The goal of this chapter is to provide an overview of microservices architecture, its taxonomy, followed by different types of communication models in microservices.

2.1 Software Architecture

As software systems grow and become larger and larger, the algorithms that are written to process the business requirement or data structure that is designed for storing data no longer constitute the major design problems. Enterprise software systems are built from many components, and the organization of the overall system, often regard as Software Architecture, presents a new set of design challenges. This level of design has been addressed in several ways, such as having informal diagrams and descriptive terms, templates, and frameworks for systems that serve the needs of specific domains[14]. Software architecture is the most fundamental organization of any software system and determines the approach for wiring each component of the entire software[15].

> *"Architecture is a tricky subject for the customers and users of software products - as it isn't something they immediately perceive."*
> *Martin Fowler*

A bad architecture can lead to the growth of unnecessary complications, redundant code, and poor performance. This then can lead to having software that is harder to modify. Consequently, new features become slower to release with more potential for bugs and defects.[1]

[1] https://www.martinfowler.com/architecture/ , Software Architecture Guide

2.1.1 Evolution of Distributed Systems

Most of today's enterprise IT applications are distributed - either at a larger-scale that operates over the internet or as a collection of local networks[16]. The demand for developing Distributed Systems has increased significantly over the past years as applications are now required to serve a wide range of consumers in different geographical locations in real-time. Initially, most software systems did not have to consider the networking layer to enable other devices to communicate with each other. As networking capabilities have advanced over the past decade, the client-server model has gained popularity to enable different users to send requests to server and demand service from it. In the client-server model, the assumption is that the client has limited computing capabilities, and therefore all the processing shall take place at the server-side. Mobile agent[17] is another paradigm for distributed systems that appeared after the client-server model. Mobile agent aimed to overcome the slow network connectivity between client and server in the client-server model as some applications needed constant contact with the server for processing and network connectivity was the bottleneck[18]. After that, Service Oriented Architecture (SOA)[19] arises, which offered an integration solution that enabled different distributed systems to communicate with each other. In SOA approach, the client communicates a message using a Remote Procedure Call such as Simple Object Access Protocol(SOAP)[20] with other components. SOA enables software components to work with each other over multiple networks independent of each other's platforms or programming languages. Service-Oriented Architecture is scalable and highly reliable while it offers platform Independence, however, SOA is complex, and slow with a high overhead as all the inputs in SOA architecture must be validated before it gets sent to the service which can lead to longer response time and machine load[18]. Due to these shortcomings more lightweight communication mechanism was needed to satisfy business needs while maintaining the scalability and reliability that SOA offers, which becomes the motive for having more lightweight communication mechanisms such as REST API and more lightweight architecture known as microservices.

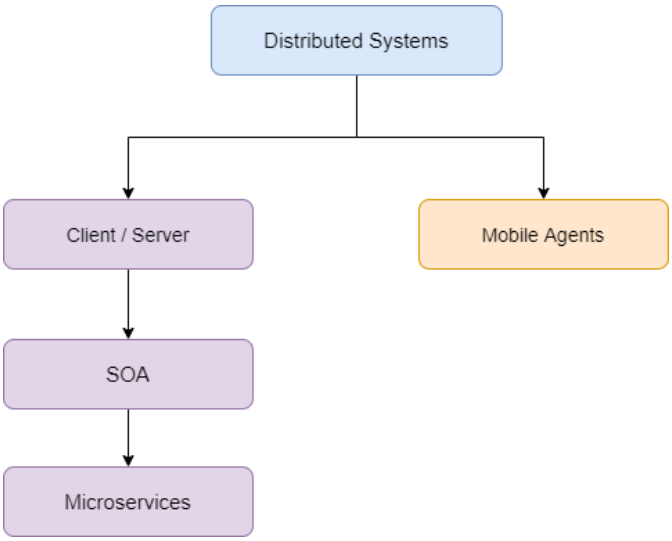

Figure 2.1: Evolution of Distributed Systems[18]

2.1.2 Microservices Architecture

Microservices are one of the architectural approach for building distributed software applications. Organizations are adopting Microservice architecture to achieve faster delivery, better resiliency, and higher safety as the scale of their systems increases[21]. Over the past decade, the term microservices has been used in different forms while they shared some common characteristics.

> *"Microservices are small, autonomous services that work together."*
> *Sam Newman*

One of the significant differences that set microservices apart from architectures like monolithic is how the entire system is broken into smaller functions that are running independently within a bounded context. Each microservice can then be extended and deployed without affecting any other microservices that are running.

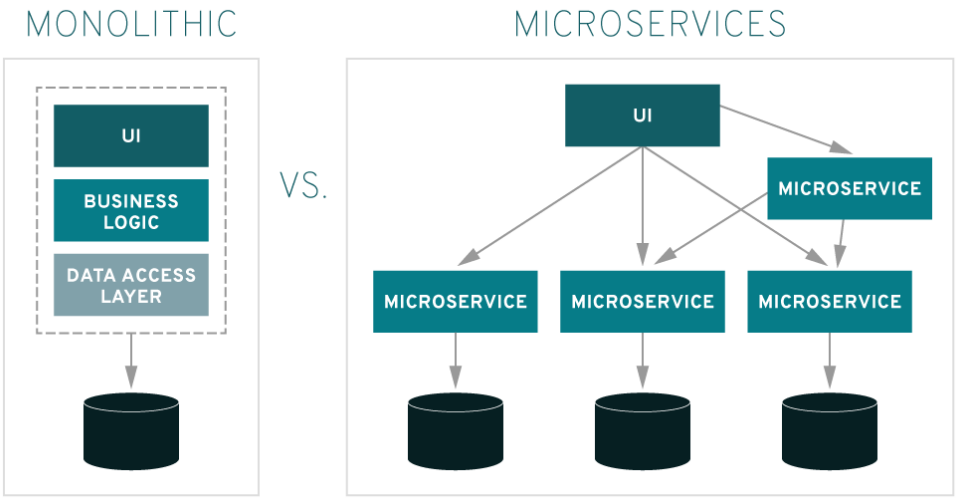

Figure 2.2: comparison between Monolithic and Microservices

As of today, there is no precise definition for the term microservices; there are however, common characteristics that every software must possess to be able to claim the term microservices for itself.[2] These characteristics are but not limited to:

- **Small in Size**: How large or small each service supposed to be is a relative term that is varying based on different factors such as whether the services are being designed around high-level business requirements like Customer Authentication/Authorization or at function level such as Customer Login/logout functionality. Having said that, based on this[22] paper's survey that was conducted by analysing different teams in 40 companies, having services that are really small with under 100 LOC, or very large with above 1000 LOC are rarely practices. It also shows that services are focused and narrowed on performing one particular task[22][23].

- **Bounded in Context**: In a microservice architecture, each service is encapsulated around a particular business capability and a subset of those capabilities creates a domain[24]. These encapsulated services operate within the boundary of the domain and does not get exposed to the rest of the system. This can then results to have better autonomy, decentralised

[2]https://martinfowler.com/articles/microservices.html

management, and easier for the services to be repeal and replace[23]. If there is a need for one of the services to communicate with outside of its domain it happens via an explicit interface at the domain level[25].

- **Distributed**: microservices is not the only architecture that operated in a distributed manner. Service-Oriented Architecture (SOA)[19]which has been around for longer is also distributed. Microservices, however, have taken this characteristic above and beyond SOA by decentralizing data between different services. Each business capability and its related data is managed by a service that is independent from other services and can be deployed and run on its own host[26].

- **Independently deploy-able**: Each microservices can be deployed independent from other services at any given time using Continues Delivery and Deployment[27] pipeline. In another term, each microservice can be deployed without any coordination with other owners of other services[28].

2.1.3 Benefits of Microservices architecture

- **Technology Diverseness**: Systems that are built based on Microservices architecture is a collection of independent services that communicate with each other over a network. This feature enables choosing the most suitable programming language, framework, or data modeling according to the requirements of that particular service instead of having a universal technology stack for building every component of the system. In this architecture, developers have the freedom to leverage technologies that make the most sense for the task and people performing that task[28].
  For instance, in an e-commerce application that is designed using a microservice architecture, the component that is responsible for customer management can use Java as its programming language and use a relational database to store and retrieve customer's data. Meanwhile, a component that is responsible for storing product catalog and performing search queries on it can use NodeJS as the programming language and non-relational database for storing and retrieval of the data to take advantage of faster performance over relation DB.

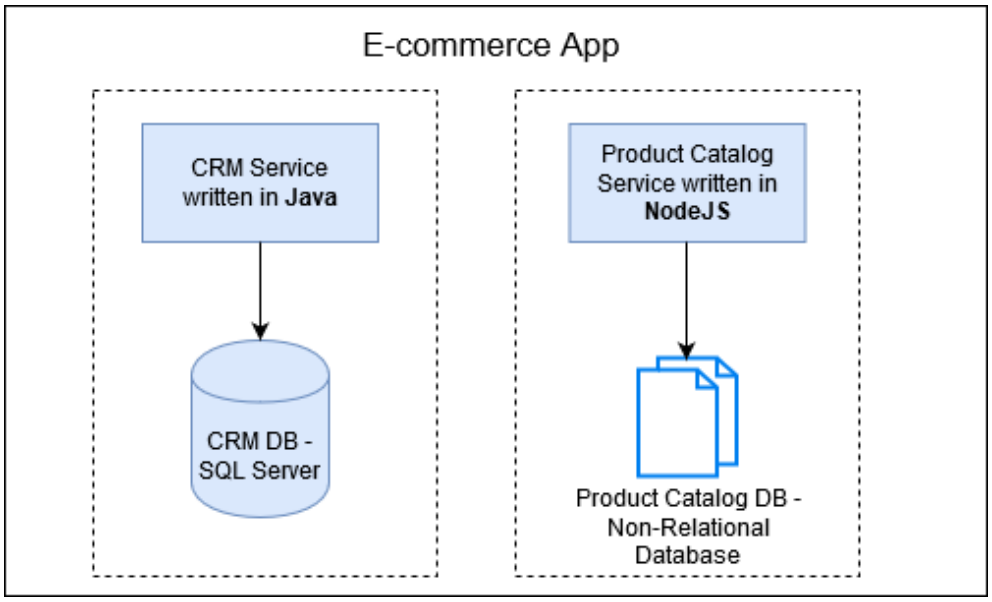

Figure 2.3: "Microservices allows to easily embrace different technologies"[25]

- **Efficient Scaling**: scalability can be regarded as one of the most important motivations for utilizing microservices architecture[28]. In Monolithic architecture when there is a need to scale, everything must scale together since all the components are bundled together. However, the microservice architecture enables targeted scaling, which is to scale the only the particular components that are in demand. Having these targeted scaling features is important for applications that are running in the cloud since most cloud providers charge their customers based on the resources they utilized. It would be expensive for companies to scale the entire application in the cloud if only one part of the application has a higher load.

- **Higher Availability**: microservices architecture can offer higher availability to software as microservices architecture can be replicated and spread across data-centers in different geographical locations and be able to split the loads and cope with failure, this is in contrast of monolithic architecture[26]. In addition, ease of deployment in microservices is another factor that can improve the system's availability. Traditionally, because the entire system was bundled together, the only way to release a small change was to release the entire system. In contrast, in

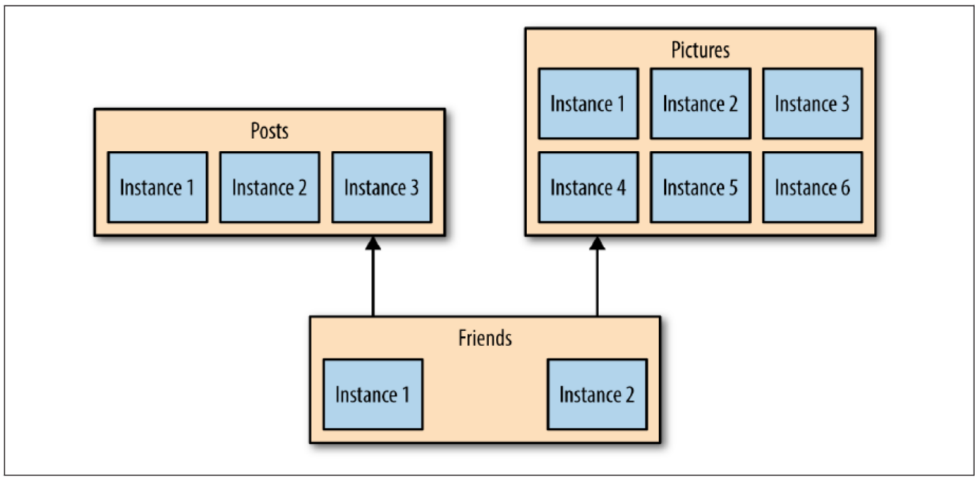

Figure 2.4: An example of targeted scaling in microservices architecture.[25]

microservices, one component can be released to Production without affecting the entire system which can result in faster deployment with less likelihood of breaking changes.

- **Organisational Alignment**: microservices architecture offers a more desired operational model. The architecture reflects the way most of today's business leaders prefer to structure and run their teams and development processes.[3] Having a large team with a large code based becomes increasingly difficult to manage its changes and can likely reduce the developer's efficiency. Microservices help to reduce the number of developers working on the same and codebase to gain better team size and higher productivity[25].

2.1.4 Challenges with Microservices

Like any other software architecture or design pattern, microservices are not a silver bullet nor a universal architectural solution for every software application. One of the key challenges of microservices is the complexity it introduces to the system. The development, deployment, testing, and debugging microservices are often more complex than monolithic-based applications due to its distributed nature.[8] Furthermore, as microservices enables each component of the application to have its own stack of technology it can lead to more overhead and higher complexity when the different team in an organization requires to coordinate with each other[18][29].
In microservices architecture, there isn't any concrete definition of how small or large each service has to be. Due to this reason, if a team incorrectly decides on the size of each microservices, it is possible to have a system that is distributed but highly coupled to each other which then defeats the purpose of microservices[8]. Finally, there will be an extra latency that gets introduced to the application as a result of process communication in microservices. This is because a single request would have to travel between multiple services to be fully processed and return a response. How to efficiently minimize the communication latency between microservices architecture is the core motive behind this research thesis.

2.1.5 Interprocess Communication in Microservices

Deciding on how microservices communicate with each other is one of the most important and fundamental decisions to make when implementing a system based on a microservices architecture[25]. Helping to find out the most suitable choice of Interprocess communication(IPC) is the key challenge that

[3]https://www.ibm.com/cloud/learn/microservices#Vartoc-how-micros-uogGcpL9 , What are microservices?

this thesis work is aiming to overcome. IPC plays a much more critical role in microservices architecture as it does in Monolithic[8], that is because in monolithic architecture module can access and invoke each other at the language-level, in contrast, microservices structure the system as a set of independent distributed service with the possibility that each service could run on a different host than the other. An important dimension when selecting an IPC mechanism for a microservices is whether the interaction between them is Synchronous or Asynchronous[30]. The following section explores these two types of interaction:

2.1.5.1 Synchronous Communication

This form of communication is often regarded as a request/response interaction style. In this mode, one microservice makes a request to another service, and wait in a timely fashion for that services to process the result and send a response back. In this style, it is common that the requester blocks it's operation while waiting for a response from the remote server.
HTTP-based REST API[31] and **gRPC**[4] are the two most common type of Synchronous communication when building microservices[8]. These two type of Synchronous communication has been selected to be assessed for this research work.

- **HTTP-based REST**: "Representational State Transfer" (REST)[31] is an architectural style that is commonly used for designing application programming interfaces (APIs) for modern web services[32].
  REST API is one the most common method for two systems to exchange data with each other regardless of their software architecture. In this method Client programs use APIs over HTTP protocol to communicate with web services provider. In a system that uses REST API for its IPC communication, each service typically has its own web-server up and running on a specific port such as 8080 or 443, and each service exposes a set of endpoints to enable the interactions with other microservices and exchange of information between them. In REST API each

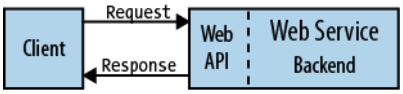

Figure 2.5: REST api directly listens and response to requests.[32]

[4]https://grpc.io/docs/

endpoint has an interface that determines the set of operations that other services can call and get a response back. This is similar to Monolithic architecture, however, in a monolithic architecture, an interface is often specified using a programming language construct like Java or C# interface. A programming language interface defines the list of methods that a consumer can invoke while keeping the implementation away from its consumers. In a microservices architecture, a service's API is a contract between the services and its consumers. Each service API has a list of operations with name, required parameters, as well as return values.
REST API mainly consists of two components: resource, and verb. A resource usually represents a single business entity or a collection of them such as Customer, Product, or Order which can be accessed from a Uniform Resource Identifier (URI), while verb refers to HTTP method by which the CRUD[5] operation can be executed[33].
REST API relies on HTTP verbs[6] to manipulate the business entities. For instance a HTTP GET call to /customer/{customerId} endpoint can return the data of the given customer, while a HTTP POST call to /customer/{customerId} can be used to update the information about that particular customer.

- **gRPC**: gRPC[7] is a modern Remote procedure call (RPC) based protocol designed and published by Google for developing cross-language client and server. RPC is a mechanism used in many distributed applications to facilitate Interprocess communication. RPC was first implemented by [34] and it has been regarded as a protocol that enables a message exchange between two processes with characteristics of low overload, simplicity, and transparency[35]. By default, when a client sends a request to a server it halts the process and waits for the results to be returned. remote procedure call is therefore considered as **Synchronous** from of communication[36]. Unlike REST API that uses HTTP/1.1 as the default transport protocol, gRPC runs over HTTP/2.0 which gives gRPC several advantages related to performance and security[33].

[5]Create, Read, Update, and Delete
[6]https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
[7]https://grpc.io/

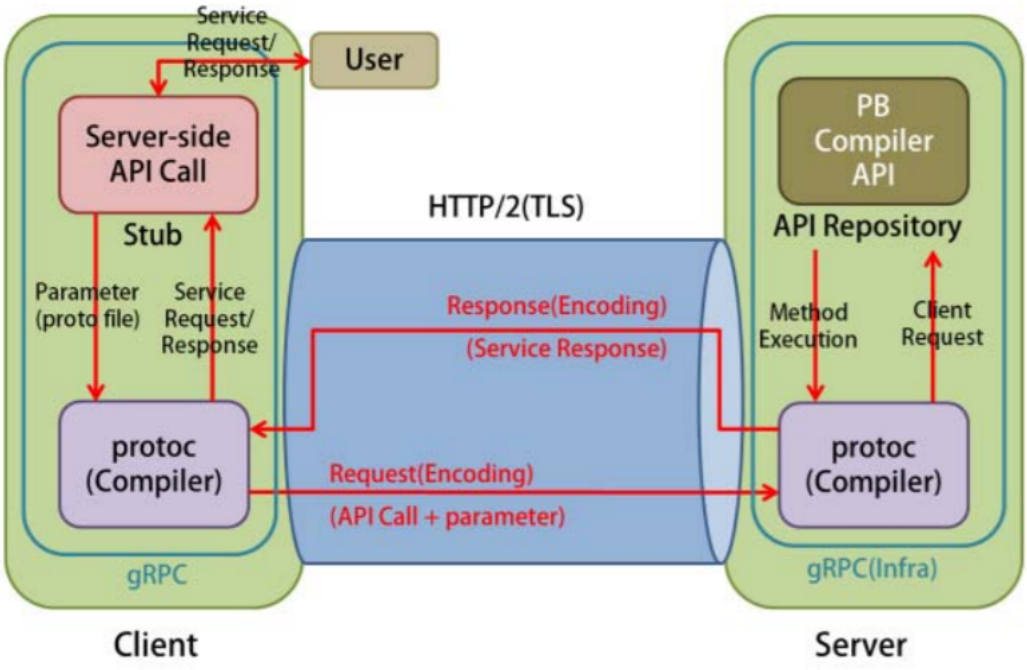

Figure 2.6: The operation process between client/server in gRPC[33]

Figure 2.6 demonstrates gRPC request/response life-cycle. The client has a local object known as "stub" that implements the methods for calling the gRPC server. As soon as the client triggers the stub's method with its required parameter, a request will be sent to the server. Upon receiving the request the server decodes the request and begins processing the request. Finally, the result is encoded by the server before it sends back to the client[33].

2.1.5.2 Asynchronous Communication

Asynchronous form of communication can be implemented in microservices when services exchange messages with each other using messaging pattern[8]. In this form of IPC, microservices have a message broker that acts as an intermediary between the services to coordinate the request and responses[8].
One of the fundamental differences in Asynchronous communication as compared to the Synchronous mode is that in Asynchronous communication the

[8]https://blogs.mulesoft.com/dev/design-dev/Asynchronous-messaging-patterns/

client no longer makes a direct call to the server and expect an immediate answer for it. Instead, the client publishes a request to the message broker. One or more than one service will take the request from the broker and process it further before placing the result back to the broker. In another word, in the Asynchronous form of communication the interaction between microservices it manged by an intermediary service known as a message broker.

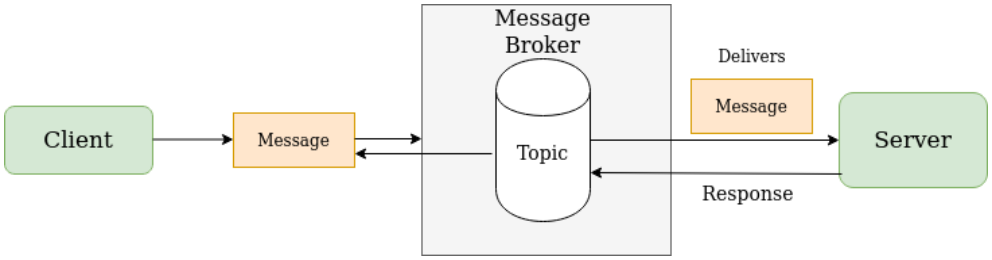

Figure 2.7: Async Communication using Pub/Sub Pattern

In this mode, because the communication is Asynchronous, the client does not block its operation while waiting for a reply.

2.2 Message format and Data Serialization

The whole idea of IPC is to allow microservices to exchange messages between each other. Messages contain information that is necessary for microservices to process their business requirements. Information such as ProductID, CustomerID, CreditCard Info, and etc. Besides choosing the method of IPC, another important decision to make is which message format should services utilize when communicating with each other.

> *"The choice of message format can impact the efficiency of IPC, the usability of the API, and its evolvability.[8]"*

Microservices can choose to exchange **text-based** formats such as JSON[9], or XML[10]. Both can offer to some extent different features such as portability, safety, security, human-readability, etc.[37] Alternatively, services can use **binary** formats such as Google Protocol Buffer[11] or Apache Avro[12] to exchange messages between each other which can offer higher efficiency and

[9]https://www.json.org/json-en.html
[10]https://www.w3.org/TR/xml/
[11]https://developers.google.com/protocol-buffers
[12]https://avro.apache.org/

lower metadata redundancy.[37]

For this thesis work, **JSON** as the text-based format and **Protocol Buffer** as the binary-based format has been selected to compare and contrast during preliminary research and implementation of the proof of concept. For binary message format Protocol Buffer has been chosen mainly because Protocol Buffer is the default message format when using gRPC as the communication method between microservices. Although it is possible to replace Protocol Buffer with a text-based messaged format such as JSON when using gRPC[13], however, it is not a practical approach as a part of gRPC performance is due to the underlying binary-format is uses to exchange message.

2.2.1 JSON

JSON was chosen over XML as XML has lost its popularity over the past few years and most modern systems no longer rely on XML to exchange information between each other.[37] JSON offers a much faster and more efficient exchange of data while being significantly less verbose than XML.[14] JSON is smaller in size than XML and it is estimated that to be serialize and deserialize one hundred times faster than XML.[38]

```
{
  "productId" : 2342344543346354,
  "productName" : "Sony 32-Inch HD Smart TV"
}
```

Figure 2.8: Example of a JSON object

Figure 2.8 demonstrates an example of JSON object that encodes *productId* and *productName* of a shopping cart object. While JSON offers a significantly higher performance than other text-based message formats in particular XML, however, there are arguments against JSON that it lacks support of namespace and input validation[38].

[13]https://grpc.io/blog/grpc-with-json/

[14]https://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html

2.2.2 Protocol-Buffer

Protocol buffer is Google's language-neutral or platform-neutral, for serializing structured data.[15] It is the default messaging format for gRPC. It is fast, and compact, binary format which is supported by a variety of programming language such as C++, Java, C#, Python, and Javascript. Every property filed in Protocol Buffer message has a type code with a unique number. The receiver of Protocol Buffer message can extract the fields that it needs while ignoring the unrecognized fields. This feature enables APIs that communicate using gRPC to expand while remaining backward-compatible[8].

Unlike JSON that is schema-less, protocol buffer enforces schema which is a way to overcome the JSON limitation for supporting namespace and input validation. Every business object that requires to be exchanged between services needs to have its own Protocol buffer schema that is defined in a ".proto" file. Figure 2.9 depicts an example of .Proto file.

```
message ProductOrder {
    required int32 productId = 1;
    required string productName = 2;
    optional string productColor = 3;
}
```

Figure 2.9: an example of ".Proto" schema file

[15] https://developers.google.com/protocol-buffers

Chapter 3

Related Work

This chapter aims to provide an overview of the existing research that has been conducted around microservices architecture and, in particular, the communication mechanism in microservices and previous investigations related to data serialization techniques that can be applied when implementing IPC for microservices-based systems.

In [39] the author has performed a thorough comparison to measure the difference in performance and error rate between monolithic architecture and microservices. The author has developed two systems that are identical to each other from a business functionality perspective but different in software architecture. The first system works on a monolithic architecture while the second system is microservices-based. In the second system, the Synchronous approach with REST API has been used for IPC communication and JSON as the data exchange format between each microservice. A load test was performed against both systems by sending a high number of requests within a specific duration. The outcome of the first test concluded that the monolithic system average response time performed better than the microservices-based system. The result was expected since in monolithic everything gets executed on the same machine while in microservices the result has to be processed and computed in different instances and be communicated over network line. In addition, to find out about the resiliency and error rate difference between these two systems, another test case was performed against both systems by sending a significantly higher load than the first experiment within the same duration as the first case. The result of the second test revealed that during high-load testing the microservices-based system performed better than monolithic by 56% on average. The outcome of this research shows that microservices architec-

ture can have a potentially higher fault tolerance as compared to monolithic architecture while also signaling a great potential for improving Interprocess communication between services to reduce the latency gap with monolithic-based systems.

Meanwhile, [40] has conducted a migration for a real-world mission-critical case study in the banking industry by transforming a monolithic-based system into microservices-based and observe how Availability and Reliability of the system changes as a result of the new architecture. The solution consists of decomposing several large components to which some of them requires to communicate with third-party services. The services in the new architecture uses message-based Asynchronous communication as its IPC model to exchange data with each other. One of the motives behind using massed-based communication was to make services decoupled from each other. The author believes that aiming to have a simple and decouple integration between services and the following principle to handler failure will eventually lead to higher Reliability in microservice architecture. The author has argued that microservices-based architecture has given the targeted system to higher Availability as the new system is now broken into several components and decoupled from each other, which makes it possible to load-balance individual services as needed. This was particularly not possible in the legacy monolithic-based system. At the same time, the new architecture offers higher Reliability and can better cope with failures this is due to the fact that in the new system the communication relies on a message-broker that can be configured to ensure all messages get delivered eventually.

In [41], the author has compared REST API performance versus Advanced Message Queuing Protocol(AMQP)[42] protocol which is one of the protocols used in message-based communication that falls under Asynchronous category. The study has been done by measuring the averaged exchanged messages for a period of time using the REST API and AMQP. The authors performed the experiments by setting up two independent software instances that constantly receives messages for a 30 minutes period with an average 226 request per second. Each instance would process the received input message and store them into a persistent database. One of the instances was based using REST API communication, and the other one was set up using AMQP. After executing the experiments, the work has concluded that for scenarios where there is a needs to receives and process-intensive amount of data, AMQP performs far better than REST API as it has a better mechanism for data loss prevention,

better message organization and utilize lower hardware resources.

In [43] research that was conducted by a team of researchers at IBM, the aim was to design an infrastructure that is optimized for running microservices architecture. The team built two versions of the sample application—one based on monolithic and the other based on microservices. The team discovered a significant performance overhead and higher hardware resource consumption in the microservices version of the application as compared to the monolithic one. The paper has marked poor design of process communication in microservices architecture as one of the significant performance degradation, and therefore unleashed the potential for further research and improvements in this topic. The paper has also pointed out that the network virtualization technique that is currently being used in a microservices architecture is another non-negligible reason behind the performance gap of monolithic versus microservices architecture. The paper, however, has not prescribed any specific solution or suggestion as to how to overcome these challenges but rather pointed out the potential future work for it.

In regards to message exchange format and data serialization, in [33], the authors have compared the two methods of the Synchronous communications, which are REST API, and gRPC for application-layer communication in Software Defined Networking (SDN)[44]. Although the comparison has been conducted for SDN and not microservices, however, the outcome of the research is highly applicable for implementing service communication in microservices architecture. The research has favored gRPC over REST API and argued that the deterioration of network latency could have a bigger negative impact when communication takes place using REST API, which runs on HTTP/1.1 protocol as compared to gRPC that runs on HTTP/2.0. Due to this difference, gRPC can take advantage of multiplexed stream function and header compression that are available in HTTP/2.0 version. Moreover, the paper outline that gRPC offers better security in communication compared to REST API by adopting TLS via creating a secure channel in HTTP/2.0. In contrast, REST API does not provide such channels out of the box.

In [45] the author performed an experiment to discover the most optimal message and data serialization format by comparing two text-based formats, which are XML and JSON and two binary-based formats which are Protocol Buffer, and Apache Thrift[1]. The experiment performed its measurement by serializing and deserializing objects that are designed to be text-heavy for

[1] https://thrift.apache.org/

a constant number of times. The research has concluded that the XML data format should be avoided at all times unless it is necessary due to its poor performance. Furthermore, the paper suggests to use JSON for the development of existing web services, while suggesting to use a binary-based format such as Protocol Buffer as the default message format for the development of new systems as it offers higher speed and lighter size.

Chapter 4

Implementation

This chapter provides details about the use-case that has been implemented in order to evaluate the impact of each IPC mechanism in microservices architecture.

4.1 Use case Background

A set of functionalities for an e-commerce scenario has been implemented in order to simulate a real microservices-based system. In this scenario, the goal is to mimic fetching all the information required to display a product page of an e-commerce system. In this use case, when a client request a product page to be loaded by providing a 'product Id' the following microservices shall get triggered:

- **Product Information Service**: This microservice is responsible for fetching the primary metadata associated with the requested product. Information such as product name, price, description, color, and image from its database.
- **Product Review Service**: This microservice is responsible for fetching the customer reviews associated with the requested product from its database.
- **Product Recommendation Service**: This microservice is responsible for fetching the product recommendations based on the requested productId from its database.
- **Product Shipping Service**: This microservice is responsible for fetching available shipment options and the delivery estimates based on the

given product from its database.

- **Customer Shopping Cart Service**: This microservices is responsible for fetching the existing items in the customer's shopping cart in order to display them to the customer.

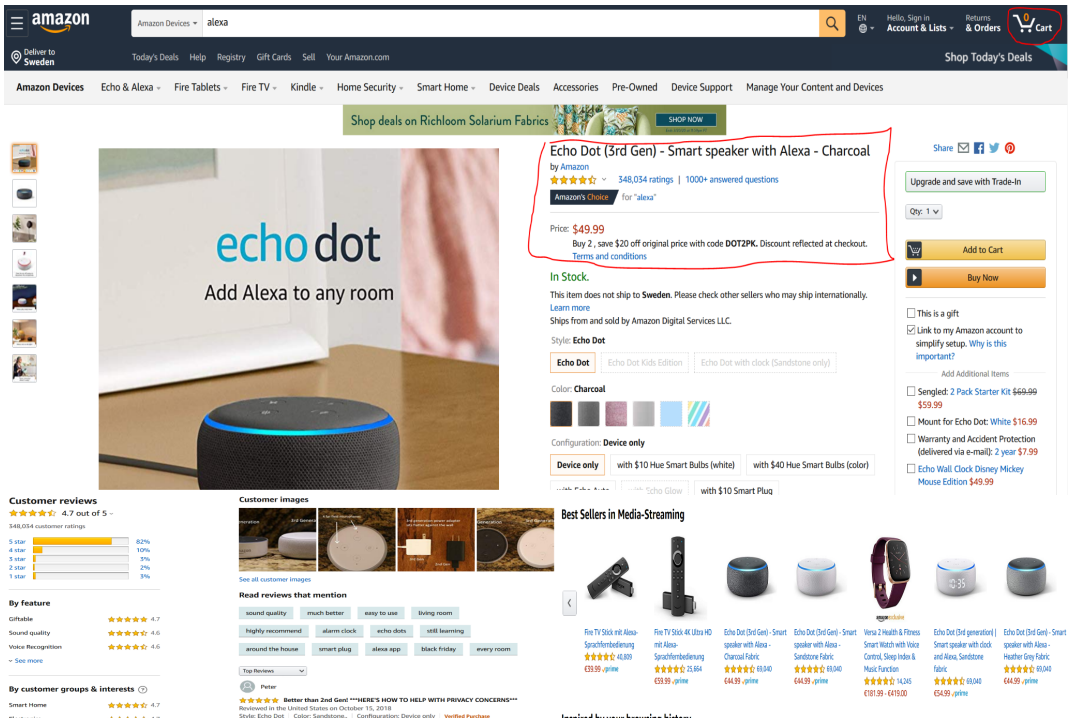

Figure 4.1: An Example of a Product Page in an E-commerce app

Figure 4.2 provides an overview of how the microservices work together in order to process the request. The API Gateway is the entry point to the system, and clients device such as mobile app or web browser send a request over HTTPS to the API gateway endpoint. The gateway then communicates back and forth with each microservice depending on which IPC method and data serialization format the system is using. Once the response from all the five microservices has returned to the API gateway, the gateway aggregates them and sends it back to the client.

Regardless of which IPC method or data serialization format API Gateway uses to communicate with microservices, the communication with the client always remains over HTTP using Rest API, and JSON format. This is because HTTP and JSON are the de-facto protocol and data format that browsers and

mobile app uses to communicate with server while other protocols such as gRPC are not supported by browsers or mobile devices.

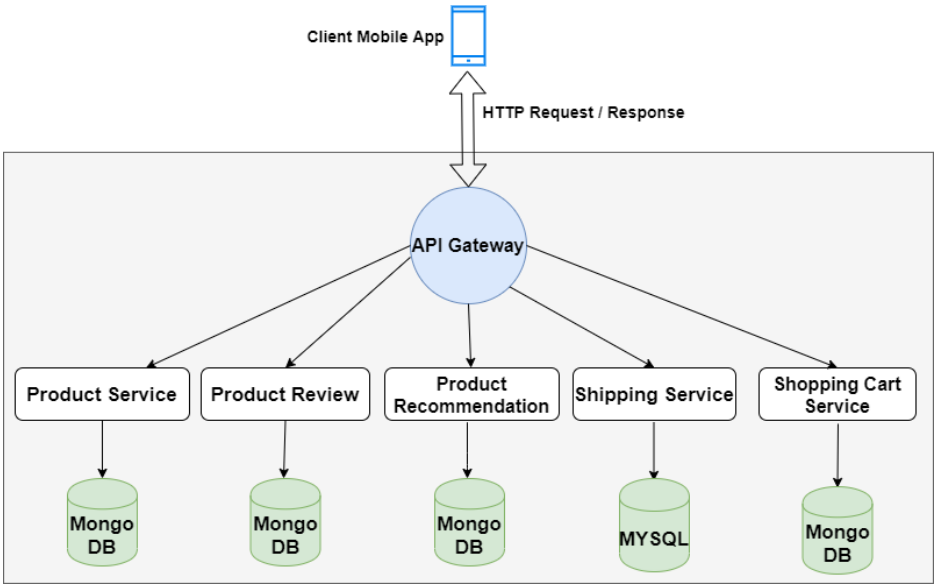

Figure 4.2: Overview of services involved in loading a product page

4.2 Test Environment Setup and Technology Stack

The proof of concept application has been deployed to Microsoft Azure Kubernetes Cluster Service[1] in order to run the test system. Having microservices to be deployed into cloud Kubernetes services instead of running them on a local computer offers better isolation for tracking performance while enabling better reproducibility for future testing.
All the microservices has been developed using NodeJs[2] as the programming language, which is a popular framework for building high-performance, concurrent programs[46]. A non-relational database system known as MongoDB[3] has been used as the database solution for all the microservices except for the service responsible for providing shipment information. Due to the nature of data required by shipping service, the shipping service uses MYSQL[4] which

[1] https://azure.microsoft.com/sv-se/services/kubernetes-service/
[2] https://nodejs.org/en/
[3] https://www.mongodb.com/
[4] https://www.mysql.com/

is a relational database. Docker[5] has been used to containerize all the microservices and enabled them to be run on Kubernetes service in the cloud. Docker enables packaging the application and all of its dependencies into a self-contained unit or so-called container[47].

Instance Type	Azure DS2-v2
vCPU	2
Memory	7 GiB
Storage	8 GiB, SSD, 6400 IOPS
Kubernetes Version	1.14.8
Node Count	3

Table 4.1: Kubernetes Cluster specification for the test system

Three Kubernetes cluster with the exact same configuration as described in Table 5.1 has been created to run the three different IPC mechanisms developed for this research work.

4.3 Synchronous communication models setup

- **REST API**: Figure 4.3 demonstrates how the IPC mechanism takes places when microservices communicate with each other via REST API. The API Gateway receives requests from the client app which includes the product ID and Customer ID[6]. The gateway then makes a call to each individual microservices over REST API by passing the *productId* and waits for each service response to come back. In this model, as the communication takes places using REST API over HTTP protocol, each microservice needs to run a web server in order to be able to handle HTTP requests.

[5] https://www.docker.com/

[6] Only ShoppingCart microservice requires Customer ID, and if the customer is not logged in it will be null.

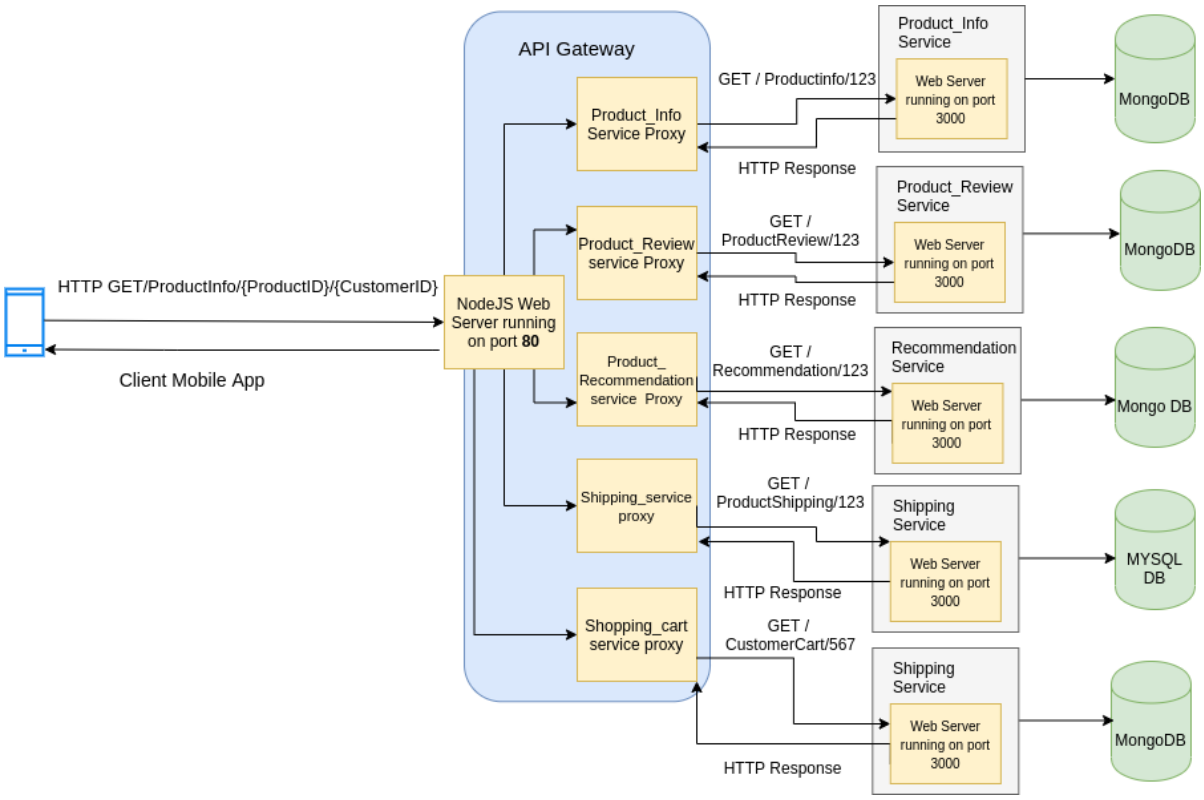

Figure 4.3: REST API-based IPC communication

- gRPC: in gRPC each microservice runs a gRPC server on a specific port[7], the API gateway runs to act as a client of every gRPC server and establish a connection with every microservices during the initial setup. Similar to the previous method, the API Gateway receives the product request from customer's mobile app or web browser over HTTP, and then take that request and calls all microservices by providing the required parameters and waits for their responses. Since gRPC uses binary-based data serialization format the results that get returned from microservices to API Gateway are in bytes. the API Gateway,therefore, converts that bytes and transform it to JSON format before returning it to the consumer.

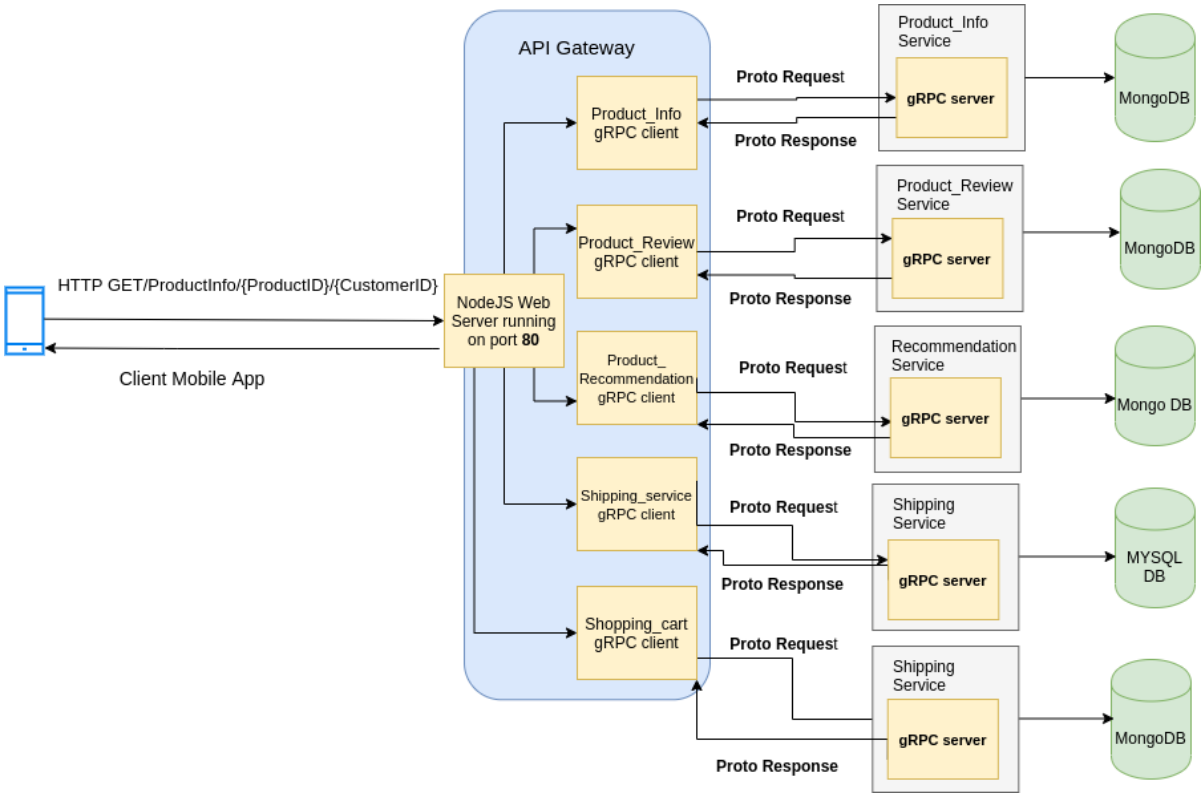

Figure 4.4: gRPC based IPC communication

[7]The default port for gRPC server is 50051

4.4 Asynchronous communication model setup

- RabbitMQ: For Asynchronous communication RabbitMQ has been selected as the message broker over AMQP protocol. In this setup unlike the Asynchronous method the API GateWay no longer have any direct communication with any of the microservices, but rather they communicate via an intermediary known as message broker. In this method the Gateway publishes a request containing the required parameters to the broker. Other microservices would pick that request and process it and place the result back to the broker. Once all microservices response are available, the Gateway will pick up the response.

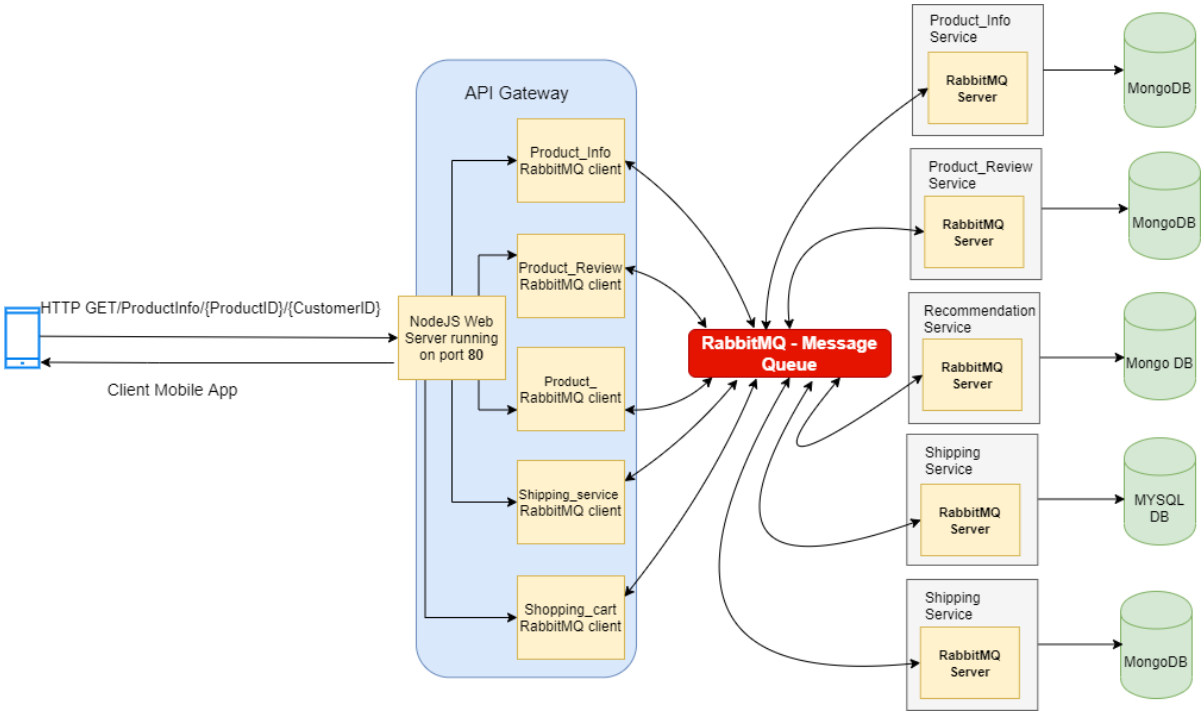

Figure 4.5: Asyncronous IPC communication using RabbitMQ

4.5 Performance Test Tool

In order to measure the **Latency**, **Scalability**, and **Availability** between different IPC methods, a load test has been executed using Apache JMeter[8]. JMeter allows to distribute the load by creating virtual users and simulate a real situation where a high number of requests from different origins arrives to the system.

[8] https://jmeter.apache.org/

Chapter 5

Results and evaluation

This first section of this chapter presents the quantitative data obtained through executing performance test against the proof of concept system that was built for this research while the latter section discusses qualitative data gathered throughout the process of developing the microservices

The quantitative data helps to answer research question related to **Performance Efficiency**, and **Availability** while the qualitative part aims to answer the remaining research questions that are related to **Scalability**, and **Complexity** of the methods.

5.1 Quantitative Results

5.1.1 Performance Efficiency

Three test cases have been designed and executed using Apache JMeter. All the test cases aim to measure the latency and throughput of each IPC method. As describe in the first chapter latency and throughput are the essential attributes for calculating performance efficiency. In all the three test experiments, the duration has been constant to 180 seconds, while the number of concurrent virtual users that continuously send requests to the system and wait for response has been varied. The first case ran with 50 concurrent virtual users, while the second case had 100, and the third case ran with 200 virtual users. The motive behind having test duration as a constant variable and number of virtual users as the controlled variable is to understand how each IPC method reacts differently when the concurrent requests and traffic to the system increase or decrease.

In every test case, *Throughput* is calculated by the total number of requests

and responses the method managed to make within the specified duration of 180 seconds; the higher the number of requests, the higher the Throughput and the better it is. In contrast, *Latency* is measured by the time each request needed to be processed. In this experiment the average response time is being evaluated. The average response time is given by summing the response time of each request and dividing it by the total number of requests the method has made. Lower average response time indicates lower Latency, which is an advantage for the particular method.

IPC Method	Test Duration	Virtual Users	Average Response Time	Total Request / Response
REST API	**180s**	**50**	**0.23 sec**	**4261**
gRPC	**180s**	**50**	**0.22 sec**	**4304**
RabbitMQ	**180s**	**50**	**0.27 sec**	**4157**

Table 5.1: 1st test case result

The result of 1st test case is presented in Table 5.1. The data indicates that gRPC has outperformed REST API, and RabbitMQ in the first case by being able to process 43 requests higher than REST API, and 104 requests more than RabbitMQ; this signifies that Synchronous form of communication can offer higher throughput than the Asynchronous method in the situation when the load to the system is relatively low. Meanwhile, the result of the first case also reveals that Synchronous form of communication can process requests slightly faster than Asynchronous form and therefore has lower latency when the number of concurrent threads[1] in the system is low.

IPC Method	Test Duration	Virtual Users	Average Response Time	Total Request / Response
REST API	**180s**	**100**	**3.5 sec**	**4373**
gRPC	**180s**	**100**	**3.3 sec**	**4453**
RabbitMQ	**180s**	**100**	**3.5 sec**	**4398**

Table 5.2: 2nd test case result

The 2nd case has double the number of virtual users as compared to the 1st one. Increasing the number of virtual users causes the number of parallel requests in the system to grow and results in longer processing time. The same data imply that gRPC has the highest throughput by processing a higher

[1] each virtual user occupies one thread in the system.

number of requests compared to RabbitMQ and REST API. In this test, gRPC managed to score the best average response time than REST API and RabbitMQ by 200 milliseconds. In this round, the processing time between REST API and RabbitMQ are equal to each other; however, RabbitMQ managed to process extra 25 requests than its Synchronous rival.

IPC Method	**Test Duration**	**Virtual Users**	**Average Response Time**	**Total Request / Response**
Rest API	**180s**	**200**	**7.8 sec**	**4334**
gRPC	**180s**	**200**	**7.2 sec**	**4348**
RabbitMQ	**180s**	**200**	**6.4 sec**	**4480**

Table 5.3: 3rd test case result

The number of virtual users in the third case has increased four times as compared to the 1st case. The outcome of the 3rd testing experiment implies considerable difference between Synchronous versus Asynchronous form of communication both in Throughput and Latency when the number of parallel requests increases. In this test, Asynchronous form of communication using RabbitMQ has outperformed the other two methods by being able to process a total of 4480 requests within the given period while gRPC managed to process 132 requests lower than RabbitMQ, and REST API processed 146 less requests than RabbitMQ.
Further, the data shows a significant latency gap between RabbitMQ and the two Synchronous methods. In this test case, the REST API's average response time took longer than RabbitMQ by 22 percent, and gRPC is higher than by 13 percent. This data is crucial for recognizing how different the performance gap can be between Synchronous versus Asynchronous IPC methods when the system is under high load.

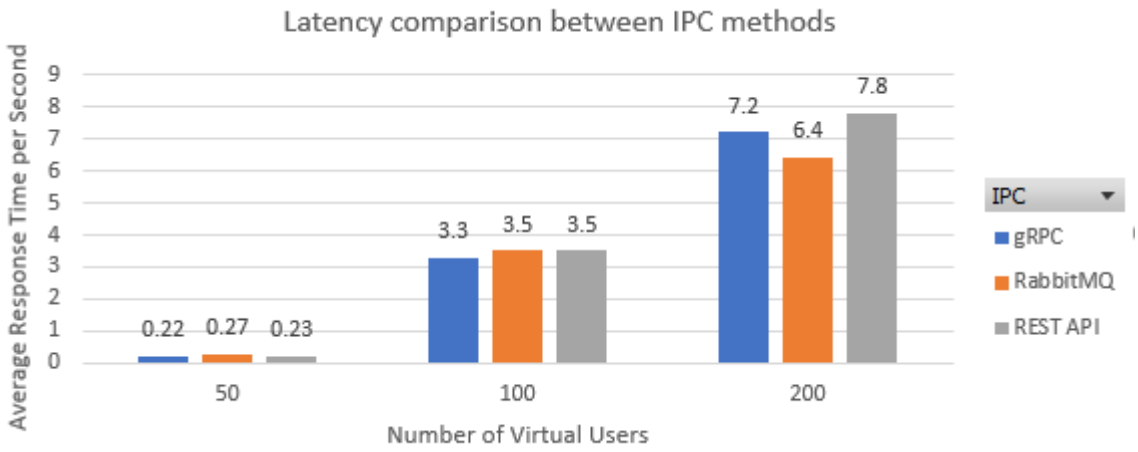

Figure 5.1: Asynchronous pattern using RabbitMQ has an advantage against its synchronous rivals for maintaining lower response time during high concurrent load.

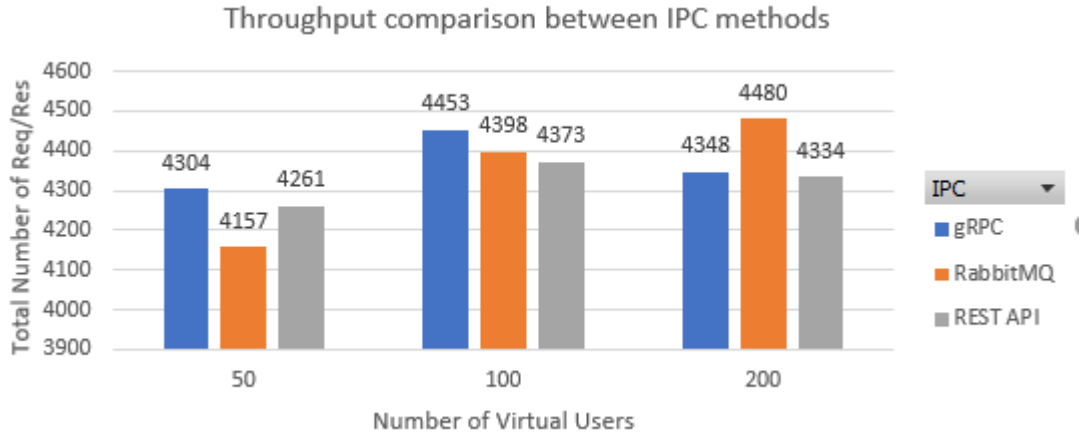

Figure 5.2: Asynchronous pattern demonstrated higher throughput as compared to its synchronous rivals when the concurrent load is high.

5.1.2 Availability

There are variety of parameters that can affect availability of a system - even hardware components can play a role in determining the availability rate of

a system. For this measurement, all the parameters outside IPC has been ignored. The availability of each IPC method has been calculated by using the following formula[11]:

$$Availability = \frac{MTTF}{MTTF + MTTR}$$

MTTF stands for "Mean Time to Failure," and MTR stands for "Mean Time to Recovery." MTTF represents the duration that the system is expected to last in operation before failure occurs. In contrast, MTR represents the duration the system requires to return to operation after a failure has occurred. The higher the MTR means, the longer it takes for the system to recover from a failure, which consequently reduces the availability of the system. Based on this formula, another test case was executed using Apache Jmeter against all the three different IPC methods to discover which one offers higher Availability. Unlike the previous test case that had a fixed duration for all the three methods, this test case had no specific duration. The test ran as long as the services became unavailable due to the high number of requests. Further, in this test, the average response time, nor the number of requests responses has not been taken into account. Further, this test was run with 200 virtual users, which is to generate a high number of parallel requests to the system. Without having this number of parallel users measuring Availability becomes more challenging as the system remains operational for a significantly longer duration.

IPC Method	Virtual Users	MTTF (second)	MTTR (second)	Availability
RabbitMQ	200	∼430	∼29	**0.95**
gRPC	200	∼290	∼21	**0.93**
REST API	200	∼275	∼19	**0.93**

Table 5.4: Measuring availability difference between IPC methods

Table 5.4 provides a summary of the test conducted for measuring Availability. During this test, it took about seven minutes for the services to become unavailable using RabbitMQ, while gRPC went down after about five minutes, and REST API took approximately four and half minutes. After the services became unavailable, the Kubernetes cluster[2] that was responsible for hosting services has been manually restarted. From that moment, both gRPC and REST API took about 20 seconds only to become available again, while RabbitMQ took ten extra seconds to become available. The main reason behind

[2]Details about Kubernetes cluster is available at chapter 4 of this document.

RabbitMQ taking longer than Synchronous form to return back to operation is the fact that it has an extra component known as a message broker that requires to be refreshed and establish a new connection with each service.From this experiment, it is possible to infer that an Asynchronous approach using RabbitMQ offers higher Availability than its Synchronous opponents.
Consequently, When microservices uses a Synchronous-based communication either REST API or gRPC both client and server must be responsive at all time, otherwise the request will fail after a specific duration depending on the configuration. In contrast to that, using Asynchronous based communication, since the consumer is loosely coupled with the server, and they do not interact with each other directly, a temporary outage of the server has minimal to no impact to the consumer. The requests can stay in the message queue and be processed at the later timing when the server is back to operation. Asynchronous pattern offers capabilities that can help the system to improve its Availability and resiliency from outage. It allows continuous operation even when there is a failure in one of the system's components without compromising the Availability of the entire system[48].

5.2 Qualitative Result

5.2.1 Scalability

Scalability is often related to how resource utilization increases as load to the system grows. It also refers to how easy it is for the system to increase its capacity by adding additional resources. Scalability is not a quantifiable attribute of software systems. Having said that, after undertaking the performance tests in the earlier section, it is possible to infer that Asynchronous methods offer higher scalability out of the box as compared to the Synchronous mode of communication. Figure 5.3 demonstrates how the response time starts to change between different methods as the load in the system increases. The latency growth in Asynchronous mode using RabbitMQ is more gradual while gRPC and REST API have more uneven latency as a result of high load in the system. It is possible to infer that Asynchronous mode is able to maintain the performance better than Synchronous mode as the number of requests increases. Further, out of the box, RabbitMQ offers advanced features for load balancing and scaling message queue as the number of messages increases.

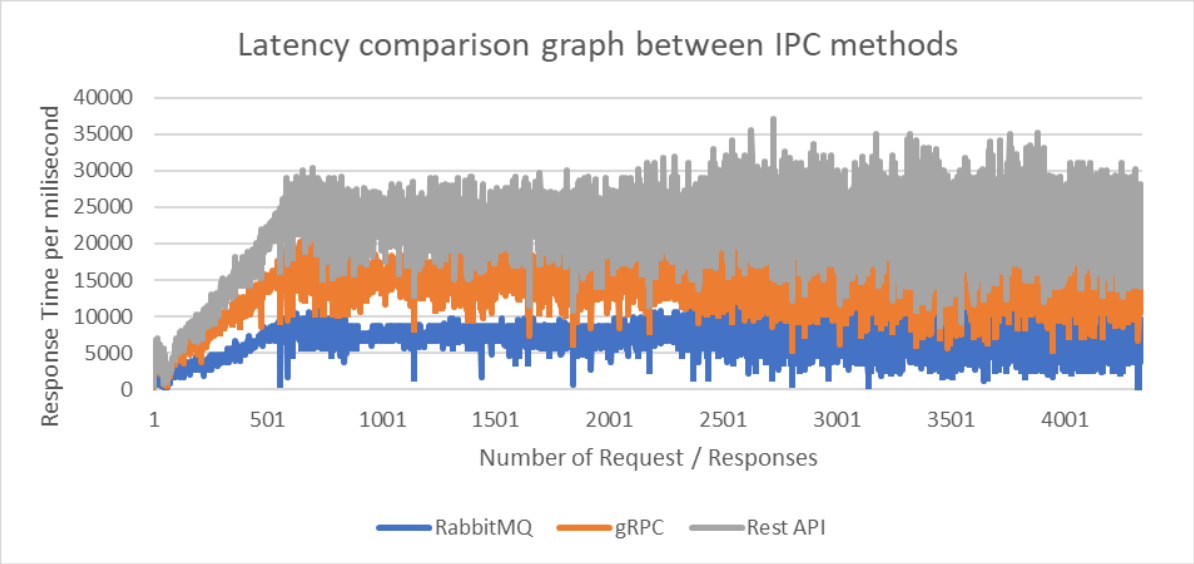

Figure 5.3: Difference in Latency growth across different IPC methods.

In the Asynchronous mode, all the requests go through a central coordinator known as Message Queue. The message Queue in RabbitMQ can be configured to run as a cluster with multiple nodes. This enables architecture enables the system to scale in order to increase the capacity and meet the increased demand efficiently. In contrast, in Synchronous form of communication, because the services are chained to each other, the only way to scale is to increase the capacity of all involved services which makes the scalability to be more difficult and less efficient.

5.2.2 Complexity

The process of implementing the proof of concept application has helped to measure the complexity of the IPC methods.

Complexity of each IPC method has been assessed through the following metrics:

- Lines of Code (LOC), and Function Points (FP): By measuring the number of code lines, and the number of function declaration each IPC method requires.
- Software Testability: This is measured by ease of testing and debugging the microservices when using one of the particular IPC.

IPC Communication based on REST API has the lowest Lines of Code and Function Point as compared to the other IPC methods. Unlike gRPC that requires schema on both Client and Server, REST API is schema-less which reduces LOC and number of files in each microservices. REST API based communication stands out for its simplicity in testing and debugging as compared to the other available methods. In REST API based communication, each microservice has an endpoint that can be trigger by providing the right HTTP verbs such as GET, or POST from other microservices or even from web browsers which makes testing and debugging of REST API more straight forward than the other two methods.
Moreover, unlike gRPC that uses Protocol Buffer as the default data format, REST API communicates using JSON by default which is human-readable without any extra decoding. In gRPC, testing and debugging can be more complicated as there are more variables involved such as the connection establishment between client and server. In RabbitMQ the debugging can even be more complicated as there is no direct communication between API Gateway and any microservices, but rather through a message queue that acts as a coordinator between the two services. All services must establish a connection with the broker in order to be able to send or receive message from the broker. Due to this architectural difference, there are more elements involved when testing and debugging microservices that are using message broker such as RabbitMQ.

Chapter 6

Discussion and Future work

6.1 Discussion

In addition to the four quality attributes that have been evaluated in the previous chapter, it is critically important to take into account the functional requirements for which microservices are being developed for. It is essential to distinguish whether the scenario requires an immediate response back from services or not. To elaborate further on this, in the proof of concept scenario that was built during this thesis work, displaying a product page for an e-commerce was simulated. In this scenario, the client sends a request to load a product page and expects an immediate result back. The result of the request can either be successfully displaying the product page or displaying an error to the user as the request was failed due to some reasons. The critical point in this scenario is that the client expects an immediate result back. In scenarios similar to loading product page, Synchronous form of communication is more suitable as these situations can not take advantage of the features that an Asynchronous form can offer.

However, not every scenario has such a requirement. Taking the check-out process of an e-commerce as an example. During the check-out process, there can be several services involves, such as payment process service, inventory management service, customer notification service, and seller notification service. When a customer completes a Check-out, the ideal flow would be to display a message that informs the customer his/her order has been received without having to wait from every back-end services to be processed completely. Once the order is submitted, each microservice can take that request and begin to process it accordingly. In this scenario, using an Asynchronous message broker is far more ideal than making a Synchronous call.

Meanwhile, the impact of IPC method on the security of the system is one of the domain that has not been discussed so far in this thesis mainly due to the fact that security of the IPC method has not been a priority for this research. However during the research, the author has came across few points related to security that are worth mentioning. The security features that each IPC method offers are differ from each other mostly due to the underlying protocol they use to transfer message. Asynchronous form of communication using RabbitMQ is capable of offering a robust authorization mechanism with access control out of the box while achieving a similar security feature using REST API requires further implementation. Meanwhile, gRPC uses HTTP/2.0 as the underlying protocol which offers additional security features such as TLS communication that are not available by default in REST API which uses HTTP/1.1 as its protocol. Furthermore, adding extra security mechanism to REST API based communication can lead to a decline in communication performance which does not occur in gRPC communication as gRPC takes advantage of multiplexed stream function and header compression that exists in HTTP/2.0[33]. In regards to data serialization format, it is possible to imply that it is more practical to follow the default message format of each IPC method. Therefore, when the system uses REST API or RabbitMQ it is best to have JSON as the data serialization format, and when the system uses gRPC, it is best to have protocol buffer as the data format. Substituting data formats can lead to more complexity and technical debt while the value it offers can often be limited.

6.2 Future Work

The result that has been obtained through the analysis and development of this thesis work has inspired further potential to expand the research topic. following are some of the potential areas to extend this research topic:

- Reliability of IPC methods: This thesis work did not measure the error rate between different IPC methods which can be helpful to calculate the difference between IPC methods from Reliability perspective. This, however, could be another important factor for many scenarios and therefore worth looking into. Further test cases with different set of loads are required in order to draw a conclusion in regards to error rate and reliability between IPC methods.

- Developing a more complex mock-system: In this thesis, displaying a product page of an e-commerce system was assumed. In this scenario,

five microservices were involved, and all of them only communicates with the API Gateway. Developing a more complex system where more microservices are involved while some communicates with each other without going through the API Gateway could potentially reveal new insight related to different IPC mechanism and their efficiencies.

- Maintainability of the IPC method: For this thesis work the complexity has been calculated by looking at Lines of Code, Function Points, as well as their ease of testing. It is worthwhile to take maintainability into account when measuring the complexity of the IPC method. Maintainability refers to the ease with which a software system as a whole or one of its components can be modified for performance improvement or bug fixes or other attributes such as adapting to changed environment[49][50].

- Evaluating different IPC in production environment: For the thesis work, the measurement and load testing has been carried out in a test system. Being able to measure IPC performance in a real production environment with real traffic instead of using Apache Jmeter to mock users and traffic can potentially reveal new insights about each IPC method.

- Measuring the effect of programming language and framework on IPC performance: For this thesis work, all the microservices for was written using NodeJS. Having an identical set of microservices from a functionality perspective but written with different languages such as Java, or Python can provide some insight into the impact of programming language, and its framework on microservices performance.

Chapter 7

Conclusion

This work argued that as of today, Interprocess communication is one of the major challenges with microservices architecture that can affect the system from different non-functional requirements. Based on this problem the following research question was constructed:

How does the choice of IPC method impact the non-functional requirements of a microservices-based system?

During the first chapter, the main research question was then broken down to a sub-question each aims at answering the impact of IPC on *Performance efficiency*, *Availability*, *Scalability*, and *Complexity*; a motive for selecting these four non-functional requirements over others were justified as well. Further, a set of microservices for an e-commerce scenario using the most common type of IPC methods that exist today has been designed and developer in order to assist in answering the research question. Extensive testing has been executed against each method to prove the choice of IPC can influence the non-functional requirements of the system. The evaluation offers solid reasoning to recommend the Asynchronous form of communication has an advantage over Synchronous form as it offers higher Performance efficiency, Availability, and Scalability even though it increases Complexity of the code and requires more development effort. In the discussion section of the sixth chapter, further guidelines on how to decide between Asynchronous versus Synchronous form for different scenarios have been discussed. In conclusion, the choice of the IPC method has to be carefully thought of as it plays a crucial role in a microservices architecture. There are scenarios where one method of communication is more suitable than the other one. Therefore, in an ideal system,

both Synchronous and Asynchronous type has to be adopted according to the functional and non-functional requirements of the specific components.

Bibliography

[1] Rajkumar Buyya. “Cloud computing: The next revolution in information technology”. In: *2010 First International Conference On Parallel, Distributed and Grid Computing (PDGC 2010)*. IEEE. 2010, pp. 2–3.

[2] Aaron Engelsrud. “Moving to the Cloud: Lift and Shift”. In: *Managing PeopleSoft on the Oracle Cloud: Best Practices with PeopleSoft Cloud Manager*. Berkeley, CA: Apress, 2019, pp. 229–242. ISBN: 978-1-4842-4546-0. DOI: 10.1007/978-1-4842-4546-0_9. URL: https://doi.org/10.1007/978-1-4842-4546-0_9.

[3] Shmuel Tyszberowicz et al. “Identifying Microservices Using Functional Decomposition”. In: *Dependable Software Engineering. Theories, Tools, and Applications*. Ed. by Xinyu Feng, Markus Müller-Olm, and Zijiang Yang. Cham: Springer International Publishing, 2018, pp. 50–65. ISBN: 978-3-319-99933-3.

[4] Claus Pahl and Pooyan Jamshidi. “Microservices: A Systematic Mapping Study.” In: *CLOSER (1)*. 2016, pp. 137–146.

[5] Björn Butzin, Frank Golatowski, and Dirk Timmermann. “Microservices approach for the internet of things”. In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2016, pp. 1–6.

[6] Wilhelm Hasselbring. “Microservices for Scalability: Keynote Talk Abstract”. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE ’16. Delft, The Netherlands: Association for Computing Machinery, 2016, pp. 133–134. ISBN: 9781450340809. DOI: 10.1145/2851553.2858659. URL: https://doi.org/10.1145/2851553.2858659.

[7] Xiang Zhou et al. “Poster: Benchmarking microservice systems for software engineering research”. In: *2018 IEEE/ACM 40th International*

Conference on Software Engineering: Companion (ICSE-Companion). IEEE. 2018, pp. 323–324.

[8] Chris Richardson. "Microservices patterns: with examples in Java". In: *Microservices patterns: with examples in Java*. Shelter Island, NY: Manning Publications, 2019. Chap. 3, pp. 65–100.

[9] Rick Kazman, Len Bass, and Paul Clements. *Software Architecture in Practice, Second Edition*. eng. 2nd ed. Addison-Wesley Professional, 2003. ISBN: 0321154959.

[10] Robert Lagerström, Pontus Johnson, and David Höök. "Architecture analysis of enterprise systems modifiability–models, analysis, and validation". In: *Journal of Systems and Software* 83.8 (2010), pp. 1387–1403.

[11] Pontus Johnson et al. "It management with enterprise architecture". In: *KTH, Stockholm* (2014).

[12] Ulrik Franke et al. "Trends in enterprise architecture practice–a survey". In: *International Workshop on Trends in Enterprise Architecture Research*. Springer. 2010, pp. 16–29.

[13] Ken Peffers et al. "A design science research methodology for information systems research". In: *Journal of management information systems* 24.3 (2007), pp. 45–77.

[14] David Garlan and Mary Shaw. "An introduction to software architecture". In: *Advances in software engineering and knowledge engineering*. World Scientific, 1993, pp. 1–39.

[15] Martin Fowler. "Who needs an architect?" In: *IEEE Software* 20.5 (2003), pp. 11–13.

[16] Winfried Lamersdorf. "Paradigms of Distributed Software Systems: Services, Processes and Self-organization". In: *International Conference on E-Business and Telecommunications*. Springer. 2011, pp. 33–40.

[17] Sophia Alami-Kamouri, Ghizlane Orhanou, and Said Elhajji. "Overview of mobile agents and security". In: *2016 International Conference on Engineering & MIS (ICEMIS)*. IEEE. 2016, pp. 1–5.

[18] Tasneem Salah et al. "The evolution of distributed systems towards microservices architecture". In: *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE. 2016, pp. 318–325.

[19] Thomas Erl. *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 1900.

[20] Don Box et al. *Simple object access protocol (SOAP) 1.1*. 2000.

[21] Mike Amundsen. *Microservice architecture*. OReilly Media, Inc, Usa, 2016.

[22] Gerald Schermann, Jürgen Cito, and Philipp Leitner. "All the services large and micro: Revisiting industrial practice in services computing". In: *International Conference on Service-Oriented Computing*. Springer. 2015, pp. 36–47.

[23] Genc Mazlami, Jürgen Cito, and Philipp Leitner. "Extraction of microservices from monolithic software architectures". In: *2017 IEEE International Conference on Web Services (ICWS)*. IEEE. 2017, pp. 524–531.

[24] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[25] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[26] Nicola Dragoni et al. "Microservices: How To Make Your Application Scale". In: *Perspectives of System Informatics*. Ed. by Alexander K. Petrenko and Andrei Voronkov. Cham: Springer International Publishing, 2018, pp. 95–104. ISBN: 978-3-319-74313-4.

[27] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices". In: *IEEE Access* 5 (2017), pp. 3909–3943.

[28] David Jaramillo, Duy V Nguyen, and Robert Smart. "Leveraging microservices architecture by using Docker technology". In: *SoutheastCon 2016*. IEEE. 2016, pp. 1–5.

[29] Sascha Alpers et al. "Microservice based tool support for business process modelling". In: *2015 IEEE 19th International Enterprise Distributed Object Computing Workshop*. IEEE. 2015, pp. 71–78.

[30] Kapil Bakshi. "Microservices-based software architecture and approaches". In: *2017 IEEE Aerospace Conference*. IEEE. 2017, pp. 1–8.

[31] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine, 2000.

[32] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[33] Sang Gyun Du, Jong Won Lee, and Keecheon Kim. "Proposal of GRPC as a New Northbound API for Application Layer Communication Efficiency in SDN". In: *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication*. 2018, pp. 1–6.

[34] Andrew D Birrell and Bruce Jay Nelson. "Implementing remote procedure calls". In: *ACM Transactions on Computer Systems (TOCS)* 2.1 (1984), pp. 39–59.

[35] Jong-Kun Lee. "A group management system analysis of GRPC protocol for distributed network management systems". In: *SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218)*. Vol. 3. IEEE. 1998, pp. 2507–2512.

[36] "Remote Procedure Call". In: *The JR Programming Language: Concurrent Programming in an Extended Java*. Boston, MA: Springer US, 2004, pp. 91–105. ISBN: 978-1-4020-8086-9. DOI: 10.1007/1-4020-8086-7_8. URL: https://doi.org/10.1007/1-4020-8086-7_8.

[37] Gurpreet Kaur and Mohammad Muztaba Fuad. "An evaluation of protocol buffer". In: *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*. IEEE. 2010, pp. 459–462.

[38] Nurzhan Nurseitov et al. "Comparison of JSON and XML data interchange formats: a case study." In: *Caine* 9 (2009), pp. 157–162.

[39] Gustav Johansson. "Investigating differences in response time and error rate between a monolithic and a microservice based architecture". MA thesis. KTH, School of Electrical Engineering and Computer Science (EECS), 2019, p. 13.

[40] Nicola Dragoni et al. "Microservices: Migration of a mission critical system". In: *arXiv preprint arXiv:1704.04173* (2017).

[41] Joel L Fernandes et al. "Performance evaluation of RESTful web services and AMQP protocol". In: *2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE. 2013, pp. 810–815.

[42] Steve Vinoski. "Advanced message queuing protocol". In: *IEEE Internet Computing* 10.6 (2006), pp. 87–89.

[43] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. "Workload characterization for microservices". In: *2016 IEEE international symposium on workload characterization (IISWC)*. IEEE. 2016, pp. 1–10.

[44] Diego Kreutz et al. "Software-defined networking: A comprehensive survey". In: *Proceedings of the IEEE* 103.1 (2014), pp. 14–76.

[45] Audie Sumaray and S. Kami Makki. "A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform". In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. ICUIMC '12. Kuala Lumpur, Malaysia: Association for Computing Machinery, 2012. ISBN: 9781450311724. DOI: 10.1145/2184751.2184810. URL: https://doi.org/10.1145/2184751.2184810.

[46] Stefan Tilkov and Steve Vinoski. "Node. js: Using JavaScript to build high-performance network programs". In: *IEEE Internet Computing* 14.6 (2010), pp. 80–83.

[47] Jürgen Cito et al. "An empirical analysis of the docker container ecosystem on github". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 323–333.

[48] Tobias Johansson. "Message-Oriented Middleware as a Queue Management Solution to Improve Job Handling within an E-Commerce System". MA thesis. KTH, School of Electrical Engineering and Computer Science (EECS), 2018, p. 91.

[49] M. Ekstedt et al. "A Tool for Enterprise Architecture Analysis of Maintainability". In: *2009 13th European Conference on Software Maintenance and Reengineering*. 2009, pp. 327–328.

[50] R. Lagerstrom and P. Johnson. "Using Architectural Models to Predict the Maintainability of Enterprise Systems". In: *2008 12th European Conference on Software Maintenance and Reengineering*. 2008, pp. 248–252.

Appendix A

API Payloads and Test Graphs

Below is a sample of response from API Gateway to the clinet:

GET http://23.14.155.200/productId/1234

arams Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON


```
{
    "TotalDuration": "34243.324",
    "ProductInfo_Response":{
        "product_id": 1234
        "product_name": "Professional Grade 1700W Steam Iron for Clothes with Rapid Even Heat Scratch Resistant Stainless Steel Sole Plate",
        "price": 17.28,
        "category": "HomeAppliences",
        "Colors": ["Red", "Blue", "White"],
        "Product_Image": ["http://example.com/image1","http://example.com/image2","http://example.com/image3"],
        "Call_Duration": 242.43
    },
    "ProductRecommendation_Response":{
        [{"product_id": 54343,
        "product_name": "Honey-Can-Do Tabletop Ironing Board with Retractable Iron Rest Honey-Can-Do Tabletop Ironing Board with Retractable Iron
        {
            "product_id": 45678,
        "product_name": "BLACK+DECKER Classic Steam Iron, F67E"
        }],
        "Call_Duration": 432.23
    },
    "ProductReview_Response":{
        [{"Review_Id": "T4564",
        "Review_Title": "This item is great!"},
        {
        "Review_Id": "T65664",
        "Review_Title": "I am happy with this purchase!"}]
        "Call_Duration": 543.23
    },
    "ProductShipping_Response":{
        "Call_Duration": 590.25
        [
            {"Shipping_ID": 2342,
            "ShippingCost": 6,
            "Type": "Standard"
            },
            {"Shipping_ID": 2342,
            "ShippingCost": 9,
            "Type": "ex[ress"
        }
        ]
    }
    "CustomerShoppingCart_Response":[{
        "itemId": "5645456",
```


Figure A.1: A Sample response from API Gateway - The Gateway is responsible for aggregating all the responses from all the microservices.

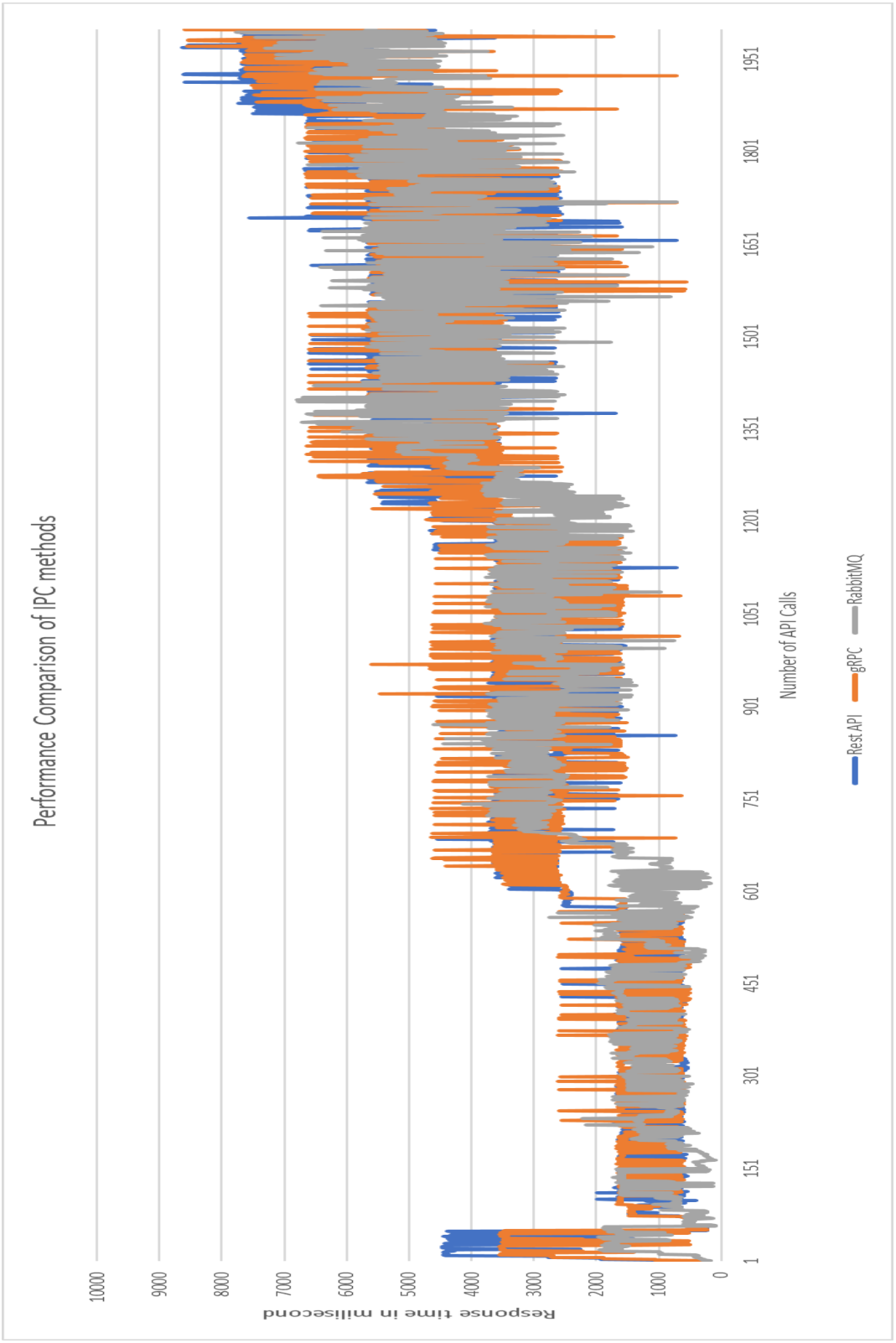

Figure A.2: Graph demonstrating the latency difference between the three IPC methods

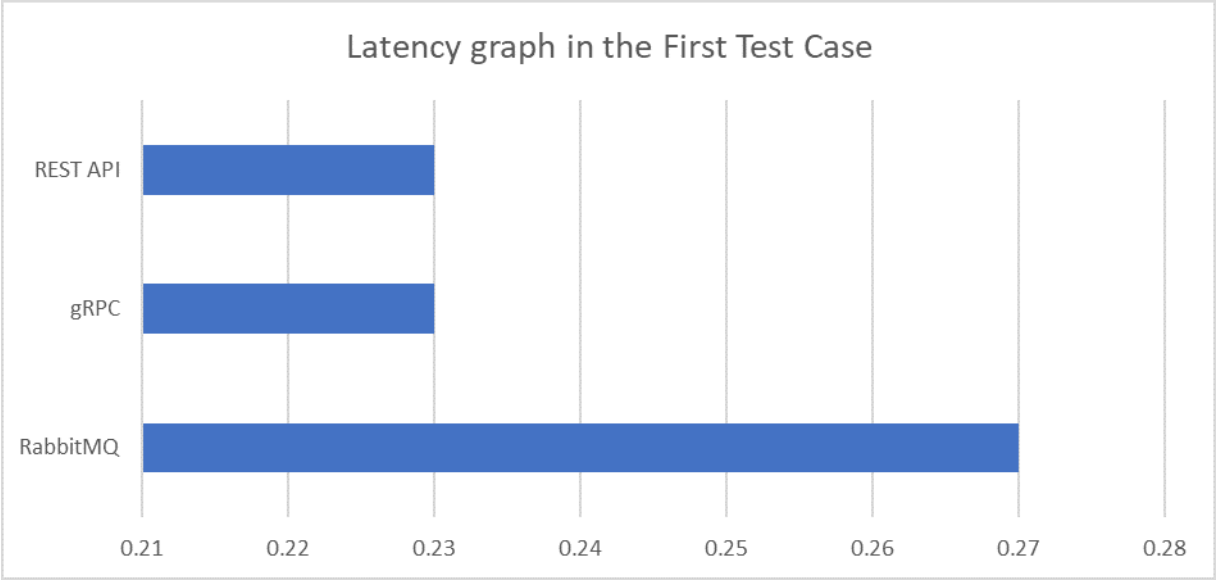

Figure A.3: Latency graph generated from the 1st test case

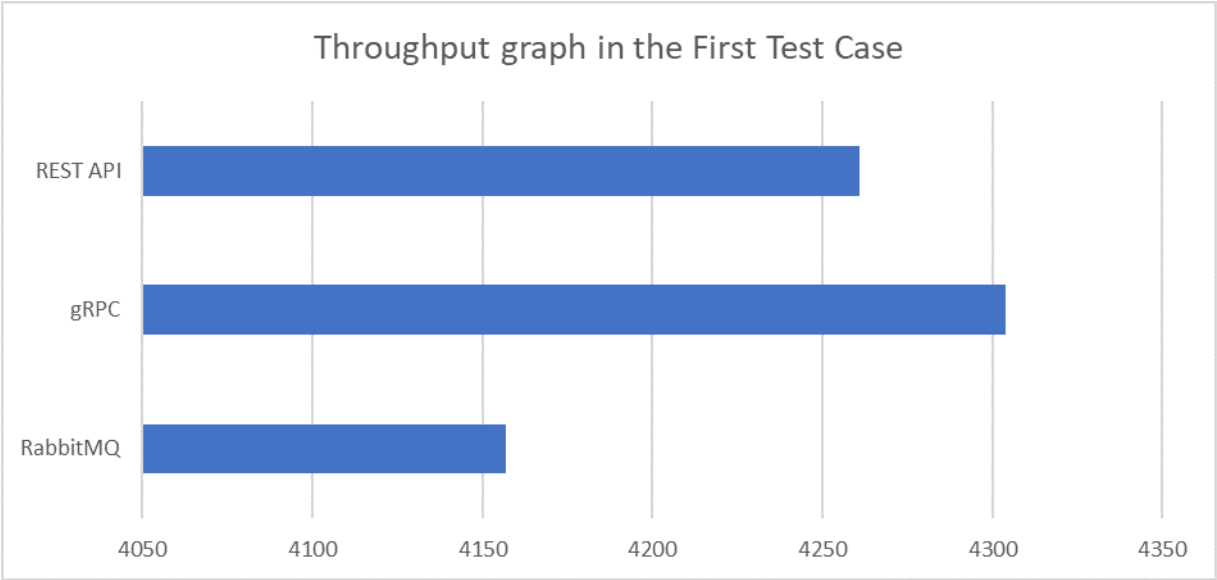

Figure A.4: Throughput graph generated from the 1st test case

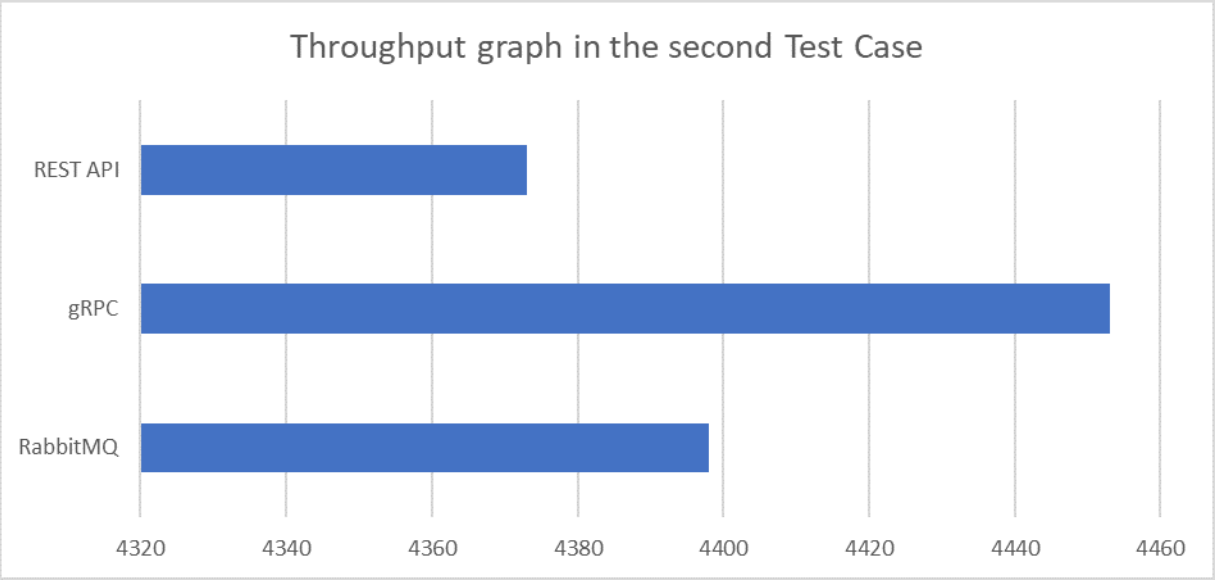

Figure A.5: Throughput graph generated from the 2nd test case

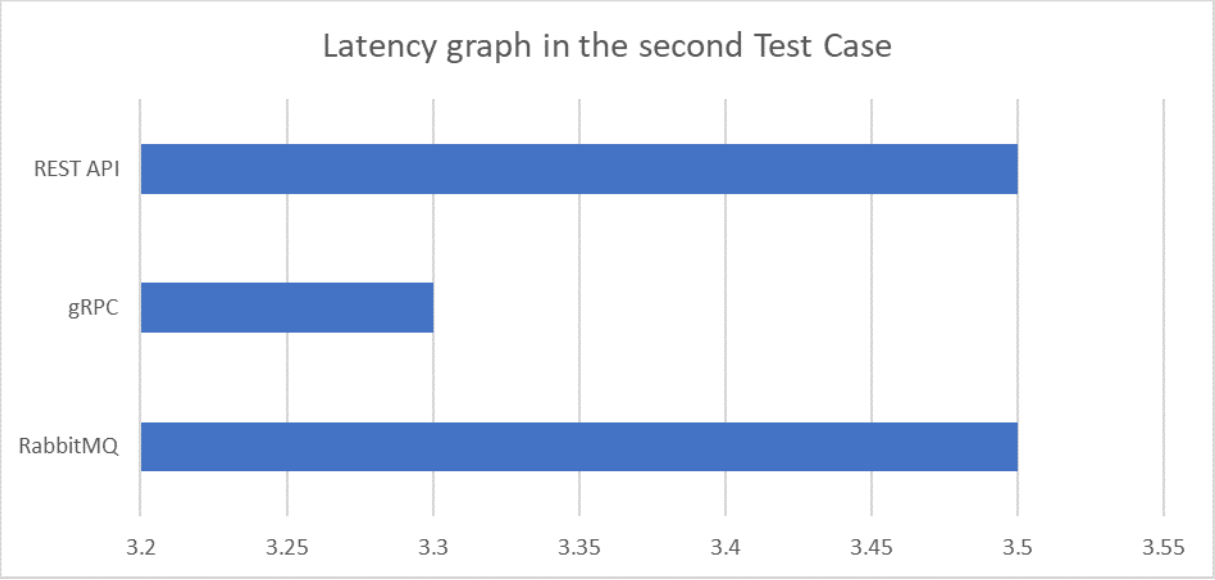

Figure A.6: Latency graph generated from the 2nd test case

TRITA-EECS-EX-2020:217