Pontifícia Universidade Católica
do Rio de Janeiro

Matheus Cunha Penso

Using Kubernetes for elasticity and load balancing of ContextNet Core Gateways for scalable mobile connectivity

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor : Prof. Markus Endler
Co-advisor: Prof. Alexandre Malheiros Meslin

Rio de Janeiro
April 2024

Pontifícia Universidade Católica
do Rio de Janeiro

Matheus Cunha Penso

Using Kubernetes for elasticity and load balancing of ContextNet Core Gateways for scalable mobile connectivity

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee:

Prof. Markus Endler
Advisor
Departamento de Informática – PUC-Rio

Prof. Alexandre Malheiros Meslin
PUC-Rio

Prof. Anderson Oliveira da Silva
PUC-Rio

Prof. Sérgio Colcher
PUC-Rio

Rio de Janeiro, April 19th, 2024

Matheus Cunha Penso

Graduated in computer engineering from PUC-Rio. Currently working as a Software Engineer at Globo.

Bibliographic data

Penso,Matheus

Using Kubernetes for elasticity and load balancing of ContextNet Core Gateways for scalable mobile connectivity / Matheus Cunha Penso; advisor: Markus Endler; co-advisor: Alexandre Malheiros Meslin. – 2024.

60 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2024.

Inclui bibliografia

1. Informática – Teses. 2. comunicação escalável. 3. IoT móvel. 4. Kubernetes. 5. cloud IoT. 6. middleware IoT. I. Endler, Markus. II. Meslin, Alexandre. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

To my family and friends, for their support and encouragement.

Acknowledgments

To my advisor Markus Endler and co-advisor Alexandre Meslin, for their patience and assistance
To the other professors at PUC-Rio, especially Noemi Rodriguez for all the teaching
To my girlfriend, Rafaela Carneiro, for her patience and understanding
To my family for supporting and motivating me
To everyone who worked with me at Globo, especially Sérgio Nazareth and Rafael Almeida, who are professional references for me
This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Penso,Matheus; Endler, Markus (Advisor); Meslin, Alexandre (Co-Advisor). **Using Kubernetes for elasticity and load balancing of ContextNet Core Gateways for scalable mobile connectivity**. Rio de Janeiro, 2024. 60p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

As the Internet of Things (IoT) increasingly incorporates mobile devices and objects, this also calls for scalable services capable of handling a growing number of concurrently connected mobile devices. Consequently, the ability to provide reliable services that are adaptable to different scenarios, efficient, and high-performing in a highly mobile environment is crucial to meet user expectations and drive widespread adoption of IoT mobile applications (IoMT). In this work, we design and implement a self-scalable and configurable architecture, allowing application administrators to configure scalability parameters according to their needs, using Kubernetes in ContextNet, a distributed IoMT middleware, and evaluate the performance of our implementation in different scalability and mobility scenarios.

Keywords

Scalable Communication; Mobile IoT; Kubernetes; Cloud-based IoT; Service Middleware.

Resumo

Penso,Matheus; Endler, Markus; Meslin, Alexandre. **Usando Kubernetes para incluir elasticidade e balanceamento de carga em Gateways do ContextNet Core visando escalabilidade de conexões móveis**. Rio de Janeiro, 2024. 60p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

À medida que a internet das coisas (IoT) incorpora cada vez mais dispositivos móveis e objetos, isso também demanda serviços escaláveis capazes de lidar com um número crescente de dispositivos móveis conectados simultaneamente. Como resultado, a capacidade de oferecer serviços confiáveis, que sejam adaptáveis a diferentes cenários, eficientes e de alto desempenho em um ambiente altamente móvel é crucial para atender às expectativas dos usuários e impulsionar a adoção em massa de aplicações móveis IoT (IoMT). Neste trabalho, projetamos e implementamos uma arquitetura autoescalável e configurável, de maneira que o administrador das aplicações consiga configurar parâmetros de escalabilidade de acordo com a necessidade, usando o Kubernetes no ContextNet, um middleware distribuído IoMT, e avaliamos o desempenho de nossa implementação em diferentes cenários de escalabilidade e mobilidade.

Palavras-chave

comunicação escalável; IoT móvel; Kubernetes; cloud IoT; middleware IoT.

Table of contents

List of figures

List of tables

List of algorithms

List of codes

List of Abreviations

IoT – Internet of Things
IoMT – Internet of Mobile Things
OMG-DDS – Object Management Group - Data Distribution Service
M-OBJ – Mobile Object
M-HUB – Mobile Hub
MR-UDP – Mobile Reliable User Datagram Protocol
MTD – Mobile Temporary Disconnect
MAPE – Monitor, Analyze, Plan, Execute
API – Application Programming Interface
CLI – Command Line Interface
HPA – Horizontal Pod Autoscaler
VPA – Vertical Pod Autoscaler
IP – Internet Protocol
GCP – Google Cloud Platform
DNS – Domain Name System
RAM – Random Access Memory
PoA – Point of Access
JVM – Java Virtual Machine
IPVS – IP Virtual Server
OSI – Open System Interconnection
LVS – Linux Virtual Server
TCP – Transmission Control Protocol
UDP – User Datagram Protocol
CPU – Central Processing Unit
JSON – JavaScript Object Notation
VM – Virtual Machine
HTTP – Hypertext Transfer Protocol
QoS – Quality of Service

1
Introduction

Mobility has become a fundamental part of our daily lives, and the Internet of Things (IoT) plays a central role in this context, primarily through integrating mobile devices. More specifically, the Internet of Mobile Things, or IoMT, encompasses the connection of smartphones, tablets, smartwatches, mobile sensors, and other mobile devices in vehicles, drones, etc., to the Internet, providing a series of functionalities and conveniences in our daily lives. This integration allows us to use our mobile devices to monitor our health and physical activity, receive real-time notifications, control home devices remotely, and much more.

The concept of the Internet of Things, together with mobility and constant connectivity, are foundational factors for smart cities. Thus, smart cities host various applications, whether public or private, aimed at collecting data through sensors and transmitting it for decision-making. Nowadays, we can see applications that use location data to track public bus networks, use location to provide real-time information on traffic conditions, work in agricultural control through monitoring via sensors of weather conditions, soil moisture, and irrigation control, and provide mobile payment services.

However, an infrastructure is needed to support mobility and scale in this distributed and growing ecosystem. As mentioned by (MESLIN; RODRIGUEZ; ENDLER, 2020) and discussed by (DELICATO et al., 2017), an IoT ecosystem is organized into three layers: (i) the lower layer consisting of devices; (ii) the upper layer consisting of static nodes, which can use cloud computing; (iii) an intermediate layer that functions with edge nodes facilitating communication between the other two layers.

As layer (i) grows with adopting IoMT in a smart city, the other layers must adapt to support this demand. That is, layer (ii) needs to have more processing power if necessary, and layer (iii) needs to be able to support a greater number of connections and more processing in the case of edge computing.

Thus, as described by (MESLIN; RODRIGUEZ; ENDLER, 2020), mobile nodes rely on these edge gateways to connect to the Internet, and an infrastructure that efficiently supports mobile users must be able to migrate connections between gateways given the mobility of the nodes and must be able to distribute the load effectively among the available gateways.

In the studies conducted by (TALAVERA et al., 2015) and (ELAZHARY,

2019), infrastructures for IoMT applications were discussed. These infrastructures utilize mobile nodes (e.g. smartphones) as gateways to connect devices to the Internet. These mobile nodes, categorized within the described layer (i), establish connections with gateways belonging to layer (iii) to transmit data from the connected devices.

Therefore, looking exclusively at the connectivity between mobile nodes and stationary gateways in scenarios of demand volatility, and aiming to ensure availability for various types of applications that a smart city may have, we identified the following problem:

Problem 1
To support the variations in the demand for mobile nodes' connectivity, it is necessary to scale the static gateways so that they can handle these mobile-cloud connections proportionally while ensuring that the elasticity of static gateways is configurable to adapt to different scenarios.

With this focus, the objective of this study is to address the following research question:

Research question 1
How can we create a self-scalable gateway architecture that does not impact established connections and mobile-cloud availability, and allows the system administrator to configure scalability parameters as needed?

To answer this question, this research will extend the previous work described in (WANOUS, 2021), which uses ContextNet, a home-brewed IoMT distributed middleware. Our contribution will be the design and implementation of a self-scalable architecture for mobile communications using Kubernetes, allowing us to evaluate the performance of our solution in different scalability and mobility scenarios (MURALIDHARAN; SONG; KO, 2019; ERMOLENKO et al., 2021; KAYAL, 2020).

2
Fundamentals

In this chapter, we will discuss the key concepts needed to understand our work separately. First, we will talk about ContextNet, which is the middleware we will use as a case study in the development and evaluation of our work. Next, we will briefly discuss autoscaling as it forms the basis for building an elastic architecture. Finally, we will discuss Kubernetes, as it is the tool we will use to deploy our application, make it elastic, and perform load balancing.

2.1
ContextNet middleware

ContextNet (ENDLER; SILVA, 2018) is a mobile-cloud middleware for IoT that can handle a large and dynamic number of mobile nodes with intermittent connectivity and provide unicast, groupcast, and broadcast communication between the ContextNet Core that runs in the cloud and the mobile devices or smartphones. The initial version of the middleware adopted an architecture that employed OMG DDS (Data Distribution Service) (PARDO-CASTELLOTE, 2003) with a single-topic Publish/Subscribe (EUGSTER et al., 2003) communication channel. However, as it evolved, we arrived at a new version of the middleware (ContextNet 3.0) (WANOUS, 2021) that uses Kafka (KREPS et al., 2011) in its core services, similar to the research seen in (SOUSA et al., 2018).

To understand how the middleware works, we can divide ContextNet into three groups of nodes, which are stationary and/or mobile, as can be seen in **Figure 2.1**. The first group represents the core nodes, which are stationary and responsible for running the ContextNet communication core services and processing nodes. At the other end of the communication are the Mobile Objects (M-OBJ), which represent peripheral IoT smart devices that have sensors or actuators, and that are accessible through Wireless Personal Area Network (WPAN) technologies independent of whether they are mobile. To bridge the communication between the M-OBJs and the ContextNet core, we employ ordinary smartphones or any mobile device capable of running Android, executing our Mobile-hub (M-HUB) middleware on top of Android. These distributed Mobile-hub nodes, which can also be stationary or mobile, must have an Internet connection (4G/5G/Wi-fi), that can be intermitent, for interacting with the core through and with WWAN and WBAN interfaces.

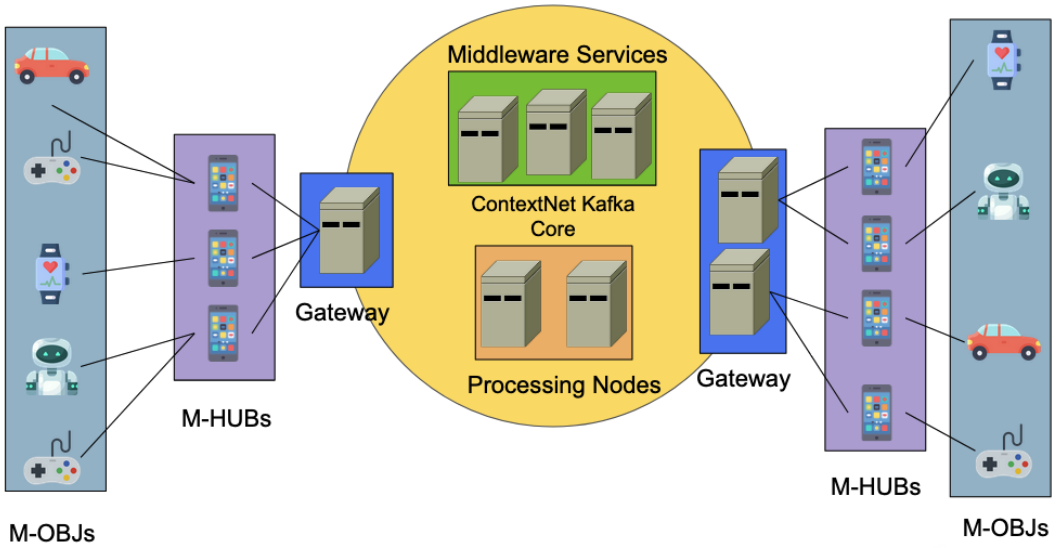


Figure 2.1: Example of mobile nodes and M-OBJs connected to and interacting through a ContextNet Core

ContextNet's core, represented by yellow circle in **Figure 2.1**, comprises microservices that interact through Publish/Subscribe on multiple topics using Kafka. Essentially, ContextNet's core runs the following infrastructure microservices:

- *Gateway*: service responsible for the connection between ContextNet's *core* and the mobile nodes, through the MR-UDP protocol (SILVA; ENDLER; RORIZ, 2013).
- *PoA Manager*: service responsible for load balancing between the Gateway nodes.
- *Mobile Temporary Disconnect* (MTD): service for temporarily storing messages addressed to mobile nodes that have their connections interrupted, and automatic re-sending of the messages as soon as the mobile node reconnects
- *Group Definer*: service responsible for grouping mobile nodes according to some context attribute (e.g., current position) and sending *groupcast* messages to all nodes whose attribute satisfies certain restrictions (e.g., all nodes placed within certain geographic boundaries).

ContextNet application developers can integrate their applications into the Kafka layer to use the middleware in a way that decouples the core. Therefore, the developer is responsible for the application on the mobile nodes that will send the data to the core and for the cloud application that processes the sent data.

Using Kafka as an asynchronous communication layer means that ContextNet can deal with concurrency, lack of global synchronization, and independent failures, essential points for a distributed system (COULOURIS; DOLLIMORE; KINDBERG, 2001). Firstly, ContextNet's architecture ensures that processes can run concurrently since each is a microservice. It also ensures that there is no need for global synchrony between them since they communicate through the message queue, which by its nature brings this asynchronous approach. Finally, Kafka's native replication mechanism ensures that ContextNet is fault-tolerant, so the failure of one of the microservices does not fail the system as a whole.

In **Figure 2.2**, we can see the connection flow of mobile nodes to the gateway through the MR-UDP (SILVA; ENDLER; RORIZ, 2013) protocol, the developers' applications connected to the Kafka layer, and the ContextNet Core layer, which is also integrated into the Publish/Subscribe communication model. Despite being containerized, this core layer lacks any automatic scaling strategy in the current middleware version.

In order to investigate the research question, we will utilize the ContextNet 3.0 gateway as a use case for implementing an testing our autoscaling architecture.

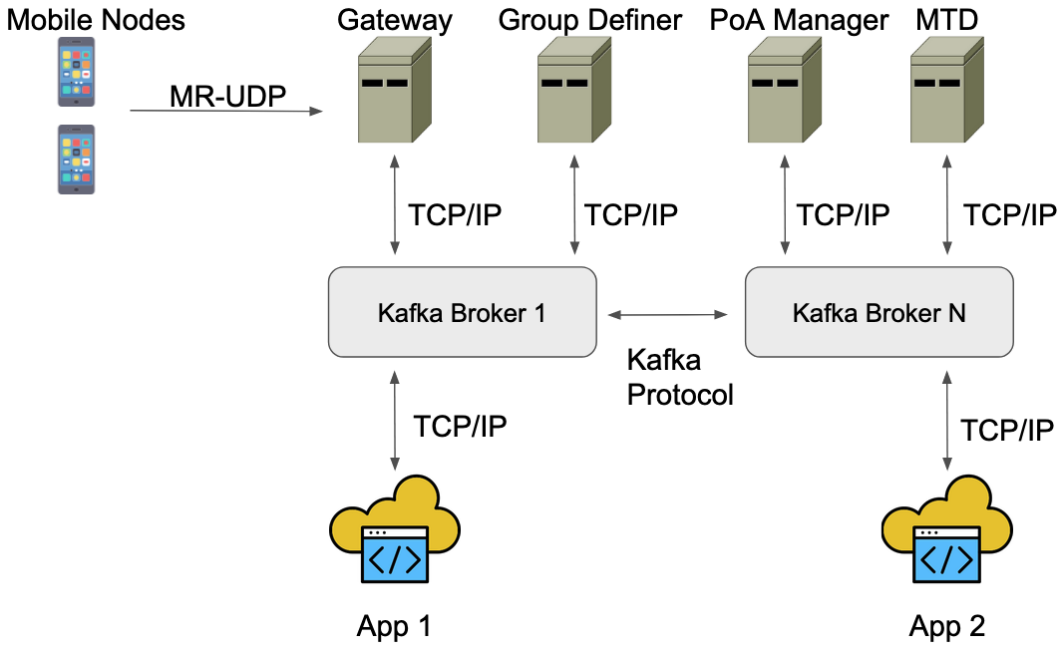


Figure 2.2: ContextNet 3.0 system

2.2 Autoscaling

Autoscaling aims to dynamically adjust processing resources based on application demand. It ensures that more resources are allocated at times of higher demand and deallocated at times of lower demand. In this way, it guarantees high availability for the application and efficiency of the resources used.

In their review, (LORIDO-BOTRAN; MIGUEL-ALONSO; LOZANO, 2014) explain that the autoscaling process adheres to the MAPE cycle. This cycle, consisting of four phases (Monitor, Analyze, Plan, Execute), guides the system in automatically adjusting resources. The first phase is monitoring, in which a system collects data on the monitored resources. The autoscaling system then analyzes this data, which uses a predefined rule to plan an action. The granularity and quality of the monitoring data directly impact the performance of the autoscaling system, so the data collected for analysis must be reliable and readily available. As classified by (LORIDO-BOTRAN; MIGUEL-ALONSO; LOZANO, 2014), the different autoscaling techniques can fall into at least one of the following groups: Threshold-based rules, Reinforcement learning, Queuing theory, and Time series analysis. The following section will explain how we applied the threshold-based rule technique to our work, the only one available in Kubernetes.

2.3 Kubernetes

Kubernetes (BURNS et al., 2016; BERNSTEIN, 2014) is an open-source platform that allows containerized applications' automation, management, and scalability, generally used with Docker.

The architecture of Kubernetes[1], as seen in **Figure 2.3**, is made up of different components that work together to provide a resilient and scalable environment. When using Kubernetes, we will deal with a Kubernetes cluster, a set of processing servers called nodes, on which pods will be hosted and where the application will run. There are two types of nodes: the Master Node, which is responsible for making scalability decisions and managing the cluster's resources, and the Worker Nodes, which are responsible for hosting the applications. Every Kubernetes cluster has at least one Worker Node.

The Kubernetes Master Node consists of three components: the API server, which exposes the Kubernetes API and acts as the entry point for

[1]https://kubernetes.io/pt-br/docs/concepts/overview/components/

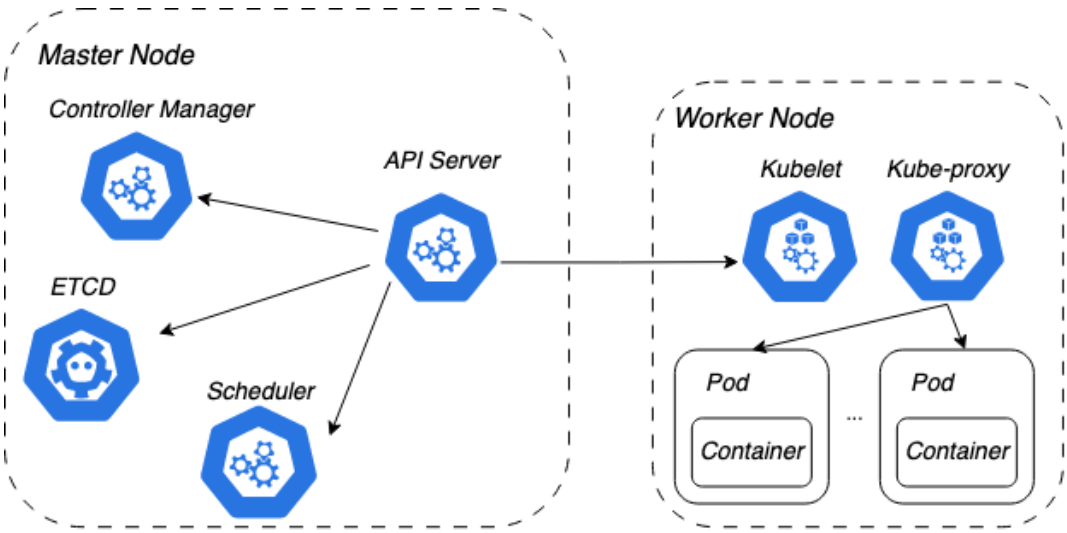


Figure 2.3: Kubernetes Architecture

all services; the Scheduler, which is responsible for monitoring newly created pods that have not yet been assigned to a node; and selecting a suitable node for them to run on; and the Controller Manager, who is in charge of managing the cluster's controllers and ensuring that the cluster's state is properly maintained.

The Kubernetes's Worker Node is also made up of three main components: Kubelet, which is an agent that runs on each of the nodes and ensures that the containers are running on a pod; Kube Proxy, which is responsible for providing network services at the pod level; and Container Runtime, which is the software responsible for running the containers, such as Docker.

Finally, the Kubernetes architecture includes ETCD, a consistent, highly available key-value storage for all the data shared in the cluster.

2.3.1 Workloads

In the context of Kubernetes, the pod[2] represents the smallest deployable unit under our control for creation and management. In its most typical configuration, a pod consists of a single container; however, the pod can span multiple containers in situations involving applications of numerous coupled containers. This possibility occurs due to the sharing of network and storage resources between all the containers contained in the pod. It's worth noting that pods in Kubernetes are ephemeral by nature, which means they can

[2]https://kubernetes.io/docs/concepts/workloads/pods/

be created, deleted, and recreated dynamically to adapt to the application's demands.

In general, pods are not directly created as individual resources within Kubernetes but are instantiated from workload resources. A workload resource[3] essentially serves as an automated mechanism for managing a group of pods. Kubernetes offers various predefined workload resources:

- Deployment: It is a way of managing *stateless* applications, i.e., where the state is not persisted, and each transaction can be treated as if starting anew.
- StatefulSet: It is a way to manage stateful applications where the state needs to be persisted.
- DaemonSet: This mechanism ensures that a specific pod instance runs on each cluster node. In the event of new nodes being added to the cluster, this workload guarantees the deployment of a new pod on the new cluster node.
- Job and CronJob: The Job provides a way to execute tasks that run once and then terminate. For scenarios where we want the same Job to run repeatedly according to a schedule, a CronJob can be employed.

2.3.2 Autoscaling

In Kubernetes, a workload can be scaled to adjust the allocated resources. This process can be made manually using kubectl[4] CLI (Command Line Interface) or automatically by providing metrics that can be used for scaling decision-making. Automatic scaling is our main focus, and Kubernetes provides two ways to scale our workload automatically:

- Horizontal Pod Autoscaler (HPA) acts by adding new pod replicas that run the same application. As a result, load balancing is carried out on more processing units, which means the load is diluted between the pods. Since it only requires adding new pods, there is no need to restart the application, which makes HPA attractive for applications that need high availability.
- Vertical Pod Autoscaler (VPA) acts by directly switching the resources of existing pods. As it is necessary to modify the existing resources available in the pods, the service must be restarted, which hinders the use of VPA for applications that need to run continuously.

[3]https://kubernetes.io/docs/concepts/workloads/
[4]https://kubernetes.io/docs/reference/kubectl/

2.3.3 Services

To make an application accessible as a network service from a collection of pods, Kubernetes provides a resource type known as a Service[5]. By utilizing this resource, we can establish a coherent group of pods and a reliable means of reaching them. The service facilitates the exposure of this pod group, making it accessible not only within the cluster but also externally if required.

Kubernetes provides the following types of services:

- ClusterIP: Allocates an internal IP address, making the set of pods accessible only internally to the cluster.
- NodePort: Exposes the service on a fixed port of each node in the cluster, providing accessibility via the IP address of the cluster node at the configured port.
- LoadBalancer: Exposes the service externally using an external load balancer. It's important to note that Kubernetes does not include a built-in load balancer component; one must be provided separately, commonly available in public cloud clusters such as the Google Cloud Platform (GCP).
- ExternalName: Maps the service to a specified DNS (Domain Name System).

[5]https://kubernetes.io/docs/concepts/services-networking/service/

3
Self-scalable system architecture based on Kubernetes

As mentioned, the gateway is the ContextNet 3.0 microservice which is responsible for communication between mobile and static nodes. This component establishes connections with mobile nodes using the MR-UDP protocol and translates messages into the Kafka protocol to deliver to other microservices. Therefore, it will be the one we will evolve as a use case for our work.

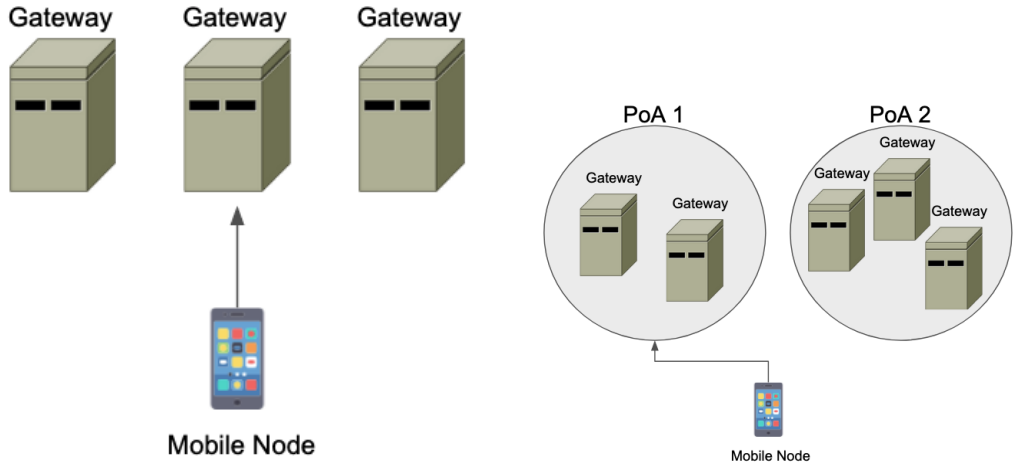
Initially, system resource bottlenecks were identified with the aim of understanding how we can leverage them in decision-making for automatic scaling. As mentioned in the previous work (WANOUS, 2021) and empirically identified by us, the system has limitations based on available resources such that variations in the amount of RAM directly impact the number of connected mobile nodes. Consequently, it was determined that each gateway can support up to 5000 mobile nodes, requiring 1.5GB of available RAM for this purpose. Adding more connections with the same amount of RAM results in system failures and significant loss of connections. It is worth noting that this configuration was used for the project based on our available resources. However, it is possible to configure gateways with more resources so that a single gateway can support more than 5000 simultaneous mobile nodes.

In Section 3.1, we will discuss the self-scalable architecture implemented using Kubernetes. Next, in Section 3.2, we will address the self-scaling system, describing its operation and the configurations made. In Section 3.3, we will discuss Kubernetes' internal load balancing and how we configured it to support MR-UDP's persistent connections. Finally, in Section 3.4, we will discuss the challenges of balancing persistent connections.

3.1
Architecture of ContextNet gateways using Kubernetes

To manage ContextNet Gateways, we created a new entity named Point of Access (PoA). A PoA consists of a Kubernetes cluster that runs gateway instances and makes them available to mobile nodes. However, with the application running in the Kubernetes cluster, we have the HPA, which adds elasticity to the gateways. With the HPA, a PoA can dynamically adjust the number of gateway instances being executed based on collected metrics and previously defined limits.

By adding this new entity to the ContextNet architecture, we have the PoA as a dynamic pool of gateways. Therefore, the proposal is to stop the mobile node from connecting directly to one gateway and start connecting to a PoA that manages several gateways. We can see this change in **Figure 3.1a** that illustrates a mobile node connecting directly to one of the gateways and in **Figure 3.1b** that show the mobile node connection to a PoA that is managing two gateways.



(a) Previous use of ContextNet gateways (b) New use gateways with PoAs

Figure 3.1: Point of Access concept.

For the architecture of a PoA, we implemented a single-node Kubernetes cluster. In this setup, a single virtual machine serves the dual role of both master and worker nodes. A deployment was set up with the gateway's Docker image. An HPA was associated with this deployment so that the gateway instances are managed automatically according to the configured rule. Finally, to access the gateways externally to the cluster, a NodePort-type service was configured, where we mapped a fixed port on the Kubernetes cluster node with a port on the service.

3.2 Horizontal Pod Autoscaler

The Horizontal Pod Autoscaler operates as a Kubernetes API resource and controller. The configuration of the resource dictates its behavior. The HPA controller runs within the Kubernetes control plane, dynamically adjusting the number of replicas based on predefined metrics. These metrics encompass pod resources, such as CPU and memory, and can extend to custom metrics derived from the application.

At its essence, the HPA functions as a continuous loop, running at customizable fixed intervals. In each iteration, the controller refers to the HPA resource definition to identify metrics for evaluation. When the metric involves the CPU or memory of the pods, the controller queries the Resource Metrics API. On the other hand, it utilizes the Custom Metrics API for any other metric. This approach offers crucial flexibility in the metrics collection, allowing the HPA to adapt to a diverse range of custom and standard pod resource metrics.

To perform queries on the Resource Metrics API, a commonly employed tool is the *metrics-server*[1], a data consolidation tool for resource usage across the entire Kubernetes cluster. Its primary function is to furnish data exclusively for horizontal and vertical scaling processes. The *metrics-server* collects resource metrics from the kubelet running on each cluster worker node and exposes this data through the Resource Metrics API.

For queries to the Custom Metrics API, it is often necessary to employ an agent in charge of providing data to the API based on application metrics. A widely adopted solution involves exposing application metrics through *Prometheus*[2] and using the *Prometheus Adapter*[3] to extract metrics from the *Prometheus* server and feed the Custom Metrics API with the data exposed by the application (SUKHIJA; BAUTISTA, 2019; SONG; ZHANG; HAIHONG, 2018; NGUYEN et al., 2020).

Hence, the architecture incorporating the *metrics-server* to provide data for the Resource API and employing the *Prometheus* strategy to feed the Custom Metrics API is illustrated in **Figure 3.2**.

This approach facilitates a robust and adaptable collection of custom metrics, seamlessly integrating with the Kubernetes ecosystem to support dynamic scaling processes tailored to the application's specific demands.

3.2.1 Configuration

In configuring the horizontal autoscaling of the application, it is crucial to understand the inherent bottlenecks within the system. This understanding guides the selection of metrics for evaluation by the HPA and establishing rules. The strategic metric selection process mirrors the application's unique nature and determines the HPA's ability to adapt to load fluctuations. Aligning metric selection with the recognition of specific bottlenecks ensures efficient

[1] https://github.com/kubernetes-sigs/metrics-server
[2] https://prometheus.io/
[3] https://github.com/kubernetes-sigs/prometheus-adapter

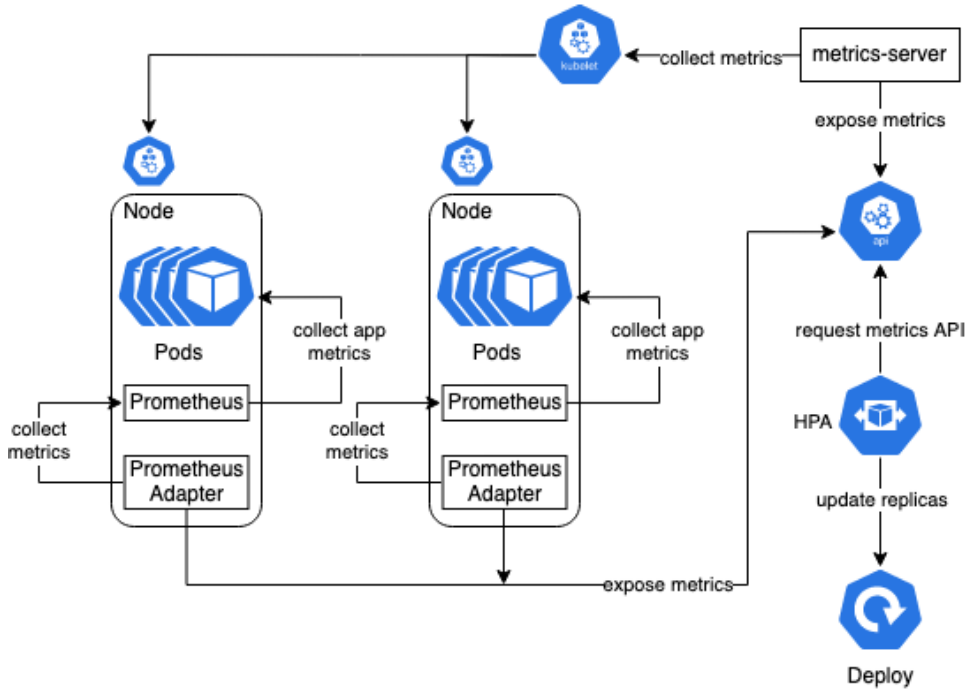


Figure 3.2: HPA metrics scrape architecture

scalability and optimized resource allocation to meet the dynamic demands of the application.

In prior research (WANOUS, 2021), Camilla empirically showed that the gateway is the component imposing the highest demand on RAM. This is attributed to the need to maintain and manage many MR-UDP connections. As a result, in most ContextNet deployments, we always choose to allocate the gateways to the machines with the most RAM available. The prior experiments have also shown that ContextNet 3.0 is highly reliable when handling up to 5000 connected mobile nodes per gateway.

Therefore, as we aim to have precise control over resources for operating a scalable system, we conducted tests to determine the amount of memory a gateway needs to connect and sustain 5000 mobile nodes sending messages every five minutes. After several tests, we found that the total RAM required by a gateway is 1.5GB.

In our assessment of suitable/minimal environment configurations, we empirically identified that CPU usage is also intensive and directly related to the volume of messages processed by a gateway. For example, in a scenario with 1000 connected mobile nodes sending messages at one-minute intervals, we observed higher CPU usage than when the same number of nodes were sending messages at five-minute intervals.

Therefore, we decided to work with gateway containers with 1.5GB of allocated RAM for the environment configuration, setting the limit to 5000

mobile nodes per gateway instance and one dedicated CPU. In response to the limit of connected mobile nodes and the CPU volatility depending on the message-sending scenario, we chose to configure the HPA with the capacity to add new pods based on monitoring both CPU usage and the number of connected nodes. We understand that monitoring memory usage is also an option. However, we chose not to do it now since the *metrics-server* monitors the operating system's memory. At the same time, the JVM (Java Virtual Machine) requests more memory than it may be using. To address this, we recognize the need to metricize memory usage from the application itself and prefer to treat this as a future task.

3.2.2 CPU Monitoring

In our scenario, we chose to assess the CPU usage of the pod rather than directly monitoring the application. This decision is secure due to the one-to-one correspondence relationship between the container and pod. In scenarios with multiple containers, the resource usage of all containers is aggregated and used as the scaling reference metric. This approach can lead to situations where a single overloaded container might not be sufficient to trigger the scaling rule.

This way, we integrated the *metrics-server* into the cluster to collect pod data and feed this information into the Resource Metrics API. This setup enables the HPA to query and act based on these metrics.

The HPA was configured with a resource metric of type CPU, specifying the scaling limit as a resource utilization percentage. It's important to note that this value may vary during testing to illustrate the behavior across different scaling scenarios.

3.2.3 Connected nodes monitoring

As previously mentioned, each gateway has a limit on the number of concurrently connected mobile nodes. Therefore, it is necessary to add new gateway containers shortly before this limit is reached. However, this metric is not inherently part of the infrastructure where the application is executed. Consequently, we need to expose this metric from the application to enable HPA to utilize it for decision-making.

To expose these metrics for utilization by the HPA, we chose to adopt the strategy of exposing application metrics to *Prometheus* and having the *Prometheus Adapter* collect and feed these metrics into the Metrics API. This

involved including both *Prometheus* and the *Prometheus Adapter* in the cluster to facilitate the desired information flow.

Subsequently, the HPA was configured with a Pods metric, using the name associated with the exposed metric to indicate the total number of connected mobile nodes. The scaling limit was set to the total number of connected mobile nodes, specifically 4500. Despite the per-gateway limit being 5000 mobile nodes, we chose a scaling limit below this threshold to prevent scenarios where the HPA takes a while to realize the need for scaling, which leads to the unavailability of new mobile node connections.

3.3 Service load balancing

As detailed in section 2.3, a service is an abstraction defining a logical group of pods and their access policies. These services are associated with a fixed IP and a DNS, simplifying client communication with the pods. Once this communication is established, effective load balancing for all requests directed to the service becomes necessary, ensuring their distribution among the pods. This process is known as service load balancing.

The kube-proxy, the component in charge of maintaining network rules and enabling communication with the pods internally and externally to the cluster, is responsible for implementing balancing. The standard implementation of this balancing occurs through *iptables rules* (PURDY, 2004), resulting in an even distribution of the load in a round-robin algorithm.

However, in our scenario, where connections are persistent, and the gateway imposes a predefined maximum limit of connections, evenly distributing new connections becomes problematic, because trying to connect to a saturated gateway results in the mobile node trying new reconnections until it connects to an available gateway.

We used kube-proxy in IPVS (IP Virtual Server) mode to address this challenge. IPVS (DU HAIBIN XIE, 2018; ZHANG et al., 2000; PHAN; KIM et al., 2022) implements load balancing at the transport layer of the OSI model (IREN; AMER; CONRAD, 1999) within the Linux kernel. Integrated with the Linux Virtual Server (LVS), it acts as a load balancer, directing TCP or UDP service requests to the servers that will process the request.

With the Kubernetes cluster operating in IPVS mode, kube-proxy dynamically monitors resources and configures IPVS rules accordingly. Simultaneously, IPVS in the Linux kernel is responsible for packet forwarding and balancing between the pods. This configuration allows us to specify the desired load-balancing algorithm. Given the challenges in connecting to the gateways,

we determined that the best option is to use the Least Connection Algorithm (SINGH; KAUR, 2018), ensuring that new connections are directed to the server with the fewest existing connections at that moment.

3.4 Dealing with persistent connections in Kubernetes

Knowing that Kubernetes does not rebalance active connections, even when adjusting the replica count, persistent connections can be a challenge that the architecture does not address.

This problematic scenario becomes apparent when the gateways increase CPU usage, mainly resulting from events of already connected mobile nodes. For instance, if the CPU increase is due to a rise in message exchange among the already connected mobile nodes, this might trigger a scale-up in the PoA. However, it does not alleviate the overload on the specific gateway because existing connections have not been migrated. Understanding this, our test scope, described in Chapter 5, always considers a fixed interval between message exchanges.

4
PoA Manager

In the original ContextNet 3.0 (WANOUS, 2021), the PoA Manager was responsible for load balancing between gateways and creating a prioritized list of active gateways. However, as explained in Chapter 3, the new architecture proposes isolating gateways through the PoA entity. This entity has a public IP where mobile nodes should connect, functioning as an elastic pool of gateways that adjust the number of its instances according to the current demand. In section 4.1, we will briefly introduce how the old PoA Manager worked, followed by an explanation of the new simplified PoA Manager in section 4.2, which we implemented to enable the use of the new architecture.

4.1
PoA Manager for non-elastic gateways

To perform load balancing among gateways, the former PoA Manager takes into account two sets of monitoring information involving mobile nodes and gateways: the Load Report and the Connection Report. Additionally, to receiving data reports from mobile nodes and gateways, there is also a need to monitor whether the gateways are active to include them in the load balancing, as seen in **Figure 4.1**. This gateway state monitoring is done through the recurring sending of Load Reports. The absence of a report within a specific time interval indicates that the gateway is inactive, resulting in its removal from the load balancing.

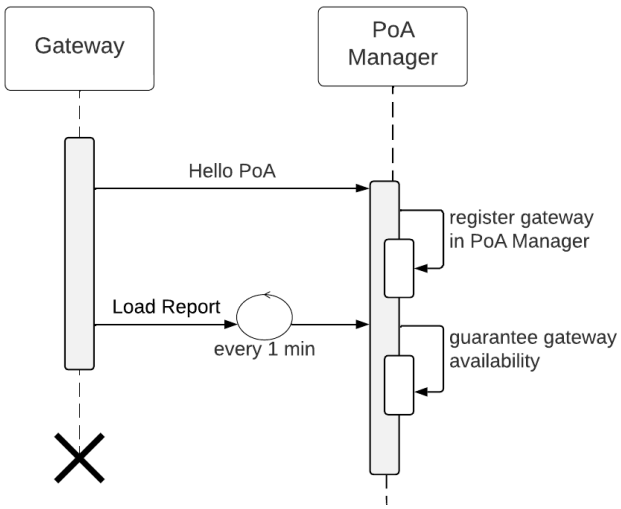


Figure 4.1: PoA Manager for non-elastic gateways gateway management process

The Load Report periodically sent by gateways provides information about the status of their hardware. Therefore, it includes details such as the utilization rate and size of the Thread Pool, CPU load, total memory, and available fraction, as well as total JVM memory and available fraction.

In the Connection Report, which is also sent periodically by the gateways, we have information regarding the connectivity of mobile nodes that are connected to the respective gateway. This report provides data on communication latency and the list of connected mobile nodes.

As a result, the PoA Manager executes a ranking algorithm that considers the load and connection reports and can be implemented by the developer. The objective of this algorithm is to generate scores for all active gateways, where a higher score represents greater availability and, consequently, a higher suitability for receiving new connections.

With the scores assigned to all active gateways, a reallocation process is initiated, taking into account both connection data and gateway scores. The purpose of this process is to calculate the number of mobile nodes that should be connected to each gateway and send reconnection requests to the mobile nodes that need to be relocated.

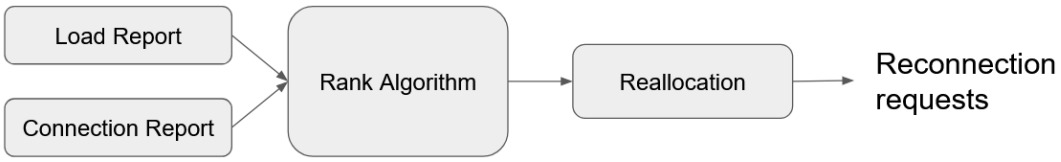


Figure 4.2: PoA Manager for non-elastic gateways reallocation process

4.2 PoA Manager for elastic gateways

With the change in the gateway architecture, the reports sent by each gateway become obsolete as the mobile node's point of connection shifts to the PoA instead of the gateway. For this reason, a restructuring of the PoA Manager was necessary to review its responsibilities and align it with the proposed new architecture.

Given that the main focus of the work is the evolution of gateways toward a self-scalable structure, when the need to restructure the PoA Manager was identified, we opted for a simple implementation to facilitate the testing and validation of the new PoA architecture. Therefore, the goal in restructuring the PoA Manager was to clearly define its new responsibilities and implement them in a way that is flexible for developers of ContextNet applications, allowing them to choose their decision-making policies. Additionally, we aimed to make

it easily extensible for ContextNet developers, as there are certain aspects to be improved to make it a more flexible component.

Looking at the new self-scalable architecture, we realize that the gateway is isolated from the mobile node so that the connection occurs between the mobile node and the PoA. Therefore, load balancing among gateways within a PoA becomes the responsibility of Kubernetes, which manages the available gateway instances per PoA. Hence, we observe that the PoA Manager now has two basic responsibilities: managing gateways to determine their associated PoA and link the sent reports to a PoA; and deciding the best PoA for a mobile node to connect to.

In subsections 4.2.1 and 4.2.2, we will thoroughly describe the two mentioned responsibilities of the PoA Manager.

4.2.1 Gateways management by PoA

Unlike gateways, a PoA is not considered a dynamic entity that frequently appears and disappears. Instead, it is viewed as a static entity that requires an installation process to exist in a real-world scenario. From our perspective, a PoA is a statically and geographically distributed access point responsible for managing a pool of gateways. Consequently, there is no mechanism for a PoA to spontaneously appear and inform the PoA Manager of its existence. Therefore, the PoA Manager needs prior knowledge of the existing PoAs.

To map the existing PoAs, the PoA Manager maintains a JSON (JavaScript Object Notation) file containing the following data for each PoA:

- Identifier: the name identifying the PoA
- IP: the public IP of the PoA, where mobile nodes should connect
- Latitude: the latitude of the location where the PoA is situated
- Longitude: the longitude of the location where the PoA is situated

In contrast to the previous PoA Manager, which used Load reports to determine whether gateways were inactive or not, the new version now relies on the "Hello" messages sent by gateways every ten seconds in a specific Kafka topic. This change was made because we want the responsibility of determining gateway liveliness to be associated with a dedicated Kafka topic focused solely on indicating the proper functioning of the system.

On the PoA Manager side, we configured a liveliness buffer with a size of two to support instabilities in the liveliness signal transmission. This ensures that a single failure in transmission is not sufficient to label the gateway as

inactive. Currently, the size of this buffer is fixed, but for the evolution of the PoA Manager, we recognize that making this value configurable per PoA could be beneficial. This is particularly relevant in regions with varying connection quality, which may result in more or fewer communication failures.

To associate a gateway with a PoA, we consider the first transmission of the "Hello" message containing the IP of the PoA to which the gateway belongs, as we can see in **Figure 4.3**. This enables us to map information from reports sent by gateways and aggregate them for decision-making by the PoA. Finally, to simplify the implementation in this version of the PoA Manager, only the load reports sent by gateways are considered. However, we acknowledge that in the future, it might be advantageous to also consider connection reports.

In summary, the new PoA Manager now determines the state of gateways solely based on the "Hello" message. It organizes gateways by PoA to aggregate reports sent by gateways within the scope of PoAs for decision-making.

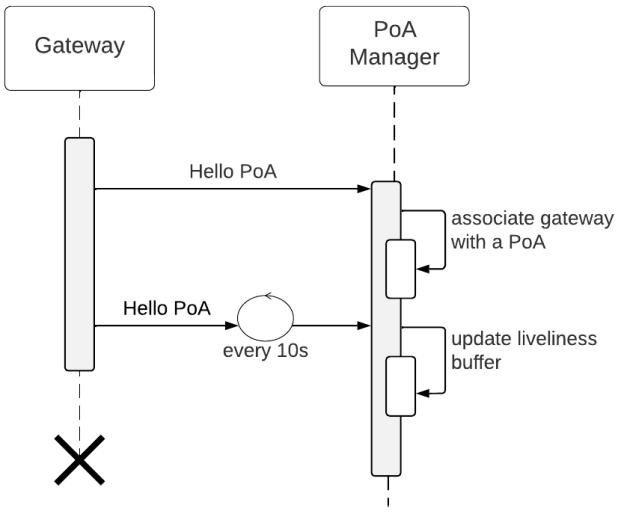


Figure 4.3: PoA Manager for elastic gateways gateways management

4.2.2 Choosing the best PoA

The PoA Manager is responsible for providing the mobile node with a prioritized list of the best PoA based on a rule defined by the developer of ContextNet's application, as we can see in **Figure 4.4**. If necessary, we delegate to Kubernetes the responsibility of allocating more resources and deploying additional gateways in scenarios where a PoA experiences high traffic.

Determining the "best" PoA is subjective and relies on the developer's specific needs. That's why we designed the PoA Manager to let the developer decide on the algorithm for selecting the best PoA. When implementing the algorithm, developers have access to all the data sent by mobile nodes in the

context message. This means they can utilize any information sent by the mobile nodes to choose the best PoA.

In our work, we used an example of a decision algorithm based on geolocation. Since each mobile node reports its latitude and longitude in its context message, and the PoA Manager knows the coordinates of the PoAs, we can use the Haversine algorithm (PRASETYA et al., 2020), which calculates the distance between two coordinates on earth using latitude and longitude, to determine the distance from the mobile node to each of the existing PoAs. This allows us to rank the PoAs based on proximity and return this list to the mobile node.

It is important to note that a geolocation-based decision is not the only valid one, but it was chosen by us for validation purposes in the architecture of autoscalable gateways. Besides the chosen decision for our validation, scenarios may exist where specific PoAs are designated for processing different types of data. In such cases, each mobile node can only connect to a group of PoAs processing the data it transfers. Scenarios may also consider the current battery level of the mobile node, directing nodes with lower battery levels to PoAs with higher availability to prevent data loss in case of failure. Another option is to assign mobile nodes that require faster updater in PoAs with greater availability.

As we can see, there are various scenarios where choosing the best PoA may need to be approached differently. For us, the most important aspect of developing the PoA Manager was to provide this flexibility of implementation for the ContextNet applications developer, as each one may have different requirements.

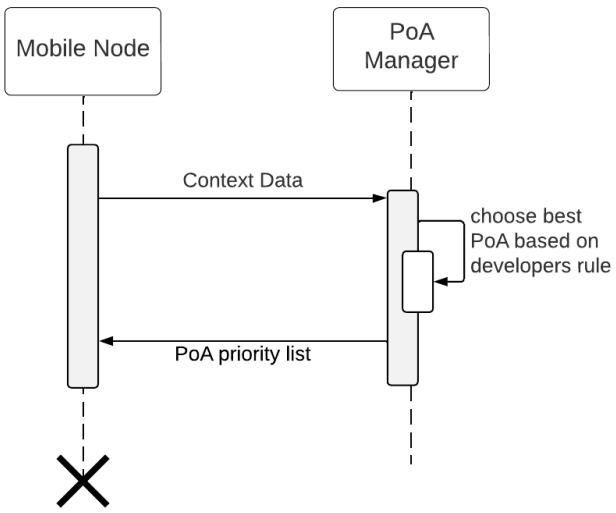


Figure 4.4: Choose best PoA

5
Experiments and performance results

5.1
Setup

Some experiments were carried out to evaluate our solution and ensure that it complies with the requirements for PoA operation without interfering with the connectivity quality of the nodes.

For the testing, we configured the environment in a distributed manner with four virtual machines (VMs) having 4 CPU cores and 16 GB of RAM each on PUC-Rio DI private cloud. These VMs were logically organized in a star network topology (MEADOR, 2008), with a separate VM as the central node responsible for packet forwarding in message exchanges.

As described in Section 2.1, ContextNet uses Kafka as the communication layer between microservices. Therefore, it was necessary to configure the environment with Zookeeper (HUNT et al., 2010). In addition to the base configuration, we set up an instance of the PoA Manager, as this service manages the created PoAs and connections of mobile nodes, leading to a significant exchange of messages with this service. To represent PoAs, two VMs were configured with a Kubernetes cluster using the minikube tool (MUDDINAGIRI; AMBAVANE; BAYAS, 2019). Finally, two VMs were allocated to simulate mobile nodes.

Thus, our test infrastructure was arranged as shown in **Figure 5.1**. The first VM runs Zookeeper, the PoA Manager, and simulates mobile nodes. The second and third VMs are exclusively responsible for implementing one PoA each. Finally, the fourth VM also simulates mobile nodes.

For all experiments described below, we used the infrastructure outlined above. Therefore, as it is a distributed architecture with asynchronous message exchange, the tests were executed multiple times to ensure a confidence level of 95%. Additionally, following the scenario described in section 3.4, the message exchange interval of the mobile nodes was configured for five minutes.

5.1.1
Fairness Index

We also considered it essential to assess how Kubernetes distributed connections among the operational gateway instances, especially in a persistent

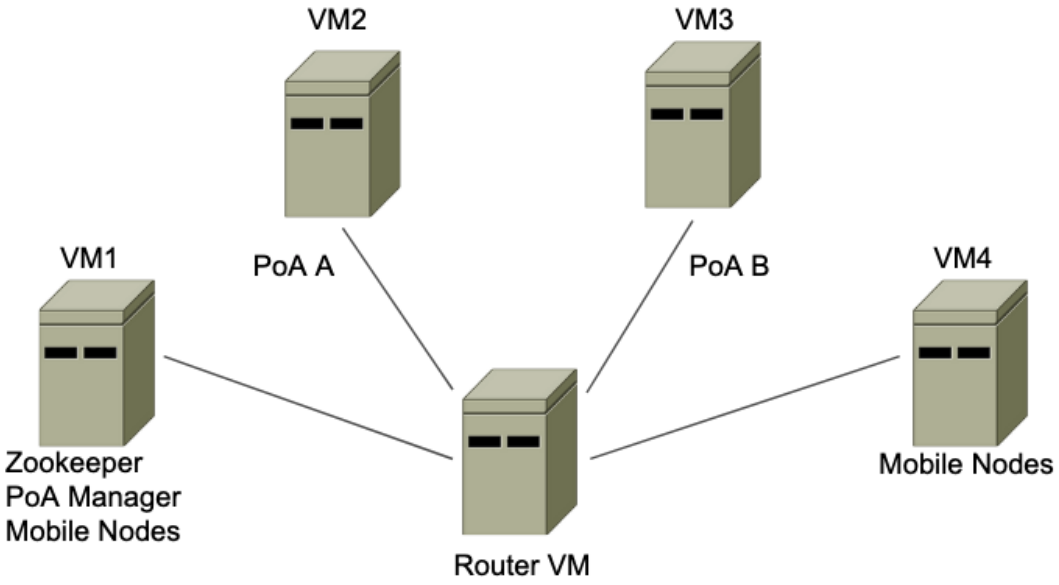


Figure 5.1: Experiments Infrastructure

connection scenario. So, in some experiments, we're using the Fairness Index (JAIN et al., 1984) to verify if Kubernetes are balancing connections properly.

As mentioned in chapter 3, section 3.3, we configured the Kubernetes cluster for the kube-proxy to operate in IPVS mode with the least connections algorithm. Consequently, new connections would always be directed to the gateway with the fewest connections. This setup is ideal since we deal with the same load across all connections and aim for fair balancing based on the number of connections rather than the processed load.

To evaluate the balancing performed by Kubernetes using our configuration, we used the Fairness Index (JAIN et al., 1984). This index provides a quantitative measure to assess the distribution of connections. The 5-1 index aims to measure the equality of the distribution of a quantity x of connections among n gateways, where each has x_i connections. The index result will be one when the distribution is 100% fair and deviates from one as the discrepancy in the distribution increases.

$$FairnessIndex = f_a(x) = \frac{[\sum_{i=1}^{n} x_i]^2}{n \sum_{i=1}^{n} x_i^2} \tag{5-1}$$

5.2 Availability and Load Balancing in PoA with and without autoscaling

This experiment aims to show the availability and load balancing in with a PoA with autoscaling, which can allocate more resources as needed when compared to a PoA without autoscaling, with a fixed resource limit.

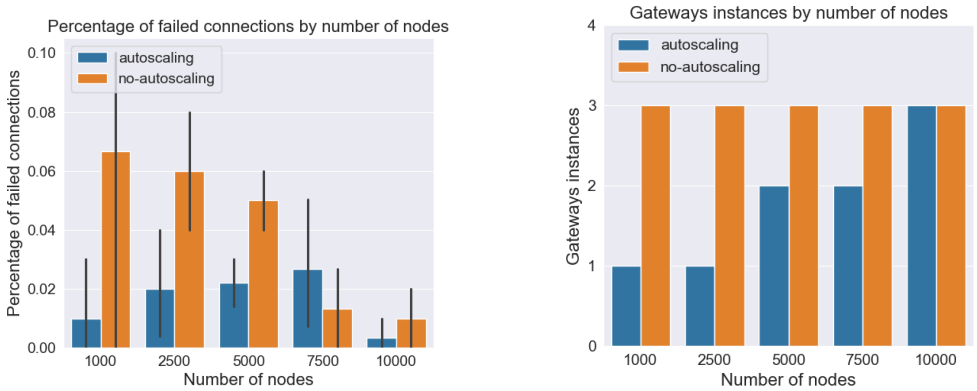
We configured the pods with one CPU core and 1.5 GB of RAM to carry out the experiment. This configuration ensures the connection of up to 5000 mobile nodes in each gateway pod, as previously described. Thus, the PoA without the autoscaling strategy will use consistently three pods, while the PoA with the autoscaling strategy is configured to use between one and three pods. To control pod usage in the autoscaling PoA, we configured the HPA to scale for 4500 connected mobile nodes and 50% CPU usage.

To experiment in different scenarios, we connected 1000, 2500, 5000, 7500, and 10000 mobile nodes with a fixed interval of 1000ms between connections and checked the percentage of successful connections for each of the PoAs and the number of gateways used, bearing in mind that each one is configured to accept a maximum of 5000 mobile nodes.

Figures 5.2a, **5.2b** and **5.3** depicts the number of connection failures for the two PoAs, the number of instances of gateways used in the tests and the connections load balance within a PoA, respectively, with a confidence level of 95%.

Figure 5.2a show that both PoA configurations can handle 100% of connections up to 10000 mobile nodes with low percentage of failed connections. We can see that the PoA without autoscaling has a higher percentage of failed connections when connecting less mobile nodes, indicating that connection failure should be an absolute value by gateway instance.

In **Figure 5.2b** we can see that while PoA without autoscaling has all gateways available since the beginning, PoA with autoscaling adapt according to new connections to guarantee availability.



(a) Percentage of failed connections for PoAs with and without autoscaling

(b) Gateways instances for PoAs with and without autoscaling

Figure 5.2: Availability analysis in autoscaling PoAs.

In **Figure 5.3**, we observe that a PoA with autoscaling impacts the

Fairness Index during an upscale event, indicated by the red line. This suggests that a PoA without autoscaling maintains a fair load balance, as all gateways are available from the start and connections are evenly distributed. While a PoA with autoscaling dynamically adjusts resources, it results in an unfair balance compared to the PoA without autoscaling. However, in terms of resource efficiency, only the autoscaling PoA can provide savings.

Thus, both configurations demonstrate the ability to handle all connections with minimal failures. Additionally, a PoA without autoscaling may achieve better connection balance, but lacks the resource-saving capabilities of the autoscaling PoA.

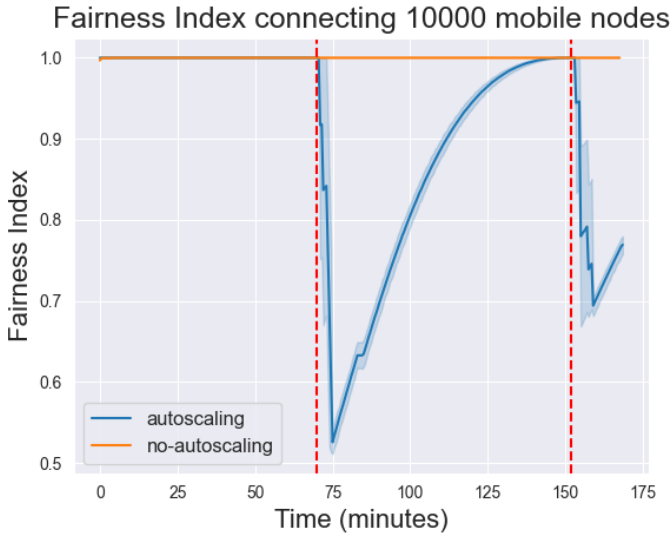


Figure 5.3: Fairness index with upscale events represented by red line

5.3 Connectivity assessment during downscale

This experiment is intended to check the connectivity of mobile nodes during the downscale process. In an ideal world, we would like no reconnection by the mobile node. However, Kubernetes does not manage the redirection of active connections, so, at a downscale moment, it will be required by all nodes connected to do the down pod disconnection followed by a reconnection.

Although it does not redirect connections, when a pod terminates, Kubernetes sends a SIGTERM signal to the container and waits for a pre-configured resolution time, called the *termination grace period*, to properly terminate the pod. During the grace period, the persistent mobile nodes do not lose connectivity with the gateway, but no new connections are directed to the pod. In the face of this, the goal is to guarantee the following three points:

- All mobile nodes must be reconnected within the termination grace period.
- Low average reconnection time so that it does not impact the system's restabilization.
- The downscale event should not impact the Fairness Index

This way, we can guarantee the system's resilience and show that the proposed solution is effective even when mobile nodes reconnect during Kubernetes' automatic adjustments.

The tests conducted for the described experiment involve a downscale simulation. To achieve this, we start a PoA with two gateway replicas and connect mobile nodes with a fixed interval of 1000ms. During this process, the established connections are evenly distributed among the gateway instances managed by the PoA. Subsequently, when all mobile nodes are connected, we force the shutdown of one of the gateway instances, causing all mobile nodes connected there to reconnect and establish a new connection with the remaining operational gateway.

As a result, we decided to vary the number of mobile nodes to be connected by 500, 1000, 1500, 2000, 2500, and 3000, and the available CPU resource for the gateway instances at 0.5 CPU, 0.75 CPU, and 1 CPU, while keeping the available RAM fixed at 1.5 GB.

Next, we will discuss the analyses conducted to ensure the two objectives mentioned in subsections 5.3.1 and 5.3.2.

5.3.1 Evaluation of system restabilization

To ensure the first point mentioned earlier, it is necessary to assess the system's restabilization time, which we understand as the time it takes for all mobile nodes to disconnect from the gateway being shut down and reconnect to another gateway. This process involves disconnecting from the PoA and reconnecting, ensuring the new connection is directed to one of the remaining gateways in service.

The gateway shutdown message is sent to all connected mobile nodes. However, due to the MR-UDP protocol and internal implementations of ContextNet components, mobile nodes receive the reconnection message at different times. Therefore, evaluating the system's restabilization ensures that the discrepancy in receiving the reconnection message does not negatively impact the reconnection of mobile nodes in this downscale process. To verify that there is no impact, we expect all mobile nodes affected by the downscale

to reconnect within a shorter time than the termination grace period. Thus, the only period of unavailability will be the reconnection time itself.

Despite the existing dispersion in the perception of mobile node disconnection due to the utilized protocol, it is possible to observe from the test results, in **Figure 5.4**, with a 95% confidence level that there is no distinct behavior given different pod configurations, nor with an increased number of connected mobile nodes.

Therefore, we ensure that in this scenario, all mobile nodes reconnect within the termination grace period of 60 seconds, affirming that no mobile node loses connectivity with its current gateway before reconnecting.

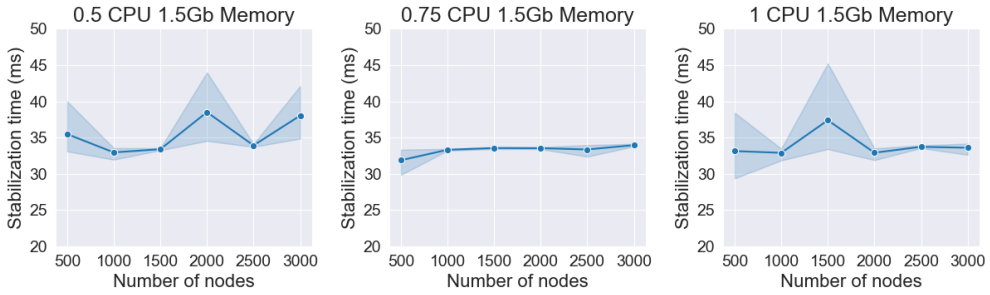


Figure 5.4: System re-stabilization analysis

5.3.2 Evaluation of reconnection time

After ensuring the system's restabilization time falls within the termination grace period interval, our next step is to analyze the reconnection times of mobile nodes. The objective is to assess the relationship between the number of reconnecting mobile nodes and the reconnection time and whether the available resources have any impact.

Analyzing the test results depicted in **Figure 5.5** with a 95% confidence level, we can see that the average reconnection time increases as more mobile nodes attempt to reconnect. A more substantial impact is noticeable regarding the available resources when limited resources (0.5 CPU per pod) exist. However, we do not observe any significant influence when comparing experiments with more abundant resources (0.75 CPU and 1 CPU). Additionally, upon examining **Figure 5.6**, we notice that the dispersion in reconnection times increases with more reconnecting mobile nodes. This suggests that some mobile nodes reconnect much faster than others in this scenario.

Given that the system's restabilization time falls within the termination grace period, we can ensure that the mobile nodes' downtime is solely the

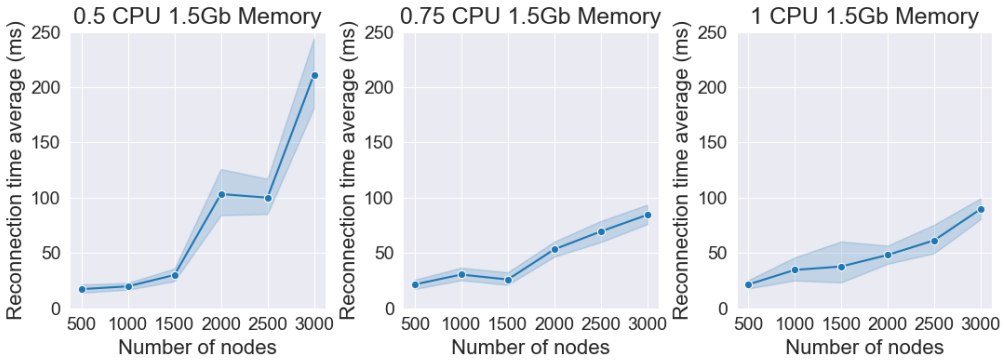


Figure 5.5: Mobile nodes reconnection time analysis

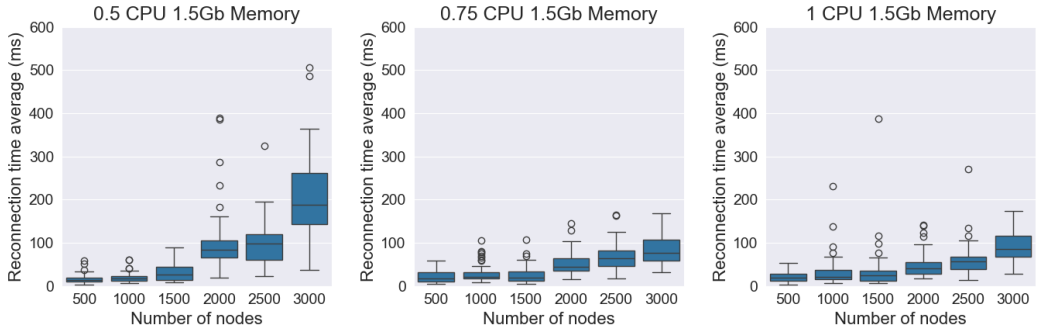


Figure 5.6: Mobile nodes reconnection time analysis

duration it takes to reconnect. Hence, in addition to assessing the average reconnection time, we find examining the maximum reconnection time crucial. This metric represents the worst-case scenario of unavailability, offering direct insights into the downtime caused by the downscale process.

In **Figure 5.7**, with a 95% confidence level, we observe a consistent pattern across the three variations of available resources. The maximum reconnection time increases as the number of connected mobile nodes rises. This suggests that the more mobile nodes are affected by the downscale, the longer it might take them to reconnect. However, the maximum reconnection time remains below one second, a level we consider satisfactory for the nature of the system under consideration.

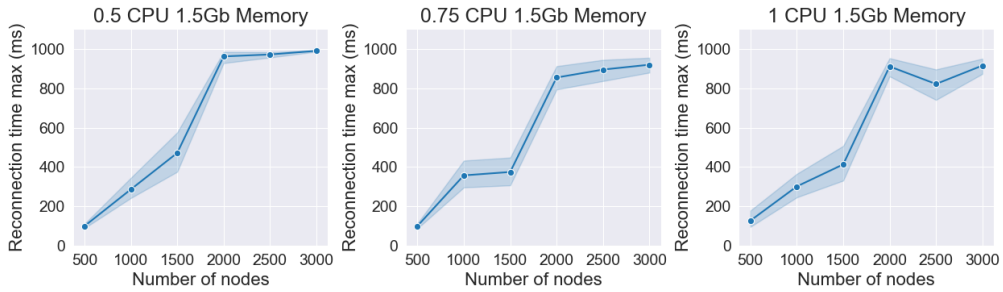


Figure 5.7: Mobile nodes maximum reconnection time analysis

5.3.3 Fairness index evaluation

During downsizing, when reconnections occur due to the shutdown of a gateway instance, we deemed it crucial to assess the Fairness Index to ensure that this event does not affect the balance of connections among operational gateways.

For this evaluation, we focused on the scenario with the most mobile nodes involved in the downsizing process, namely when 3000 mobile nodes are utilized. Additionally, we restricted our analysis to the scenario where each gateway instance has 1 CPU and 1.5GB of RAM, as the amount of resources available in the gateway does not impact the distribution of connections since balancing connections is the responsibility of the kube-proxy.

In **Figure 5.8**, with a 95% confidence level, we observed that during the connection of the mobile nodes, the Fairness Index remains close to one. After the downsizing event, represented by the marker on the graph, we maintained the index at the same levels. This indicates that, despite a third of the connections being migrated, the balance remained fair among the gateway instances that continued operating.

5.3.4 Discussion

The experiment aims to show that the system does not experience downtime exceeding the reconnection time during downscales, which should be insignificant given the application's nature, and to ensure that the downsizing event does not impact the Fairness Index.

We observed that the configured termination grace period of 60 seconds is sufficient for the tested scenarios. However, in scenarios where gateways have more resources and consequently more concurrently connected mobile

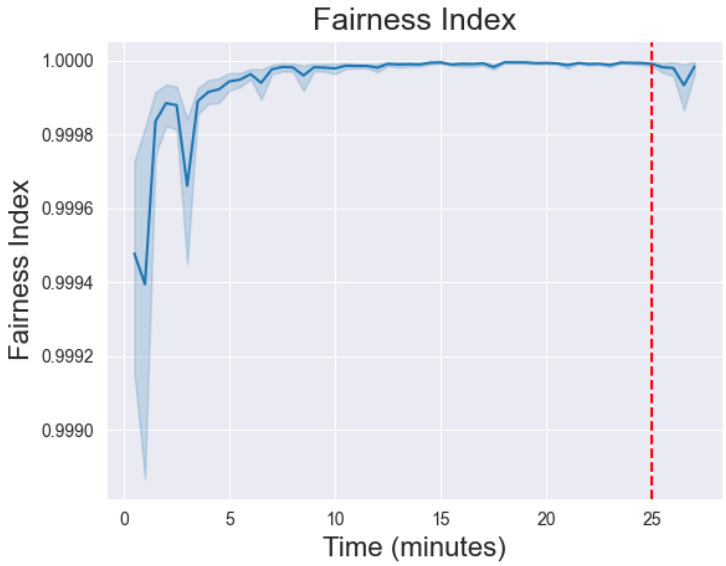


Figure 5.8: Fairness Index during downscale event represented by the red line.

nodes, an adjustment to this time may be necessary. This flexibility in the configuration indicates that our system functions for the tested values and can be adapted to support scenarios beyond our current evaluations.

Furthermore, we noted that the individual reconnection time for mobile nodes increases alongside the number of mobile nodes attempting to reconnect. However, we also identified that the maximum time in these scenarios does not exceed one second, indicating that it has no significant impact.

Finally, verifying that the downsizing event does not impact the Fairness Index was possible, demonstrating that downsizing does not affect the balancing of connections within a PoA.

5.4 Upscaling during high connection demand scenarios

As outlined in the previous tests, connections were gradually established, with mobile nodes connecting in 1000ms intervals. However, it is crucial to understand how our service handles scenarios involving high connection demand, as this may impact up-scaling behavior. This experiment assesses up-scaling performance in scenarios with varying quantities of connected mobile nodes and different CPU scaling activation thresholds. We recognize that the connection of many mobile nodes in burst directly influences the application's CPU usage, necessitating the addition of more instances to distribute the load effectively. Furthermore, we want to analyze how the upscale event impacts the Fairness Index since we have persistent connections that are not rebalanced after adding a new gateway.

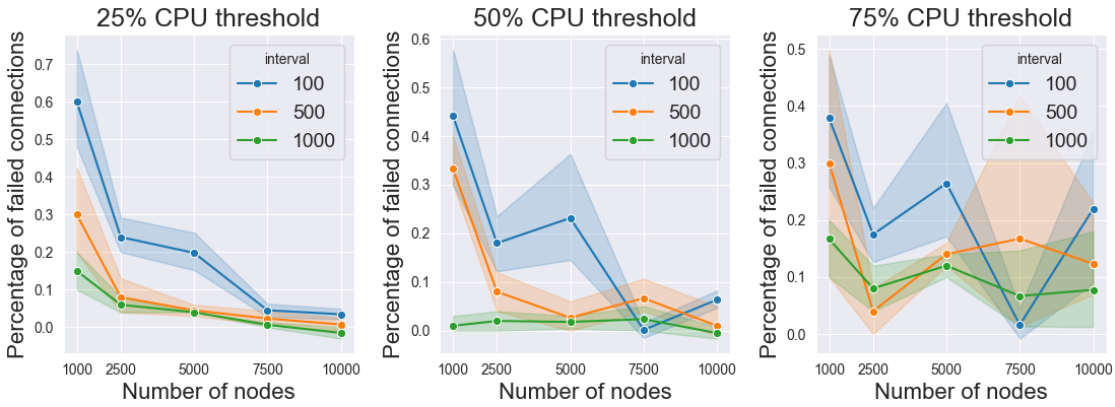


Figure 5.9: Percentage of failed connections in moments of high demand of connections

Table 5.1 outlines the tested scenarios (Case 1 to 3), where we varied the quantity of connected mobile nodes and the interval between each connection. Each test was conducted with CPU scaling thresholds set at 25%, 50%, and 75%. Additionally, we configured node quantity scaling, set at 4500 connected mobile nodes, in conjunction with CPU-based scaling.

Table 5.1: Connections burst experiment configuration.

Number of mobile nodes	Case 1	Case 2	Case 3
1000	1000 ms	500 ms	100 ms
2500	1000 ms	500 ms	100 ms
5000	1000 ms	500 ms	100 ms
7500	1000 ms	500 ms	100 ms
10000	1000 ms	500 ms	100 ms

5.4.1 Connectivity Assessment

First, we assessed the impact of the interval between connections on the connectivity of mobile nodes with the PoA. It was crucial to determine whether the scenario involving numerous simultaneous connections would affect the establishment of connections. During periods of high demand, the gateway's increased CPU consumption could potentially pose challenges in establishing new connections.

As depicted in **Figure 5.9**, we can see the percentage of failed connections for each connection interval in each CPU threshold case.

We observe a trend of improved connectivity in the graphs for 25% and 50% CPU when the interval between connections is larger, significantly as we increase the number of connected nodes. Enlarging the intervals between

connections has a positive impact, reducing the load on gateways and decreasing the likelihood of failures. Additionally, as the number of connected nodes increases, the percentage of failures decreases since failures are typically in absolute values. This indicates that, as the number of mobile nodes increases, the amount of connection failures does not increase proportionally, which can only be achieved using the proposed autoscaling architecture.

However, the graph for 75% CPU exhibits a similar but slightly more diverging trend, especially at points with 5000 and 10000 connected mobile nodes. This occurs because this configuration allows the gateway to operate with higher CPU usage, resulting in more connection failures, especially in scenarios where the gateway is close to its limit of connected nodes, as in the mentioned cases. Notably, this higher variability pattern is more pronounced in the test with a connection interval of 100ms, as this interval significantly increases CPU usage.

5.4.2 Scaling Evaluation

Next, we thought it should be essential to evaluate the number of gateway instances used in each test scenario, as the impact on CPU usage is a factor that can lead to the addition of new gateway pods. We understand that in a scenario with high demand for new connections, it is acceptable to work with more gateways to meet the momentary demand and avoid unavailability. If possible, downsizing can be performed later once the peak load has subsided.

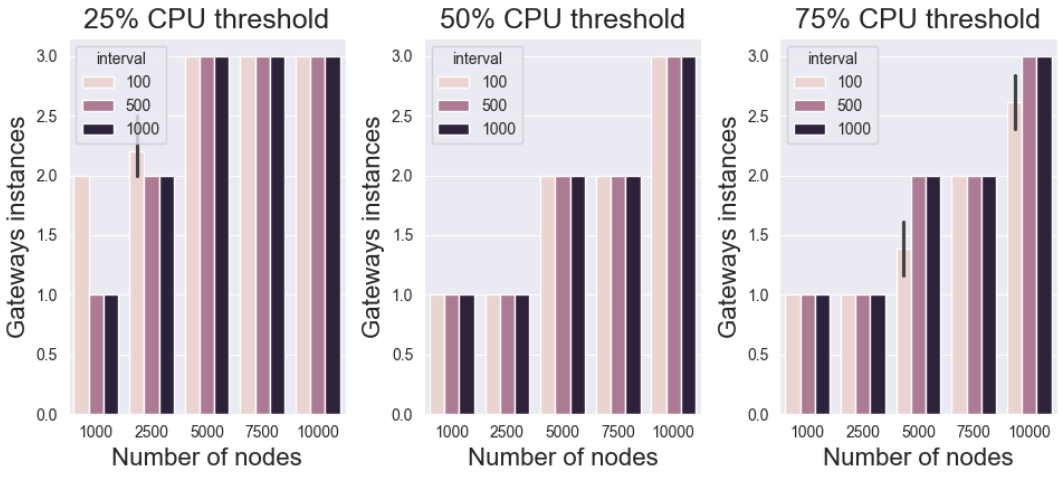


Figure 5.10: Scaling analysis in moments of high demand of connections

As illustrated in **Figure 5.10**, we categorized the graphs based on the configuration threshold for activating scaling, allowing us to assess different connection intervals with three scaling configurations, similar to the previous analysis.

For the 25% CPU usage configuration, we observe consistent behavior across the three variations of connection intervals. The pattern reflects a proactive scaling approach, resulting in the utilization of more gateway instances without approaching each one's limit. For example, scenarios with 2500 connected nodes use two instances, even though a gateway supports up to 5000 connected nodes. Notably, the scenario with a 100ms interval to connect 1000 mobile nodes stands out, indicating that shorter connection intervals lead to faster scaling events due to increased CPU usage on the gateway.

In the 50% CPU usage configuration, we observe similar behavior for the three evaluated intervals. However, for smaller quantities of connected mobile nodes, fewer gateway instances are employed. For example, in the scenario with 2500 mobile nodes, which previously used two instances, now only uses one. This happens because we need more CPU usage to trigger HPA scaling when compared with the previous scenario.

The final configuration with 75% CPU usage shows similar behavior to the configuration with 50%, except for the 100ms intervals in tests approaching the gateway's connection limit. Connecting 5000 mobile nodes in a short time sometimes doesn't provide enough time for the HPA to identify the scaling need based on the number of connected nodes before the test concludes. This is reflected in the variation in the number of gateway instances in these tests.

It is crucial to emphasize that, in addition to the number of instances used, the timing of scaling influences the distribution of mobile nodes among these instances. In the 25% CPU configuration, despite employing more gateway instances, there is a more effective distribution of connections because scaling events occur more rapidly. Conversely, in the 75% CPU configuration, gateway instances are utilized up to their limit before scaling occurs. This observation during the tests indicates that a higher CPU threshold for scaling increases the likelihood of gateway overload. A potential strategy to address this situation is to reduce the scale threshold based on the number of nodes to be connected.

5.4.3 Fairness Index

During the upscale, when a new gateway instance is added within a PoA, we consider it essential to assess the Fairness Index to understand how the event impacts the balance of connections among operational gateways.

For this analysis, we focused on the scenario with the highest number of mobile nodes involved, namely the one involving the connection of 10000 mobile nodes. We analyzed the three connection intervals already described for

this case: 100ms, 500ms, and 1000ms. Additionally, we restricted our analysis to the scenario where the CPU usage threshold for scaling is 50%.

In **Figures 5.11, 5.12, and 5.13**, we were able to verify, with a 95% confidence level, the test results. In them, we observed that in the three evaluated intervals, the behavior of the Fairness Index follows the same pattern, indicating a drop in the index with each upscale event followed by its recovery in the following minutes.

This behavior aligns with what is expected by the designed system. We have a less fair balance after inserting a new gateway instance because the old connections are not distributed to the new gateway. However, as new connections are made, the index tends toward one again, as these new connections are exclusively directed to the new gateway since it has fewer active connections.

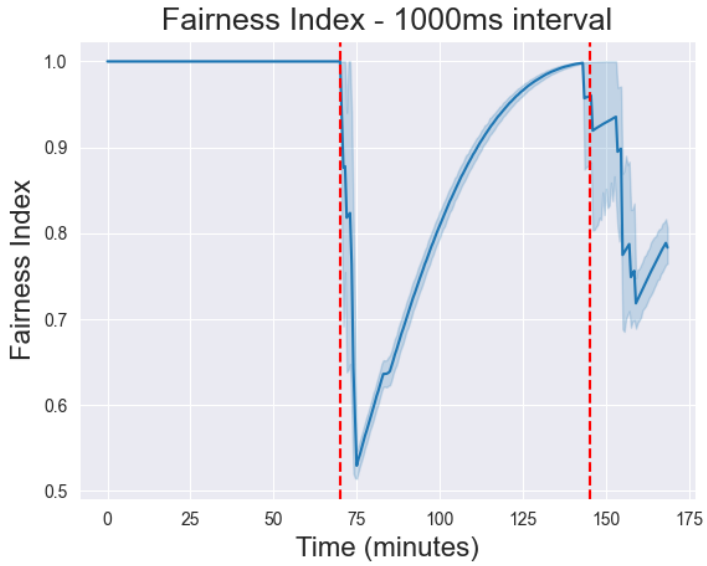


Figure 5.11: Fairness Index in upscale events, represented by the red line, with 1000ms interval between connections

5.4.4 Discussion

In conclusion, this experiment provided an insightful evaluation of the upscale behavior in scenarios with high connection demand. While previous tests focused on gradual connections, this experiment aimed to comprehend the system's response to many mobile nodes connecting at short intervals. The objective was to analyze how scaling performs under varying load conditions, considering different quantities of mobile nodes and distinct CPU usage thresholds to trigger scaling.

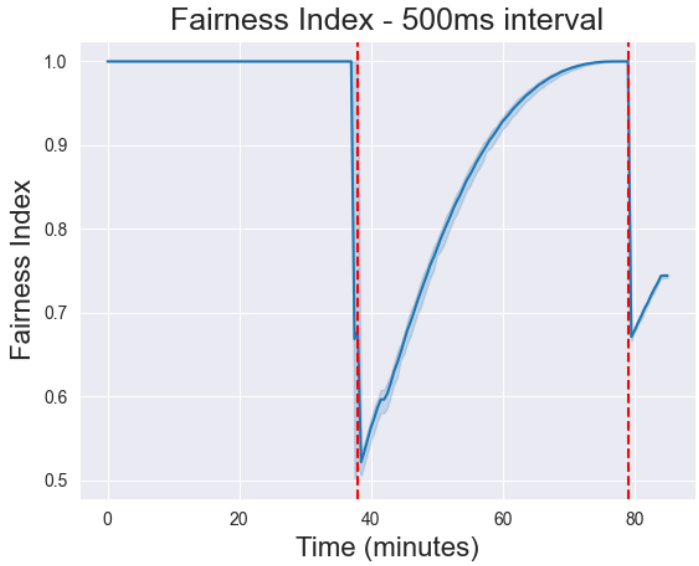


Figure 5.12: Fairness Index in upscale events, represented by the red line, with 500ms interval between connections

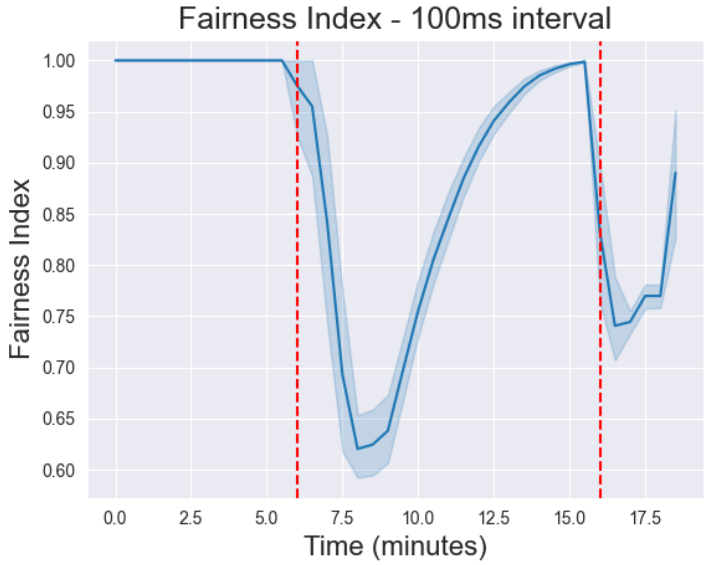


Figure 5.13: Fairness Index in upscale events, represented by the red line, with 100ms interval between connections

The results indicate that increasing the interval between connections and the number of mobile nodes contributes to more stable connectivity, particularly in configurations with 25% and 50% CPU usage. Extending intervals between connections helps to reduce the load on gateways, resulting in fewer connection failures. Moreover, the percentage of failures tends to decrease as the number of mobile nodes increases, suggesting that the number of failures does not rise proportionally with the number of connected mobile nodes.

In the scaling assessment, it was observed that configurations with 25%

Table 5.2: PoA's configurations.

Identifier	Latitude	Longitude	IP
Gramado - RS	-29.39461097	-50.79763940	172.16.2.202/24
Rio de Janeiro - RJ	-23.00040296	-43.33866122	172.16.3.202/24

and 50% CPU usage adopt a preventive approach, utilizing more gateway instances without approaching their limits. However, the 75% CPU configuration, especially at 100ms intervals, exhibits more turbulence, highlighting more intensive CPU usage. This condition may lead to connection failures with loads closer to the gateway limits. Furthermore, the distribution of mobile nodes among instances is influenced by the scaling timing, being more effective in configurations with a lower CPU threshold for scaling activation.

Finally, based on the analysis of the Fairness Index, even though our system does not rebalance active connections during upscale events, we ensure that it always balances new connections in a way that they are distributed as fairly as possible.

As intended, the goal of this experiment is not to provide a definitive answer on how to configure CPU scaling but rather to furnish developers with data to make informed decisions based on their specific scenarios. Considering the results, the suggestion is to adopt scaling approaches that consider the number of connected nodes and the predicted connection flow, adjusting the threshold based on these criteria to mitigate the likelihood of gateway overload.

5.5 Migration between PoAs

The previous experiments involved a single PoA, where we tested availability and connectivity during downscale and upscale moments. However, we also consider it essential to evaluate a scenario involving multiple PoAs, simulating a real-world scenario of mobile node migration to assess the scalability of the involved PoAs and the effectiveness of the architecture proposed in this work.

Therefore, two PoAs were considered for the test: the source PoA (Gramado - RS) and the destination PoA (Rio de Janeiro - RJ). Each PoA has a different IP, representing various geographic regions for the PoA Manager based on latitude and longitude. As mentioned in Section 4.2, the algorithm to determine the best PoA for a mobile node was Haversine, where we considered latitudes and longitudes to calculate the distance between points and create the priority list. The PoAs for tests were configured in the PoA Manager according to **Table 5.2**.

The test was designed so that once a mobile node connects to the source PoA, it has a consistent chance of migrating to the destination PoA. The success of the migration means that the mobile node will not be moved back to the source—effectively a one-way migration.

Furthermore, mobile nodes are connected to the source PoA at intervals of 500 ms, and the likelihood of migration is tied to sending a context message containing the mobile node's latitude and longitude, which is sent every five minutes. Consequently, every five minutes after connecting to the source, the mobile node has a fixed probability of undergoing migration. Lastly, the configuration of the PoAs ensures scaling limits of 4500 connected nodes or 50% CPU usage.

5.5.1 Scalability and multi PoA architecture effectiveness

To assess the solution's scalability, we varied the number of mobile nodes involved in the test to 1000, 2500, 5000, 7500, and 10000. We also varied their migration probability to 25%, 50%, and 75%. The goal was to examine the demand for gateways in each scenario and evaluate the proposed architecture's effectiveness.

In **Figures 5.14**, **5.15**, **5.16**, **5.17**, and **5.18**, we can observe the results for each test scenario.

Figures 5.14 and **5.15** show, with a 95% confidence level, that during the experiment with 1000 and 2500 mobile nodes, there was no need for scaling by either of the two PoAs in any variation of migration probability. This indicates that, in the described and tested situation, there was no CPU overload on the gateways, resulting in the addition of new instances, and there were no more than 4500 mobile nodes to trigger scaling based on the number of mobile nodes. Lastly, we also verified that 99% of the mobile nodes concluded the experiment at the destination PoA, indicating that the test succeeded.

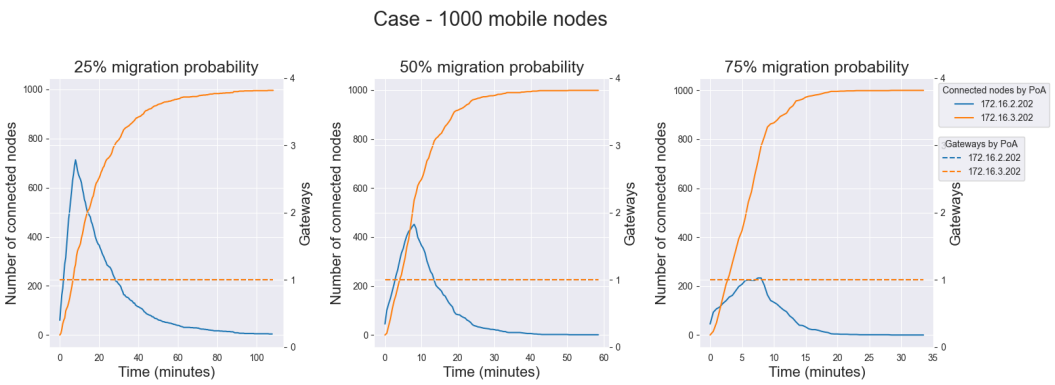


Figure 5.14: 1000 mobile nodes migration

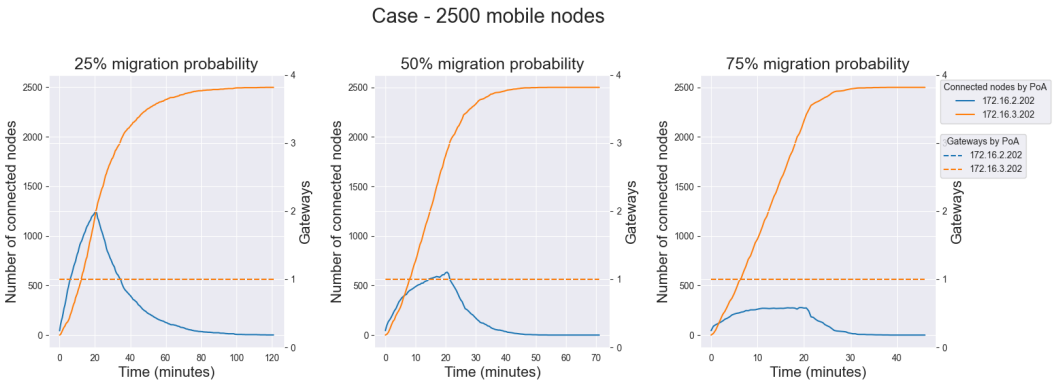


Figure 5.15: 2500 mobile nodes migration

Figures 5.16 and **5.17** show, with a 95% confidence level, that during the experiment with 5000 and 7500 mobile nodes, scaling was required for two gateways in the destination PoA in all variations of migration probability. The graph shows that scaling occurs within the range of 4500 to 5000 connected mobile nodes in the destination PoA, indicating that scaling occurred based on the number of connected mobile nodes. Regarding the connectivity of migrating mobile nodes, it was confirmed that in these test scenarios, 99% of the mobile nodes concluded the experiment at the destination PoA, indicating that the test succeeded.

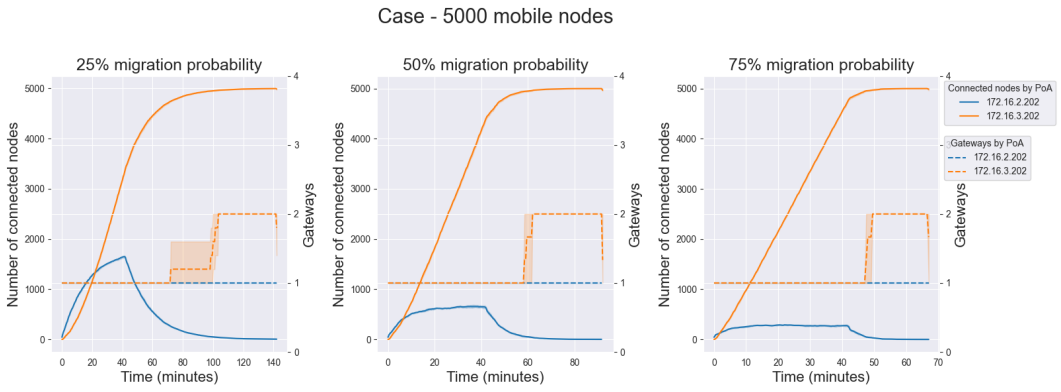


Figure 5.16: 5000 mobile nodes migration

Finally, **Figure 5.18**, with a 95% confidence level, shows that during the experiment with 10000 mobile nodes, scaling was necessary for three gateways in the destination PoA in all variations of migration probability. The graph shows that scaling occurs initially within 4500 to 5000 connected mobile nodes and subsequently within 9000 to 10000 connected mobile nodes, a behavior similar to that described in the scenarios with 5000 and 7500 mobile nodes. This indicates that scaling occurred based on the number of connected mobile nodes. Like the other tests, we also confirmed that in these test scenarios,

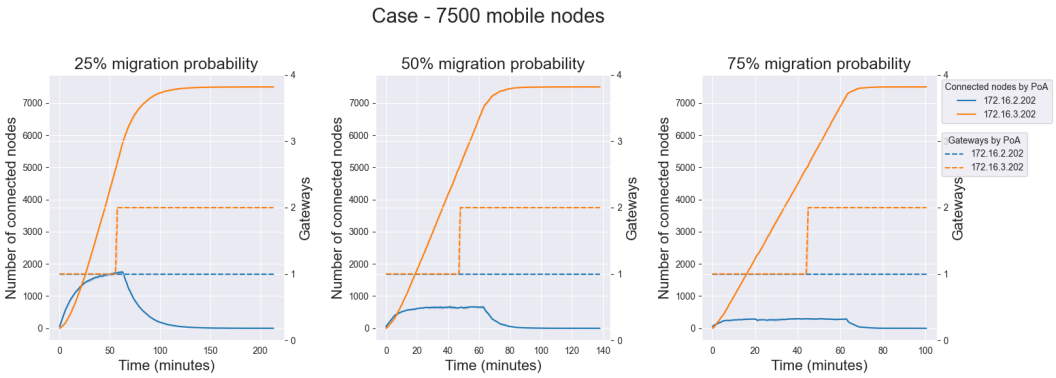


Figure 5.17: 7500 mobiles nodes migration

99% of the mobile nodes concluded the experiment at the destination PoA, indicating that the test succeeded.

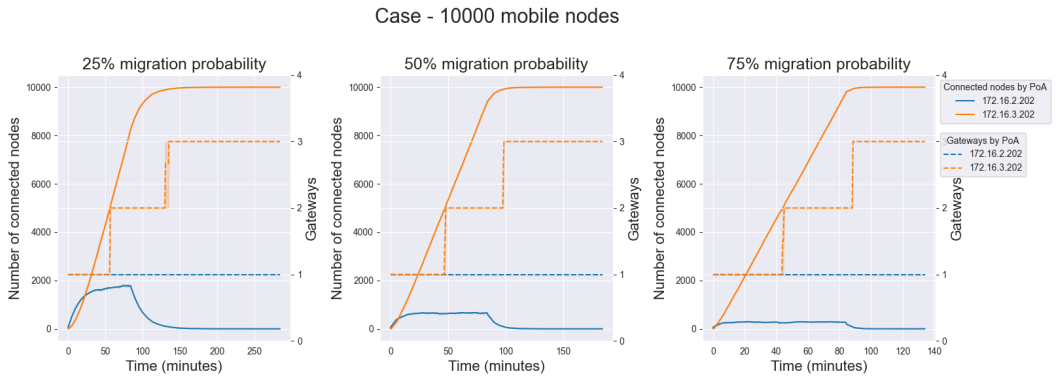


Figure 5.18: 10000 mobile nodes migration

5.5.2 Discussion

As mentioned earlier, based on the experimental data, it can be concluded that the scalable multi-PoA architecture works for the described scenarios. It is also evident that the transition between PoAs for a mobile node occurs as expected according to the geolocation rule used in this work.

However, it is essential to highlight the impact of variations in migration probabilities and their significance for the study. In the test, we used probabilities of 25%, 50%, and 75% for a mobile node to migrate. This results in more excellent retention of nodes in the source PoA for lower probabilities, while for higher probabilities, the retention at the source is lower. Therefore, a scenario where the migration probability is 5% could lead to significant retention at the source, triggering scaling. This is a plausible scenario and demonstrates that the system functions correctly. Additionally, it is essential to note that mobile

nodes were connected at fixed intervals in the experiment, and reducing this interval could also impact the scaling of the source PoA.

To illustrate the meaning of this experiment using the geolocation-based algorithm, we can make an analogy with a real-world scenario: imagine a city where people's movement reflects the probability of migration. At a high probability time, we would have an intense flow of people moving, while at a low probability time, the flow would be reduced. Additionally, there are other possible scenarios, such as using an algorithm that decides the best PoA based on the battery level of mobile nodes. In this case, the probability of migration would be associated with the battery levels set for migration.

6
Related Work

In this section, we present and discuss some research work on autoscaling systems that does not necessarily focus on mobile IoT.

An API gateway was proposed by (SONG; ZHANG; HAIHONG, 2018) as an entry point for backend applications in a microservices architecture, using Kubernetes as the autoscaling system and *Prometheus* as the pod's resource monitoring system. In their approach, they pointed out that the API Gateway is the client's only access point to the backend services, making it a potential bottleneck during periods of high demand with many simultaneous requests. Similar to ContextNet's gateway, the API Gateway presented requires high availability since any bottleneck in the system could result in the entire system being unavailable.

Although the main objective of the (SONG; ZHANG; HAIHONG, 2018)'s experiments was to validate the system's ability to dynamically adjust the number of pods using the Kubernetes HPA in response to variations in service load, no connectivity-related experiments were conducted, since the system in question did not involve client connection persistence. On the other hand, in our experiments, we investigated both the scalability and connectivity of the mobile nodes, ensuring a more comprehensive view and addressing the specific challenges related to mobility and connectivity in our application scenarios.

Sharing some similarities with our work, (DICKEL; PODOLSKIY; GERNDT, 2019) deals with the obstacle of the gateway in a scalable IoT platform, as it is a component that needs to handle many simultaneous connections. Their study addressed the problem of the increase in devices correlating directly with the increase in latency, timeout, and disconnections. To solve this problem, they carried out an experimental study focusing on three metrics: CPU usage, number of concurrent active connections, and throughput per gateway, which were used by the autoscaling system to make horizontal scale decisions.

Our study faced a similar challenge but focused on enabling new connections after gateway saturation. Similar to the study by (DICKEL; PODOLSKIY; GERNDT, 2019), we also observed that the number of simultaneous connections and individual messaging rate influenced the load. However, we chose to rely solely on CPU usage and simultaneous connection metrics for scaling decision-making.

We can see in **Table 6.1** in which aspects the works relate to our study.

Table 6.1: Comparison with related work.

Paper	**Type of autoscaling**	**Metrics**	**Related to IoT/IoMT**
(SONG; ZHANG; HAIHONG, 2018)	HPA	CPU usage, QPS	No
(DICKEL; PODOLSKIY; GERNDT, 2019)	HPA	CPU usage, active connections, throughput of messages	Yes
Our work	HPA	CPU, current connections	Yes

Besides research similar to our proposed work, it was also possible to verify in the work conducted by (ELAMIN; PAARDEKOOPER, 2021) the effect of persistent connections in autoscaling scenarios using HPA. Even after adding new pods, the traffic remains restricted to the initial pod due to the configuration of HTTP session persistence. Their conclusions raised the point that not all traffic patterns benefit from horizontal autoscaling, and in their tests, the scenario of persistent connections was identified as one that does not.

In our scenario, connections between mobile nodes and the gateway are persistent UDP connections. As mentioned earlier, the increase in the number of messages sent by mobile nodes directly impacts the CPU usage of the gateway. Therefore, we might encounter a similar scenario described by (ELAMIN; PAARDEKOOPER, 2021). This fact motivated us to narrow our test scope to a fixed interval for sending messages by mobile nodes.

Related to autoscaling techniques, (VERMA; BALA, 2021) conducted a literature review on IoT cloud applications, focusing on evaluating different QoS (Quality of Service) metrics. The review details threshold-based autoscaling that is the strategy adopted in our study, and discusses various works that use this strategy as a basis for autoscaling, especially in cloud systems that use public providers.

As in our study, most of these systems that adopt the same strategy also use the CPU metric as a decision criterion for autoscaling. Furthermore, like our work, most of the systems reviewed also employ reactive rather than proactive techniques since proactive approaches require more intelligence to predict events before they occur.

7
Conclusions and future work

As the popularity of IoT and IoMT continues to grow, the challenges associated with this domain, particularly those related to real-time communication services and high availability, are becoming increasingly prominent. This study assessed the implementation of a self-scalable architecture using Kubernetes for the ContextNet 3.0 middleware, aiming to make its gateway component self-manageable in response to demand.

The conducted experiments aimed to demonstrate the feasibility of the solution, focusing on the impact that the architecture could have on connectivity and the need for flexibility in configuration for system management, as described in the research questions in Chapter 1.

Our experiments demonstrated the advantages of using an elastic architecture in terms of resource savings compared to a non-elastic architecture. They also showed that the upscale and downscale events resulting from the proposed architecture do not significantly impact the connectivity of mobile nodes or the balancing of connections among operational gateways. Finally, we demonstrated the performance of the multi-PoA autoscalable architecture in different scenarios of mobile node migrations with the aim of guiding ContextNet application developers to configure the system according to needs. Thus, we demonstrate that the architecture proposed in the work addresses Research question 1.

Research question 1
How can we create a self-scalable gateway architecture that does not impact established connections and mobile-cloud availability, and allows the system administrator to configure scalability parameters as needed?

Therefore, the work addresses the solution to the problem below identified in Chapter 1 for the described scenarios, allowing us to conclude that the proposed architecture is a way to deal with the scalability issues of static gateways focusing on establishing connections.

Problem 1
To support the variations in the demand for mobile nodes' connectivity, it is necessary to scale the static gateways so that they can handle these mobile-cloud connections proportionally while ensuring that the elasticity of static gateways is configurable to adapt to different scenarios.

Finally, it is worth noting that the architecture implemented with Kubernetes to add elasticity and load balancing to the gateways of the ContextNet Core for scalable mobile connectivity can be implemented in various other systems where elasticity is needed for resource savings purposes.

7.1 Future Work

As part of future work, we can outline aspects related to the evolution of the multi-PoA architecture and advancements in the ContextNet middleware.

We recognize the potential to incorporate more sophisticated analyses by varying the scaling parameters introduced thus far regarding the architecture's evolution. Moreover, we aim to explore additional scaling parameters beyond the number of connected mobile nodes and CPU usage. Different metrics may offer better scaling performance depending on the specific system context.

In advancing the ContextNet 3.0 middleware, the next phase involves assessing the auto-scaling architecture for protocols not reliant on persistent connections. This exploration entails investigating alternatives beyond connection persistence, as indicated by the limitations identified in this study.

Additionally, while our focus has primarily been on developing and testing autoscalable gateways, we have pursued a simplified implementation of the new PoA Manager deemed necessary. However, we have identified areas for improvement that demand attention in the future.

Primarily, the current configuration of the PoA Manager fixedly sets gateway liveliness. We recognize the need to tailor this configuration to each PoA, especially in scenarios with varying connectivity. This approach would enhance fault tolerance, allowing for greater resilience in PoAs with poorer connectivity and tighter control in PoAs with better connectivity.

Regarding the architecture of the PoA Manager, we want to provide mobile node connectivity data and PoA load data as additional options while implementing ranking algorithms. While the initial implementations only utilized data sent by mobile nodes' context messages, incorporating these additional data can enrich decision-making processes.

8
Bibliography

BERNSTEIN, D. Containers and cloud: From LXC to docker to kubernetes. **IEEE cloud computing**, IEEE, v. 1, n. 3, p. 81–84, 2014.

BURNS, B. et al. Borg, omega, and kubernetes. **ACM Queue**, v. 14, p. 70–93, 2016. Disponível em: <http://queue.acm.org/detail.cfm?id=2898444>.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed Systems: Concepts and Design Edition 3**. [S.l.]: Addison-Wesley.–2001.–779 p, 2001.

DELICATO, F. C. et al. **Resource management for Internet of Things**. [S.l.]: Springer, 2017. v. 16.

DICKEL, H.; PODOLSKIY, V.; GERNDT, M. Evaluation of autoscaling metrics for (stateful) iot gateways. In: IEEE. **2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA)**. [S.l.], 2019. p. 17–24.

DU HAIBIN XIE, W. L. J. **IPVS-Based in-Cluster Load Balancing Deep Dive**. 2018. <https://kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive/>, Last accessed on 2024-01-11.

ELAMIN, M.; PAARDEKOOPER, P. Scaling of containerized network functions. 2021.

ELAZHARY, H. Internet of things (iot), mobile cloud, cloudlet, mobile iot, iot cloud, fog, mobile edge, and edge emerging computing paradigms: Disambiguation and research directions. **Journal of network and computer applications**, Elsevier, v. 128, p. 105–140, 2019.

ENDLER, M.; SILVA, F. S. e. Past, present and future of the contextnet iomt middleware. **Open Journal of Internet Of Things (OJIOT)**, RonPub, v. 4, n. 1, p. 7–23, 2018.

ERMOLENKO, D. et al. Internet of things services orchestration framework based on kubernetes and edge computing. In: IEEE. **2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)**. [S.l.], 2021. p. 12–17.

EUGSTER, P. T. et al. The many faces of publish/subscribe. **ACM computing surveys (CSUR)**, ACM New York, NY, USA, v. 35, n. 2, p. 114–131, 2003.

HUNT, P. et al. {ZooKeeper}: Wait-free coordination for internet-scale systems. In: **2010 USENIX Annual Technical Conference (USENIX ATC 10)**. [S.l.: s.n.], 2010.

IREN, S.; AMER, P. D.; CONRAD, P. T. The transport layer: tutorial and survey. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 31, n. 4, p. 360–404, 1999.

JAIN, R. K. et al. A quantitative measure of fairness and discrimination. **Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA**, v. 21, 1984.

KAYAL, P. Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope. In: IEEE. **2020 IEEE 6th World Forum on Internet of Things (WF-IoT)**. [S.l.], 2020. p. 1–6.

KREPS, J. et al. Kafka: A distributed messaging system for log processing. In: ATHENS, GREECE. **Proceedings of the NetDB**. [S.l.], 2011. v. 11, n. 2011, p. 1–7.

LORIDO-BOTRAN, T.; MIGUEL-ALONSO, J.; LOZANO, J. A. A review of auto-scaling techniques for elastic applications in cloud environments. **Journal of grid computing**, Springer, v. 12, p. 559–592, 2014.

MEADOR, B. A survey of computer network topology and analysis examples. **Washington University**, p. 2–3, 2008.

MESLIN, A.; RODRIGUEZ, N.; ENDLER, M. Scalable mobile sensing for smart cities: The musanet experience. **IEEE Internet of Things Journal**, IEEE, v. 7, n. 6, p. 5202–5209, 2020.

MUDDINAGIRI, R.; AMBAVANE, S.; BAYAS, S. Self-hosted kubernetes: deploying docker containers locally with minikube. In: IEEE. **2019 international conference on innovative trends and advances in engineering and technology (ICITAET)**. [S.l.], 2019. p. 239–243.

MURALIDHARAN, S.; SONG, G.; KO, H. Monitoring and managing iot applications in smart cities using kubernetes. **Cloud Computing**, v. 11, 2019.

NGUYEN, T.-T. et al. Horizontal pod autoscaling in kubernetes for elastic container orchestration. **Sensors**, MDPI, v. 20, n. 16, p. 4621, 2020.

PARDO-CASTELLOTE, G. Omg data-distribution service: Architectural overview. In: IEEE. **23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.** [S.l.], 2003. p. 200–206.

PHAN, L.-A.; KIM, T. et al. Traffic-aware horizontal pod autoscaler in kubernetes-based edge computing infrastructure. **IEEE Access**, IEEE, v. 10, p. 18966–18977, 2022.

PRASETYA, D. A. et al. Resolving the shortest path problem using the haversine algorithm. **Journal of critical reviews**, Innovare Academics Sciences Pvt. Ltd, v. 7, n. 1, p. 62–64, 2020.

PURDY, G. N. **Linux iptables Pocket Reference: Firewalls, NAT & Accounting**. [S.l.]: " O'Reilly Media, Inc.", 2004.

SILVA, L.; ENDLER, M.; RORIZ, M. MR-UDP: Yet another reliable user datagram protocol, now for mobile nodes. **Monografias em Ciência da Computação, nr**, v. 1200, p. 06–13, 2013.

SINGH, G.; KAUR, K. An improved weighted least connection scheduling algorithm for load balancing in web cluster systems. **International Research Journal of Engineering and Technology (IRJET)**, v. 5, n. 3, p. 6, 2018.

SONG, M.; ZHANG, C.; HAIHONG, E. An auto scaling system for api gateway based on kubernetes. In: IEEE. **2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)**. [S.l.], 2018. p. 109–112.

SOUSA, A. B. de et al. Uma plataforma de iot para integração de dispositivos baseada em nuvem com apache kafka. In: SBC. **Anais Estendidos do XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. [S.l.], 2018.

SUKHIJA, N.; BAUTISTA, E. Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus. In: IEEE. **2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)**. [S.l.], 2019. p. 257–262.

TALAVERA, L. E. et al. The mobile hub concept: Enabling applications for the internet of mobile things. In: IEEE. **2015 IEEE International conference on pervasive computing and communication workshops (percom workshops)**. [S.l.], 2015. p. 123–128.

VERMA, S.; BALA, A. Auto-scaling techniques for iot-based cloud applications: a review. **Cluster Computing**, Springer, v. 24, n. 3, p. 2425–2459, 2021.

WANOUS, C. A. **Reengenharia do ContextNet utilizando Kafka**. Tese (Mestrado em Ciência da Computação) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2021.

ZHANG, W. et al. Linux virtual server for scalable network services. In: **Ottawa Linux Symposium**. [S.l.: s.n.], 2000. v. 2000.