National College of Ireland

Enhancing Resilience in Spring Cloud Gateway using a Dynamic Heartbeat Algorithm in Eureka Service Registry

MSc Research Project
Msc Cloud Computing

Soumya Mohanan
Student ID: x23104767

School of Computing
National College of Ireland

Supervisor: Yasantha Chamara Samarawickrama

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Soumya Mohanan
Student ID:	x23104767
Programme:	Msc Cloud Computing
Year:	2024
Module:	MSc Research Project
Supervisor:	Yasantha Chamara Samarawickrama
Submission Due Date:	12/08/2024
Project Title:	Enhancing Resilience in Spring Cloud Gateway using a Dynamic Heartbeat Algorithm in Eureka Service Registry
Word Count:	7147
Page Count:	20

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

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

Signature:	Soumya Mohanan
Date:	12th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

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

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Enhancing Resilience in Spring Cloud Gateway using a Dynamic Heartbeat Algorithm in Eureka Service Registry

Soumya Mohanan
x23104767

Abstract

In modern distributed systems, the microservices architecture is preferred due to scalability, flexibility, and resilience. Spring Cloud Gateway is quite important in management and routing of requests in these environments. However, maintaining availability and scalabilities particularly during network outage and services disruptions poses significant challenges. This research thus presents a heartbeat system that is dynamic in nature for the Eureka Service Registry of the microservices to improve on their communication . The system dynamically changes the heartbeat intervals in accordance with the load of the microservices which depends on the rate of CPU usage, memory usage and request throughput using OSbean libraries. This approach is to minimize the number of times the Eureka server is called for in the process by reducing response during low traffic and improve system response during high traffic. Additionally, as for the retry mechanism, this study aims at improving the existing retry mechanism by utilizing the Eureka registry to improve retries and fallbacks. Therefore, the effectiveness of the method is assessed by load testing under different parameters, failure scenarios and comparison testing with traditional configurations. Based on the findings of this study, it can be concluded that the dynamic heartbeat intervals enhance the system performance and responsiveness to a greater extend. This way, the system is able to identify problems as intervals are varied depending on the load. as fast as every 9 seconds under high load and, as a result, enhancing the performance and issues can be identified early contrary to the default 30 seconds default heartbeat transfer.

1 Introduction

The landscape of cloud-based applications has undergone a significant transformation over the past decade, moving from monolithic architectures to microservices architectures. This evolution is driven by the need for more scalable, manageable, and flexible software development strategies. Unlike monolithic architectures, where all components are interconnected and interdependent, microservices architecture breaks down applications into smaller, loosely coupled services. Every one of them concentrates on a single functionality which makes the development, deployment and management of the services more feasible Blinowski et al. (2022).

Microservices architecture offers several advantages. It allows independent deployment of services, enabling continuous integration and continuous deployment (CI/CD) practices. Moreover, it supports technology variety, because the different services can be divided and developed using various programming languages and frameworks provided they comprise a single integrated system. nicate through standardized APIs Adhikari et al. (2012). However, this architecture also introduces complexity in managing inter-service communication, monitoring, and failure handling. Ensuring robust communication and fault tolerance mechanisms is critical to maintaining the availability and reliability of the system. Research Problem

One of the key challenges in microservices architecture is managing service requests and handling failures. In this context, Spring Cloud Gateway plays a pivotal role in routing and managing requests within microservices architectures. Despite its capabilities, ensuring the availability and scalability of systems using Spring Cloud Gateway remains a significant challenge, especially in the face of network failures and service disruptions.

Circuit breaker patterns are commonly employed to address these challenges by implementing fail-fast mechanisms that prevent cascading failures, allowing systems to degrade gracefully. However, the effectiveness of circuit breakers is highly dependent on the configuration of retry mechanisms and the discovery servers. These mechanisms enhance system availability and scalability by efficiently managing retries, detecting failures, and adapting to dynamic workload variations Hlybovets and Paprotskyi (2024). Despite their importance, the integration of retry mechanisms and heartbeat detection with circuit-breaker strategies is not well-explored.

Research Question and Objectives

Can the retry mechanism and heartbeat detection be set up to enhance the availability and scalability of Spring Cloud Gateway and the microservices utilizing a circuit-breaker strategy?

To address this question, the research focuses on the following objectives:

1) Create a dynamic heartbeat detection algorithm incorporating the existing spring framework.

2) Evaluate the effectiveness of the proposed framework in enhancing the availability and scalability of microservices architectures.

3) Suggest best practices with regard to the setup of retry cycles and the ability to detect heartbeat of microservices.

In so doing, this research advances knowledge in several specific ways. Firstly, it provides in depth analysis of retrying and heartbeat techniques as plans to break and reset in microservices architectural design. It has the feature of deeply practical solution on to improve the availability and scalability of Spring Cloud Gateway and associated microservices. Thirdly, the implications of the findings and recommendations that will be made in this study will may be of a great help to the developers and system architects trying to implement fault-tolerance mechanisms in their microservices deployments. Figure 1 depicts the Spring Microservices Architecture.

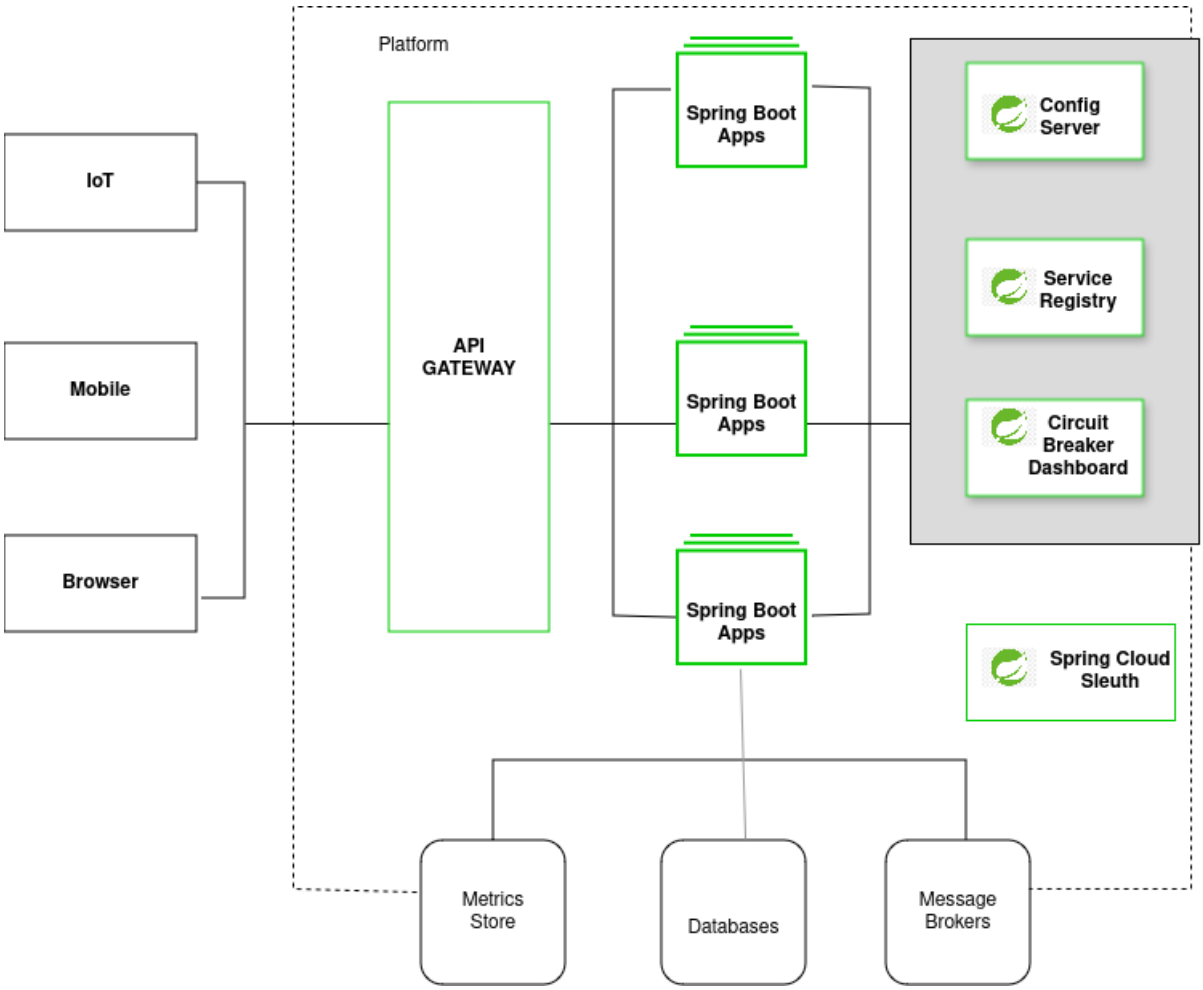

Figure 1: Springboot Mircoservices Architecture

Following is an overview of this work's content. In Section 2, a comprehensive literature review on microservices architecture, circuit-breaker patterns, retry mechanisms, and heartbeat detection, offering a thorough comparison of key articles in the field is provided. In Section 3 and Section 4, the paper's research methodology is described, structural and conceptual designs of the proposed framework and architecture diagrams is detailed. Section 5 details the implementation of the framework within Spring Cloud Gateway, highlighting the integration heartbeat detection. The performance metrics and analysis of the framework are presented in Section 6, including the configuration of scenarios and evaluation of results. An in-depth analysis of the outcomes, comparing the

dynamic heartbeat with the normal static approach, is also provided. Finally, Section 7 summarizes the study's findings and offers recommendations for future research, discussing the study's contribution to improving resilience in microservices architectures.

The report aims to provide a comprehensive understanding of the role of retry mechanisms and heartbeat detection in enhancing the reliability and scalability of microservices architectures, particularly in the context of Spring Cloud Gateway.

2 Related Work

2.1 Latency Minimization

Many works have suggested methods for achieving the best possible latency, and one of the primary objectives in use cases following the microservices pattern is scalability. In this architecture, the one service is duplicated with multiple copies, which are, in turn, implemented across the various physical systems. Requirements often arise where certain tasks have to be performed in these service instances, which may be located at different computers. When a set of services performs activities serially/step-by-step, deciding on the ideal service instances is vital to avoid latencies and enhance the API's overall time. Selvakumar et al. (2023) put forward an algorithm that was aimed at identifying optimal services that should be invoked for given request with the main goal of enhancing efficiency through quantitatively evaluating the resource consumption based on the system's view of each instance of the service while execution of a particular request is ongoing. Thus, here the load balancing is provided by the algorithmic solution that is coupled with machine learning as the number of requests per second is growing; and in terms of optimal response time, the results are comparable to the traditional load balancing approach. Though the above said method varies slightly from a certain fixed limit, it has been found that the improvement substantially brings down the latency period, especially for large service calls. According to this approach, one has clearly observed that the approach works well in the following aspects in general: Demonstrated impressive results in terms of latency reduction; More specifically, the results indeed show improvement in the capacity of service sequences with the added bonus that the service sequences had longer durations.

These studies underscore the significance of optimizing resource allocation and load balancing strategies, which will be considered in this research which are essential components in enhancing the performance and scalability of Eureka's adaptive heartbeat detection mechanism.

When implementing priority queues in a long chain of microservices, Rahman et al. (2014) in their research work was able to minimize the latency and also cope with rivalry issues in relation to the resource competition among divergent service chains. They also used a message queue to differentiate between microservices by using different priority levels while processing requests; this way, the priority of resources can be always changed in dependence on current load. There were several priority queues at the edge cloud server; thus, a multi-level feedback mechanism was used. Duration with respect to priority in queues suggested higher priority for shorter microservice chains than the larger ones, presumably because duration was used to prioritize packets between the higher and lower priority queues based on the size of the packets. However, for the cases with a lot of instances of microservices, this approach of load balancing may prove quite problematic

and could very easily end up being a bottleneck.

In the case of latency, Selvakumar et al. (2023) has brought up an adaptive load balancing technique to handle the reality of microservices. This technique entail estimation of the waiting time for various types of requests and selectivity of the service instances depending on the degree of utility of the system resources like CPU, memory, band width and others. This has to result in optimization of overall latency because the corresponding service instances are chosen to have minimum interconnect delays between physically distinct machines hosting different services in the system. For the prioritization of the service instances, Load Balance Indicator (LBI) is presented to measure the load on service instances to select the best instance accepting the request. The LBI takes into account different consumption indicators of system resources and the number of messages to be exchanged to perform a task. The algorithm gives more priority to the high importance service chains or to chains that require more subdued latencies in which important services are processed with less latency.

Since, a number of methods have shown that there is a rise in latency for ordinary requests when optimizing complicated ones, Rui et al. (2021) developed a technique that estimates latency for dealing with the interaction workloads and the multiple queuing systems. They created a feedback loop so that there was some measure of fairness regarding workload and to safeguard non-interactive workloads. They proposed a load balancing algorithm based on the concept of a task chain that targeted service call and information exchange between servers. This algorithm combined Particle Swarm Optimization, Simulated Annealing, and Genetic Algorithm for the improvement of load balancing where the problem can be observed in unbalanced loads as well as long times of completion in containerized microservices applications. Further, they used service discovery as well as performance monitoring through the Optimized Ant Colony Algorithm, which highlighted a marked decrease in the subjected energy consumptions of workload. It also examined the power consumption in conjunction with the latency optimization utilizing black-box monitoring system.

These studies point to the fact that it is appropriate to emphasize the problem of resource distribution and sharing and applying appropriate measures for load balancing. These are the aspects that are critical in improving the efficiency and effectiveness of Eureka's adaptive heartbeat detection mechanism that shall form the basis of my study.

2.2 Dynamic Load Balancing and Heartbeat Detection

According to the manner in which real-time status information is dealt with, load balancing algorithms are generally categorized in the static and dynamic categories. Static algorithms like the weighted round-robin analyze the load and through prior studies assign weights to the nodes in a cluster which is also applicable in round-robin max-min among others. Although these algorithms are quite uncomplicated to use, they might not be so efficient in addressing such changes becoming perhaps slow in matters of node failure or node overload and so on since they are not programmed with real-time view of nodes. Other authors such asWen et al. (2015) have introduced new concepts that may help in the management of resources namely distributed VM migration strategies like the ant colony optimization algorithm that independently checks on the usage of the existing resources and sets off the migration process when it is necessary. This was one of the first studies that used a distributed VM migration strategy which was reliable and scalable However, it was unable to change the threshold values depending on the load

and depended only on threshold values only. Thus, Cui et al. (2017) also used the ant colony algorithm for the scheduling process in the context of cloud computing. Other enhancements like the max-min and colony algorithm's enhanced version were established to emphasize execution time when balancing loadmaking it efficient.

These studies did not dwindle much on the heartbeat system and this is very essential in identifying faults in cluster systems. It is concluded that the quality of a heartbeat system has a strong correlation to the performance of high-availability clusters. When the system or network is congested, the heartbeat messages may take a while to be sent or received, and there are high chances that such messages may get lost, and hence those specific problems may go unnoticed and therefore the reliability of the system will have dropped. However, in order to detect the heartbeat more frequently, additional resources, namely the bandwidth and computing power are used. This was done by Hao et al. (2023) in a Nginx-based Dynamic Feedback Load Balancing Algorithm with Adaptive Heartbeat Detection where the cluster manager determines the load degree of the individual cluster node, using the following statistic; the rate of services, node response rate, CPU utilization and memory utilization. This feedback system was used to enhance load balancing because;

This work will be extended due to the focus on dynamic heartbeat detection in the Eureka servers using the algorithm different from the one employed in the beforementioned works, which does not utilize the controller server but instead relies on the data from the Virtual Machines. The heartbeat mechanism proved to be effective in diagnosing problems in cluster systems but if the network is congested, heartbeat messages could get dropped, delayed and therefore real time diagnosis of problems is difficult thus making the entire system less reliable. Hence, good design of smart load balancing system, which involves the variations of frequency of the heartbeat checks depending on the load is critical for optimum performance of the system.

2.3 Circuit Breakers and Retry Mechanism

Circuit breakers make microservice more reliable, but in the complex structure, it is seen that it can hamper throughput. To address this, retry controller can effectively retry on failures, improving on the overall system performance. Service response time, failed requests and state of the circuit breaker are some of the performance measures that feed the retry controller. Through the management of the above metrics, the controller ensures that the service performance, its reliability and availability are improved mostly to warrant a situations whereby users are served with timeouts or whereby they have to wait for long times due to failed requests. The circuit breaker design pattern is popular among software developers to detect failures and to avoid giving out timeout or gateway errors during maintenance or when the external systems are briefly, temporary outages or unexpected issues. This helps the system to fail faster when needed. This pattern was officially introduced in a research paper Michael (2007).The first library that practiced the above pattern was Hystrix – Java's code is wrapped in a mechanism managed by the circuit breaker. Surendro et al. (2021) offered a literature review on circuit breakers wherein the author presented the field and regarding potential future research. Circuit breakers improve microservices' resilience, but in intricate systems, they could reduce a rate of throughput. To address this, there can be a retry controller that can effectively retry the failed requests improving the general performance of the system.

As proposed in Sedghpour et al. (2023) retry controller utilising a circuit breaker to enhance microservice systems. The retry controller gets its data from service KPIs i.e., service Response Time, failed Requests, and circuit breaker status. Thus, by varying the retry settings according to these ratings, the controller increases the service availability, dependability, and efficiency, so that users do not experience the delay or timeouts because of failed requests. As in the Hystrix/Resilience4j, in Spring Cloud Gateway, when the circuit breaker is triggered, it affects the detection of heartbeats as the continuation of certain operations; it may additionally increase the failure rate of further heartbeats. This interruption may also pose a threat on the health checks of service and may lead to false alarms on the actual service failure or may lead to a delay in their discovery.

The above studies are based on creating an additional component such as a cluster manager for adaptive feedback and a circuit breaker controller. These approaches require a significant amount of data to be collected and processed, which may not be feasible in all scenarios. Also, machine learning and control theory algorithms require computational resources, which may impact the system's response time and throughput.
This research aims to fill this gap by optimizing the inbuilt retry and heartbeat mechanism by proposing a novel algorithm for dynamically adapting heartbeat mechanisms in Eureka service registry servers and optimizing circuit breakers and retry controllers in Spring Cloud environments.

The above studies are based on the development of a new component as a cluster managed extra layer of complexity to the problem of resource allocation by adding a cluster manager for adaptive feedback and a circuit breaker controller. These approaches require a certain amounts of data being to be collected and analyzed for which it may not be possible in all cases. Moreover, machine learning and control theory algorithms entail computational increasing or decreasing of the resources may affect the overall time response of the system and converter throughput. This research seeks to address this research question by enhancing the inbuilt retry and heartbeat mech- anism, a new algorithm for the heartbeat adaptation is suggested in Eureka service registry servers and fine tweaking circuit breakers as well as retry controllers in Spring Cloud environments.

3 Methodology

The aim of this research will be to assess the adherence and efficacy of a dynamic heartbeat algorithm for microservices using spring boot, spring cloud, eureka and Zuul. This algorithm tries to improve dependability and accessibility of the service by modifying the heartbeat interval depending upon the system load so that accurate detection and recovery could be made. Some of these components were Custom Eureka Client. A client that is able to send heartbeat to Eureka server with flexible time gap. And a Dynamic Heartbeat Service that decides on correct rate of heartbeat in relation to the current system load, (e.g. CPU usage, memory usage or response timings). The Dynamic Heartbeat mechanism is beneficial for managing and boosting microservices systems architectural and resource usage. Figure 2 shows the request handling and heartbeat requests from eureka server to all the microservices. Traditional static heartbeat intervals can lead to inefficiencies, frequent heartbeats may bring about traffic and load on the Eureka server whereas infrequent heartbeats may take time before it can discover the presence of problems in high load instances. The heartbeat intervals thus change with the real-time load

metrics which include CPU utilization, memory utilization and throughput of requests to the Eureka server. This makes it reduce on costs such as overhead, makes the service registry to be more responsive and at the same time makes the use of resources to be more efficient. As a result, it improves reliability and performance of microservices by extending a more robust and reliable architecture well suited for increasing loads flexibility and scalability.

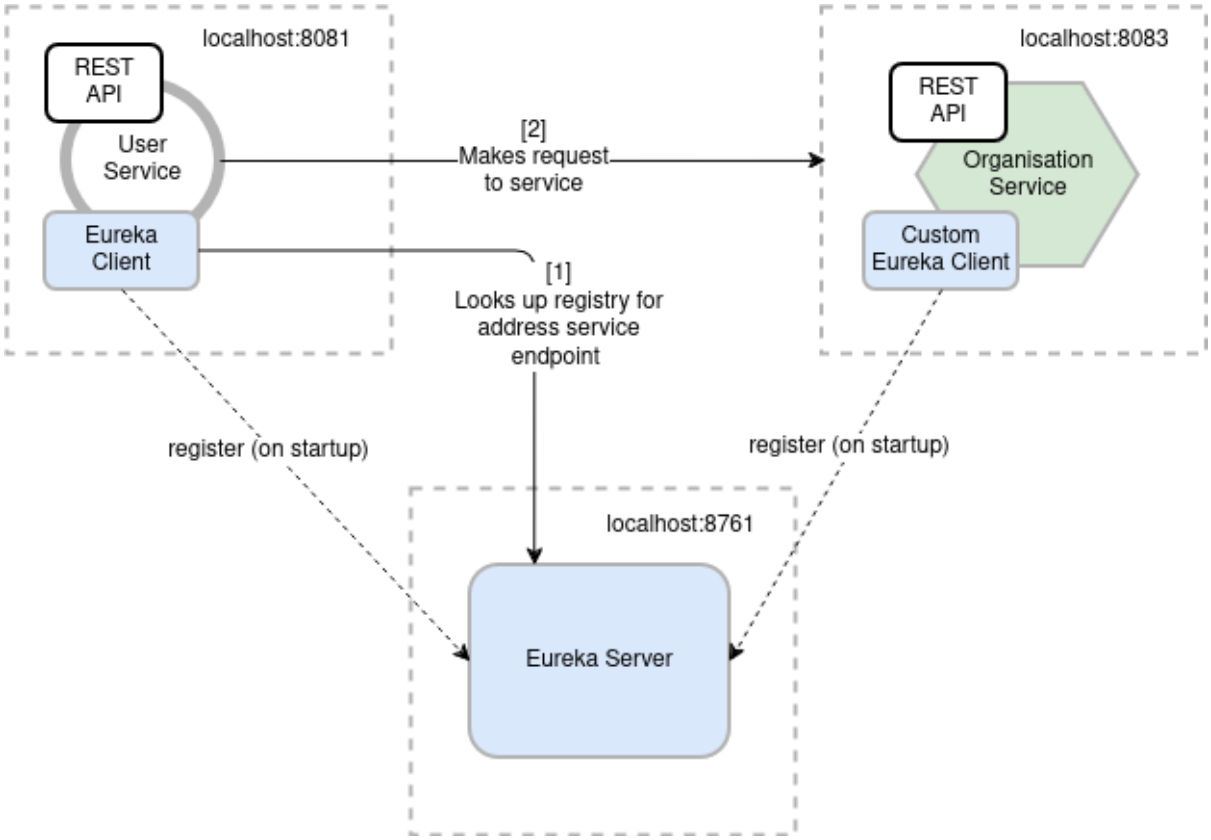

Figure 2: Discovery Service and Eureka Client Communication Architecture

3.1 Metric Collection

To the first means of applying the dynamic heartbeat detection system of the microservice architecture, each microservice is placed into an AWS EC2 instance to analyze its performance in real time. This is done with help of the OSBean library, which is rather useful for capturing a broad range of system characteristics on the level of JVM. The primary metrics collected include:The primary metrics collected include:

CPU Usage: This metric expresses the frequency of CPU usage as a percentage, offering insights into how much of the processing power of the structure is being utilized by the microservice at any one particular moment in time. High CPU usage usually points to the fact that the microservice in question is working on a large processing or handling large amounts of data or load in doing specific tasks or in solving problems.

Memory Usage: This is the total memory being used by the microservice as a way of determining memory consumption. Memory utilization is also important for getting information on the resource consumption of the microservice and guarantee that the

microservice runs within the allocated memory constraints to avoid memory printing and or out of memory conditions for the microservice.

Request Throughput: This is the rate of the number of requests served by the microservice, or, in other words, the frequency of the requests' flow. It is an essential metric of the microservice workload and performance that describes the microservice's readiness for traffic and request processing.

Gathering these metrics in real-time allows the system to have better and correct perception of the microservice status, which in turn is used as a foundation for dynamic heartbeat changes.

3.2 Weighted Average Load Calculation

After the various performance metrics have been obtained it is possible to estimate for each microservice a weighted average loads.

Assigning Weight Factors: Every of the above stated measurement is then given a weight factor depending on the proportionality of the every measurement the significance in the total load assessment has also been emphasised in this research. These weights are established depending on factors that may include the specific needs that the particular application will serve, and the nature of those applications. Because one would set different weights, the system on its own can be at the liberty of prioritizing some indices over others, and in turn can make an imbalanced load calculation. In this experiment the following weights have been used, this might not be the same in other experiments according to the specific application scenarios and important factors:

Weight CPU: 0. 5 which signifies that the utilization of CPU is vital out of all the parameters.

Weight Memory: 0. 3 because memory usage which is significant but not as important as CPU in this case.

Weight Throughput: 0. 2, thereby highlighting the authorities request handling capacity.

Calculating the Weighted Average Load: Using the assigned weights, the weighted average load is calculated with the help of the formula:

$$\begin{aligned} \text{Weighted Load} = &(\text{CPU Usage} \times \text{Weight}_{\text{CPU}}) \\ &+ (\text{Memory Usage} \times \text{Weight}_{\text{Memory}}) \\ &+ (\text{Throughput} \times \text{Weight}_{\text{Throughput}}) \end{aligned} \tag{1}$$

These values are summed up, to produce a general load value which provides an exact depiction of the microservice's status. Thus, the weighted average load is useful for making decision on changes of heartbeat interval.

3.3 Threshold Determination

To allow dynamic changes, a set of standard values is set to classify the load into different levels. The problem of load can be subdivided into different levels. These less quantities are necessary for the delimitation of certain borders between low, medium, and high load conditions:

Threshold Low: This threshold reflects a low degree of utilization and can mean that the microservice is idle or that it has been loaded well below its maximum allowable limit. Settings within this range seek to cut unwanted calls to the Eureka server by extending the interval of the heartbeat.
Threshold Medium: This threshold is that the microservice has average load, in other words, the load at which the microservice can work without affecting its performance due to a large number of requests. In this state, the current interval of heartbeat goes on the same with the intention to keep some standard and make continuous communication with the Eureka server.
Threshold High: This threshold means very high load, and it indicates that the microservice is almost at its limit in terms of capability. As for the fundamental function of the program to monitor whether the Eureka server can operate effectively, the heartbeat interval is lowered so that the handling of the heartbeat will be faster and the reaction to certain problems will be quicker.

These thresholds should be set more or less scientifically based on the measured data and characteristics of the microservices in question.

3.4 Heartbeat Interval Adjustment

Based on the calculated weighted load and the predefined thresholds, the system dynamically adjusts the heartbeat interval for each microservice. The adjustments are as follows:

- **If Load $<$ Threshold Low:** When the load is below the low load threshold, the above functions are implemented to increase the heartbeat interval. This is because heartbeats are transmitted less often meaning that overall the workload is lighter and less resources are used where the microservice is less active.

- **If Threshold Low $\leq$ Load $<$ Threshold Medium:** When the load is of the medium value the current interval between the heartbeats is preserved. This is important in minimizing the changes that are sought to be introduced to the existing system so that their stability is enhanced.

- **If Threshold Medium $\leq$ Load $<$ Threshold High:** If the amount of the load is over the medium threshold and below the high threshold, the heartbeat interval is reduced. This leads to more frequent heartbeats enabling the Eureka server make updates as often as possible about the microservice state and possible performance problems that need to be dealt with immediately.

This adaptive mechanism means that the communication with the Eureka server is going to be dynamic, thus making the best out of actual load in terms of performance as well as resource consumption.

3.5 Custom Logging

For the support of the dynamic heartbeat mechanism new specific logging techniques are implemented. These logs capture detailed information about the system's operation, including:

Real-time Metrics Data: Logs include the performance data gathered at runtime , and provide a complete response from each microservice, indicating the current state of the

system at any given moment. Adjustments Made to the Heartbeat Intervals: Logs record all the changes that occurs to the intervals between the beats, the changes depending on the load thresholds.

4 Design Specification

The distributed version of an application with two microservices developed using the Springboot framework is taken as the application for the setup. It utilizes Spring Boot for building microservices and consists of eight services - Discovery Server, Zuul Gateway Server, Organisation Service and User Service. This contains Spring Cloud for making cloud native patterns such as the service discovery with eureka server and client-sie load balancing with ribbon and Zuul Gateway for load balancing. The communication between the microservices is done by REST API.

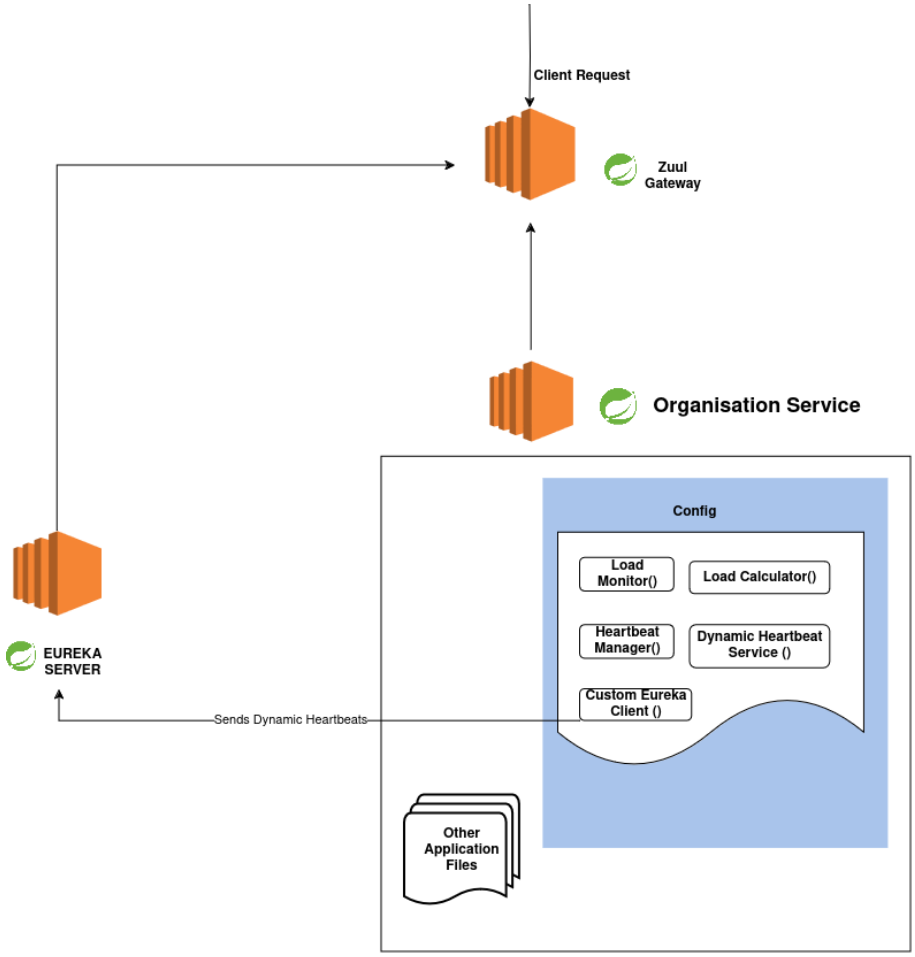

Figure 3: Architecture of proposed algorithm

Figure 3depicts the request flow of the implemented architecture and the configuration file structure of the dynamic heartbeat algorithm within the microservice - Organisation Service. The metrices for the mechanism is extracted from the com.sun.management package which is the OperatingSystemMXBean. This is then takes the real time resource utilisation data. The mechanism is built as a config file within the main spring file and consists of the five functions required for making this dynamic algorithm which are the LoadMonitor, LoadCalculator, HeartbeatManager, CustomEurekaClient, DynamicHeartbeartService. Detailed file structure of the created mechanism is shown in Figure 4

Figure 4: Springboot Application File Structure with the dynamic heartbeat functionality defined in the Config Folder

5 Implementation

The dynamic heartbeat approach involves several components and steps to ensure efficient and responsive communication with the Eureka server.

5.1 Initialize Components

The first step in the dynamic heartbeat algorithm is to initialize the necessary components. This involves instantiating several key classes that will work together to manage

and adjust the heartbeat intervals based on system load. The components include:

LoadMonitor: This component monitors system metrics such as CPU usage, memory usage, and request throughput.

LoadCalculator: Utilizing the data from LoadMonitor, this component calculates a load score that reflects the current system load.

HeartbeatManager: Based on the load score from LoadCalculator, this component determines the optimal heartbeat interval.

CustomEurekaClient: This component is responsible for sending heartbeats to the Eureka server and updating the metadata with heartbeat details.

Once these components are instantiated, they are integrated into the DynamicHeartbeatService, which will manage the overall heartbeat scheduling process.

Each microservice, equipped with the dynamic heartbeat mechanism, is deployed to an AWS EC2 instance. The dynamic heartbeat algorithm is implemented within a Spring Boot microservice architecture as part of the microservices configuration code. Although metrics could be collected directly from the cloud instances, here the OSBean library is choosen for its direct integration capabilities and finer granularity in metric collection, ensuring precise and real-time adjustments. Additionally, detailed custom logging is implemented to capture logs and outputs, providing comprehensive insights into system performance and facilitating troubleshooting and optimization.

5.2 Start Heartbeat Service

When the application starts, a **ContextRefreshedEvent** is triggered. This event is handled by the DynamicHeartbeatService, which invokes its start method. The start method is crucial as it initiates the heartbeat scheduling process. This ensures that the heartbeat mechanism is set up and ready to adapt to the system load from the moment the application becomes operational.

Schedule First Heartbeat

Upon starting, the DynamicHeartbeatService calls the **scheduleNextHeartbeat** method. This method is responsible for determining the initial heartbeat interval. It does this by consulting the **HeartbeatManager**, which calculates the interval based on the current system load. But for the first heartbeat is scheduled to start at 30 seconds which is the maximum when the service registers with the eureka server. This sets the stage for the dynamic adjustment of heartbeat intervals.

5.3 Heartbeat Interval and Scheduling

The **HeartbeatManager** class calculated the heartbeat interval. The HeartbeatManager used the **LoadCalculator()** to assess the system load. The LoadCalculator()

gathered data on CPU usage, memory usage, and request throughput from the LoadMonitor().
Based on the load score, if the load score is less than 20, the interval is set to the maximum value of 30,000 milliseconds. If the load score exceeds 80, the interval is set to the minimum value of 1,000 milliseconds. For load scores between 20 and 80, a proportional interval is calculated to ensure a smooth adjustment based on the load. This adaptive approach ensured that the system can react promptly to varying load conditions, optimizing resource usage and responsiveness.

When it is time to send a heartbeat, the DynamicHeartbeat Service logs the action and calls the **sendHeartbeat()** method of the **'CustomEurekaClient'**. This method sends the heartbeat to the Eureka server to signify that the service is active. Updates the instance metadata with the current timestamp and the interval used for the heartbeat.
After a heartbeat is sent, the DynamicHeartbeatService immediately calls scheduleNextHeartbeat again. This method recalculates the next interval using the updated load metrics and schedules the subsequent heartbeat. The sendHeartbeat method logs the action of sending a heartbeat and delegates the actual sending to the CustomEurekaClient class. This loop continues indefinitely, ensuring that heartbeat intervals are continuously adjusted based on real-time system load.

Logging at the application level ensured the record of the heartbeat events and the intervals at they are sent.

6 Evaluation

Experiment has been conducted by using AWS EC2 services with Ubuntu Server 20.04 LTS. For evaluation, Eureka service was deployed on one EC2 instance, a Springboot microservice application - 'User service' without the dynamic heartbeat features was deployed on an EC2 instance. Another service named 'Organisation Service' was deployed on a EC2 instance. The Organisation service code has a custom eureka client and the necessary classes for the dynamic heartbeats as mentioned in the implementation section. These two independent microservices (User and Organization), which register themself in service discovery (Eureka Server), and communicate with each other by declarative REST client (Open Feign). The whole system is hidden behind Zuul API gateway which is also deployed on an ec2 instance. Zuul API Gateway will forward the request to the specific microservice based on its proxy configuration. Such request will also be load balances by ribbon client. Figure 5 depicts the architecture of Springboot Mircoservices Application created for the experiment. The evaluation involved running multiple test scenarios to compare the dynamic heartbeat algorithm against a fixed interval heartbeat approach. The scenarios included a baseline test which included running the microservices with a fixed heartbeat interval of 30 seconds. Dynamic Heartbeat Test included running the microservices with the dynamic heartbeat algorithm enabled. The experiment conducted was a load testing experiment using Apache JMeter.

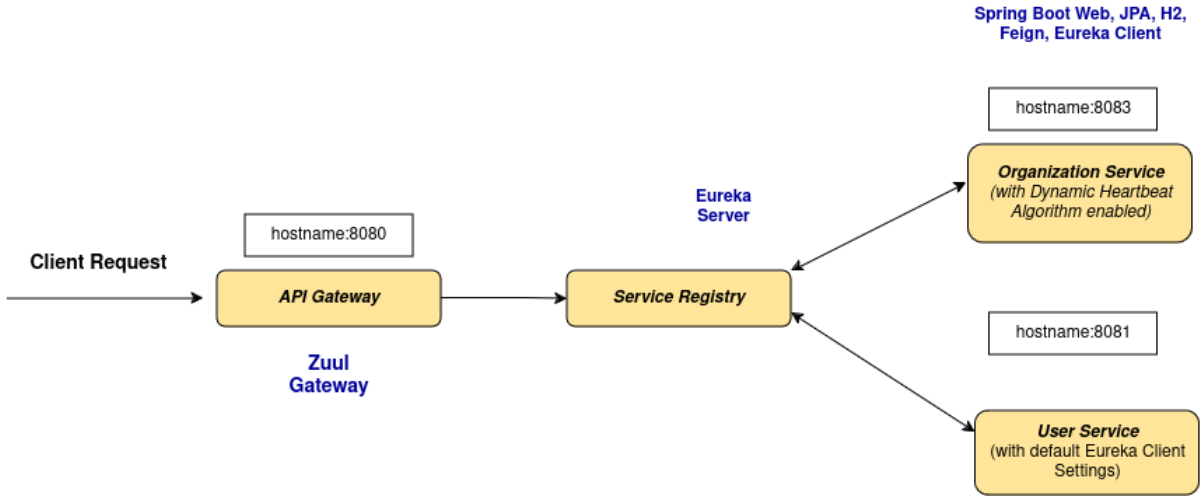

Figure 5: Architecture of Springboot Mircoservices Application created for the experiment.

6.1 Baseline Test (Fixed Interval)

In the baseline test scenario, the microservices were configured to operate with a fixed heartbeat interval of 30000 ms. Spring Cloud and Netflix Eureka,frameworks for microservices architecture come with default configurations that include a fixed heartbeat interval for service registry and discovery. In this setup, each microservice sends a heartbeat signal at regular intervals (30 seconds) to the Eureka server to confirm its availability and health status. Figure 6 shows the results of a default setting. This setup was considered in this experiment to serve as a control scenario, providing a stable reference point for comparison against dynamic heartbeat intervals.

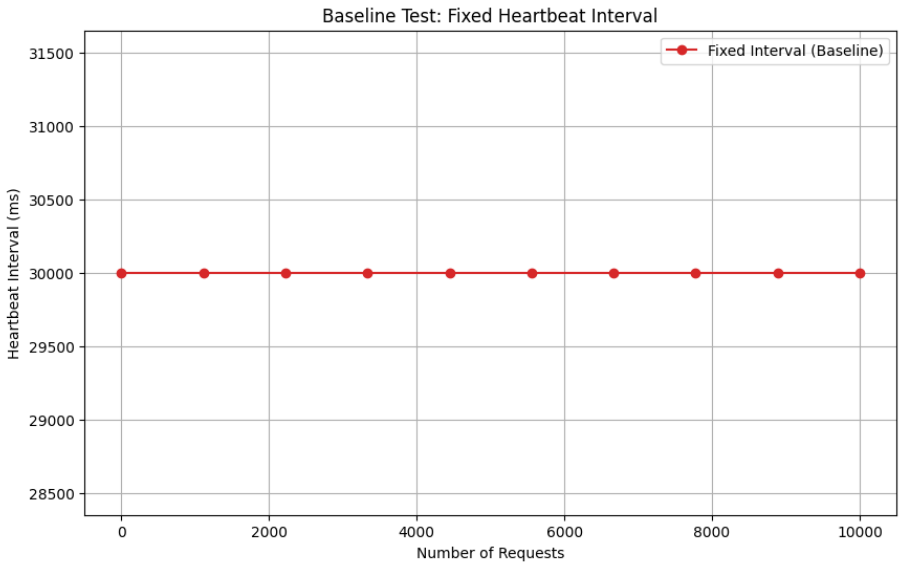

Figure 6: Fixed Interval Test (Default)

6.2 Case Study 2 - Sudden Increase in traffic

In the Jmeter for testing the Thread Group was configured with 50 virtual users, a ramp-up period of 172 seconds, and a loop count of 1. These values allowed a gradual increase in load. Figure 7 depicts the graphical representation of this outcome.

Table 1: Dynamic Heartbeat Intervals Observed

Interval (ms)	Approximate Time (sec)	Approximate Number of Requests
10972	11	10000
8652	9	9000
8650	9	8000
8596	9	7000
8594	9	6000
12483	12.5	5000
17216	17	4000
22884	23	3000
22641	23	2000
23009	23	1000

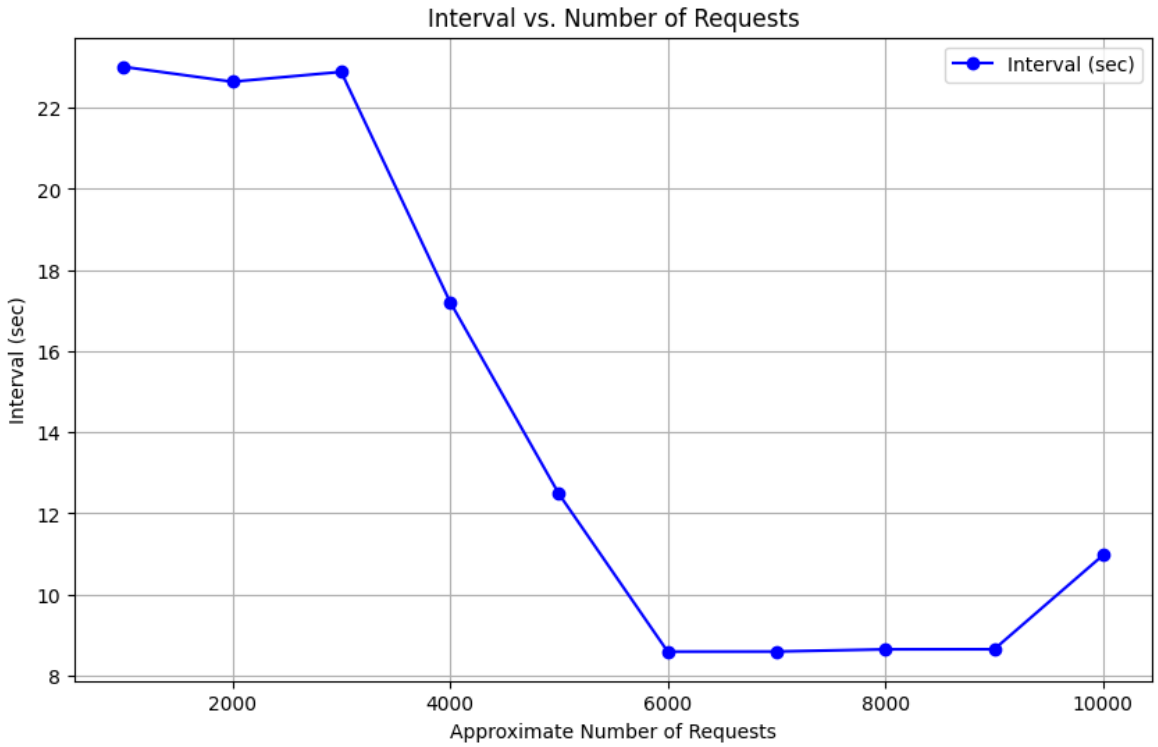

Figure 7: Sudden Increase in traffic

Key Observations

Initial High Frequency: The initial intervals were significantly shorter, ranging from 9 to 12.5 seconds. This indicates a higher frequency of heartbeats in the early stages of the test, due to the higher initial load.

Adaptive Mechanism: As the load decreased, the intervals between heartbeats gradually increased. This adaptive behavior suggests that the system adjusts the heartbeat frequency to balance load and performance, responding quickily to any failure during high load conditions.

Transition Period: There was a noticeable transition period where the intervals shifted from approximately 9 seconds to 23 seconds. This indicates a dynamic adjustment phase where the system responds to increasing load by extending the intervals.

Stabilized Intervals: In the latter part of the test, the intervals stabilized around 23 seconds. While this is still below the static interval of 30 seconds, it demonstrates the system's capability to maintain a balanced state under sustained high load.

6.3 Case Study 3 - Gradual Increase in Load

For gradual increase in the load the JMeter was configured with a maximum of 10,000 virtual Users and ramp-up period of 540 seconds.

Table 2: Test Results with Gradual Increase in Requests

Interval (ms)	Approximate Time (sec)	Approximate Number of Requests
22543	22.5	500
21567	21.6	1000
19876	19.9	2000
18734	18.7	3000
16045	16.0	4000
14567	14.6	5000
12789	12.8	6000
10987	11.0	7000
8954	9.0	8000
7543	7.5	10000

Key Observations

Decreasing Intervals with Increasing Load: The interval duration decreases as the number of requests increases. This indicates that the system reduces the time between heartbeats as the load increases to maintain performance and responsiveness.

Adaptive Behavior: The system demonstrates adaptive behavior by dynamically adjusting heartbeat intervals based on load. Shorter intervals during higher loads help the system monitor its state more frequently, allowing for quicker detection and response to issues.

Non-linear Adjustment: The above modifications to the intervals are periodic, with steep declines in the interval's length as the number of requests increases. The interval is extended or reduced by about 8 seconds from the first to the last interval in order to demonstrate that the system requires updates at greater frequency under elevated load.

Balance Between Load and Monitoring Overhead: To address this the system is composed in a way that offers high monitoring frequency on the heartbeats when the system is experiencing high loads this is balanced against the overall cost incurred on processing these heartbeats. Such balance helps prevent system performance from degrading due to effect of frequent heartbeat messages. Figure 8 depicts the graphical representation of this outcome.

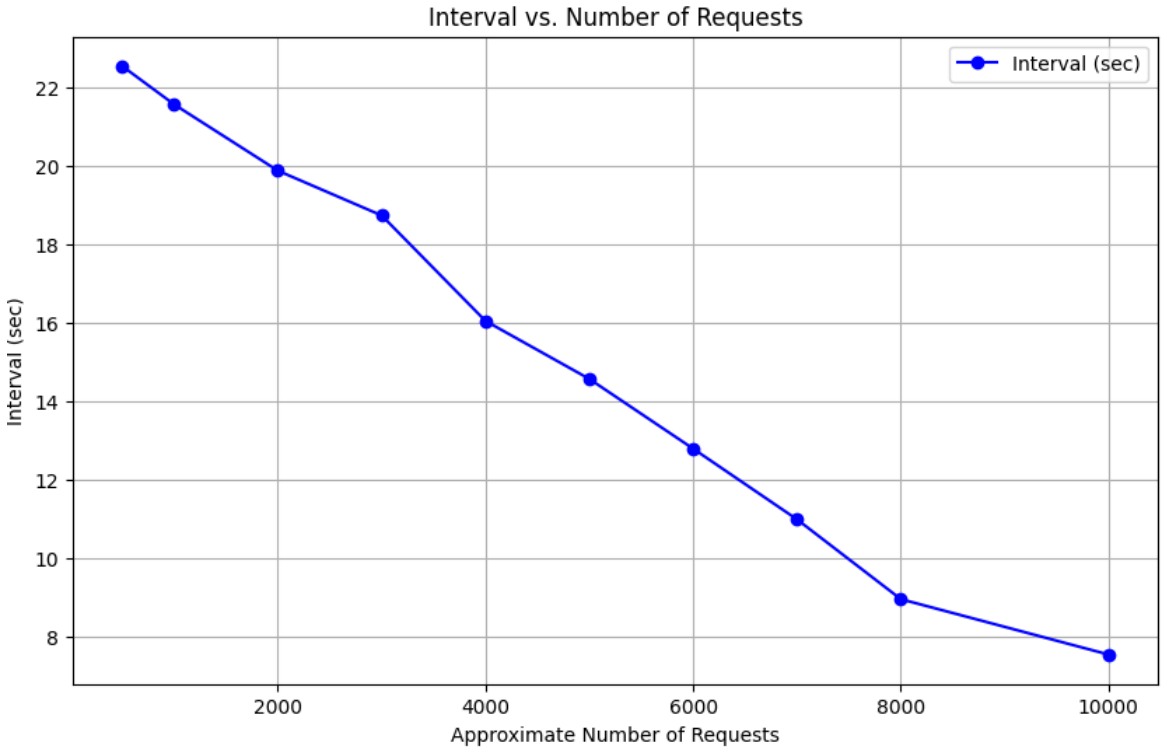

Figure 8: Gradual Increase in Load

6.4 Discussion

The experiments were planned in a way that woud allow for the examination of the time-varying dynamics of the heartbeat intervals and their consequences That can be attributed to differences in the detection speed and system resilience of the two. The examined dynamic intervals varied within the span of about 9-23 seconds. This means that in periods of high loads, **the system has switch to the short interval to quickly detect problem during the high load improves the system's resilience.** The system's default setting of 30 seconds for heartbeat intervals means issues can only be detected at this interval, potentially delaying critical responses and actions in a high-load environment. This fixed setting may not be adequate for high-load scenarios where quicker detection is necessary. The ability to detect issues as quickly as every **9 seconds**. The minimum 9 seconds is from results of our test scenarios but the sample size and duration of the experiments were limited. Conducting the tests over a longer period and with varying conditions could provide more robust results. Also, this experiments primarily focused on high-load scenarios, the results could further vary if various failure scenarios, such as network partitioning or service crashes will be considered. This is crucial for cloud applications using the Spring Cloud Gateway Framework, as it helps in maintaining service availability and reliability by quickly identifying and addressing issues.

7 Conclusion and Future Work

The main objective of this study and experiment was to optimise and create an adaptive feature in the discovery registry for the microservices created using spring framework. The study was conducted to see if the retry mechanisms and the heartbeat detection be set up to improve the resilience of the Spring Cloud Gateway which utilizes circuit breaker strategy. Retry mechanisms and heartbeat detection are essential for keeping distributed systems reliable and available. When a problem is detected through heartbeat monitoring, a retry mechanism can automatically try to fix it by resending the request or taking a different action. This helps the system recover quickly from temporary issues without needing someone to step in manually. Spring Cloud Gateway uses Eureka for finding and routing services, but the retry mechanism itself doesn't get availability information directly from Eureka. Instead, it uses local settings, like how many times to retry, how long to wait between retries, and under what conditions to retry.

In this setup, Spring Cloud Gateway and Eureka work together to create a stronger and more reliable communication system. By adding Spring Cloud Circuit Breaker to Spring Cloud Gateway and using the metrics collected by the heartbeat detection in Eureka, the system can decide whether to retry requests or use a backup plan, ensuring that services remain available even when there are problems.

It is evident from the experiments performed that the algorithm has significant responsiveness in showing the availablity of the service but there could be certain limitations in the real-world scenarios. Since the dynamic heartbeat algorithm is tightly coupled with the Eureka for service registraion and heartbeat monitoring , this approach has a high dependency on the Eureka Server. Therefore if the environments use some other service registries such as Consul or Zookeeper, this approach would require a lot of modifications or could be not applicable at all.

Even though the dynamic adjusment of heartbeat intervals optimize resource usage, there is high chance of introducing some overhead since LoadMonitor is continuously gathering system metrics which in turn could add to the CPU and memory load. Although the overheads are negligible during the experiments, it should be considered when deployed in resource constrained environments.

For future work this approaches performance impact and overall system latency should be studied because This system needs to scale proportionally as the deployed system scales. With hundreds or thousands of microservices, continuous adjustment might bring scaling challenges. Further study has to be conducted for getting the right balance between stability and responsiveness of the system because there is risk of over optimisation. In some scenarios the algorithm could adjust too frequently leading to system instability or could result in more logging and monitoring unnecessarily.

Since there is a growing demand for cloud native application and microservices architecture are a popular approach in cloud native applications where resilience is very important, this approach could prove helpful and innovative in these environments.

References

Adhikari, V. K., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M. and Zhang, Z.-L. (2012). Unreeling netflix: Understanding and improving multi-cdn movie delivery, *2012 Proceedings IEEE Infocom*, IEEE, pp. 1620–1628.

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

Cui, H., Liu, X., Yu, T., Zhang, H., Fang, Y., Xia, Z. et al. (2017). Cloud service scheduling algorithm research and optimization, *Security and Communication Networks* **2017**.

Hao, G., Qiongbing, Z., Xuan, L. and Junchao, C. (2023). A nginx-based dynamic feedback load balancing algorithm with adaptive heartbeat detecting, *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, pp. 673–679.

Hlybovets, A. and Paprotskyi, I. (2024). Increasing the fault tolerance in microservice architecture, *Cybernetics and Systems Analysis* pp. 1–9.

Michael, N. (2007). Release it!–design and deploy production-ready software.

Rahman, M., Iqbal, S. and Gao, J. (2014). Load balancer as a service in cloud computing, *2014 IEEE 8th international symposium on service oriented system engineering*, IEEE, pp. 204–211.

Rui, X., Wu, J., Zhao, J. and Khamesinia, M. S. (2021). Load balancing in the internet of things using fuzzy logic and shark smell optimization algorithm, *Circuit World* **47**(4): 335–344.

Sedghpour, M. R. S., Garlan, D., Schmerl, B., Klein, C. and Tordsson, J. (2023). Breaking the vicious circle: Self-adaptive microservice circuit breaking and retry, *2023 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, pp. 32–42.

Selvakumar, G., Jayashree, L. and Arumugam, S. (2023). Latency minimization using an adaptive load balancing technique in microservices applications., *Comput. Syst. Sci. Eng.* **46**(1): 1215–1231.

Surendro, K., Sunindyo, W. D. et al. (2021). Circuit breaker in microservices: State of the art and future prospects, *IOP Conference Series: Materials Science and Engineering*, Vol. 1077, IOP Publishing, p. 012065.

Wen, W.-T., Wang, C.-D., Wu, D.-S. and Xie, Y.-Y. (2015). An aco-based scheduling strategy on load balancing in cloud computing environment, *2015 Ninth international conference on frontier of computer science and technology*, IEEE, pp. 364–369.