Effectivity of message brokers in communication between microservices

A THESIS SUBMITTED TO
THE FACULTY OF ARCHITECTURE AND ENGINEERING
OF
EPOKA UNIVERSITY

BY

GERILD PJETRI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF BACHELOR OF SCIENCE
IN
COMPUTER ENGINEERING

JUNE 2019

Approval Sheet
Given Separately by the Department

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Gerild Pjetri

Signature:

ABSTRACT

EFFECTIVITY OF MESSAGE BROKERS IN COMMUNICATION BETWEEN MICROSERVICES

Pjetri, Gerild
Ba.Sc., Department of Computer Engineering
Supervisor: Prof.Dr. Dimitri A.Karras

In software world new techniques,methodologies and solutions emerge in an extremely rapid rate with respect to our reasonable capabilities of knowing and mastering at least the most popular ones.

Having the need to academically examine such techniques empirically not subjectively is a must for engineers who desire to provide the business with undisputable results and consequently convince the real world on strength and weakness of the popular methodologies which they are about to use.

The Microservice Architecture is an emerging architectural pattern that has attracted much attention in the industry for architecting backend systems.

Various articles and materials are written about architecting and migrating to this trendy architecture, but as with anything else that becomes trendy in humans , developers and engineers grow a tendency in usage of the methodologies without knowing empirically the benefits that goes with the implementation.

In this thesis a study of effectivity in communication between microservices is treated aiming to clarify engineers or business analysts on the power of three major possible techniques, especially messaging technique used in communication between microservices.

Such study has taken place in local machines and must not be regarded as a final conclusion for business decisions but it offers a clear picture on performace metrics respectively throughput, processing time and response time of the requests thrown upon microservices.The analysis is obtained by examining three most common types of communication: Http Synchronous Rest, Http Asynchronous Rest and Kafka messaging.

. . .

Keywords: Microservice,Interservice-Communication,Throughput,Processing Time,Response Time,Apache Kafka,Effectivity, Message Broker,Performance

ABSTRAKT

EFEKTIVITETI I BROKERAVE TE MESAZHEVE NE KOMUNIKIMIN MIDIS MIKROSHERBIMEVE

Pjetri, Gerild
Bachelor Shkencor, Departamenti i Inxhinierisë Kompjuterike
Udhëheqësi: Prof.Dr. Dimitri A.Karras

Në botën e softuerit, teknikat, metodologjitë dhe zgjidhjet e reja dalin në një shkallë shumë të shpejtë në lidhje me aftësitë tona të arsyeshme të njohjes dhe të zotërimit të të paktën atyre më të njohura.

Duke pasur nevojë për të shqyrtuar akademikisht këto teknika empirikisht jo subjektivisht, është një domosdoshmëri për inxhinierët që dëshirojnë të sigurojnë biznes me rezultate të padiskutueshme dhe si rrjedhojë ta bindin botën e vërtetë për forcën dhe dobësinë e metodologjive popullore të cilat ata do të përdorin.

Arkitektura Microservice është një model arkitektonik në zhvillim që ka tërhequr shumë vëmendje në industrinë për sistemimin e sistemeve backend.

Artikuj dhe materiale të ndryshme janë shkruar rreth arkitekturës dhe emigrimit në këtë arkitekturë të modës, por si me çdo gjë tjetër që bëhet e modës në njerëz, zhvilluesit dhe inxhinierët rriten një tendencë në përdorimin e metodologjive pa e njohur në mënyrë empirike përfitimet që shkojnë me zbatimin.

Në këtë tezë është trajtuar një studim i efektivitetit në komunikimin midis mikroservices duke synuar sqarimin e inxhinierëve apo analistëve të biznesit në fuqinë e tre teknikave më të mëdha të mundshme, veçanërisht teknikën e mesazheve që përdoret në komunikimin ndërmjet shërbimeve mikro.
Një studim i tillë ka ndodhur në makinat lokale dhe nuk duhet të konsiderohet si një rezultat përfundimtar për vendimet e biznesit, por ofron një pasqyrë të qartë mbi performancën e matricave përkatësisht të xhiros, kohën e përpunimit dhe kohën e përgjigjes së kërkesave të hedhura në mikroservice. Analiza është marrë duke shqyrtuar tre llojet më të zakonshme të komunikimit: Http Synchronous Rest, Http Asynchronous Rest dhe Kafka mesazheve.

Fjalët kyçe: Mikroshërbim, InterKomunikim midis mikrosherbimeve, Koha e Procesimit, Apache Kafka, Efektiviteti, Brokerat e mesazheve, Performanca

Dedicated to My Family

ACKNOWLEDGEMENTS

I would like to express my special thanks to my supervisor Prof.Dr.Dimitrios A.Karras for his continuous guidance, encouragement, motivation and support during all the stages of my thesis. I sincerely appreciate the time and effort he has spent to improve my experience during my undergraduate final year.

I am also deeply thankful to my brother Erald Pjetri for giving me engineering insight on thesis completion.

I am especially grateful to my friend Arbi Elezi for his continuous advice on the type of technology which is used to fulfill the thesis's purpose.

I would like to thank my family for the unwavering support on my simple but meaningful research jorney.

Table of Contents

List of Figures

LIST OF ABBREVIATIONS

HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
API	**A**pplication **P**rogramming **I**nterface
REST	**Re**presentational **S**tate **T**ransfer
URI	**U**niform **R**esource **I**dentifier
TCP	**T**ransmission **C**ontrol **P**rotocol

CHAPTER 1

INTRODUCTION

1.1 Topic

In science and technology people strive after innovation,advancement on thinking patterns , advancement of design philosophies and improvement of current systems.Microservice Architecture is the trendy and most widely spoken backend architecture for the moment in Tech communities.Microservice are build accordingly representing business capabilities in order to provide a more efficient way in the structuring of the system. The need for these business capabilities to communicate with each other is obvious in most real world applications. Consequently inter-service communication between microservices is an important topic of the developer jorney to build reliable systems. There are three major strategies in communication between such services. Particularly Sunchronous Http Communication , Asynchronous Http Communication and Messaging Strategy.Although messaging technique is used for use cases which require event driven paradigm as primary basis of reasoning for building the system,the aim of the study conducted in this thesis is not to focus on what decision to take about the paradiigms of software development. The scope of the thesis is to give a simple and meaningful study on effectivity regarding communication betwen services and to show espacially how effective is the message broker technique with respect to other techniques on the basis of empirical data generated by multiple tests on both quality and quantity level.The services will be build just to be able to conduct

the experiments and to provide the necessary results which will be used to draw the conclusions. The focus of the study is to generate reliable data which can extrapolate quickly the differences between the techniques.

1.2 Problem Description

There is an infinite number of use cases in real software world. Many approaches and different styles of requirements can show up each time an engineer wants to build microservices. The problem is not with the potential technology he/she can use but how much is he/she aware of their effectivity regarding empirical undisputable data. On such conditions inter-service communication, when the uses case of software development requires it , must be a major topic regarding the choice of the technique which has to be implemented . In order to make an effective decision,an effective study must be held on each technique .This can happen only if the conclusions are obtained based on reliable and meaningful performance metrics. To make possible such study an examination of the possible strategies will take place on the thesis.

1.3 Motivation

This project will contribute to the overall understanding of inter-service communication techniques by emphasizing the importance of messaging technique as very effective among other types of communication.

It's objectives are :

- To make a thorough analysis on performance metrics chosen as a basis for reasoning the conclusions.
- Clarify the effectivity of a message broker in communication between microservices among other methods of communication.

- Metrics used to draw conclusions on research questions are common performance metrics such as throughput, response time and processing time.

1.4 Requirements

The conditions under which the study is to be held should take place in personal machines not on platform as service premises.

1.4.1 Functional Requirements

- The system should implement three kinds of inter-service communication
- Four independent projects should be build specifically one client service for synchronous communication , one client service for asynchronous communication , one server service for serving two client services and one project for building producer-consumer services together with kafka implementation.
- The testing should be performed on local machines with different input data.
- The system should not contain business services only services for study-purpose.

1.4.2 Non-Functional Requirements

- The system should not involve extra unnecessary line of codes that exced the scope of the study.
- Testing should be performed on optimal conditions of operating system. While the services are up a minimum of other resources in the computer should be running so as to minimize unnecessary overhead in the sytem.
- The testing should produce effective data on reasonable time.

1.5 Research Questions

This project will aim at answering the following research questions:

- What is the impact of a message broker in the communication between microservices?
- How much do the inter-service communication strategies differ between each other with respect to empirical data obtained by clear performance metric?
- How much beneficial is the study for business decision on the different use cases in real world?
- What is the impact of empirical data obtained by performance metric in offering knowledge on which technique of communication is better?

1.6 Planned Contribution

This project will contribute in the area of inter-service communication in Microservice Architecture. The study will provide insight on the effectivity of the usage of message broker in inter-service communication. Such study will give general but detailed results on performance of the techniques used in communication between service. In this project conclusions may be found helpful regarding decision making about which strategy to be selected for different real world use case. The study doesn't provide insight on which use case it might be helful but it provides clear data on effectivity of each technique used in communication between microservices.

CHAPTER 2

LITERATURE REVIEW

2.1 Microservice Architecture

The microservice architecture pattern is a widely spread architectural pattern and has attracted much attention in the industry of building backend systems. Multiple tech companies such as LinkedIn,Netflix,Amazon and SoundCloud had large, monolithic architectures which constricted their abilites in terms of scalability and extensibility. These monolithic where then divided into a set of smaller, independent service that communicate together. These small and independent peaces of software are referred as loosely coupled services with bounded context. Currently there is no consensus on a concrete definition of the microservice pattern itself.The pattern can be described as a way to develop a system consisting of small, independent services that are centered around business capabilites and run autonomously from each other. Despite being autonomous from one another , the services can still communicate in a synchronous or asynchronous fashion. Some key benefits of the microservice architecture comes from the independence nature of the services : each service can be developed ,tested and deployed independenlty.

Advantages of microservice architecture are:

- Modularity
- Service-Specific Database
- Fault Isolation
- Scalability
- Technology/Language Flexibility
- Freedom to independently develop and deploy services
- Small teams can develop a microservice
- Code for different services can be written in different languages
- Easy integration and automatic deployment
- Easy to understand and modify for developers .New team becoms more productive
- API can be versioned more effectively

Disadvantages of microservice architecture are:

- Existing tools are not designed to work with service dependencies.
- Hard maintenance
- As each service has its own database, transaction management and data consistency can become a complex.
- The initial refactoring of a monolithic application can be exceedingly complex for large enterprise applications.

- Number of processes can grow exponentially when load balancing and messaging middleware are considered.
- Increases in documentation overhead as organization has to keep schemas and interface documents up to date.
- Numerous microservices patterns exist, will have to determine which is the best fit for your application.

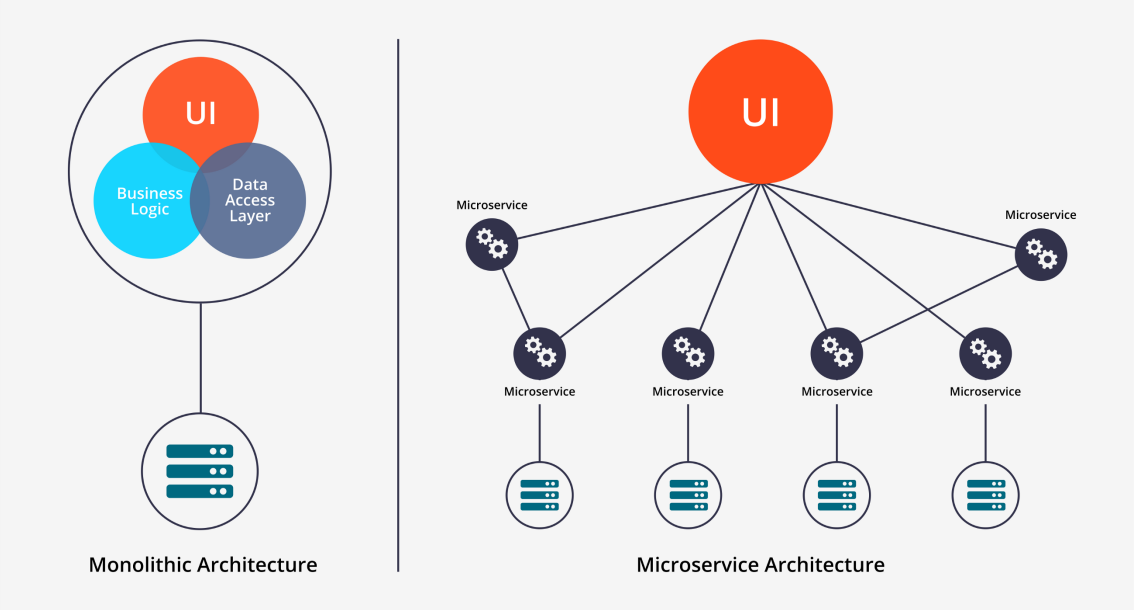


Figure 2.1: Conceptual Image On Microservice Architecture

2.2 Inter-Service Communication

There are various techniques for inter-service communication. Communication can be synchronous or asynchronous. Synchronous communication happens when the call from the service consumer is blocked until the operation of the requested task has finished execution and the result is given back. This is a common characteristic for RESTful HTTP APIs where the entire operation is finished once the service consumer receives the response (Request/Response).Compared to synchronous communica- tion, with asynchronous, the request is processed at a later point in time, outside the request-response procedure by the service consumer. The request may be put in a message

queue and later processed by a consumer service. The producer service may or may not receive any notification of the completion of the task. It is possible to integrate asynchronous communication using REST APIs (request/asyn- chronous response), where the worker node could perform a request back to the service consumer through a push mechanism, or by having the service consumer pull for updates.

Practically identification of two different styles of communication between microservices exist. It is possible to classify them into two dimensions. The first of them is a division into synchronous and asynchronous communication protocols. The essence of asynchronous communication is that the client should not have blocked a thread while waiting for a response. The well-known protocol for that type of communication is AMQP. While there can be asynchronous implementation of HTTP protocol in inter-service communication it is not preferred because it requires stateful request/response. The most common usage in inter-service communication is synchronous HTTP protocol. The thesis encompasses three types of communication and offers an examination of their effectivity with respect to predifined metrics.

2.3 Message Brokers

A message broker by definition is an intermediary software module that converts a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver.

A message broker is an architectural pattern for message validation, transformation, and routing. It sustains communication among applications, reducing the mutual awareness that applications should have of one another in order to be able to exchange messages optimally by implementing decoupling.

The primary purpose of a broker is to receive incoming messages from applications and apply some action on them. Message brokers can decouple end-points, coordinate non-functional requirements, and facilitate reuse of intermediary functions.

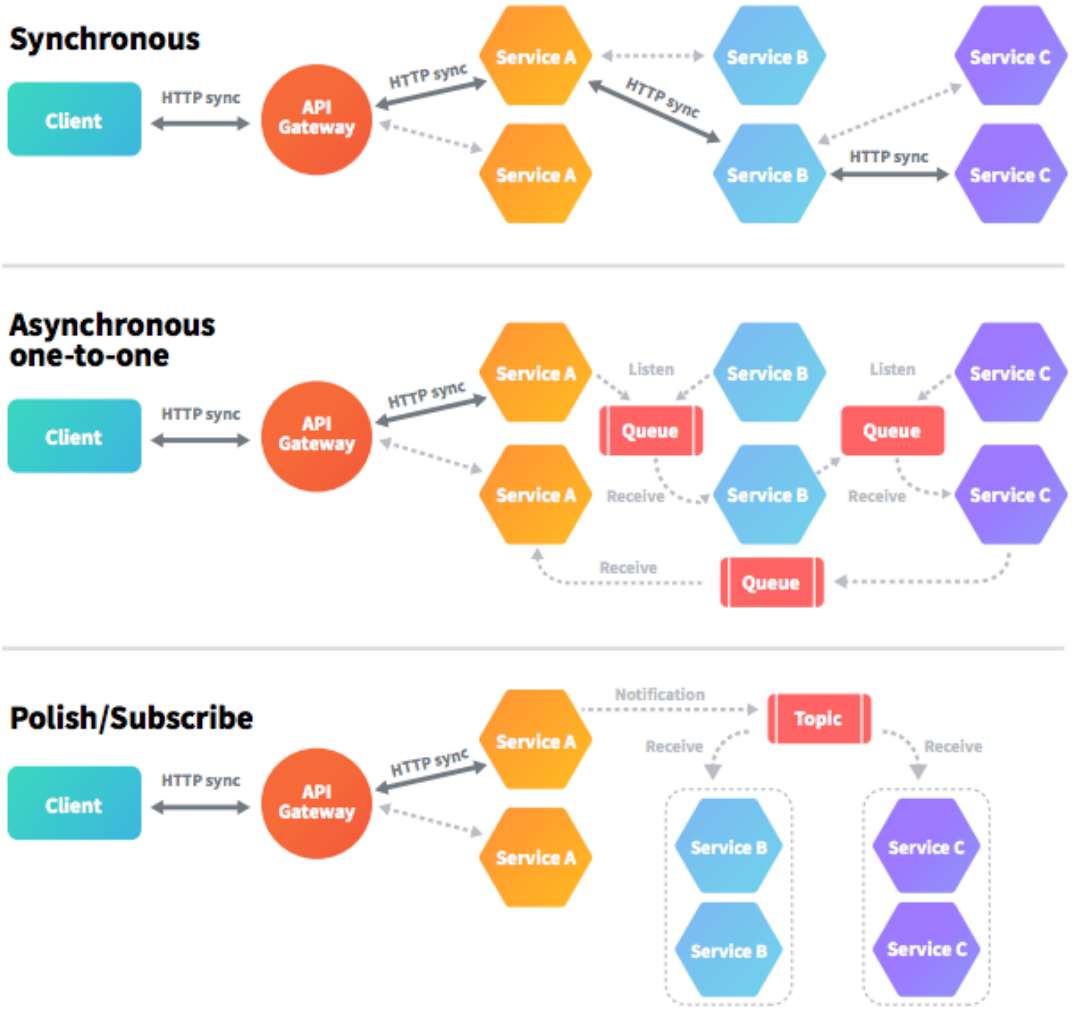


Figure 2.2: Inter-Service Communication In Microservice Architecture

In kafka architecture the following components are involved:

Producers: are the primary data contributors that produce messages and push them into the message queue so that data consumers can later retrieve them. Producers directly communicate with one of the brokers in the queuing system and receive information about message partitioning and split forwarding data and retaining them to particular nodes accordingly within the queueing cloud. When retaining data, a topic must be settled and the consumers retrieve all data within that topic.

Brokers: are primary storage entity that composes the entire queueing network. They receive data sent from data producers, store them then dispatch them when consumers make requests. In a traditional Kafka broker system, a cluster of machines running

ZooKeeper system will maintain the coordination, data partitioning and consumer offset info processing and fault tolerance for all broker nodes.

Consumers: are usually request data as consumer groups. Consumers subscribe to a particular topic and retrieve all available messages which are under the topic range. Each consumer from a consumer group will receive data from one or more brokers that store messages on the requested topic. The number of consumers cannot be more than the number of partitions granted to that topic.

ZooKeeper Architecture: ZooKeeper acts simply as an entity-data information table that dictates

- which brokers messages under a certain topic are stored
- what are the current available
- if replica is on, which brokers are leaders and which are backups
- at what progress (offsets) have consumers already gone through on each broker.

Figure 2.3: Message Broker Conceptual View

CHAPTER 3

MATERIALS AND METHODS

3.1 Materials

List of materials used during the study:

- Two laptop computers
- Intellij Ultimate IDE
- Matlab Software
- Built in measuring tools techniques

3.1.1 Technologies Involved

- Java Programing Langage
- Spring Boot Framework
- Apache Kafka Message Broker

Spring Boot General Information :

Spring Boot is dedicated to running standalone Spring applications, the same as simple Java applications, with the java -jar command. The basic thing that makes Spring Boot different than standard Spring configuration is simplicity. This simplicity is closely related to the first important term neccessary to know about, which is a starter. A starter is an artifact that can be integrated in the project dependencies. It supports a set of dependecies to other artifacts that have to be included in the application in order to achive the desired functionality.

Figure 3.1: Spring Boot

Apache Kafka General Information :

Kafka was initially built at LinkedIn as its centralized event pipelining platform, substituting a disparate set of point-to-point integration systems. Kafka is created to manage high throughput (billions of mes- sages). In its design, special attention has been given to the efficient managing of multiple consumers of the same stream that read at different speeds. The resulting system is a scalable publish-subscribe messaging system designed around a distributed commit log. High- throughput is one advantage of the design of log aggregation systems over most messaging systems. Kafka aims to provide a unified, high-throughput, low-latency platform for managing real-time data feeds.

Figure 3.2: Apache Kafka

3.1.2 Description Of Hardware Devices

Local machines used in the project

Dell Laptop Computer : Linux gerild-Latitude-E7250 4.15.0-20-generic 21-Ubuntu SMP Tue Apr 24 06:16:15 UTC 2018 $x86_64 x86_64 x86_64 GNU/Linux$

$Operating Sytem -> Linux Mint 19.1 Tessa$

$Cpu -> Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz 1 physical processor; 2 cores; 4 threads$

$Motherboard -> Dell Inc.0G9CNK$

$Storage -> ATA SAMSUNG SSD PM87$

HP Laptop Computer : Machine Name-> DESKTOP-OPHG0A2

Operating Sytem -> Microsoft Windows 10 Pro

Cpu -> Intel(R) Core(TM) i7- 5500U CPU @2.40GHz ,2401 Mhz, 2 cores; 4 Logical Processing Unitsds

Total Physical Memory-> 7.90 GB

3.2 Methods

On the study of effectivity regarding inter-service communication three basic type of metrics are used, respectively :throughput, total processing time and response time metrics

3.2.1 Systematic Approach

The evaluation of the result is obtained by supplying the sender services with an automated peace of software which is able to send bundles of requests sequentially through a loop programming structure. In order to have a clearer view on the testing phase for each technique the same amount of requests is tested. In this study the number of requests changes. Particularly the testing phase is done with N=10,100,1000,10000,100000,100000 number of requests on each strategy of communication.

The increasing number of requests measures the capacity of each technique to handle different amounts of requests.

The purpose is to find how much does the increasing number of requests affects each strategy.It becomes easy to assess the effectivity by evaluating the behaviours of each communication technique at different amount of requests.

Throughput is obtained calculating the number of requests which are sent over the total processing time.

Total processing time is calculated as the difference between the starting time at the moment the first requests is sent from sender service with the ending time at the moment the last requests is served properly.

Response time on message broker is considered the time it takes the message to go from

the sender to kafka and from kafka to receiver,while in the HTTP synchronous and HTTP asynchronous communication the response time is calucalted as the difference between the start time of the outgoing individual request and the end time of the acknowledge of that request in the sender service.

These 3 variables are enough to calculate and to evaluate the importance of each technique regarding their general performance. Obtaining such empirical data makes possible the accurate decision taking on which technique to choose with respect to the use case requirements.

CHAPTER 4

IMPLEMENTATION

In this chapter the implementation of the services will be explained.There are 4 small projects build separately in order to provide environment for the 3 kinds of communication to take place.

4.1 Explanation

4 components are respectively:

- restclient service (sender service of synchronous communication)
- clientasync service (sender service of asynchronous communication)
- server service (which serves restclient and clientasync)
- server service (which serves restclient and clientasync)

4.1.1 Synchronous HTTP Communication

The restclient service is the microservice which will automate the sending requests. In this microservice data regarding synchronous HTTP communication will be gathered. The whole microservice is composed of three main java classes:

- RestclientApplication.java
- Example.java which serves as a data transfer model
- Environment.java interface which holds the url of the server service.

RestclientApplication.java holds the main logic in which througput , processing time and response time of each request is obtained.

Since the microservice is build on spring boot it obeys the configuration needed to make it a valid service.

Such configuration include annotations used to make possible the communication between microservices.

Annotations used :

- @SpringBootApplication
- @Data
- @NoArgsConstructor
- @AllArgsConstructor
- @Builder

Figure 4.1: Configuration Of RestClient Microservice

4.1.2 Asynchronous HTTP Communication

The clientasync is the service which will be responsible for the generation of performance data regarding HTTP asynchronous communication. The whole service is composed 4 main Java Classes:

- ClientAsyncApplication.java
- RequestThread.java which serves as thread provider class
- Example.java interface which serves as data transfer model
- MetricsCache.java which serves as a supporting class to measure neccessary performance metrics

Annotations used :

- @SpringBootApplication
- @Data
- @NoArgsConstructor

- @AllArgsConstructor
- @Builder
- @Configuration
- @Bean

```
    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-webflux</artifactId>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Figure 4.2: Configuration Of Client-Async Microservice

4.1.3 Kafka Messaging Communication

Kafka-prod-cons is the poject which implements the message broker between 2 common conceptual common services producer and consumer.In this project the performance metrics are measured at the moment of sending the message and immediately after the consumer receives the message.Throughput and Total Processing time are measured at the moment when the automated structure for generating requests is finished.

The whole project is composed of 6 Java classes:

- DataComponent.java serves to measure performance metrics
- KafkaConfig.java which serves to configure kafka approach
- KafkaProdConsApplication.java the main class which makes possible the initiation of automated process of metric measurement
- KafkaSimpleController.java (this class holds the automated procedure which is responsible for the initiation of the requests)
- MoreSimpleModel.java serves as a data transfer model
- SimpleModel.java serves as a data transfer model

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka-test</artifactId>
            <scope>test</scope>
        </dependency>

        <!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.8.5</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
```

Figure 4.3: Kafka Configuration

4.2 Coding

The coding section will provide visual insight on the automated process involved in measurement of performance metrics.

4.2.1 Synchronous Client Service

Exaplanation of the automated procedures in the code

```
@SpringBootApplication
public class RestclientApplication implements CommandLineRunner {


    public Logger log = Logger.getAnonymousLogger();

    public static void main(String[] args) { SpringApplication.run(RestclientApplication.class, args); }

    @Override
    public void run(String... args) throws Exception {

        RestTemplate restTemplate = new RestTemplate();
        ArrayList<Long> startTime = new ArrayList<>();
        ArrayList<Long> endTime = new ArrayList<>();


        Example ex = Example.builder().field("ex").build();

        long startTtime = System.currentTimeMillis();

        int N = 1000000;

        for(int i = 0; i < N; i++){

            startTime.add(System.currentTimeMillis());

            restTemplate.postForEntity(Environment.uri, ex, Example.class);

            endTime.add(System.currentTimeMillis());
        }

        long endTtime = System.currentTimeMillis();
```

Figure 4.4: Part 1 Rest Client Automated Procedure For Performance Measurement

```
        long endTtime = System.currentTimeMillis();

        for(int i = 0; i < startTime.size(); i++){
            long diff = endTime.get(i) - startTime.get(i);
            log.info( msg: "StartTime: " + startTime.get(i) + "  EndTime: " +endTime.get(i) + " difference " + diff);
        }

        double throughput = N/(double)(endTtime - startTtime);

        log.info( msg: "Throughput: " + throughput);
        log.info( msg: "Round Trip Time:"+(double)(endTtime-startTtime));

        PrintWriter out = new PrintWriter(new FileOutputStream( name: "file.m"));

        StringBuilder str = new StringBuilder();
        str.append("x = [ ");

        for (int i=0;i<N;i++){
            str.append(endTime.get(i)-startTime.get(i)+" ");
        }

        str.append(" ];");

        str.append(" N = 1:100000;  figure, plot(N, x) title('Response Time For Each Request When N=10') xlabel('N^{th} requests') ylabel('Resp

        out.println(str);
        out.flush();
    }
}
```

Figure 4.5: Part 2 Rest Client Automated Procedure For Performance Measurement

4.2.2 Asynchronous Client Service)

Code Explanation.

```
Logger  log = Logger.getGlobal();

public static void main(String[] args) { SpringApplication.run(ClientAsyncApplication.class, args); }

@Override
public void run(String... args) throws Exception {

    System.out.println("sth");

    int N = 1000;

    RequestThread requestThread[] = new RequestThread[N];

    for(int i = 0; i < N; i++){
       requestThread[i] = new RequestThread(metricsCache);
       requestThread[i].start();
    }


    for(int i = 0; i < N; i++){

        requestThread[i].join();

    }


    Thread.sleep( millis: 10000);


    metricsCache.calc();
    metricsCache.calcThroughput();
    metricsCache.generateMatlab();

}
```

Figure 4.6: AsyncClient Microservice

```
public void addEnd(Long time) { endTime.add(time); }

public void calcThroughput() throws Exception{
    Long diff = endTime.get(endTime.size() - 1) - startTime.get(0);
    int N = endTime.size();

    double throughput = N/(double)diff;

    log.info( msg: "Throughput: " + throughput);
    log.info( msg: "Total Processing Time:"+diff);
}

public  void generateMatlab() throws FileNotFoundException {
    PrintWriter out = new PrintWriter(new FileOutputStream( name: "file.m"));
    int cnt=0;
    StringBuilder str = new StringBuilder();
    str.append("x = [ ");

    diffTime.forEach(
            a -> {

                str.append(a +" ");

            }

    );

    str.append(" ];");

    str.append("Size of threads " + diffTime.size());

    str.append(" N = 1:1000;  figure, plot(N, x) xlabel('N^th requests') ylabel('Response Time of N^th request') title('Response Time For Each Request N=10') ");

    out.println(str);
    out.flush();
```

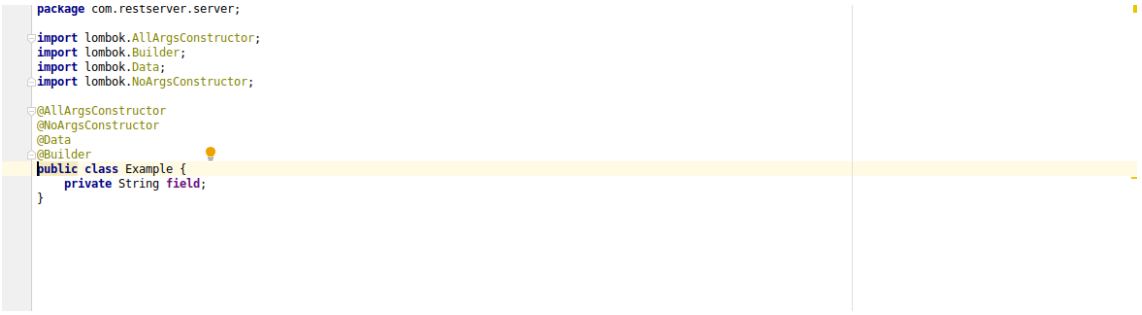
Figure 4.7: Class Responsible For Metrics Operations

4.2.3 Server Service

Code explanation.

```
package com.restserver.server;

import org.springframework.scheduling.annotation.Async;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api")
public class MyController {

    @Async
    @PostMapping
    Example post(@RequestBody Example ex) { return ex; }


}
```

Figure 4.8: Server Controller



```java
package com.restserver.server;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@AllArgsConstructor
@NoArgsConstructor
@Data
@Builder
public class Example {
    private String field;
}
```

Figure 4.9: Data Transfer Model On Server Service

4.2.4 Kafka Message Broker Producer-Consumer

Code Explanation.

```java
        Gson jsonConverter,
        DataComponent dataComponent
){
    this.kafkaTemplate = kafkaTemplate;
    this.jsonConverter = jsonConverter;
    this.dataComponent = dataComponent;
}

@GetMapping
public void post() throws InterruptedException {
   SimpleModel sm = new SimpleModel("field1", "field2");

    String str = jsonConverter.toJson(sm);

    int N = 1000000;

    for(int i = 0; i < N; i++){
        dataComponent.addStartTime(System.currentTimeMillis());
        kafkaTemplate.send( topic: "myTopic", str);
    }

    Thread.sleep( millis: 2000);

    dataComponent.calculate();
}


@KafkaListener(topics = "myTopic")
public void getFromKafka(String simpleModel){


    //System.out.println(simpleModel);
    dataComponent.addEndTime(System.currentTimeMillis());
    //SimpleModel simpleModel1 = (SimpleModel) jsonConverter.fromJson(simpleModel, SimpleModel.class);
```

Figure 4.10: Kafka Controller

```java
package com.simpleexample.kafka_prod_cons;

import ...

@SpringBootApplication
public class KafkaProdConsApplication implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(KafkaProdConsApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        RestTemplate restTemplate = new RestTemplate();

        restTemplate.getForEntity( url: "http://localhost:8084/api/kafka", SimpleModel.class);
    }
}
```

Figure 4.11: Kafka Application Initiation

```java
public class KafkaConfig {

    @Bean
    public ProducerFactory<String, String> producerFactory(){

        Map<String, Object> config = new HashMap<>();

        config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);

        return  new DefaultKafkaProducerFactory<>(config);
    }

    @Bean
    public KafkaTemplate<String, String> kafkaTemplate() { return new KafkaTemplate<>(producerFactory()); }

    @Bean
    public ConsumerFactory<String, String> consumerFactory(){
        Map<String, Object> config = new HashMap<>();

        config.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        config.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        config.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        config.put(ConsumerConfig.GROUP_ID_CONFIG, "myGroupId");

        return new  DefaultKafkaConsumerFactory<>(config, new StringDeserializer(), new StringDeserializer());
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory(){
        ConcurrentKafkaListenerContainerFactory<String, String> concurrentKafkaListenerContainerFactory = new ConcurrentKafkaListenerContainerFactory<>();

        concurrentKafkaListenerContainerFactory.setConsumerFactory(consumerFactory());
```

Figure 4.12: Kafka Configurations

```java
    private ArrayList<Long> startTime = new ArrayList<>();
    private ArrayList<Long> endTime = new ArrayList<>();


    Logger log  = Logger.getAnonymousLogger();


    void addStartTime(long start) { startTime.add(start); }

    void addEndTime(long end) { endTime.add(end); }

    void calculate(){
        int N = endTime.size();

        for(int i = 0; i < N; i++){
            long diff = endTime.get(i) - startTime.get(i);
            log.info( msg: "StartTime: " + startTime.get(i) + " EndTime: " + endTime.get(i) + "diff  " + diff );
        }

        long roundTrip = endTime.get(N - 1) - startTime.get(0);

        double throughput = N/(double)roundTrip;

        log.info( msg: " Throughput: " + throughput);
        log.info( msg: "Total Processing Time:"+roundTrip);


        PrintWriter out = null;
        try {
            out = new PrintWriter(new FileOutputStream( name: "file.m"));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
```

Figure 4.13: DataComponent Class In Producer-Consumer

CHAPTER 5

RESULTS

5.1 Throughput

The graphs show the throughput results when automated procedure took place for N=10,100,1000,10000,10000,100000,1000000

5.1.1 Synchronous HTTP Communication

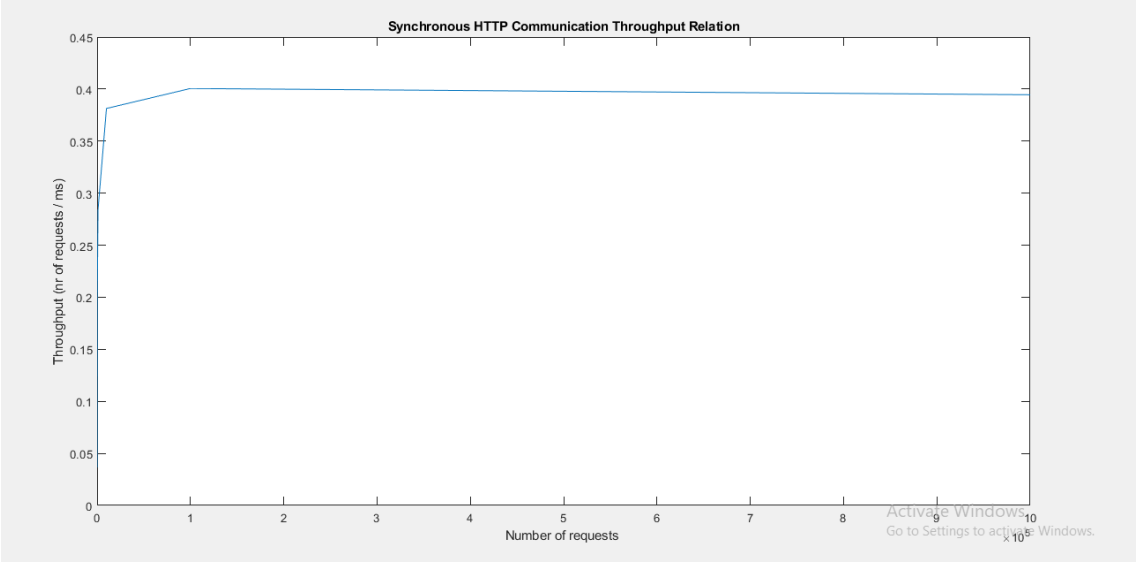


Figure 5.1: Throughput For Synchronous Communication

5.1.2 Asynchronous HTTP Communication

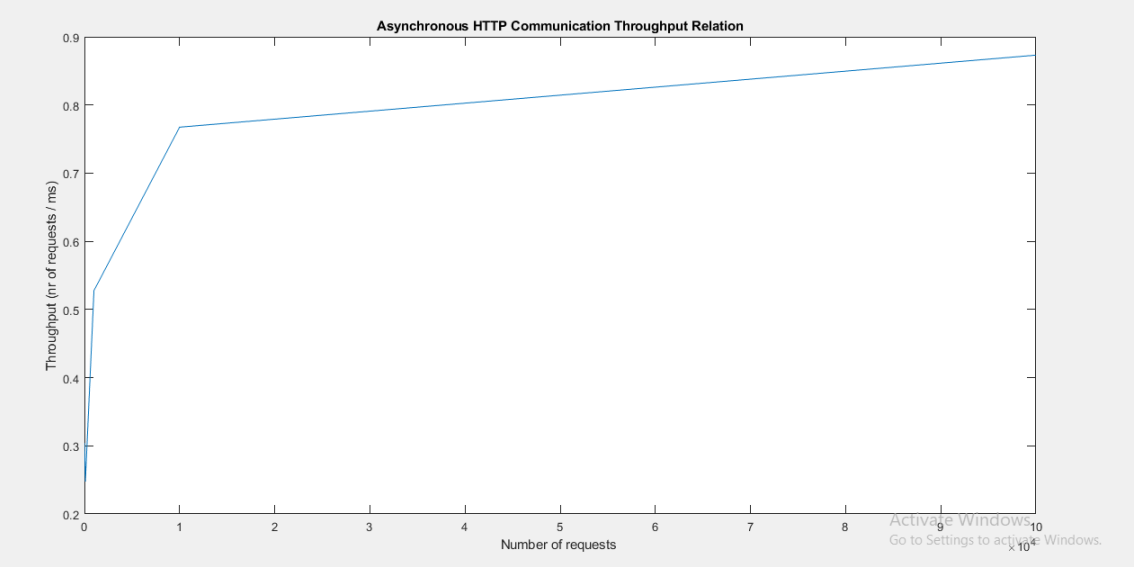


Figure 5.2: Throughput For Asynchronous Communication

5.1.3 Kafka Messaging

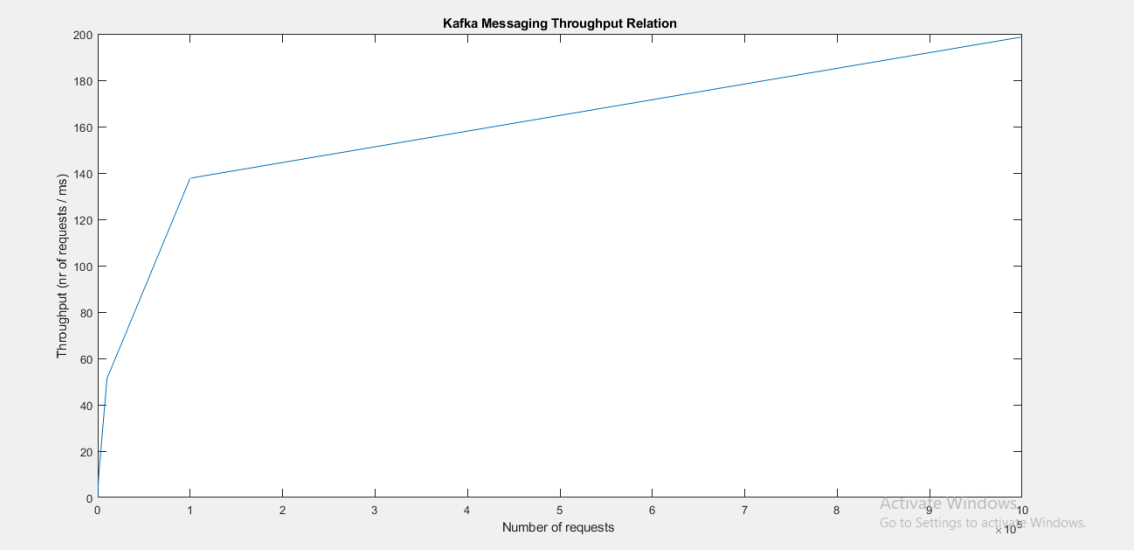


Figure 5.3: Throughput For Kafka Messaging

5.2 Processing Time

5.2.1 Synchronous HTTP Communication

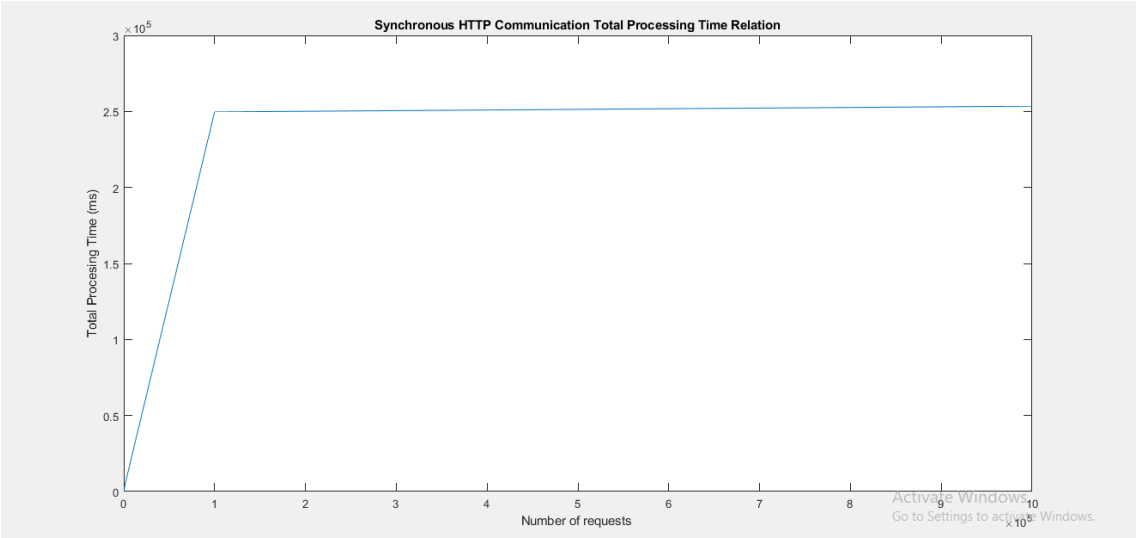


Figure 5.4: Total Processing Time For Synchronous HTTP Communication

5.2.2 Asynchronous HTTP Communication

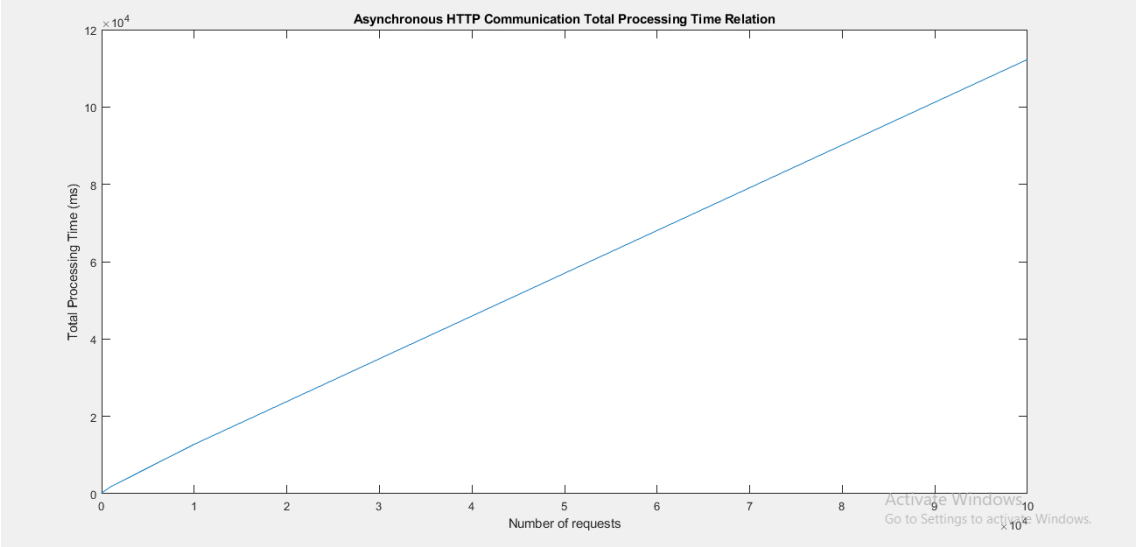


Figure 5.5: Total Processing Time For Asynchronous HTTP Communication

5.2.3 Kafka Messaging

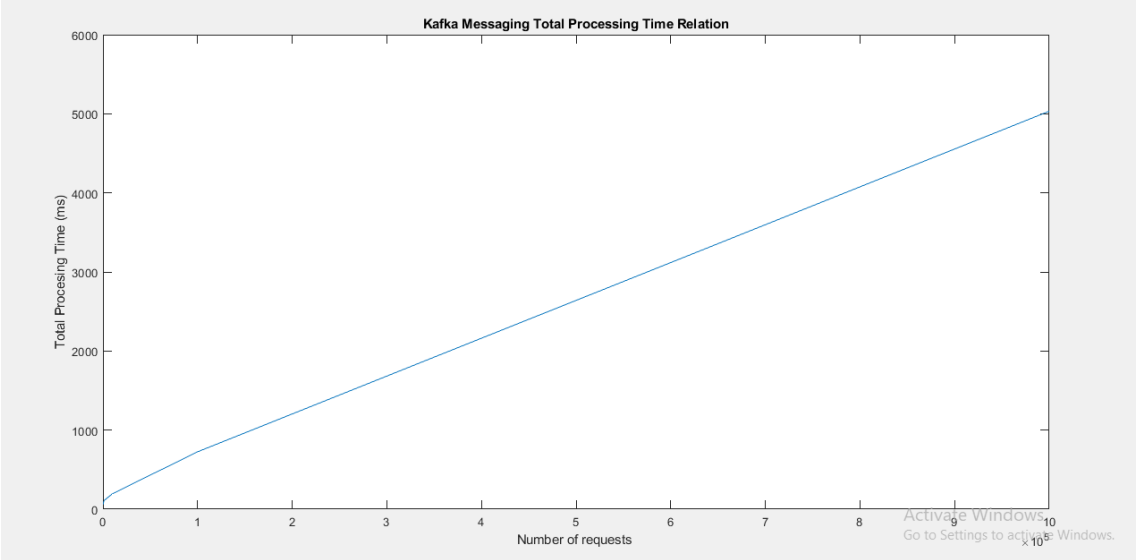


Figure 5.6: Total Processing Time For Kafka Messaging

5.3 Response Time

5.3.1 Synchronous HTTP Communication

For N=10:

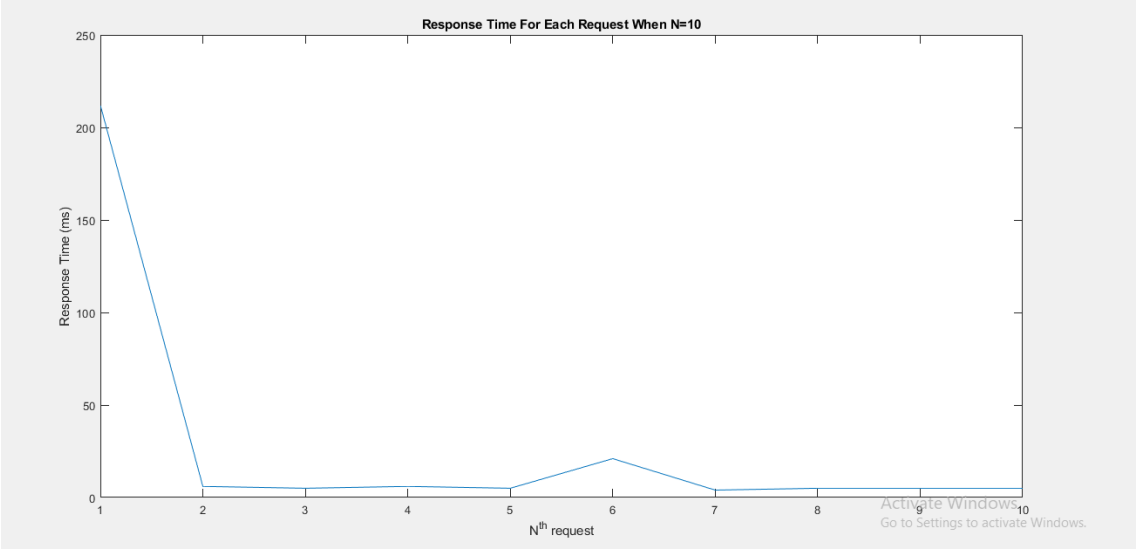


Figure 5.7: Response Time Relation Synchronous Communication N=10

For N=100:

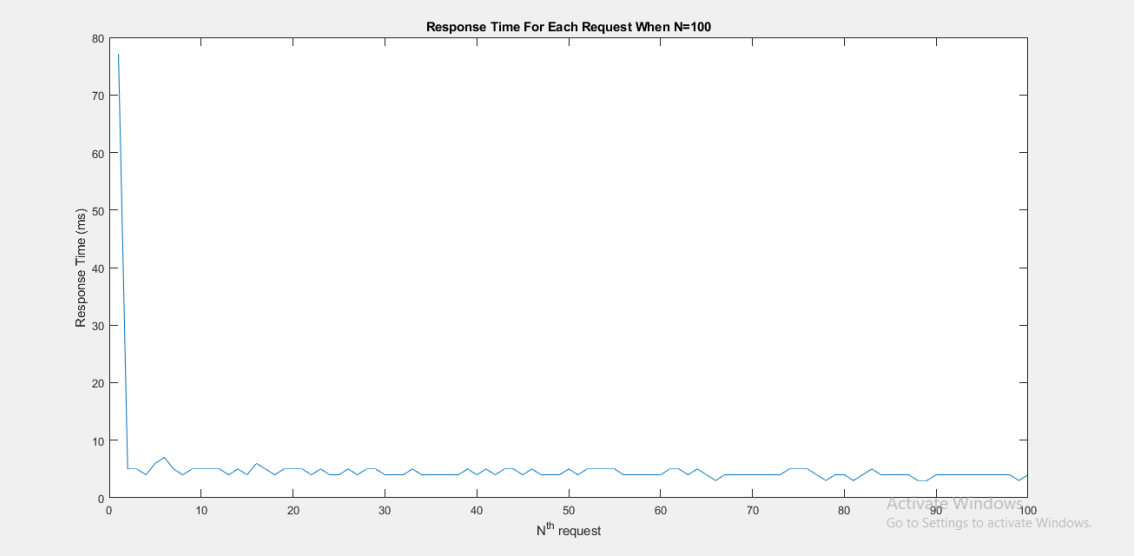


Figure 5.8: Response Time Relation Synchronous Communication N=100

For N=1000:

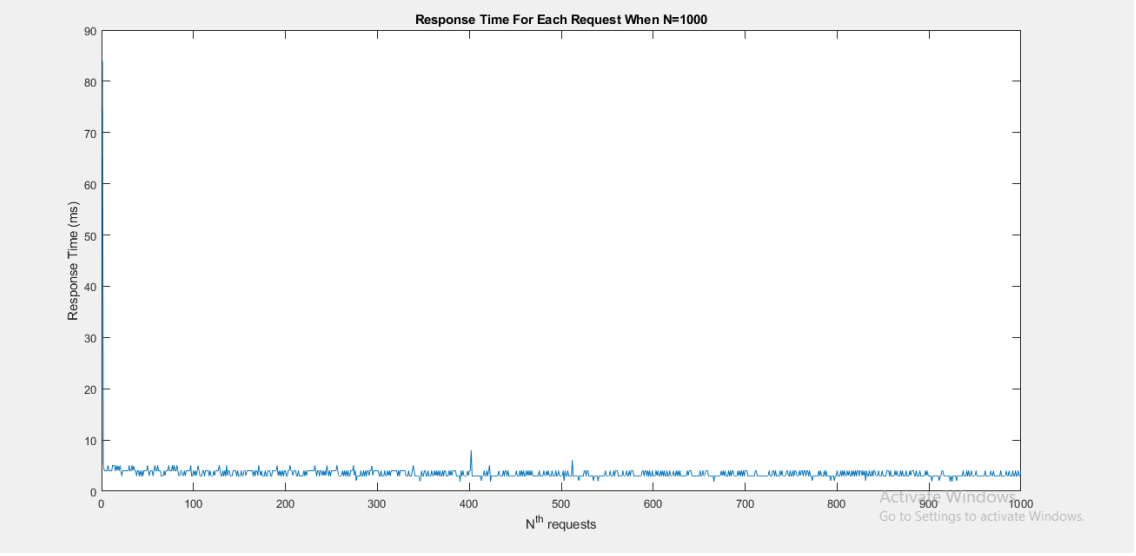


Figure 5.9: Response Time Relation Synchronous Communication N=1000

For N=10000:

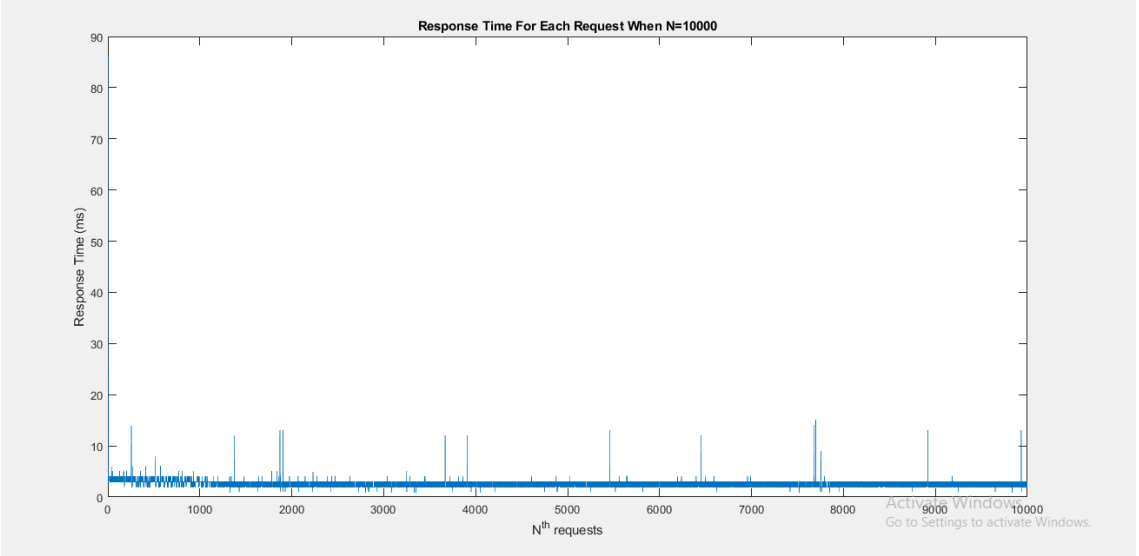


Figure 5.10: Response Time Relation Synchronous Communication N=10000

For N=100000:

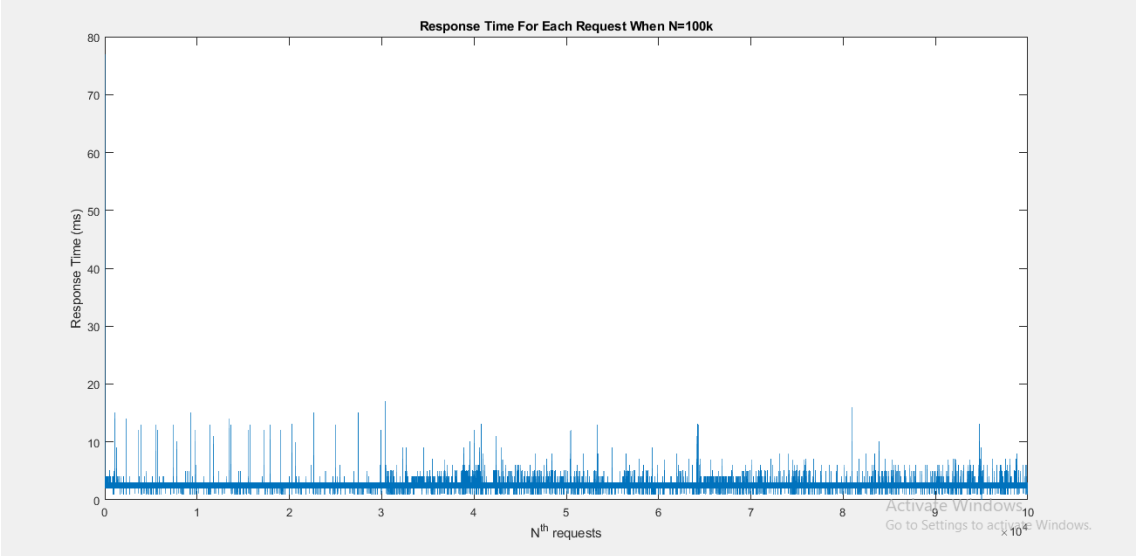


Figure 5.11: Response Time Relation Synchronous N=100000

For N=1000000:

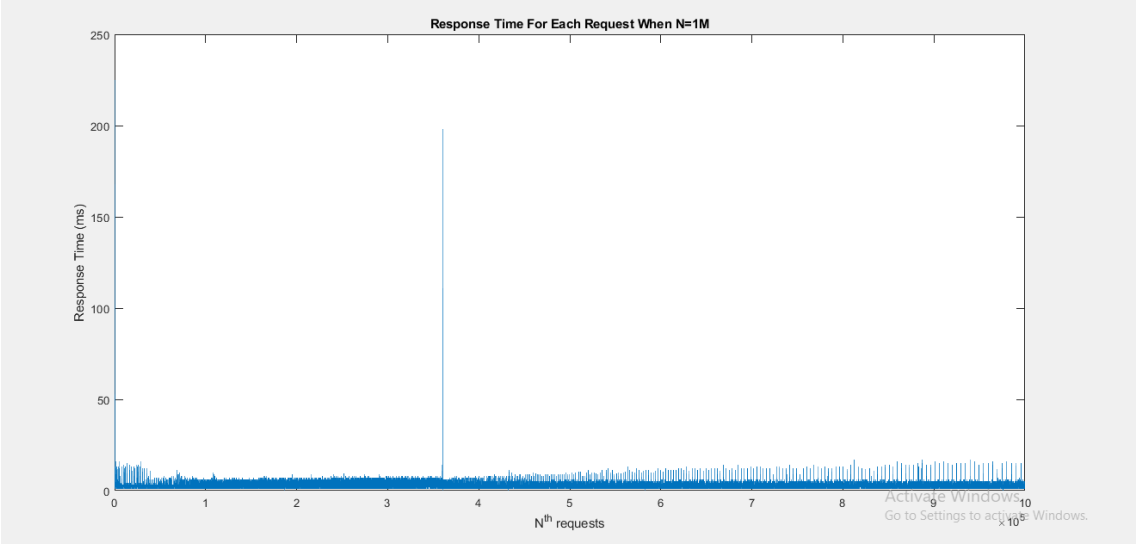


Figure 5.12: Response Time Relation Synchronous Communication N=1000000

5.3.2 Asynchronous HTTP Communication

For N=10:

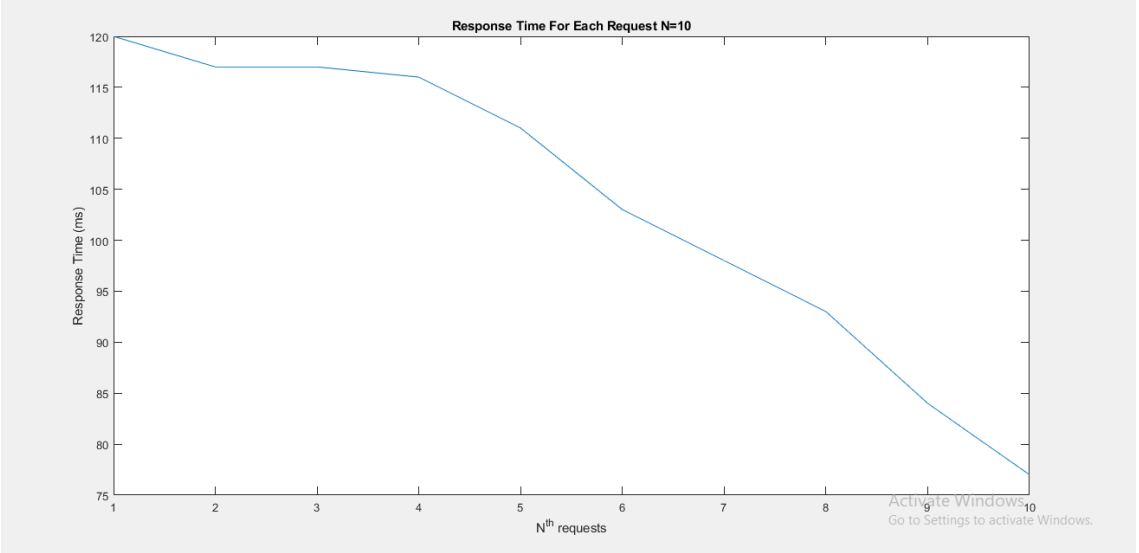


Figure 5.13: Response Time Relation Synchronous Communication N=10

For N=100:

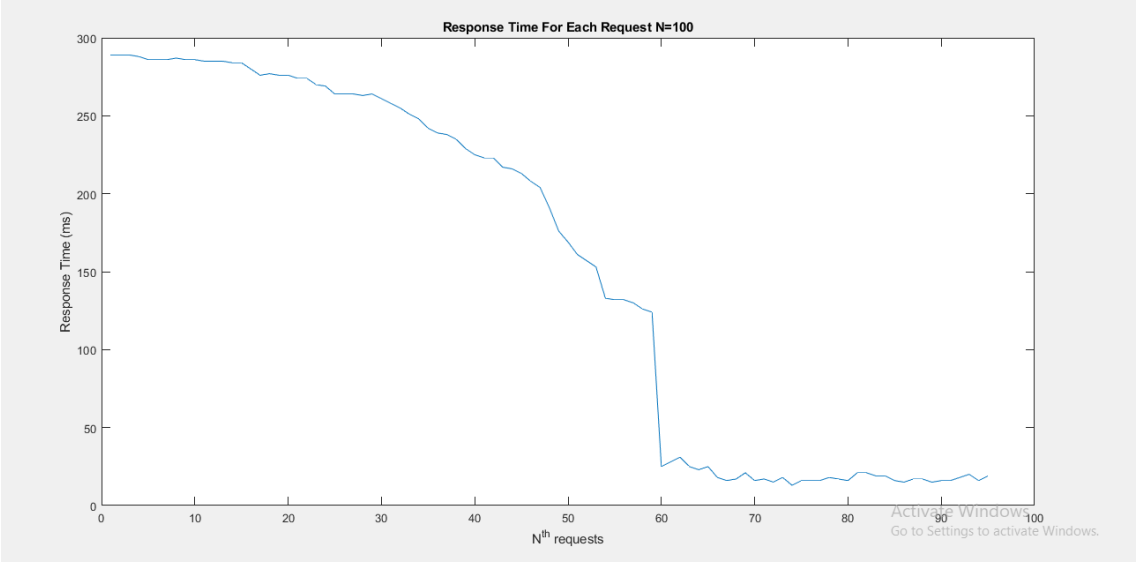


Figure 5.14: Response Time Relation Asynchronous Communication N=100

For N=1000:

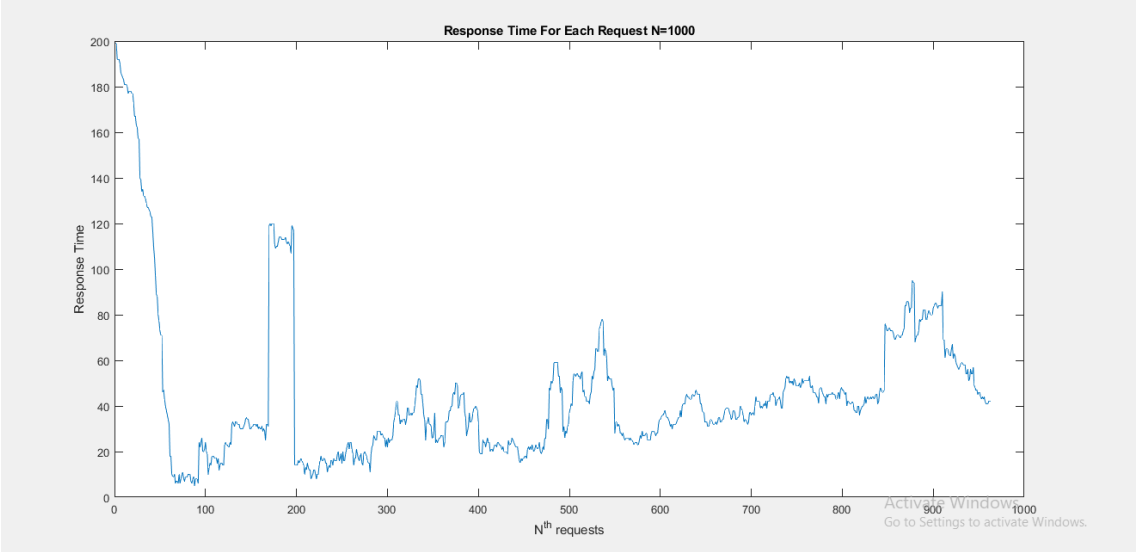


Figure 5.15: Response Time Relation Asynchronous Communication N=1000

For N=10000:

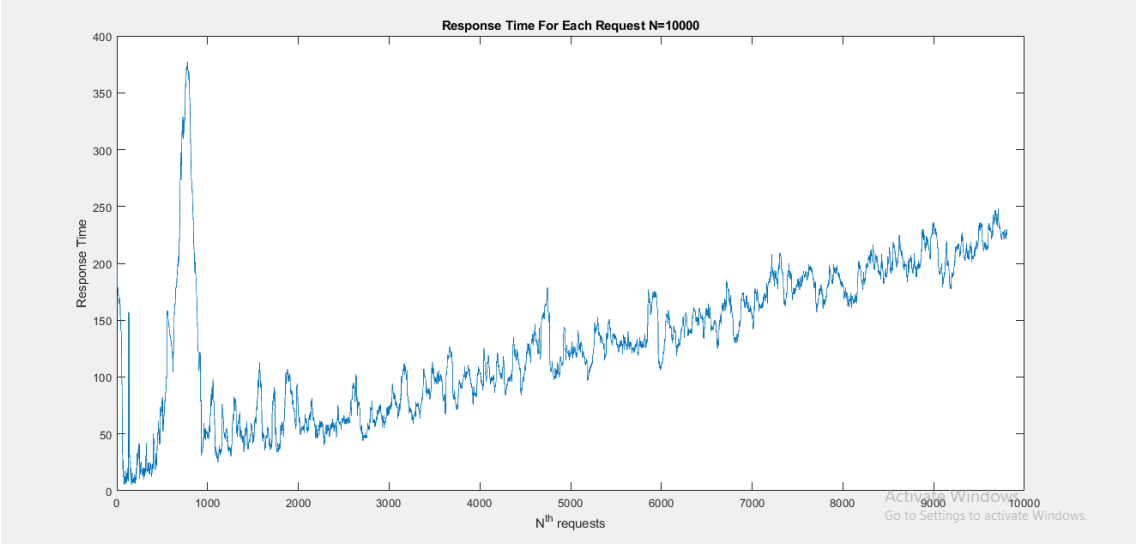


Figure 5.16: Response Time Relation Asynchronous Communication N=10000

For N=100000:

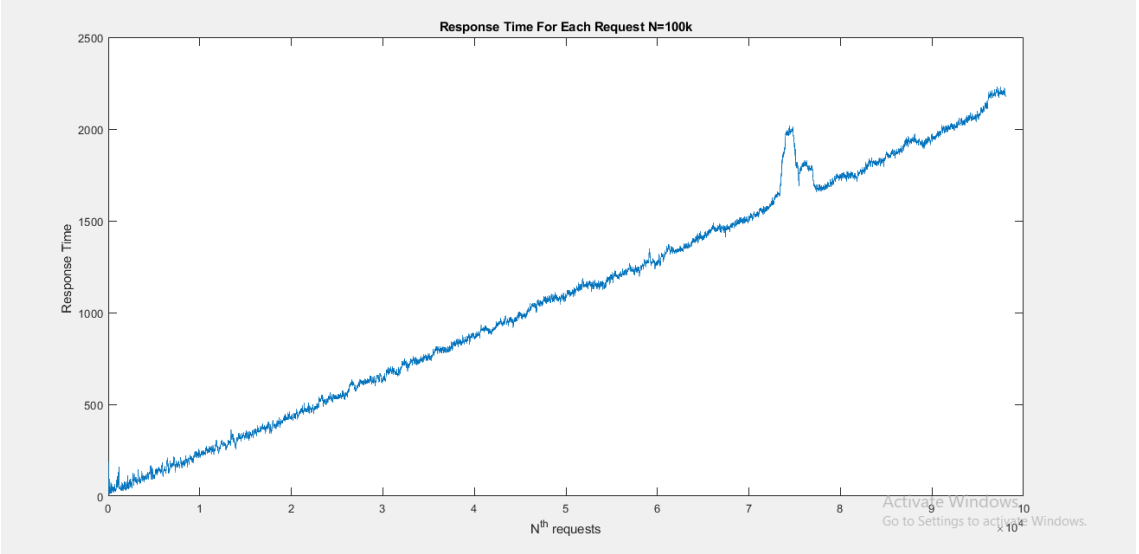


Figure 5.17: Response Time Relation Asynchronous Communication N=1000000

5.3.3 Kafka Messaging

For N=10:

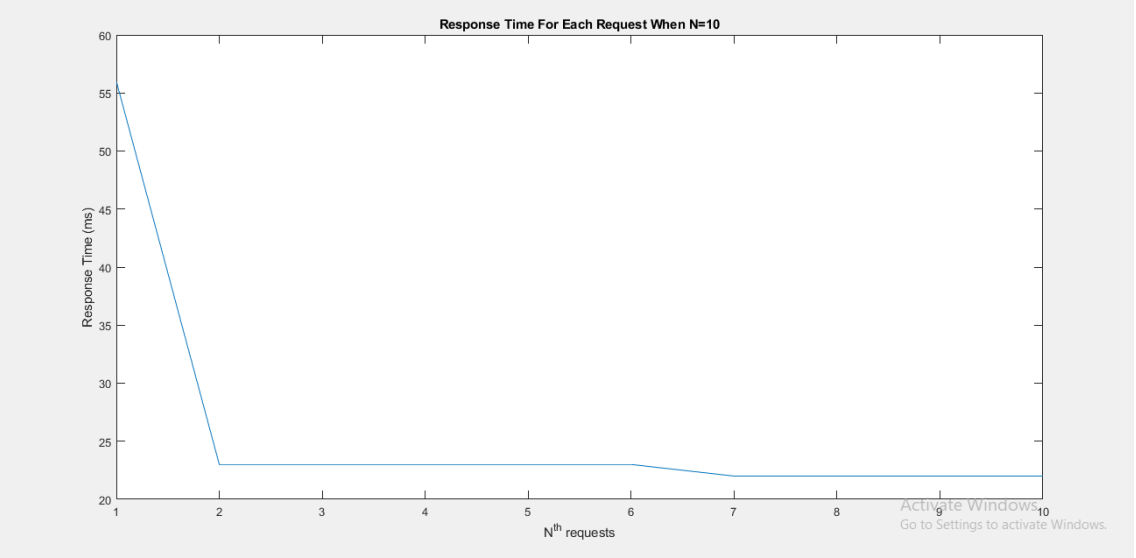


Figure 5.18: Response Time Relation Kafka Messaging N=10

For N=100:

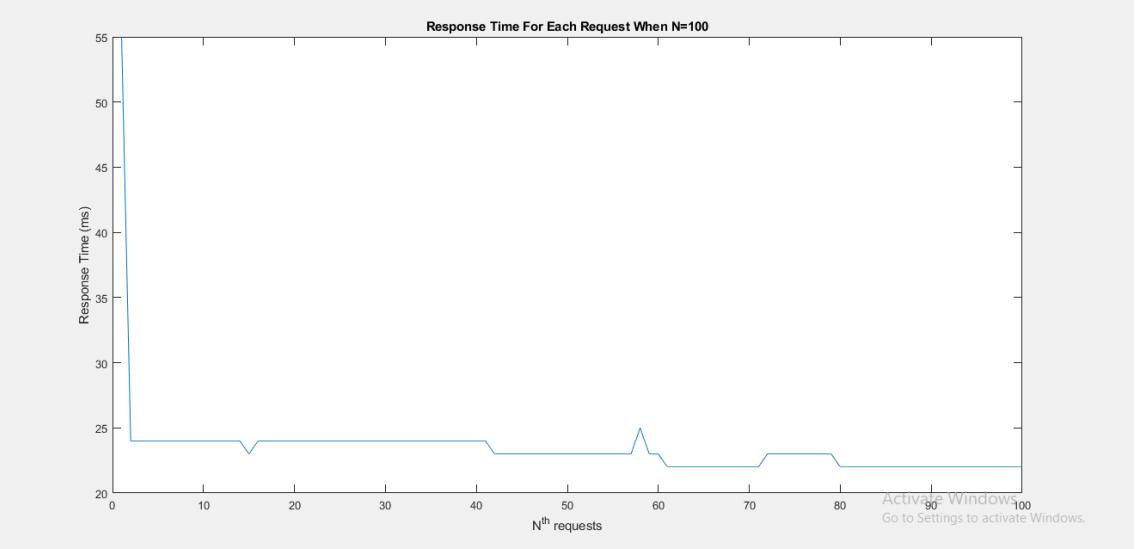


Figure 5.19: Response Time Relation Kafka Messaging N=100

For N=1000:

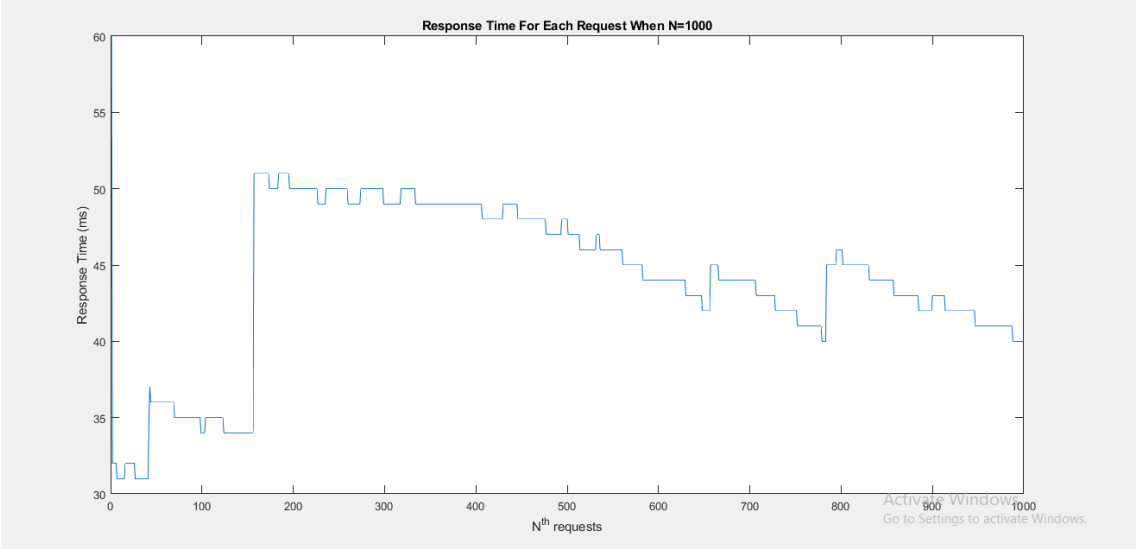


Figure 5.20: Response Time Relation Kafka Messaging N=1000

For N=10000:

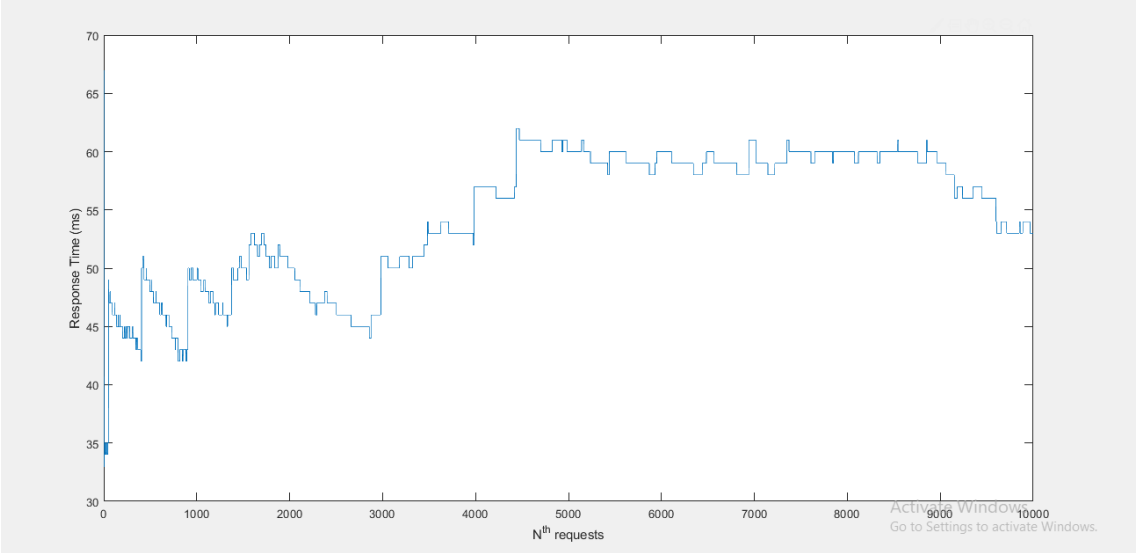


Figure 5.21: Response Time Relation Kafka Messaging N=10000

For N=100000:

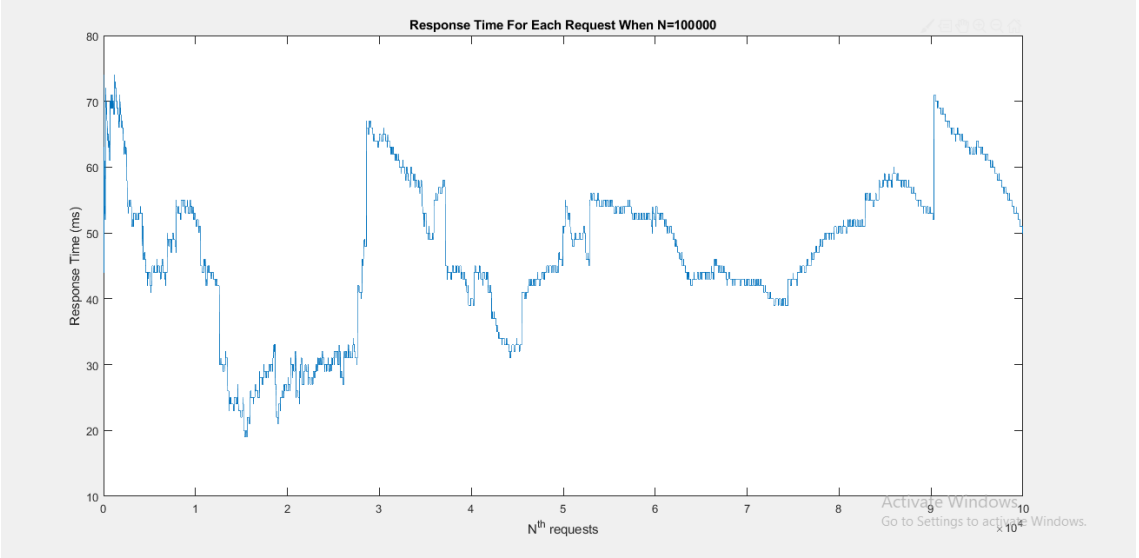


Figure 5.22: Response Time Relation Kafka Messaging N=100000

For N=1000000:

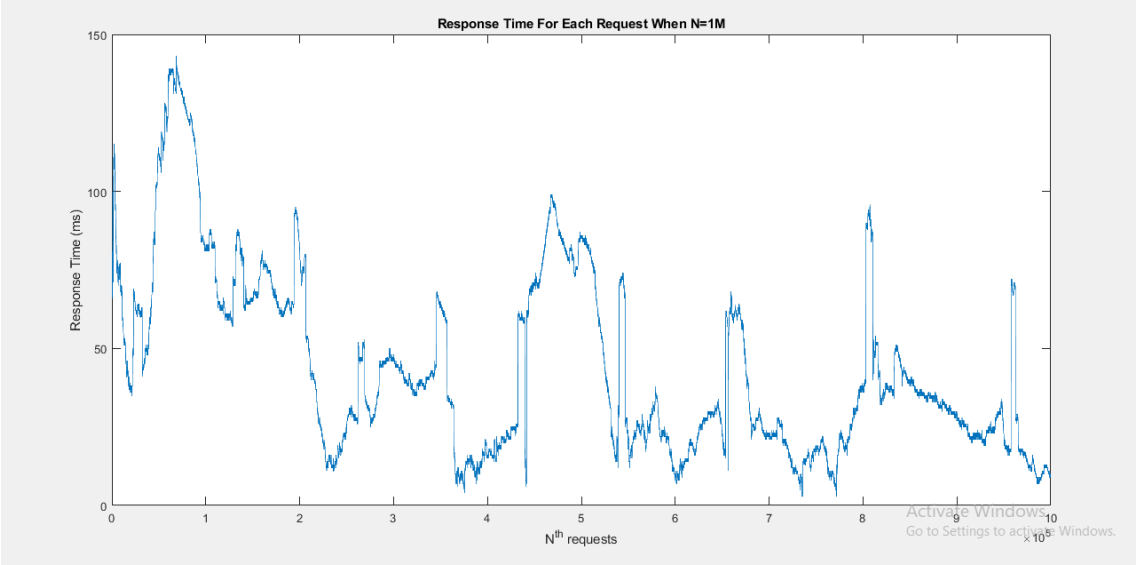


Figure 5.23: Response Time Relation Kafka Messaging N=1000000

CHAPTER 6

CONSLUSIONS AND FUTURE WORK

Once the above results are evaluated , the power of Kafka message broker becomes clear. Regarding throughput Asynchronous HTTP communication is faster and better on a subtle but evident difference in magnitude than HTTP Synchronous Communication. Kafka Broker proves to have a very high throughput with respect to the compared inter-service communication techniques. Even in abundance of requests thrown to the services Kafka has 138.689 requests/seconds while the two are methods still process sequentially at very low throughput even though the asynchronous communication is aided by the concurrent threads in sending the testing requests. The testing was done by automated structure of programming and consequently it must no be regarded as accurately showing the real capacities of the techniques of communication. Such methodology only proves the dominance of message brokers in throughput and in total processing time.

Total processing time is evidently faster. Taking into account the 1000000 requests it clearly separates the characteristics of Message Broker in a communication between microservices with respect to Synchronous HTTP communication which took nearly 40 minutes to finish the task of 1 million requests thrown to the server service.

The only metrics where the two other techniques overcome kafka are in response time of individual requests because kafka involves delay on it's queue system.In the end the response time is not of much value for as long as the broker provides the communication with a very effective approach in both Throughput and Total Processing Time

Future work may be projected on a more detailed analysis on the types of inter-service communication by implemented different protocols of communication and by automating the testing on platform as service industrial competitors. This approach can provide more effective conclusions while being implemented on real environments which are supposed to serve millions of people not on local machines. The study of effectivity of inter-service communication is beneficial for both academic growth and business capabilities.

REFERENCES

[1] Spring.io. (2019). Spring Projects. [online] Available at: https://spring.io/projects/spring-boot [Accessed 24 Jun. 2019].

[2] Spring Initializr. (2019). *Spring Initializr.* [online] Available at: https://start.spring.io [Accessed 24 Jun. 2019].

[3] Estrada, R. (2017). *Apache Kafka 1.0 cookbook.* Birmingham, UK: Packt Publishing., p.44.

[4] Mastering Spring Cloud Book. *Piotr Mińkowski.*Build self-healing, microservices-based, distributed systems using Spring Cloud

[5] Efficient Communication With Microservices by Petter Johansson [June 14, 2017],Master's Thesis in Computing Science

[6] Sookocheff, K. (2019). *Kafka in a Nutshell.* [online] Kevin Sookocheff. Available at: https://sookocheff.com/post/kafka/kafka-in-a-nutshell/ [Accessed 24 Jun. 2019].

[7] Kreps, J. (2019). *Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines).* [online] Engineering.linkedin.com. Available at: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines [Accessed 24 Jun. 2019].

[8] Philippe Dobbelaere and Kyumars Sheykh Esmaili (2018)Industry Paper: Kafka

versus RabbitMQ ,A comparative study of two industry reference publish/subscribe implementations

[9] Martin Storø Nyfløtt , Optimizing Inter-Service Communication Between Microservices, master thesis [2017]

[10] Jerry Gao, Ph.D., Chandra S. Ravi, and Espinoza Raquel,Measuring Component Performance Using A Systematic Approach and Environment [2016]

Appendix A

OUTPUT DATA

A.1 Data Generated

A.1.1 Synchronous HTTP Communication

For N=10:

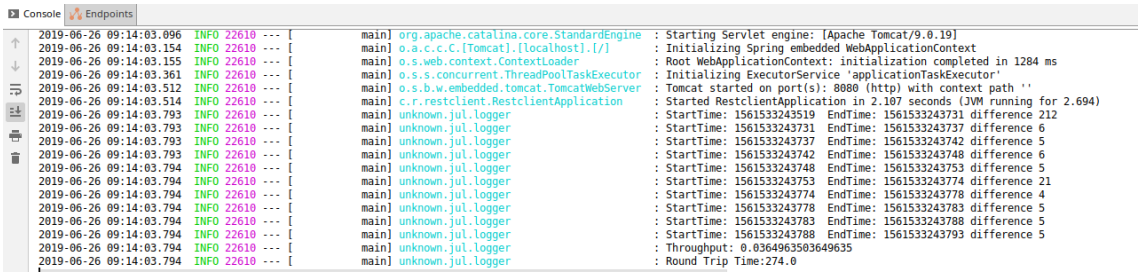


Figure A.1: Synchronous Communication N=10

For N=100:

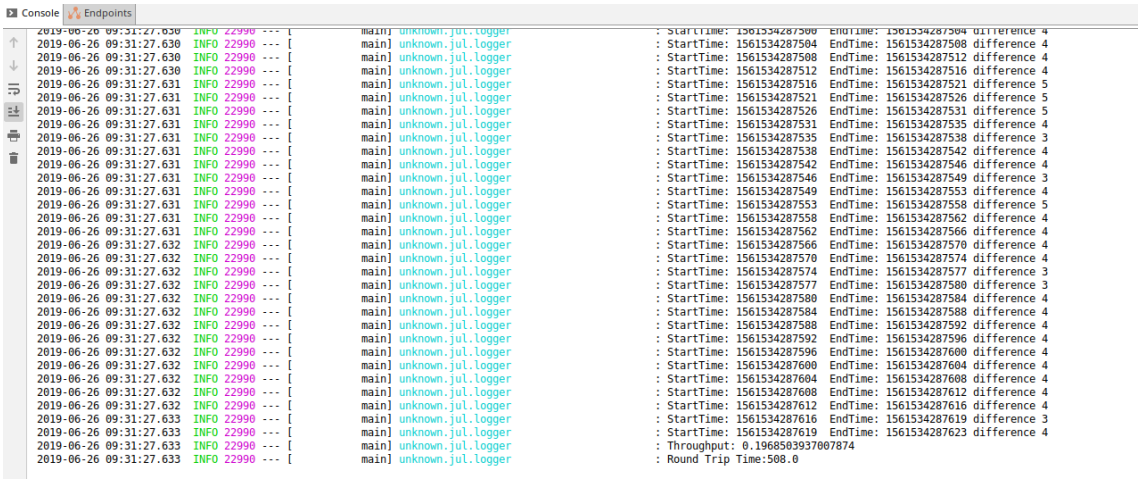


Figure A.2: Synchronous Communication N=100

For N=10000:

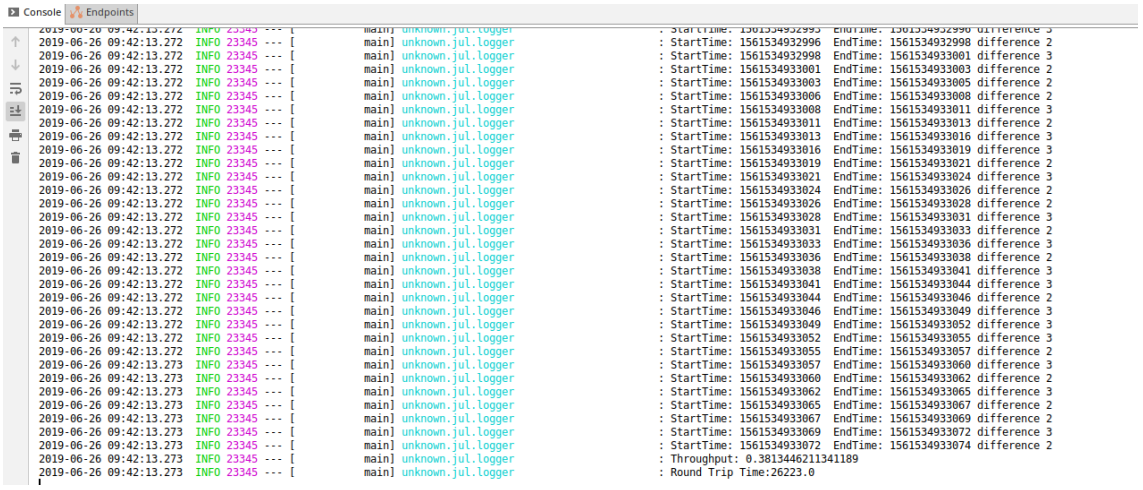


Figure A.3: Synchronous Communication N=10000

For N=100000:

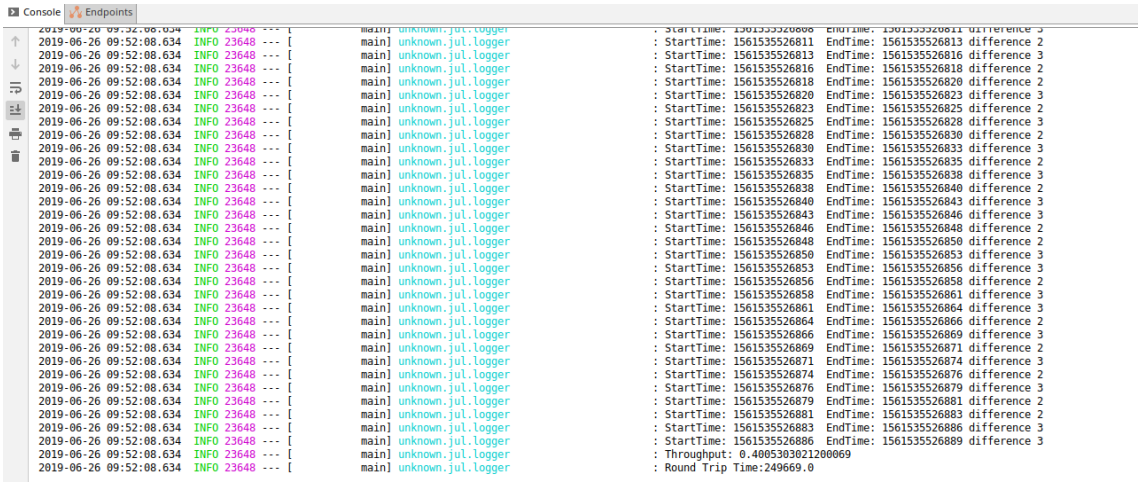


Figure A.4: Relation Synchronous N=100000

For N=1000000:

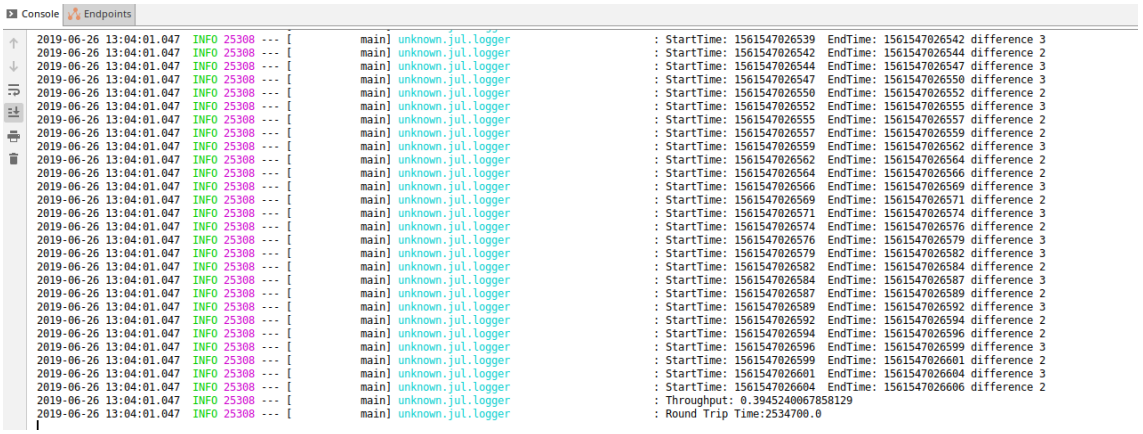


Figure A.5: Relation Synchronous Communication N=1000000

A.1.2 Asynchronous HTTP Communication

For N=10:

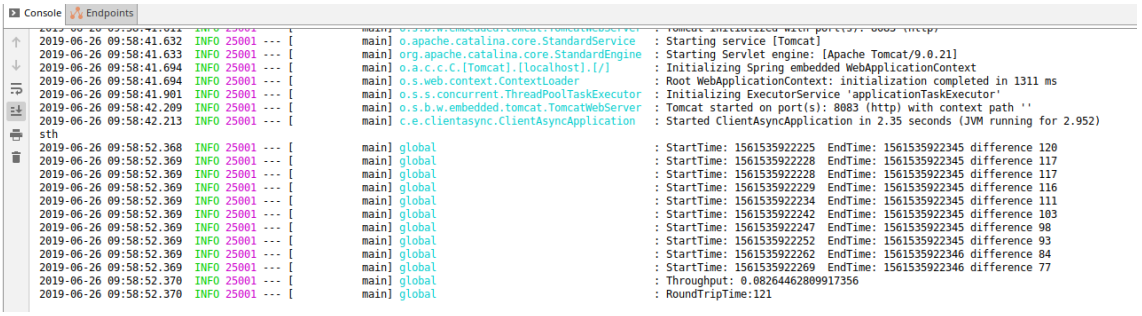


Figure A.6: Asynchronous Communication N=10

For N=100:

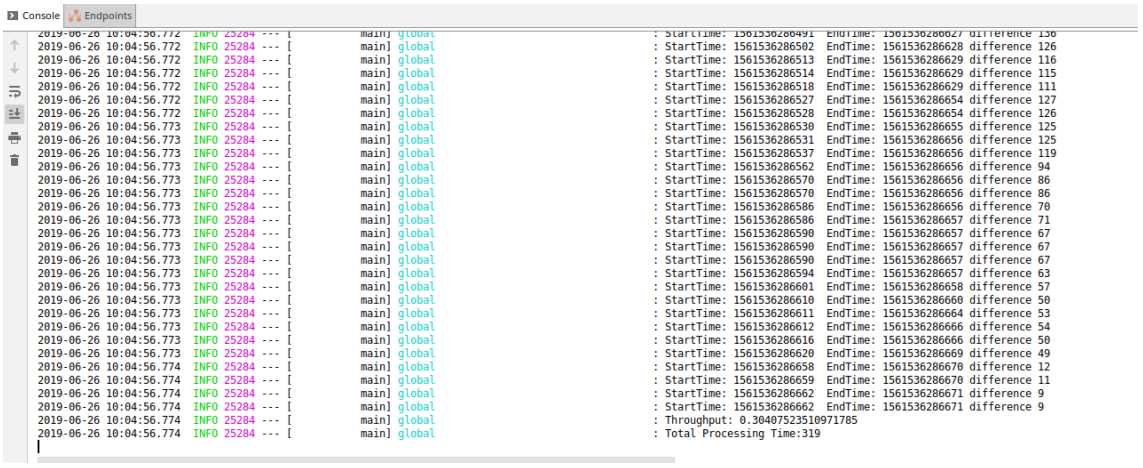


Figure A.7: Asynchronous Communication N=100

For N=10000:

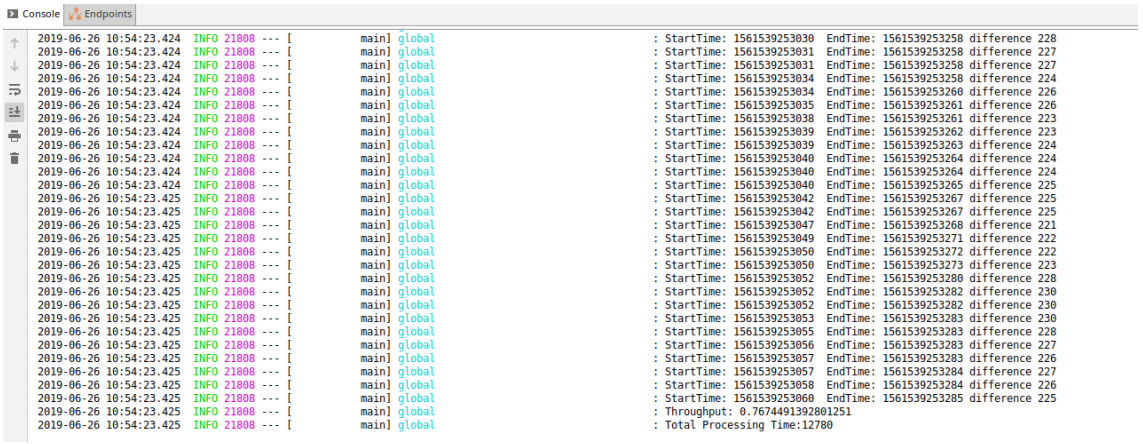


Figure A.8: Asynchronous Communication N=10000

For N=100000:

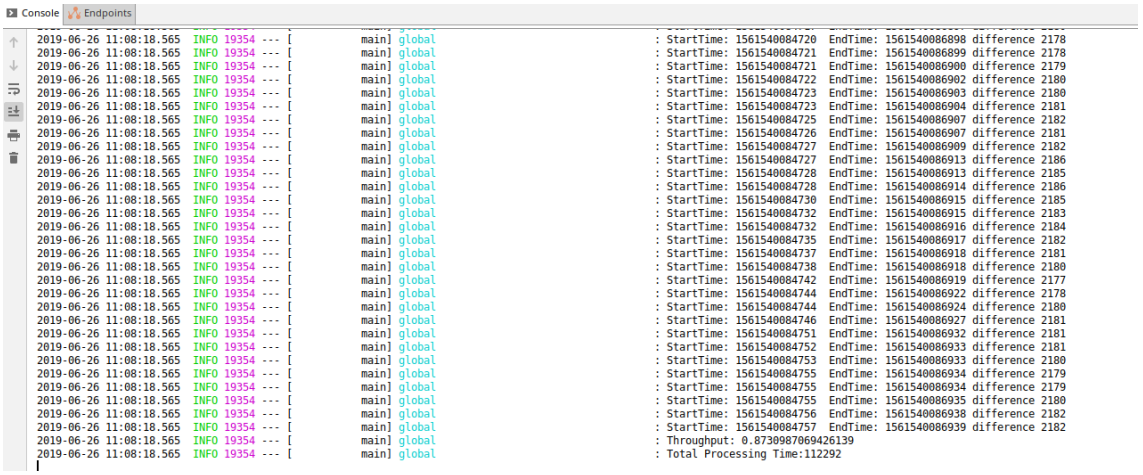


Figure A.9: Asynchronous Communication N=1000000

A.1.3 Kafka Messaging

For N=10:

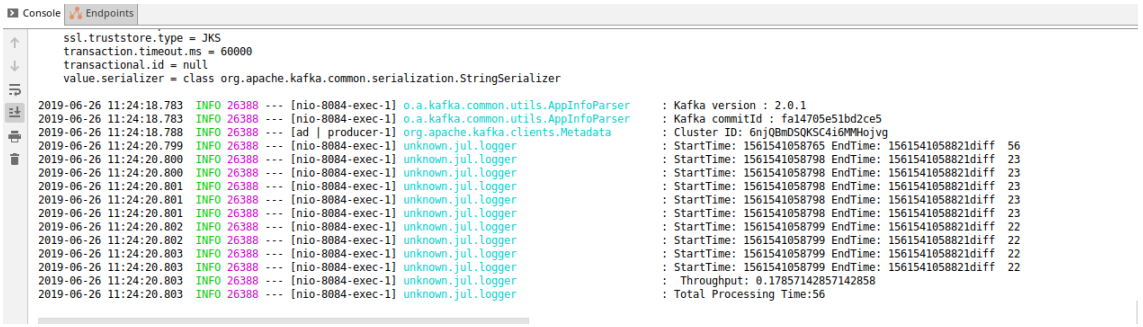


Figure A.10: Kafka Messaging N=10

For N=100:

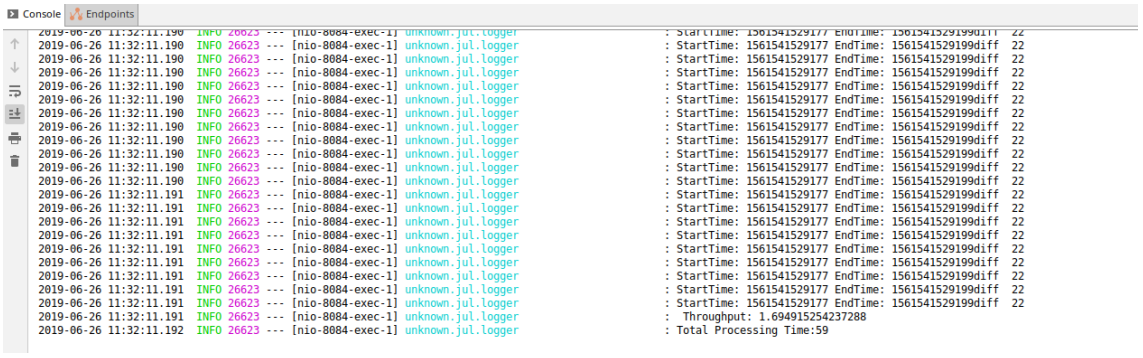


Figure A.11: Kafka Messaging N=100

For N=1000:

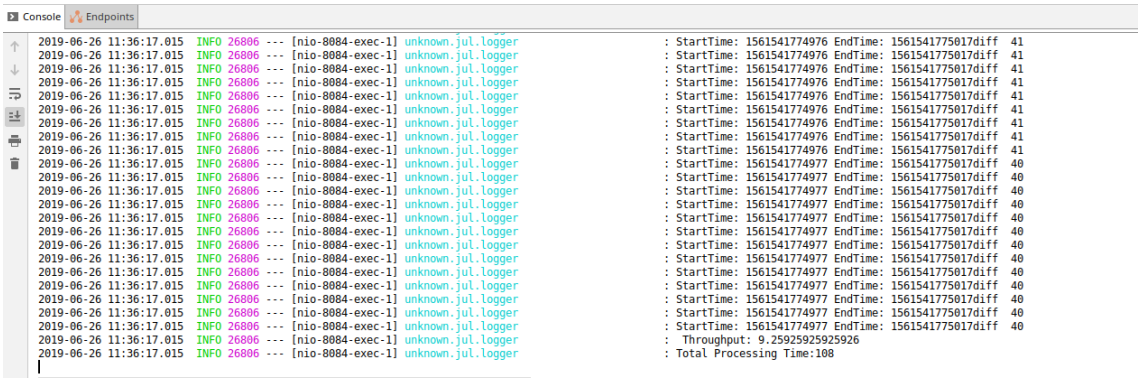


Figure A.12: Kafka Messaging N=1000

For N=10000:

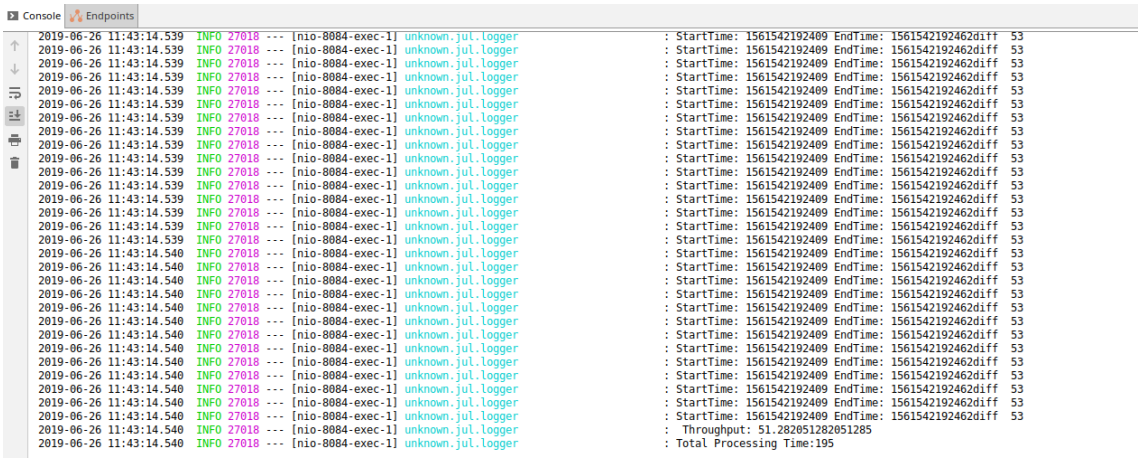


Figure A.13: Kafka Messaging N=10000

For N=100000:

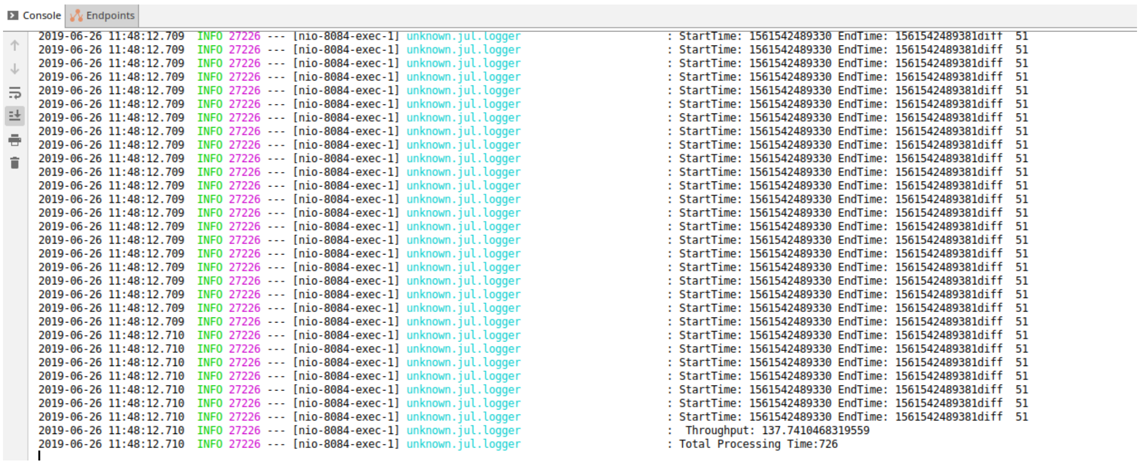

```
Console  Endpoints
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.709  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.710  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.710  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.710  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.710  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.710  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.710  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.710  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542489330 EndTime: 1561542489381diff  51
2019-06-26 11:48:12.710  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  :  Throughput: 137.7410468319559
2019-06-26 11:48:12.710  INFO 27226 --- [nio-8084-exec-1] unknown.jul.logger  : Total Processing Time:726
```

Figure A.14: Kafka Messaging N=100000

For N=1000000:

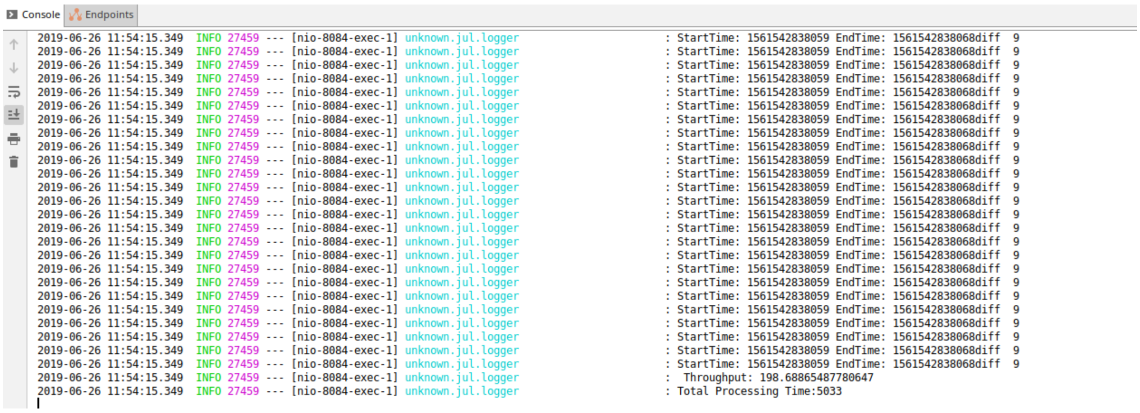

```
Console  Endpoints
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : StartTime: 1561542838059 EndTime: 1561542838068diff  9
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  :  Throughput: 198.68865487780647
2019-06-26 11:54:15.349  INFO 27459 --- [nio-8084-exec-1] unknown.jul.logger  : Total Processing Time:5033
```

Figure A.15: Kafka Messaging N=1000000

Appendix B

INSTALLATION AND RUNNING MANUAL

Below are the necessary steps to install and run components described :

1. Use a Linux/Unix machine as the system will not work otherwise
2. Make sure Java 8 is installed in that machine.
3. Download Apache Kafka software from: https://kafka.apache.org/downloads and unzip it somewhere in your machine where you have full administrative rights.
4. To run Kafka go inside the Kafka folder and open a terminal and type: "bin/zookeeper-server-start.sh config/zookeeper.properties" and leave the terminal open.
5. After doing that, in the same location, open a new terminal and type: "bin/kafka-server-start.sh config/server.properties" and leave the terminal open.
6. Make sure that Maven is installed in your machine. If not install it using the appropriate package manager. To check if it is installed open a terminal and type: "mvn -v".

7. For each individual project, browse where the "pom.xml" file is located, open a terminal there and type: "mvn clean", "mvn package", "mvn install", "mvn spring-boot:run" and leave every terminal window/process open.

8. Run the individual projects on run button and the process will automate itself