Universidade de Aveiro
2023

Rodrigo de Larmand Alvim Leal Rosmaninho

Estratégias de Orquestração para Serviços de Cidades Inteligentes com Restrições Temporais

Orchestration Strategies for Time-Constrained Smart City Services

Universidade de Aveiro
2023

Rodrigo de Larmand Alvim Leal Rosmaninho

Estratégias de Orquestração para Serviços de Cidades Inteligentes com Restrições Temporais

Orchestration Strategies for Time-Constrained Smart City Services

"*Nobody said it was easy. No one ever said it would be this hard.*"

— Coldplay, The Scientist

Universidade de Aveiro
2023

Rodrigo de Larmand Alvim Leal Rosmaninho

Estratégias de Orquestração para Serviços de Cidades Inteligentes com Restrições Temporais

Orchestration Strategies for Time-Constrained Smart City Services

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Pedro Filipe Vieira Rito, Investigador Auxiliar do Instituto de Telecomunicações de Aveiro, e do Doutor Duarte Miguel Garcia Raposo, Investigador Auxiliar do Instituto de Telecomunicações de Aveiro, e colaboração da Doutora Susana Isabel Barreto de Miranda Sargento, Professora catedrática do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho aos meus pais, Alexandra e Francisco

agradecimentos / acknowledgements

Em primeiro lugar quero agradecer aos meus pais. De todas as secções desta dissertação, esta é sem dúvida a mais difícil de exprimir corretamente em palavras: Tudo aquilo que sou e todos os pequenos sucessos que vou alcançando devem-se quase exclusivamente à vossa educação impecável e a todas as maravilhosas aventuras e experiências que me proporcionaram ao longo dos anos. Vocês são e sempre foram incríveis como pais. Nunca duvidem disso. Muito obrigado pelo vosso apoio e amor incondicionais durante este mestrado e, claro, em tudo o resto. E peço desculpa pela frustração dos últimos meses.

Ao Eurico Dias, Pedro Valério, Gonçalo Perna, João Trindade, e Ricardo Carvalho quero estender um profundo obrigado por todas as noitadas que passámos a gerir o eterno malabarismo de projetos e entregas. Tenho um orgulho tremendo pelo caminho que fizemos juntos, as adversidades que enfrentámos, e tudo aquilo que conseguimos construir. Vocês são engenheiros exemplares e conseguem transpor qualquer desafio. Mas o curso não se resume apenas a estudo e projetos, e portanto também quero agradecer a todos os outros colegas de matrícula que ajudaram a tornar estes 6 anos numa experiência verdadeiramente memorável, em especial ao Pedro Valente, Joaquim Ramos, Pedro Almeida, e André Alves. Espero que todas estas amizades perdurem durante muito mais tempo. Agradeço igualmente a todos os colegas que integraram o Grupo de Linux da Universidade de Aveiro e a Comissão de Curso do MIECT, e também ao meu patrão da faina académica, João Pedro "Planck" Fonseca, que se não existisse teria definitivamente que ser inventado.

Nestes últimos 3 anos e meio tive a sorte tremenda de integrar a equipa incrível do grupo Network Architectures and Protocols do IT-Aveiro, onde ganhei a vasta maioria dos conhecimentos e experiência que me permitiram desenvolver este trabalho. Cada dia traz novas oportunidades e desafios, mas o que permanece constante é a aprendizagem e a companhia. Quero agradecer a todos os colegas que me ajudaram com os meus problemas e me permitiram aprender com os deles, em especial à Andreia Figueiredo, Rui Lopes, Pedro Teixeira, e Christian Gomes. Vocês são incríveis, e mais pessoas deviam ter a sorte de vos conhecer.

O mesmo também se estende, naturalmente, aos professores. Os últimos meses foram um pouco atribulados e causaram, sem dúvida, alguma frustração. No entanto, nunca deixei de sentir o vosso apoio, que foi muito acima de qualquer expectativa.

Ao Dr. Duarte Raposo: Sem ti este trabalho não teria existido. A tua ajuda foi imprescindível em todos os níveis. Muito obrigado por toda a tua orientação, as discussões, e a motivação sincera nos momentos mais complicados.

Ao Dr. Pedro Rito: É um prazer poder trabalhar contigo todos os dias. Não há nenhum obstáculo ou dificuldade técnica que sobreviva a uma pequena reunião contigo. Estabeleces uma barra altíssima para qualquer outro "manager" que eu venha a ter no futuro. Muito obrigado.

À Prof. Dra. Susana Sargento: Como já tive oportunidade prévia de dizer, não existe nenhuma pessoa que tenha feito parte do meu percurso académico na UA a quem eu deva mais agradecimentos do que a professora. Muito obrigado por todas as oportunidades e pequenos empurrões.

Por fim, queria agradecer ao Instituto de Telecomunicações e ao projeto IMMINENCE (POCI-01-0247-FEDER-112314), financiado através do Fundo Europeu de Desenvolvimento Regional (FEDER), através do Programa Operacional Competitividade e Internacionalização (COMPETE 2020) do Portugal 2020.

Palavras Chave

Cidades Inteligentes, Computação Edge, Redes Veiculares, Orquestração, Kubernetes, Escalonamento em Tempo Real

Resumo

À medida que a escala e a complexidade da infraestrutura de uma cidade inteligente aumentam, também aumenta a necessidade da existência de estratégias de orquestração que maximizem a automação, eficiência, tolerância a falhas, flexibilidade, e observabilidade do sistema. No entanto, estes tipos de infraestruturas apresentam, por natureza, um conjunto de desafios que devem ser devidamente considerados. Para alguns tipos de serviços críticos de cidades inteligentes, manter um limite superior de tempo para a produção de resultados pode ser mais importante do que os resultados em si. Sem uma planificação cuidadosa, estas restrições temporais podem ser frequentemente violadas devido a interferência causada por outras aplicações que estejam a ser executadas no mesmo nó com recursos escassos, que tipicamente compõem a infraestrutura de computação na Edge de uma cidade inteligente. A instalação de uma solução de orquestração pré-existente, sem modificações, pode exacerbar estes problemas devido ao overhead introduzido e ao facto do lançamento dos serviços ser sub-ótimo, consequente da falta de discriminação entre serviços críticos e serviços pouco prioritários.

Esta dissertação propõe uma estratégia tripartida para endereçar estes problemas. Em primeiro lugar, as próprias aplicações devem ser construídas de acordo com boas práticas que aumentam a eficiência do processo de orquestração e minimizam a latência de processamento e de comunicação. Como caso de estudo, foi implementada uma nova suite protocolar ETSI C-ITS utilizando uma arquitetura baseada em microserviços, que representa uma mudança de paradigma na forma como as aplicações C-V2X são implementadas, como comunicam com as redes veiculares e entre si, e como podem ser orquestradas de forma eficiente.

Em segundo lugar, as aplicações críticas devem ser executadas com configurações especializadas para mitigar a interferência proveniente de outros processos, como, por exemplo, o escalonamento em tempo real. Foi realizada uma avaliação experimental detalhada com o objetivo de estudar os efeitos que a utilização de vários tipos diferentes de configurações de runtime têm no desempenho de uma aplicação a ser executada numa *single board computer* de computação edge.

Finalmente, as ferramentas de orquestração para cidades inteligentes devem ser estendidas com lógica específica ao domínio e novas funcionalidades e de orquestração dinâmica, que promovem uma alocação mais ótima de serviços para nós, e a estabilidade do *cluster* a longo termo.

Para validar o funcionamento da sua implementação, cada componente desenvolvido foi submetido a testes individuais e costumizados num cluster virtualizado. Os resultados demonstram que todos os componentes apresentam níveis de desempenho dentro do esperado e que, no geral, representam melhorias significativas face ao comportamento nativo do Kubernetes.

Keywords

Smart City, Edge Computing, Vehicular Ad-hoc Networks, Orchestration, Kubernetes, Real-Time Scheduling.

Abstract

As the scale and complexity of a smart city infrastructure increases, so does the need for comprehensive orchestration strategies that maximize automation, efficiency, fault-tolerance, flexibility, and observability. However, this type of infrastructure inherently presents some challenges that must be considered before opting to deploy one of the existing orchestration solutions. For some types of critical smart city services, maintaining an upper bound for the time required for producing the results can be more important than the results themselves. Without careful planning and tuning, these timing constraints can frequently be violated due to interference from other applications running on the resource-constrained nodes that typically comprise a smart cities' edge computing infrastructure. Deploying one of the freely available orchestration solutions, without modification, can actually exacerbate these issues by introducing overhead and mismanaging deployments due to not being able to discriminate between critical and low priority services.

This dissertation proposes a three-pronged approach to address these issues. Firstly, the applications themselves should be designed using best practices that increase orchestration efficiency and minimize processing and communication latency. As a case study, a new ETSI C-ITS protocol stack was implemented using a microservice architecture that represents a paradigm shift in the way that C-V2X applications are designed, in how they interact with each other and with the VANET, and also in how they can be orchestrated in an efficient manner.

Secondly, critical applications should be executed using specialised configurations that mitigate interference by other processes, such as Real-Time scheduling. A comprehensive experimental evaluation was performed in order to study the effects of several different runtime configurations on the performance of an application in an edge computing single board computer.

Finally, smart city orchestration tools should be extended with domain-specific logic, new functionalities and dynamic orchestration that promote a more optimal placement of services in worker nodes and the long-term stability of the cluster.

In order to validate their implementation, each component was individually tested on a virtualised cluster using bespoke methodologies for each one. Results show that all the components performed within the levels that were expected and, in general, exhibited significant improvements relative to Kubernetes's default behaviour.

Contents

List of Figures

List of Tables

Abbreviations

5GAA	5G Automotive Association
5GAA	5G Automotive Association
ATCLL	Aveiro Tech City Living Lab
BL	baseline
CAM	Cooperative Awareness Message
CFS	Completely Fair Scheduling
Cgroup	Control group
C-ITS	Cooperative Intelligent Transport Systems
CTLTA	Cross-Traffic Left-Turn Assist
CCDF	Complementary Cumulative Distribution Function
EBW	Emergency Brake Warning
EDF	Earliest Deadline First
HDSS	Automated Driving Lane Change
HP	High Priority
IMS	Intersection Movement Assist
IoT	Internet-of-things
IPC	Inter-process communication
ITS	Intelligent Transportation Systems
KASAN	Kernel Address Sanitizer
STIBP	Single Thread Indirect Branch Predictors
LP	Low Priority
MAPEM	MAP Extended Message
VAM	VRU Awareness Message
MEC	Multi-Access Edge Computing
ND	Native as Docker
OBU	On-Board Unit
OS	Operating System
RSU	Road Side Unit
RT	Real-Time
RTOS	Real-Time Operating System
RTSA	Real-Time Situational Awareness
SBC	Single-Board Computer
SDK	Software Development Kit
SoC	System on a Chip
SP	Same Priority
STP	See-Through for Passing
TC	Traffic Control
URLLC	Ultra-Reliable and Low-Latency
V2X	Vehicle-to-everything
VANET	Vehicular Ad-hoc Network
VEC	Vehicle Edge Computing
VHP	Very High Priority
VM	Virtual Machine
VRU	Vulnerable Road User
WCET	Worst-Case Execution Time
CPU	Central Processing Unit
RAM	Random Access Memory
GPU	Graphics Processing Unit
RTT	Round-Trip Time
DNAT	Destination Network Address Translation
gRPC	Google Remote Procedure Call
API	Application Programming Interface
DNS	Domain Name System
IP	Internet Protocol
VM	Virtual Machine
EDF	Earliest Deadline First
CFS	Completely Fair Scheduler
FIFO	First-In First-Out
CRI	Container Runtime Interface
CNI	Container Network Interface
CLI	Command Line Interface
HTTP	Hypertext Transfer Protocol
TC	Traffic Control
CSV	Comma-Separated Values
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
CNI	Container Network Interface
URL	Uniform Resource Locator
LAN	Local Area Network
MQTT	Message Queuing Telemetry Transport
DDS	Data Distribution Service
ETSI	European Telecommunications Standards Institute
JSON	JavaScript Object Notation
GPS	Global Positioning System
POSIX	Portable Operating System Interface
SysV	Unix System V
RSSI	Received Signal Strength Indication
CUDA	Compute Unified Device Architecture
CR	Custom Resource

CHAPTER 1

Introduction

1.1 Motivation

As the scale and complexity of a smart city infrastructure increases, so does the need for comprehensive orchestration strategies that maximize automation, efficiency, fault-tolerance, flexibility, and observability. However, this type of infrastructure inherently presents some challenges that must be considered before opting to deploy one of the existing orchestration solutions.

For some types of critical smart city services, maintaining an upper bound for the time required for producing the results can be more important than the results themselves. Without careful planning and tuning, these timing constraints can frequently be violated due to interference from other applications running on the resource-constrained nodes that typically comprise a smart cities' edge computing infrastructure.

To address this issue, further research is required in order to analyse the performance impacts of this type of interference, specifically in edge computing hosts, as well as any potential performance and stability gains that can be attained with the use of different mitigating strategies such as, for example, the use of Real-Time scheduling. Additionally, since most orchestration solutions require applications to be packaged and executed in containerised environments, this research effort must also determine the amount of overhead introduced with such runtime environments, and how the aforementioned mitigating strategies can be applied in that scenario.

In order to support some of its most promising use cases (e.g., autonomous vehicles, C-V2X services), smart city orchestration tools should support these delay-sensitive application requirements. Instead, current container-based virtualisation technologies are mainly designed for scalability and mostly target cloud environments where these problems are not so relevant. Therefore, deploying one of the freely available orchestration solutions, without modification, can actually exacerbate these issues by introducing overhead and mismanaging deployments due to not being able to discriminate between critical and low priority services.

As such, the use of these technologies in this context also requires further research concerning how they can be extended with domain-specific logic to enable time constraint awareness in orchestration decisions, and possibly apply the aforementioned mitigating strategies automatically.

Finally, an additional challenge that hinders the effective use of orchestration strategies in smart city environments is the fact that legacy applications are not always designed and implemented in a way that can most benefit from the orchestration paradigm. This lack of effectiveness usually results in sub-optimal deployment allocation, misused resources, and higher processing and/or communication latency. This problem is especially noticeable in some particular contexts like, for instance, C-V2X applications that are built using traditional monolithic architectures that lead to redundant processing and mis-spent storage space. For that reason, such services should be re-designed to follow a more modern architecture with orchestration support taken into account.

1.2 Goals and contributions

1.2.1 Goals

The main goal of this Dissertation is to propose orchestration strategies for smart city services with time-constrained requirements in a cloud-edge-based approach. To address this objective, several steps have to be taken, starting with the study and evaluation of existing inter-process interference mitigation strategies, up to the proposal of a dynamic orchestration approach that autonomously addresses those constraints.

This dissertation therefore encompasses 3 different objectives. The first one comprises the study of different service architectures and design principles, and their overall suitability for deployments in an edge computing cluster environment. This analysis will focus on the effectiveness of a service's system resource usage and communication with its dependencies, as well as other strategies that can improve the synergy between a service and the orchestrator, such as how the application can be dynamically configured and how its performance can be measured and reported. This will be exemplified through the implementation of a vehicular communication network service, *Vanetza-NAP*, which will be used as a case study of a microservice-oriented architecture.

The second objective aims to assess how edge and fog concepts have been applied to vehicular networks under VEC, to perform computational offloading, as well as content caching, data management, flexible network management, security, and others. This will be accomplished by assessing different strategies of VEC service deployments in resource-constrained nodes to understand their impact to the deterministic nature of some critical VEC services, mainly through service prioritisation and preemption.

Finally, the third objective is the proposal of an orchestration framework that includes various strategies, applying the knowledge acquired in pursing the first two objectives. The aim of this approach is to automate the configuration of real-time scheduling parameters for critical processes; schedule workloads to nodes, taking into account several factors such as the latency and application-level metrics of their dependencies; automate the migration of workload allocations that become sub-optimal over time; and load balance network requests, also taking into account the latency and application-level metrics of the destination service.

1.2.2 Contributions

Considering the aforementioned objectives, which were all addressed in this dissertation, the following contributions can be highlighted:

- *Vanetza-NAP* has been proposed, as an extension to the *Vanetza* project, to provide better support for MEC scenarios, with improved scalability, and envisioning time-sensitive applications.
- *Vanetza-NAP* is now being used in several ongoing research work, including other Master's degree dissertations and PhD Theses focusing on Vehicle-to-everything (V2X) use cases such as collective perception, vulnerable road users, and virtual traffic lights. It is also used as an educational tool in the Autonomous Networks and Systems course of the Master's degree in Computer and Telematics Engineering at the University of Aveiro. Finally, it is currently deployed in the production environment of the Aveiro Tech City Living Lab (ATCLL) project, where it is included in all of the Road Side Unit (RSU)s and On-Board Unit (OBU)s within the infrastructure.
- Understanding of the performance of several data brokers (MQTT, ZeroMQ and DDS) in the face of real-time and deterministic applications.
- Understanding of the performance impact of different runtime environments and configurations (Native, Containerised, PREEMPT_RT, the impact of cgroup level, and the use of real-time prioritisation).
- Orchestration framework that is able to perform automatic configuration of real-time scheduling parameters for critical processes depending on the worker nodes and network conditions, and the priority of the running services. Several algorithms and plugins have been proposed, and an extensive evaluation is also a contribution of this work.

The work presented in this dissertation results is several demonstrations and publications. First, it has been a great basis for the V2X communication and orchestration in the Aveiro Tech City Living Lab platform, and therefore, the following paper is also a result of this work:

- P. Rito, A. Almeida, A. Figueiredo, C. Gomes, P. Teixeira, R. Rosmaninho, R. Lopes, D. Dias, G. Vítor, G. Perna, M. Silva, C. Senna, D. Raposo, M. Luís, S. Sargento, A. Oliveira, and N. B. de Carvalho, "Aveiro Tech City Living Lab: A Communication, Sensing and Computing Platform for City Environments," IEEE Internet of Things Journal, pp. 1–1, 2023.

The second paper also makes use of this work to provide the basis for the cooperative perception and the interaction with the autonomous shuttle, which paper is the following:

- Joao Amaral, Joao Viegas, Bruno Lemos, Pedro Almeida, Rodrigo Rosmaninho, Gonçalo Perna, Pedro Rito, Susana Sargento "Autonomous Shuttle Integrated in a Communication and Sensing City Infrastructure", IEEE International Conference on Mobility: Operations, Services, and Technologies (MOST), Detroit (MI) USA, 17-19 May 2023.

The third paper is the result of the work on the performance of different runtime environments and configurations, and the use of real-time prioritisation in V2X and VEC environments:

- Rodrigo Rosmaninho, Duarte Raposo, Pedro Rito, Susana Sargento, "Time Constraints on Vehicular Edge Computing: A Performance Analysis", 36th IEEE/IFIP Network Operations and Management Symposium (NOMS 2023), Miami, Florida (USA), 08-12 May 2023.

Finally, a conference paper about the *Vanetza-NAP* and the broker-based approaches, and a journal article about the orchestration strategies will be submitted in the near future.

1.3 Document structure

In total, this dissertation is organised into seven chapters. The remaining chapters are described in the following paragraphs.

Chapter 2 - Background and Related Work - This chapter presents the background information and concepts required to provide context on the topics that are central to this dissertation. These concepts include Smart Cities, VANETs, Real-Time Scheduling, Service Deployment Strategies, and Kubernetes, among others. Finally, it also presents an analysis of related works in this field.

Chapter 3 - Vehicular Communications in Microservice Architectures - This chapter discusses application design principles that help reduce latency and increase the orchestrator's efficiency. This discussion uses Vehicular Communications as a case study, presenting background concepts on ETCI C-ITS protocol stacks and describing the proposed *Vanetza-NAP* solution. The chapter ends with the evaluation of several broker-based solutions in microservice-oriented architectures.

Chapter 4 - Evaluation of Latency Reduction Strategies - This chapter presents a comprehensive experimental study on the performance of edge computing services (in this case *Vanetza-NAP*) in several different runtimes and configurations, and with the use of inter-process interference mitigation strategies.

Chapter 5 - Intelligent Service Orchestration Proposal - This chapter describes the orchestration solution envisioned for this dissertation, including its requirements and the proposed architecture. Moreover, it provides an extensive discussion regarding the implementation of each major component, the rationale behind the decisions that were taken, and the proposed plugins and algorithms.

Chapter 6 - Results and Evaluation - This chapter presents the methodologies used to validate the various components described previously, and presents and discusses the respective results.

Chapter 7 - Conclusion and Future Work - This chapter discusses the proposed approaches, their algorithms and components, and their final results. Finally, it presents some avenues for future work.

CHAPTER 2

Background and Related Work

This chapter presents some background information relevant to the work of this Dissertation, as well as related work on each area to be addressed. In the next chapters, relevant related work is also included to better position the proposed approaches.

2.1 Smart City Services and Technologies

Throughout the years and around the world, the definition of a Smart City has been in a state of constant flux, shaped not only by technological progress, but also by the range of services and applications provided to their residents. Although the concept has been more widely implemented in Europe than in other regions, there is a growing tendency towards the development of living labs within Smart City environments as part of their evolution.

To enable advanced Internet of Things (IoT) applications like self-driving cars, smart city surveillance, and virtual/augmented reality, it is essential to have cloud services in close proximity to IoT devices for optimal performance. For example, to ensure safe autonomous driving, vehicular applications must have a service delay of only a few milliseconds, as any delay in information could cause dangerous conditions on the road. An enhancement for the Smart City service and technologies is the possibility to use edge computing. Edge computing brings communication, storage, and computational capabilities closer to the end-users, allowing the reduction of network load and latency, and consequently enabling real-time operations. Edge devices can collaborate to perform functions like data collection, processing, model training, caching, and data analytics. This is achieved through methods such as edge caching, edge training, and edge offloading, which enhance the intelligence of IoT services at the network edge [1].

Some challenges of bringing edge computing to production environments are QoS, data management and network scalability. Additionally, more challenges are arising lately related to: security and privacy, communications reliability, applications placement and resource allocation, integration in the future 5G and beyond ecosystem [2]. Moreover, we need to consider that edge computational and storage capabilities are limited. For example, for real-time ITS (Intelligent Transportation Systems) big data analytics, both cloud and edge computing should be used. In that case, we need to investigate how to orchestrate a virtualization environment with characteristics to handle this heterogeneous scenario [3]. For instance, in [4] the authors concluded that there is no ideal approach, and the decision of edge-only, hybrid, or cloud-only should be made considering the application characteristics. In [5], resource allocation between the edge and the cloud is studied, considering the maximum acceptable delay. At [6], the authors proposed a framework based on fog nodes to efficiently monitor the network, avoiding unnecessary processing and data upload.

The authors discussed in [7] the significance of the computation location and elaborated on two architectures: cloud and MEC. They weighted the advantages and disadvantages of the former and emphasized the benefits of the latter in a 5G setting. In addition, they delved into the relationship between MEC and SDN/NFV, and emphasized the crucial role of the edge location. The authors also outlined the essential criteria for deploying MEC in the network.

The impact and implications of 5G on ITS, a vital smart city application, were discussed in [8]. The authors highlighted the technological advancements and economic benefits associated with 5G that would affect various industries in smart cities, including public transport, manufacturing, health care, and energy.

An example of a Smart City platform which supports edge computing is the ATCLL. The ATCLL comprises a large number of Internet-of-things (IoT) devices with communication, sensing and computing capabilities constituting a Smart City infrastructure [9]. The communication technologies, built on fiber and Millimeter-wave (mmWave) links, integrate multiprotocol networking with radio terminals (WiFi, ITS-G5, C-V2X, 5G and LoRa), spread throughout 44 connected points of access in the city. Additionally, public transportation has also been equipped with vehicular communication and sensing units. All these points combine and interconnect a set of sensors, such as mobility (radars, LiDARs, video cameras) and environmental sensors. The RSU combine communication and computing devices such as PCEngines APU2 units[1] and NVidia Jetsons, forming a VEC platform in a real environment. With this platform, the ATCLL supports the deployment of a wide range of services and applications: IoT, emergency and safety, ITS and assisted driving, and environmental monitoring.

The main platform for conducting real experiments on the developed solutions in this dissertation will be ATCLL.

2.2 Time Constraints

Edge devices with processing capabilities can be used to perform several fog computing tasks like offloading, content caching, data management, security and privacy. Each task presents specific requirements that need to be carefully identified to select the appropriate service priority and location, like for instance, when deployed on VEC devices -RSU or OBU. The work presented in [10] splits current vehicular use cases in three categories: road

[1] `https://www.pcengines.ch/apu2.htm`

Table 2.1: Time constraints in VEC services [11]

Use Case	**Message Size**	**Reliability**	**Upper-bound Latency**
CLCTA-US1	300B CAM	90%	100 ms
CLCTA-US2	1000B Extended CAM	99.9%	10 ms
IMS	300B BSM/CAM	99.9%	120 ms
EBW-US1	200B-400B CAM/BSM	99.9%	120 ms
EBW-US2	200B-400B CAM/BSM	99.9%	120 ms
RTSA	300B BSM/CAM	99%	100 ms
VRU	20-40Mbps (depends C-ITS)	99.9%	20 ms / 100 ms
HDSS	Process and unprocessed data	99.9%	10 ms
STP	15Mbps (video streaming)	99%	50 ms

safety, traffic efficiency and value-added applications, such as infotainment and internet access. Road safety applications are services with the aim of lowering the risk of accidents; traffic efficiency applications minimize travel time and alleviate traffic congestion; and lastly, value-added applications provide infotainment, path planning and internet access. A more detailed definition is presented in [11]. The 5G Automotive Association (5GAA) has published two white papers where automotive use cases and application's requirements are evaluated in terms of latency and reliability [11][12].

The 5GAA starts by the definition of the information and the service level latency. In the time domain, service level latency is defined *"by the time measured between the occurrence of the event to the beginning of the resulting action"*. Such time could have additional latency which needs to be taken into consideration: 1) the latency of the processing of the event by the information generator; 2) the communication of the information using the VANET; 3) the processing of the information by the end-user; and lastly, 4) the time to actuate.

Deterministic requirements are introduced by the notion of upper-bound latency. In [11] this definition could be found in several examples (see Table 2.1) like in the Cross-Traffic Left-Turn Assist (CTLTA); Intersection Movement Assist (IMS); Emergency Brake Warning (EBW); Real-Time Situational Awareness (RTSA); in the awareness of the presence of Vulnerable Road User (VRU) near potentially dangerous situation; Automated Driving Lane Change (HDSS); and in See-Through for Passing (STP). As an example, EBW performs data fusion tasks between several sensors like radar and camera vision, and needs to send an alert within an upper-bound latency of 120 ms. A unique service or a chain of services could be used to perform the use case (object detection, data fusion service, communication service, etc).

Another example is the awareness of the presence of VRUs, that could use several Cooperative Intelligent Transport Systems (C-ITS) to process the received raw data and generate the alert to the user. The alert should be generated within an upper-bound of 100 ms, with a recommended communication latency of 20 ms. By surveying the works presented in [13]–[15], some real measurements can be found. In [13] and [14] the authors present measurements of the IEEE 802.11p communication delay, showing values of couples of milliseconds. In [15], it is considered the communication delay, and also the processing of the ITS-G5 stack messages, for instance CAM, implemented with Alex Voronov's GeoNetworking stack [16]. The results present an average delay of 50 ms, showing how important it is to consider the latency for processing the exchanged information on the edge, which depends on the deployment technologies explored in Chapter 4.

2.3 Real-Time Scheduling

When attempting to guarantee stable and predictable task execution times, it is important to consider how the underlying operating system schedules the allocation of CPU time between all the active threads. Since Linux is a general purpose operating system, by default, tasks are scheduled using the Completely Fair Scheduler (CFS) algorithm [17], which strives to maintain the balance (fairness) in providing processor time to tasks. However, the fairness approach can make critical services vulnerable to resource starvation [18] in systems with a high number of concurrent low priority tasks that also require CPU time, causing unexpected delays. In fact, general purpose operating systems are considered ill-suited for running tasks that require deterministic guarantees on their Worst-Case Execution Time (WCET). As such, those types of tasks are generally run on more specialized Real-Time Operating System (RTOS), designed to use scheduling algorithms with strictly predictable behaviour, to maximize preemptability, and to provide a robust set of tools to classify each task's priority, periodicity, and/or deadline constraints (e.g., Xenomai and VxWorks [19]).

Fortunately, Linux also provides native support for a set of Real-Time (RT) scheduling policies that coexist with the regular CFS scheduling, and allow the developer to assign a RT priority to their processes, the RT scheduler. Or, alternatively, to set a period and deadline for execution, by using the Earliest Deadline First (EDF) scheduler [17]. Using this model, RT scheduling takes precedence over CFS. As such, any RT tasks that are ready to run will be given priority over regular or low priority tasks. Furthermore, lower priority tasks that have been allocated CPU time will be preempted by higher priority tasks that become ready. This approach greatly increases the level of consistency of the execution time of critical tasks running on Linux. Nevertheless, there are a number of strategies that can further improve the system's predictability. One such strategy is to apply the PREEMPT_RT [20] patch to the Linux kernel, which maximizes the preemptability of the kernel code itself, so that it can also mostly be superseded by real-time tasks. Alternatively, there are solutions such as Xenomai, which implement a Co-Kernel that runs alongside the regular Linux kernel and is responsible for the scheduling of any real-time tasks [21]. Crucially, the Co-Kernel has a higher priority than the regular kernel's functions, and, as such, every time that an RT task needs to be run, the Linux kernel is preempted in its entirety. This technique results in even greater levels of preemptability, but, on the other hand, it involves a more complex installation effort, and increased overhead due to frequent context switches when RT tasks are migrated to Linux to allow the usage of Linux syscalls.

2.4 Service Deployment Strategies

The use of different technologies in service deployments can impact time-critical VEC services. In this section, service deployment techniques are explored. Typical solutions for native service deployments are presented, together with the use of virtualization, containerization and hypervisors approaches.

2.4.1 Using Bare-Metal

Services can generally be deployed using several different methodologies, depending on several factors such as the nature of the workloads, their constraints, scale, resources and the complexity of the edge computing infrastructure [22]. In small scale edge computing environments, it may be preferable to deploy these services directly on the host machine's Operating System (OS), usually Linux. Such deployments may be configured, using the *crontab* to schedule the execution of a binary on each boot; or by leveraging *systemd* to register the program as a system or user service, which will provide more comprehensive features such as inter-service dependency control, automatic failure detection and restarts, and easier log access [23].

These simpler approaches present the smallest learning curve and avoid any virtualization overhead. They could also prove necessary in cases where services require access to specialized resources, interfaces, or shared memory, which might be impractical or impossible in virtualized contexts [24]. However, these types of deployments can become increasingly harder to manage and maintain, particularly in scenarios involving compute nodes that are heterogeneous in terms of system architecture, operating system, and other factors. In fact, given that each service typically has its own set of constraints and strict dependency requirements, some workloads may prove impossible to run natively on a given type of node, and, furthermore, dependency version incompatibilities may occur between services that should run on the same node.

2.4.2 Using Virtualization

Most of previous challenges can be solved by virtualization based deployments, in which the software is pre-packaged with all of its dependencies and runs in isolation from the host OS' environment and other services (see Figure 2.1). In some cases, this isolation may extend to the kernel level and to the use of Virtual Machines (VMs). A clear downside of this approach is the increase in processing and resource access overhead, the extent of which heavily depends both on the type of virtualization technology, and on the characteristics of the specific workloads being considered.

The widespread use of full VMs with virtualized hardware is generally not feasible on edge computing architectures that mostly consist of resource limited Single-Board Computers (SBCs), as is frequently the case on VEC scenarios. Alternatively, containerization is a promising way of leveraging the benefits of virtualization while potentially avoiding most of the overhead [25]. The containerized app and its dependencies are isolated and run on top of the host's kernel and OS, and they can share any common libraries, thus cutting on

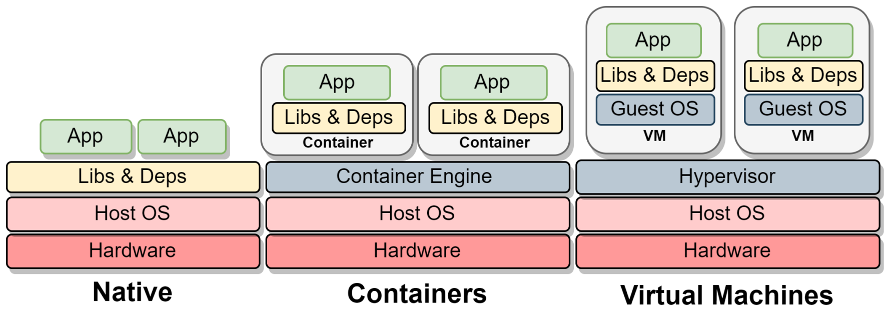


Figure 2.1: Native, container-based and hypervisor-based virtualization in VEC

excess storage usage for duplicate dependencies. The sharing is achieved by using a specialized layered filesystem and two important Linux kernel features: Namespaces [26] and Control groups (Cgroups) [27]. Namespaces virtualize global resources (e.g., processes, network, inter-process communication) to promote isolation and guarantee that the processes within the container do not have visibility over the rest of the host's resources and running processes [26]. Cgroups, on the other hand, provide a mechanism for aggregating and partitioning sets of processes, and all their subsequent children, into hierarchical groups with specialized behaviour, resource allocation, and resource utilization limits [27]. As such, all the processes across the entire system are organized hierarchically, and available resources are evenly distributed at each layer, unless otherwise specified. Each container is allocated to its own dedicated Cgroup within the parent "Docker" Cgroup, in the case of the popular Docker Container Engine.

2.4.3 Using Real-Time Scheduling in Container Environments

Containerized processes may also take advantage of Linux's real-time scheduling capabilities. As before, the simplest method to achieve this is by starting the container, finding the resulting host system PIDs, and then manually assigning a real-time priority or attributes using *chrt* on the host system. Such a manual process is unsuitable for any real world application. Alternatively, it is also possible to add a *chrt* call to the entrypoint script, or to configure real-time scheduling parameters in the application code itself, provided that the SYS_NICE capability [28] is added to the container. However, these approaches still do not offer the level of flexibility required for a complex deployment orchestration system like the ones explored in the context of this dissertation.

2.5 Orchestration Tools

As the scale and complexity of an edge computing architecture increases, so does the need for comprehensive deployment strategies that maximize automation, efficiency, fault-tolerance, flexibility, and observability. Such solutions typically involve a cluster of compute nodes managed by a centralized orchestrator that is capable of, among others: automatically deploying containerized applications; monitoring their states; dynamically scaling and load-balancing services; creating virtual networks; scheduling deployments based on the nodes' state and resource utilization; and lastly, migrating application between nodes, in the case of failures.

In addition to the virtualization and containerization drawbacks mentioned earlier, these clustering and orchestration based strategies usually present a substantial learning curve and increased resource consumption overhead, due to the fact that the client software that clustered edge nodes is required to run. Network utilization/bandwidth may also be affected by in-band control plane communications. This section presents some of the most well-known orchestration tools, finishing with a comparative analysis, that identifies some limitations.

2.5.1 Docker Swarm

Within the Docker ecosystem, a swarm is defined as a set of machines which are running the Docker engine and have been configured to join together as a cluster of worker nodes. These swarm nodes can be physical hosts or Virtual machines. Once the cluster is formed, developers and administrators continue using the usual Docker engine CLI commands that they would use on any individual Docker host, but, under the hood, the containers are scheduled to any of the nodes belonging to the cluster. Additionally, the network requests exchanged between applications are automatically load-balanced between replicas of the same service.

One of the main advantages of using Docker Swarm is the tight integration with the existing Docker tooling, which presents several advantages: the fact that it comes pre-bundled with every Docker engine installation, which greatly simplifies the setup process; and the fact that developers can leverage their existing knowledge of the Docker CLI, instead of learning an entirely new CLI. Swarm also works seamlessly with other Docker tools such as Compose. However, this simplicity and ease-of-use also means that Swarm's functionality is, in some respects, limited, especially when compared with other orchestration solutions which present a more robust feature set and enable a much greater level of customisation, extensibility, and automation.

2.5.2 Hashicorp Nomad

Nomad by Hashicorp[2] is an alternative solution that implements the archetypical design and feature-set of a service orchestration tool. Each host executes Nomad's agent daemon and is joined to a cluster consisting of manager nodes that control the state of the deployments and the underlying cluster, and worker nodes which run the applications themselves. As with other solutions, Nomad includes rich Container Networking Interface (CNI) support and service discovery via DNS, in order to enable service-to-service networking.

Compared to some of the more popular alternatives, Nomad is built using a more lightweight and simplified approach in order to reduce overhead and increase adoption and ease of use. Another major area where Nomad distinguishes itself from competitor's solutions is in its support for orchestrating applications that execute on legacy runtimes such as Windows/Linux services, static binaries, Java JARs, or simple OS commands. Whereas traditional orchestrators only support deploying applications as containers, or in some cases, full virtual machines.

[2]https://www.nomadproject.io/

By making use of this feature, legacy applications that have not yet been migrated to containerised runtimes, or cannot feasibly run as containers for compatibility or operational reasons, can still be included in the orchestration process. This represents a net benefit to the infrastructure and deployments as a whole, as it means that legacy applications are also taken into account when scheduling new applications to worker nodes, which in turn promotes the overall balance between worker nodes in terms of resource utilisation. Furthermore, given that containerised applications typically incur a small but non-negligible resource consumption overhead, it could be beneficial to strategically deploy applications to legacy runtimes in cases where the target worker node is significantly resource-constrained.

Despite these advantages, as is the case with Docker Swarm, Nomad's relative simplicity comes at the cost of advanced features and extensibility, meaning that custom domain-specific automations and/or scheduling decisions are much harder to implement.

2.5.3 Kubernetes

One of the most popular orchestration solutions is Kubernetes[3], an open-source project which works as a cluster of control nodes that manage the cluster state, and worker nodes that run containerized workloads specified by users using Kubernetes' declarative API. However, since Kubernetes was developed primarily for cloud computing environments containing much more powerful hardware, its official release is not very suited for use in low-powered devices. Instead, those types of deployments should use alternative distributions such as MicroK8s or Rancher k3s, which include several optimisations in terms of overall resource overhead.

One of Kubernetes' standout features is its focus on providing extensible components and APIs, so that users can exert a high level of control over the default behaviour throughout the various steps of the orchestration process. One such approach is the Operator Pattern, which consists on implementing domain specific logic which can leverage the Kubernetes API to automate complex tasks, and continuously monitor and reconcile the state of specific deployments within the cluster. In section 5.4.3, a fog operator is proposed to orchestrate time-critical city services.

The process through which the Kubernetes control plane allocates containers to specific worker nodes, known as Scheduling, is also extensible. Users have the possibility to develop plugins that extend the functionality of the default scheduler by filtering and/or sorting potential worker nodes using custom logic and metrics. In terms of allocating real-time scheduling attributes and priorities to processes, the same methodologies and caveats discussed in section 2.3 apply to Kubernetes as well. Nonetheless, Kubernetes also enables a few additional approaches (require the SYS_NICE capability) which lead to a much higher level of automation:

- by using container lifecycle hooks, specifically the PostStart hook, to automatically execute a predefined chrt command as soon as the container starts running;

[3]https://kubernetes.io/

- by using the Kubernetes Exec API to run chrt commands on the container whenever required, thus allowing for dynamic priority changes based on performance metrics and external factors.

2.5.4 Comparative Analysis

As described in section 2.5.1, from the three solutions, the Docker Swarm project holds distinct advantages in the simplicity and ease-of-use domains due to its integration with the existing Docker tooling. Moreover, the worker node software is substantially more lightweight to run when compared with other solutions, since it is packaged within the base Docker Engine. This also ensures compatibility with a large number of environments and CPU architectures. Finally, Docker Swarm is, in general, capable of deploying instances of applications at a faster rate than competing solutions, which is also a by-product of its simplicity.

On the other hand, Swarm lacks some of the more advanced features and algorithms supported by its competitors like Kubernetes, and is typically reserved for simpler scenarios that involve smaller clusters and less critical applications. These limitations include, for instance, the lack of automatic scaling mechanisms or integrated monitoring, and the relatively rudimentary workload scheduler implementation. In fact, its scheduler exclusively employs a *spread* strategy that strives to keep a balanced resource usage throughout the cluster, whereas solutions like Kubernetes include other considerations such as image locality, taints and tolerations, topology spread, and preemption. Furthermore, developers are not able to add custom implementations of these missing strategies, since Swarm's API does not provide mechanisms to extend core components.

The Hashicorp Nomad project strives to maintain a lightweight footprint while also introducing a more advanced feature-set that enables its use in complex production scenarios. The tool includes automatic metric-based autoscaling and a workload scheduler that includes both resource-balancing and topology-spread strategies, similar to the ones implemented on Kubernetes, as well as preemption support. The most significant benefit, however, is its native support for orchestrating applications in legacy runtimes, which complies with the requirements in section 2.2, and is exceedingly rare amongst its competing solutions.

Nevertheless, Nomad is still a maturing project that does not have yet complete feature parity with the most advanced alternatives like Kubernetes, particularly in terms of networking capabilities and overall tooling. Crucially, the system's extensibility is also very lacking, since there is no plugin support for core components such as the scheduler or load balancer, resulting in a lack of adaptation for new scenarios. As a result, Nomad is currently not as widely used as some alternatives like Kubernetes. As a consequence, there is a relative shortage of documentation, community support, and FOSS projects that could help guide both the implementation effort and the future use of the finished framework by developers. Thus, from the three solutions, Kubernetes is the most used, extensible and community accepted orchestration tool for cloud serviced. As presented in section 5.1, the modular architecture and easy adoption by the community allows the development of specific city fog services, plugins and operators for the ATCLL infrastructure.

2.6 Related Work

To meet time-sensitive requirements for edge services, there are three key areas that must be addressed. The first area relates to the design of the microservice itself, including its architecture, programming language, and communication protocol. This topic is covered in chapter 3 of the dissertation, which evaluates the performance of critical services like vehicular communication stacks using MQTT, ZeroMQ, and DDS protocols. The second area that must be addressed is the deployment environment. Chapter 4 explores this topic, discussing the selection of suitable running environments to ensure optimal scheduling and isolation. It also presents various mechanisms that can be used in microservice architectures to isolate them from other services in containerization environments. Finally, it is important to consider the microservice or service-chain requirements when orchestrating these services in a fog environment. This is why chapter 5 and 6 offer multiple approaches for managing time-critical fog services from an orchestration perspective. The related work presented in this section is organized according to these three areas.

From the network performance perspective, several studies can be found that compare the performance of the ITS-G5 and C-V2X technologies, from a theoretical perspective [29], using simulation environments [30] or on real deployments [31]. For instance, the work presented in [29] analyzes the evolution of the two protocol stacks on the physical and MAC layers, ending with a comparison of the access layer technologies of LTE-V2X and NR-V2X in C-V2X. However, studies that compare the performance and architecture of each available stack are missing. Additionally, like presented in [29] there are some studies that present some of the performance limitations of current publish/subscribe protocols; however, current V2X studies only address the time-critical requirements from the network latency and jitter perspective. Chapter 3 presents the comparison between two open-source V2X stacks that could be used on real deployments, together with the impact of publish/subscribe protocols in the solution design in terms of the micro-service latency.

Several studies have investigated the impact of kernel configurations and patches on virtualization technologies in the context of the running environment. In [32], the PREEMPT_RT kernel patch is described in terms of its objectives, design choices, strengths, and limitations, and compared to other Co-Kernel-based solutions such as Xenomai. The authors also surveyed other studies to identify viable scenarios for using PREEMPT_RT in different hardware configurations, as well as summarized their efforts in benchmarking the patch's performance in key metrics. Similarly, the study in [33] evaluated several hypervisors to assess their ability to handle real-time workloads, for critical industrial environments. However, studies that evaluate the impact of this type of techniques in containerization environments are missing. Chapter 4 addresses this gap by presenting an evaluation of some of these strategies utilized in the deployment of microservices in container environments.

Orchestration solutions typically involve a cluster of compute nodes managed by a centralized orchestrator with the capabilities to deploy, monitor, migrate and scale services, and load-balance requests. One of the most popular orchestration solutions is Kubernetes.

However, since Kubernetes was developed primarily for cloud computing environments, its official release is not suited for use in low-constrained devices. In [34], the authors analysed the various scheduling features available in Kubernetes and present an extension to the default Kubernetes scheduler in order to make the scheduling process aware of network bandwidth and latencies. Results show that the proposed network-aware scheduler can significantly improve the service provisioning of the default scheduler by achieving a reduction of 80% in terms of network latency. However, as presented in chapter 6, the processing latency introduced by the service also needs to be considered, together with the network latency. Furthermore, in [35] the authors use Kubernetes as a fault-tolerant framework to run Vehicular Ad-hoc Network (VANET) applications, in scenarios where one or more RSU worker nodes (or their communication links) could unexpectedly fail. The authors only focus their proposal on the impact of fault-tolerant mechanisms in terms of latency and throughput. Lastly, from a more architectural perspective, the work presented in [36] discusses the usage of Kubernetes as an orchestration platform for Edge Computing and presents some limitations with regard to resource awareness and architectural shortcomings, as well as potential solutions for each one. These solutions include the use of lightweight Kubernetes distributions, custom metric servers, and Scheduler extensions. It also presents a comparison and analysis of several Kubernetes-based Edge orchestration architectures proposed in other works.

As conclusion, to allocate real-time scheduling attributes and priorities, the same methodologies and caveats discussed in section 2.3 could also be applied to Kubernetes as well. Nonetheless, Kubernetes also enables a few additional approaches which lead to a much higher level of automation: 1) by using container lifecycle hooks (i.e., PostStart), to automatically execute a predefined *chrt* command; 2) by using the Kubernetes Exec API to run *chrt* commands on the containers, allowing dynamic priority changes based on performance metrics and external factors (both require SYS_NICE). One of Kubernetes' standout features are their focus on providing extensible components and APIs, and users can exert a high level of control over the default behaviour (i.e., operator pattern, scheduling). Given these capabilities, Kubernetes was extended in chapter 5 with domain-specific logic and management time priorities, in order to minimize the latency of critical services. This is the final novelty presented in this dissertation, with the proposal of the architecture in chapter 5 and its evaluation in chapter 6.

2.7 Summary

This chapter presented the background information and concepts required to provide context on the topics that are central to this dissertation. These concepts include Smart Cities, VANETs, Real-Time Scheduling, Service Deployment Strategies, Vehicular egde computing, and Kubernetes, among others. Finally, it also presented an analysis of related works in this field, with a major relevance on the works related to V2X technologies, stacks and pub-sub protocols, virtualization technologies and the possibility to support real-time services, and the dynamic orchestration of services in an edge computing environments. Each of this topic is complemented with information on how it will be addressed in this dissertation.

CHAPTER 3

Vehicular Communications in Microservice Architectures

3.1 Motivation

Chapters 2 and 4 present several task scheduling techniques that can, in general, be applied to any type of service in order to potentially provide a meaningful improvement on performance and stability. Likewise, chapter 5 proposes various orchestration systems to automate the application of the aforementioned techniques, maintain a balanced cluster, place services in the best nodes, and shape application traffic between dependencies. Despite this, one of the most crucial aspects of minimising latency and assuring predictable performance for critical services is the design and implementation of the service itself. These aspects include: how the service uses system resources, how wide its scope is, how it communicates with its dependencies, how its performance can be measured and reported, among many others. Furthermore, it is also important to consider aspects that facilitate the application's orchestration, such as how the application is packaged and how it can be configured. In order to explore this topic and provide a relevant example, a network application such as a Vehicular network protocol stack service is a suitable candidate to be used as a case study of a set of services that can be deployed using the proposed orchestrator.

This chapter discusses the drawbacks of traditional VANET application architectures, and proposes a microservice-oriented alternative that better suits an edge computing cluster environment. Within this architecture, the most crucial service is the protocol stack, which also serves as an example of a critical service which is depended on by multiple other applications, and therefore, could benefit from intelligent scheduling and real-time priorities. In chapter 4, this new protocol stack is subject to a comprehensive set of experiments to evaluate different processing latency reduction strategies.

Finally, this chapter also provides meaningful context to several of the architectural and implementation decisions made while designing the orchestrator framework that is proposed later in chapter 5.

The organization of this chapter is the following. Section 3.2 presents some vehicular network architectures. Section 3.3 presents the Vanetza vehicular services, and section 3.4 proposes the extensions to Vanetza to provide a better support for MEC scenarios, with better scalability, and envisioning time-sensitive applications. Section 3.5 describes the evaluation of *Vanetza-NAP*, and an evaluation between several data brokers MQTT, ZeroMQ and DDS, to gather conclusions about which technology is more appropriate to real-time and scalable applications. Finally, section 3.6 presents the conclusions of this chapter.

3.2 Vehicular Edge Computing Architectures

The main reasons to consider the VEC and Multi-Access Edge Computing (MEC) paradigms as promising solutions are the need to have lower latency in the service level and to reduce congestion of the backhaul network [37]. However, edge devices, like those used in VEC scenarios, are typically constrained, when compared with the devices available in cloud systems. Thus, the migration of vehicular services from the cloud to the edge shall be carefully addressed, considering all components that can add delay to the system. This section surveys some of the characteristics of current V2X platforms found in commercial and open-source solutions, identifying some limitations to addressing time-constrained applications.

Overview of available architectures

Figure 3.1 presents the most typical architectures found in V2X solutions, that differ in several aspects. In general, commercial solutions tend to be closed to third-party modifications. Applications must run on proprietary hardware, and use a Software Development Kit (SDK) to encode and decode messages, as well as to send and receive them (e.g., Cohda Wireless MK5 [38] and Unex V2XCast ecosystem [39], [40]). Production-ready platforms, in (1) on the figure, are usually shipped as a closed V2X solution, supporting only a minimal set of V2X applications that could not be easily extended.

Nevertheless, there are also slightly different solutions, in (2) on the figure, that allow Intelligent Transportation Systems (ITS) application developers to use their own hardware coupled with a mPCIe module (e.g., Unex SOM-3xx System-On-Module [40]). In that scenario, an application still needs to use a proprietary SDK to build a correctly formatted message and interact with the mPCIe module, but the System on a Chip (SoC) within the module itself handles the ASN.1 encoding/decoding, and the transmission and reception of encoded messages is performed through the IEEE 802.11p interface. These types of solutions should yield great stability, given that part of the process runs on isolated purpose-built hardware, and therefore cannot suffer interference from other processes. However, given the proprietary nature of the software and, in some cases, the lack of full access to the OS, it can be difficult for users to attempt to improve performance by, for instance, applying some of the techniques explored in Section 2.3.

In contrast, open-source solutions (3-4 on the figure) may be freely installed on any machine and use any wireless network interface card with IEEE 802.11p support. Examples of open source V2X stacks are Alex Voronov's GeoNetworking library [16], and *Vanetza*, which

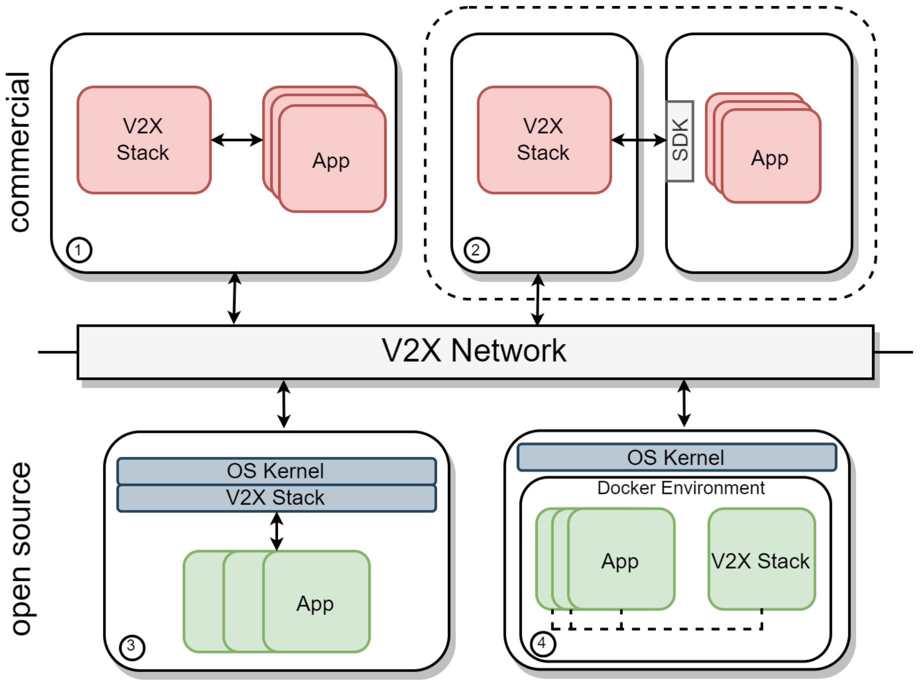


Figure 3.1: Commercial and Open-source VEC platforms architectures

is developed as part of ongoing research work at Technische Hochschule Ingolstadt [41]. These solutions are also typically used as a library that ITS applications include.

Despite, perhaps, not having the same level of quality assurance and testing as a commercial solution, these open-source alternatives have the advantage of granting users the freedom to make changes to the underlying code and to deploy them in different sets of hardware and configurations. As such, the open-source solutions may support different runtime environments, either in bare-metal hosts (in 3), or within virtualisation environments such as Containers (in 4). Thus, developers could optimize V2X applications that address time-critical services.

Monolithic architectures

As previously mentioned, the most typical architecture used by ITS applications that opt to make use of existing open-source libraries follows a monolithic paradigm where the use case's implementation will, by necessity, include a third-party library or SDK that performs the encoding and decoding of ASN.1-encoded messages, as well as their transmission and reception through the raw 802.11p-enabled wireless network interface.

This approach presents several notable drawbacks, the most significant of which being the fact that each incoming C-ITS message received by the host system will need to be processed and decoded by the protocol stack layer of every active ITS application, even if a given application's particular vehicular network use case does not require the use of that specific message type. Since there is no filtering mechanism that can be applied at a lower level of the Linux networking stack to automatically discard unwanted packets by their C-ITS message type, the only means of ascertaining the message type of an incoming transmission is to fully decode it first.

In practice, this results in several redundant processing cycles spent performing the exact same operations multiple times, only for some of the results to be immediately discarded. This is especially relevant in the context of a smart city edge-computing cluster comprised of resource-constrained SBCs, where any extraneous processing should be avoided.

In addition, since the message transmission and reception logic is also included in each application as part of the imported SDK, every application that manages a vehicular network use case is restricted to deployments on worker nodes that support 802.11p connectivity capabilities, which limits the orchestrator's ability to balance the resource usage throughout the cluster's worker nodes.

The use of this architecture in containerised environments raises additional issues such as excessive storage usage, since each container requires access to a full copy of the protocol stack library, as opposed to deployments using the native runtime, where a single copy is linked to each application's binary target. This issue can be successfully mitigated by the container engine through the sharing of filesystem layers to multiple containers. Nevertheless, this requires the careful installation of the library in specific ways.

Furthermore, since, as previously mentioned, the message transmission and reception logic is also included in each application as part of the imported SDK, every ITS application container requires host mode networking capabilities in order to access the raw 802.11p-enabled wireless network interface. This restriction hinders the overall security of the system by decreasing the level of isolation between running workloads, and may also preclude the applications from accessing advanced network features that are only available to containers with namespaced network environments.

Alternatively, vehicular network use cases can be implemented using an even stronger monolithic approach that combines every use case into a single codebase, thus mitigating all of the aforementioned problems. However, this solution inevitably brings several development and usability issues stemming from the size of the codebase and the inherent difficulties in maintaining it. And, crucially, the proposal still can not be considered an orchestration or cluster-friendly architecture, since it severely limits the avenues that the orchestrator can use in order to manage its performance and maintain cluster balance.

Finally, in both alternatives, the choice of programming language/framework when implementing new ITS applications is restricted by the need to use the protocol stack library, which hinders developers' freedom, unless somewhat complex language interoperability techniques are used, when possible.

3.2.1 Microservice-oriented proposal

As an alternative, this work proposes a microservice-oriented approach that decouples the protocol stack from the implementation of each vehicular use case. Under this proposal, the protocol stack is deployed as its own standalone service that serves as the system's single point of entry/exit for C-ITS messages and performs all the respective encoding and decoding operations. In turn, vehicular use cases are implemented as individual applications that interact with the ITS stack through Inter-process communication (IPC) mechanisms, in order to receive incoming messages or instruct the stack to send new outgoing ones. This communication is accomplished using a simpler, universal, representation of ITS message contents (e.g., in JavaScript Object Notation (JSON) format), thus avoiding the inclusion of extensive ASN.1 coding logic in every application.

This strategy successfully addresses the aforementioned drawbacks of monolithic solutions in regards to their ineffective use of storage space and processing power, and greatly simplifies the development of applications that implement vehicular use cases. Furthermore, this separation of the ITS monolith into a set of individual microservices also presents several advantages when considered in the context of cluster environments. Namely:

- **Differentiated scheduling and eviction constraints and behaviour -** Since each ITS application is subject to a separate worker node scheduling process, the cluster's orchestration system can maintain a balanced cluster state in terms of resource utilisation more effectively. This is especially relevant in cases where the orchestrator determines that one or more ITS applications can not feasibly be allocated to the same node as the protocol stack service, and must therefore be placed in neighbouring nodes to ensure stable performance levels, even if the application incurs a slight latency penalty as a result. Additionally, ITS services can frequently require multiple dependencies with varying levels of importance, and which are not necessarily present in the same worker node as the protocol stack. In a microservice architectural paradigm, the orchestrator can act in accordance to these types of constraints for each specific application, without affecting the others.
- **Differentiated resource requests and limits -** Each ITS use case implementation is able to specify its own particular set of resource requests, which allows for a more effective scheduling process and, consequently, a cluster state that is better optimised. Deployments may also specify differentiated resource limits, which improves the granularity of this constraint when compared to setting a single upper bound on the resource usage of an entire monolithic application, therefore improving the stability and resiliency of the cluster in unexpected scenarios.
- **Differentiated replication behaviour for increased load-balancing and fault-tolerance -** Each ITS application is subject to a replication behaviour that respects its particular requirements and constraints, and (possibly) acts in response to its specific current performance level.
- **Improved isolation/security -** Since only the protocol stack service requires access to raw host-level network interfaces, ITS applications no longer require the use of host-mode networking, which increases the isolation of each service, and overall system security.

Under this microservice-oriented architecture, the communication between the protocol stack service and its client ITS applications can feasibly be conducted using any available backend technology. Given this dissertation's focus on reducing latency as much as possible, the most apparent choice is to use direct connections (e.g., TCP/UDP sockets) between each producer-consumer pair, thus minimising traffic hops and overall delay. However, this type of messaging solution is deemed to be overly restrictive in this context, since the incoming

C-ITS messages are expected to be consumed by multiple different ITS applications and other services, and, ideally, the addition of a new consumer shouldn't require any changes to the protocol stack's implementation.

Instead, this proposal suggests the use of technologies that implement the Publish and Subscribe (PubSub) paradigm, such as the Message Queuing Telemetry Transport (MQTT) protocol, which allows for a virtually unlimited number of producers and consumers that can be integrated into the communication architecture in a plug-and-play fashion, without requiring any modifications to the stack's base code. Moreover, using MQTT enables the effortless integration of the data contained in these C-ITS messages in cloud-based dashboards and persistent databases, and also greatly improves the observability that developers have into the messages being exchanged, for monitoring and/or debugging purposes.

Nevertheless, despite these advantages, the centralised nature of the MQTT protocol also presentes a very significant drawback, since it adds the MQTT broker as a mandatory additional network hop, thus incurring higher latency values on each message that is exchanged. On balance, given the substantial ease-of-use advantages, this is deemed an acceptable compromise for most generic applications. However, this is not the case for time-sensitive critical services which must use alternative solutions in order to ensure compliance with their specified upper-bounds. Some of these alternative strategies are explored later in this chapter.

3.3 Vanetza

The protocol stack implementation presented in this chapter is primarily focused on enabling a paradigm shift in the way that V2X applications are designed, in how they interact with each other and with the VANET, and also in how they can be orchestrated in an efficient manner. Since the underlying message encoding/decoding logic of ITS protocol stacks is not central to achieving these goals, this implementation does not re-implement it.

Instead, the proposed stack uses an existing free and open-source library as a foundation to build upon. After considering the available alternatives, the most promising option to fulfil this role was *Vanetza*[1].

Vanetza is an open-source implementation of the ETSI C-ITS protocol suite. Among others, it comprises the following protocols and features:

- GeoNetworking (GN)
- Basic Transport Protocol (BTP)
- Decentralized Congestion Control (DCC)
- Security
- Support for ASN.1 messages (Facilities) such as CAM and DENM

Though originally designed to operate on ITS-G5 channels in a Vehicular Ad Hoc Network (VANET) using IEEE 802.11p, *Vanetza* and its components can be combined with other communication technologies. This means that running several instances of a containerized version of *Vanetza* in a Docker bridge is sufficient to emulate a basic V2X environment.

[1]https://www.vanetza.org/

Another of its relevant features is *Socktap*, which is an included application that illustrates the usage of the *Vanetza* library's API. This demo application can be executed on systems with ordinary IEEE 802.11p-enabled Wi-Fi modules, without the need for specialised V2X hardware. *Socktap* serves as an experimental application that highlights some of *Vanetza*'s capabilities, but should not be viewed as a fully developed ITS-G5 station. Thus, although it is an example of how to interact with *Vanetza*, it should not be used as a reference design with which to test its real-time performance and behaviour, for the purposes of this work.

Finally, *Vanetza* is implemented in C++, whereas Alex Voronov's aforementioned GeoNetworking library was developed in Java. C++ is a low-level programming language that allows for direct memory manipulation, fine-grained control over the hardware, and a small memory footprint. This makes it a powerful language for developing high-performance applications, particularly on resource-constrained devices such as edge devices. In contrast, Java is a higher-level language that runs on a virtual machine, which can introduce some overhead and reduce performance compared to C++.

3.4 Vanetza-NAP

The new protocol stack implementation presented in this chapter, built upon the open-source *Vanetza* library and its *Socktap* application example, was named *Vanetza-NAP*. Having "NAP" included in the name reflects an important contribution of the work of this dissertation: developing a new "de facto" VANET stack to be used in the ongoing research work in the Network Architectures and Protocols (NAP) research group of the Instituto de Telecomunicações - Aveiro.

The following subsections describe the set of features that are introduced in order to extend the original *Vanetza* library into a standalone service with better support for MEC scenarios, improved scalability, and an architecture that supports time-sensitive applications.

3.4.1 ETSI C-ITS Message Support Extension

As discussed previously, the protocol stack uses a JSON representation in order for other V2X applications to be able to send and receive messages. Naturally, the schema of these representations needs to be known and agreed upon by all applications and developers.

One possible approach would be to design a bespoke JSON payload for each message. This would require significant effort, but would likely result in smaller payload sizes when compared to alternative techniques, improving inter-service communication. However, forcing V2X applications to use a customised message format would introduce friction in the adoption of this new paradigm. Additionally, C-ITS messages typically have several optional fields and containers of fields that may or may not be present on each given message. Handling this variability would greatly increase the complexity of any custom JSON payload format. Instead, the schemas used by *Vanetza-NAP* are an exact match to the respective ASN.1 specification in terms of message layout, object and sub-object structure, and field names. This way, application developers only need to follow the standards set by European Telecommunications Standards Institute (ETSI) in order to interface with the system. Moreover, *Vanetza*'s C-ITS message data structures are auto-generated from the ASN.1 specification files using the ASN.1C project and, as such, closely follow the specification, which simplifies the process of converting information between JSON and the data structures.

However, *Vanetza* does not natively support JSON representations of messages and, unlike other languages, C++ JSON libraries require that the marshalling and un-marshalling logic be explicitly declared and implemented, usually in the form of toJSON and fromJSON functions for each individual datatype. Since C-ITS messages are typically composed of a large quantity of data fields encapsulated in several layers of containers, *Vanetza*'s internal representation of messages is composed of hundreds of different data structures, making it highly impractical to manually implement the conversion logic.

To address this, *Vanetza-NAP*'s development included the creation of a Python script that reads the relevant ASN.1 specification files, and automatically generates a C++ output file containing over 430 JSON encoding and decoding functions for different data structures. This output file is then compiled as part of the overall project, enabling the seamless conversion of the following supported C-ITS message types to and from JSON:

- *Cooperative Awareness Messages (CAM)*
- *Decentralised Environmental Notification Message (DENM)*
- *Collective Perception Message (CPM)*
- *Vulnerable Road User Awareness Message (VAM)*
- *Signal Phase And Timing Extended Message (SPATEM)*
- *MAP (topology) Extended Message (MAPEM)*

3.4.2 Communication strategies

As introduced in the previous subsection, V2X applications interact with *Vanetza-NAP* through messages in JSON format which closely replicate the respective ASN.1 structure. To instruct the protocol stack to send C-ITS messages, these applications build JSON representations of the given messages and publish them in a specific MQTT topic, which *Vanetza-NAP* subscribes to. Likewise, applications that need to receive incoming messages do so by subscribing to the respective MQTT topics, that are published to by the stack. The overview of this architecture is depicted in Fig. 3.2. In order to minimize the communication latency between *Vanetza* and any critical producer/consumer applications, *Vanetza-NAP* also supports publishing and subscribing to OMG Data Distribution Service (DDS) topics. The DDS implementation is explained in more detail later in this section.

Fig. 3.3 presents a diagram with all the topics used by the *Vanetza-NAP* implementation. These topics can be aggregated into the following groups:

- **vanetza/in** — Used by V2X applications to publish vehicular messages to the protocol stack, which are then sent through the V2X interface to other VANET stations.
- **vanetza/own** — Used by *Vanetza-NAP* to publish, for reference, the contents of CAM messages which are auto-generated by the protocol suite when it is installed in known SBCss used within the research group, and sent at regular intervals. This is possible since the interface with the GPS module and other data sources are known, and so it was embedded in the *Vanetza-NAP* development as an optional feature.

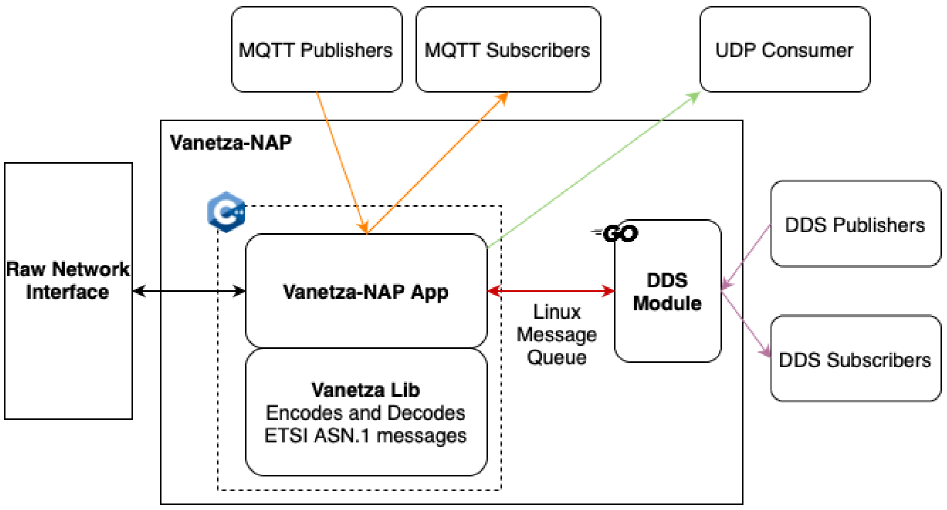


Figure 3.2: Overview of Vanetza-NAP's architecture

- **vanetza/out** — Used by V2X applications to subscribe to incoming vehicular messages from other VANET stations.
- **vanetza/time** — Used by *Vanetza-NAP* to publish, for reference, the timing information for the processing delay incurred while handling messages received through *vanetza/in*. This is described further in the following section.

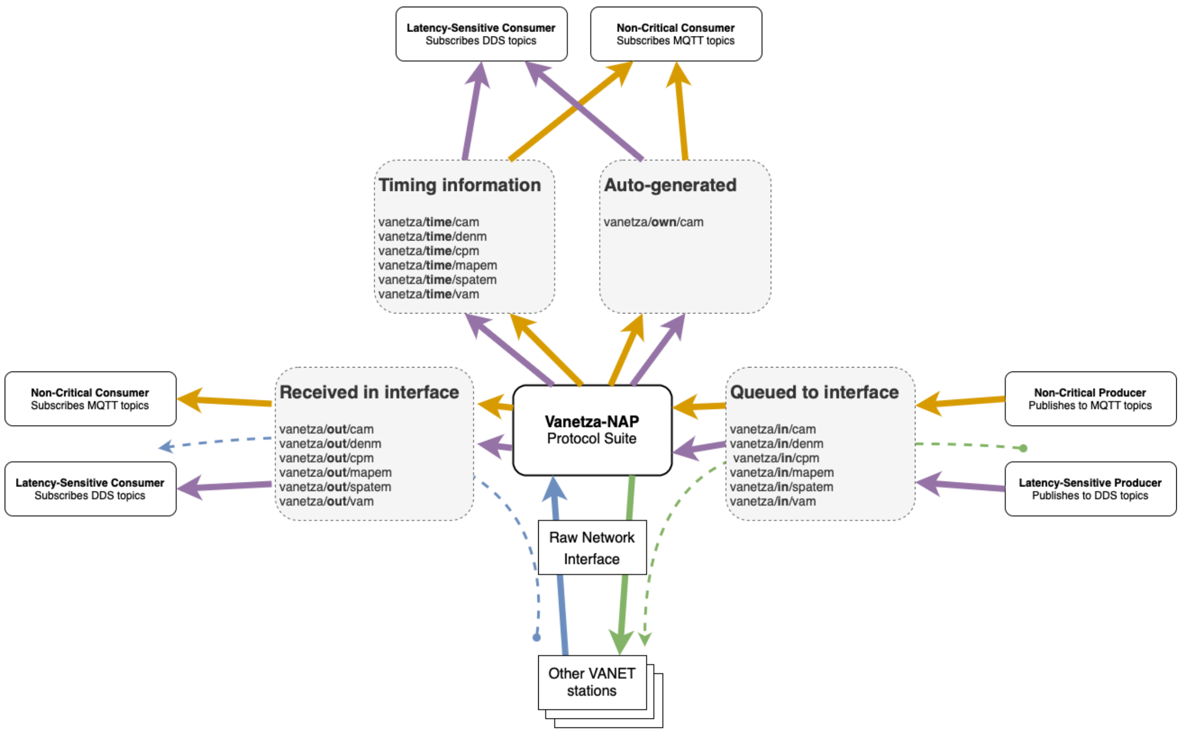


Figure 3.3: Overview of the MQTT and DDS topics supported by Vanetza-NAP and their flows

Measuring processing performance

One of the considerations in the development of this protocol stack was the inclusion of a mechanism to provide readily available feedback to developers and client applications regarding the processing performance of the protocol stack, which enables a more comprehensive understanding of end-to-end latencies experienced by ITS use cases. Firstly, each JSON payload that is output by the protocol stack contains two different timestamps in UNIX time format, whose names and meanings depend on the topic to which the message is published. Namely:

vanetza/out/xxx:

- “timestamp”: The instant at which the ASN.1 encoded message was received at the network interface.
- “test/json_timestamp“: The instant at which *Vanetza* finished preparing the JSON payload in order to send it via MQTT and/or DDS.

The difference between these timestamps represents the total time elapsed while performing the decoding, parsing, and JSON generation tasks. The difference between the “test/json_timestamp” field and the time at which a given client receives a message on this topic represents the total latency introduced by MQTT/DDS and the underlying network.

vanetza/own/cam:

- “timestamp”: The instant at which the internal timer scheduled the transmission of a new self-generated CAM.
- “test/json_timestamp“: The instant at which *Vanetza* finished preparing the JSON payload in order to send it via MQTT and/or DDS.

The difference between these timestamps represents the total time elapsed while performing the encoding and queueing of the ASN.1 message and the JSON generation pipeline. The difference between the “test/json_timestamp” field and the time at which a given client receives a message on this topic represents the total latency introduced by MQTT/DDS and the underlying network.

vanetza/time/xxx

- “timestamp”: The instant at which the JSON payload was received in the “vanetza/in/xxx” MQTT/DDS topic.
- “test/wave_timestamp”: The instant at which Vanetza finished preparing and sending the ASN.1 encoded message through the network interface.

The difference between these timestamps represents the total time elapsed while parsing the JSON message and encoding and queueing the ASN.1 message. The difference between the “test/wave_timestamp” field and the time at which a given client receives a message on this topic represents the total latency introduced by MQTT/DDS and the underlying network.

A final note to this implementation is that the “json_timestamp” fields need to be included in the JSON payload itself. This adds additional processing to the pipeline, introducing a slight error margin that is unavoidable in this scenario.

Data Distribution Service (DDS) Integration

In order to minimize communication latency between *Vanetza* and any critical producer/consumer applications, *Vanetza-NAP* also supports publishing and subscribing to Data Distribution Service (DDS) topics (concurrently to MQTT)[2]. This technology consists of a middleware standard that aims to enable dependable, high-performance, interoperable, real-time, scalable data exchanges using a brokerless, decentralised, publish–subscribe pattern. It is typically used in real-time and embedded systems.

The support for this feature is implemented as a Golang micro-service that interacts with *Vanetza-NAP* using *SysV* Message Queues. This approach allows *Vanetza* to use Go's $rticonnextdds-connector-go$ library[3] which is simple to use and offers great interoperability with Python and NodeJS DDS applications that use equivalent libraries. It also does not raise any licensing issues, unlike most C++ alternatives that were considered. This module is represented in the architecture Fig. 3.2 as "DDS Module".

UDP Socket

In cases where critical V2X applications have timing constraints that preclude the use of MQTT but have not yet implemented or are otherwise incompatible with DDS communication functionality, *Vanetza-NAP* supports sending the JSON representations of decoded C-ITS messages directly via a UDP socket. This is done concurrently with the Pub-Sub based alternatives discussed above, and is mainly intended to be used by applications running within the same host, where packet loss is generally not a concern. This communication alternative is represented in the architecture Fig. 3.2 as "UDP Consumer".

Simultaneous implementation of MQTT and DDS

While *Vanetza-NAP* supports the simultaneous usage of MQTT and DDS, this may introduce a slight processing delay and might not be feasible to support in time-sensitive services. Instead, those services will likely make exclusive use of DDS, which reduces overhead but also restricts the amount and types of applications that can interact with the service. In fact, to consume the time-sensitive service's data, non-critical applications would be forced to implement DDS as well.

Within the scope of this dissertation, this issue is addressed with the implementation of a Golang-based service that subscribes to a configurable set of OMG DDS topics and publishes every new message to the respective MQTT topics, a DDS-MQTT Relay Service. This enables less critical services to effectively subscribe to a DDS topic via MQTT, and also allows developers easy access to the exchanged messages, for debugging or monitoring purposes.

Fig. 3.4 presents an example of an architecture where this solution would be useful. In this scenario, several services, with different requirements, subscribe to the output messages of the Latency-Sensitive Publisher Service (for example *Vanetza-NAP)*. The Latency-Sensitive

[2]https://www.dds-foundation.org/

[3]`https://github.com/rticommunity/rticonnextdds-connector-go`

Consumer Service receives the messages directly through DDS without additional processing, as the publisher is transmitting the messages natively using that protocol. In the case of a Non-Critical Consumer Service which does not implement DDS, these messages are accessible through the local MQTT broker via the Relay service. Since it represents a service without time-sensitive requirements, it may tolerate the additional latency created by the DDS-MQTT Relay Service and the Pub-Sub mechanism through the local MQTT broker. The same is valid for services in the cloud environment, which may also access the data indirectly via MQTT.

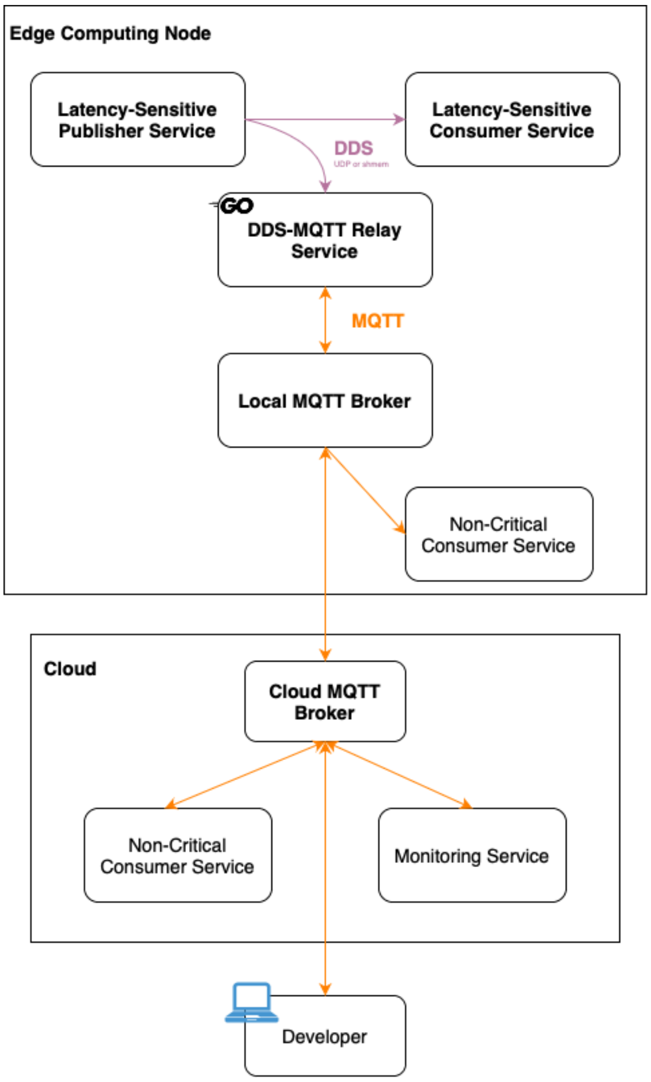


Figure 3.4: Overview of the PubSub communication architecture on a Smart City infrastructure with the DDS-MQTT Relay Service

3.4.3 Orchestration features

Beyond the previous discussions present in this chapter, and envisioning the work presented in Chapter 5, there are additional considerations that help a service improve its synergy with the orchestration framework with which it will be deployed and managed.

Metrics

In order for the orchestration systems to make informed decisions and react efficiently to potential issues, applications must continuously provide additional domain-specific context on their current performance and other relevant statistics. These values can also be used to automatically alert developers when an anomaly occurs. An industry standard approach to achieve this is generating and exposing application-level metrics within the application code itself, which allows for the monitoring of any statistic deemed relevant by the developer.

In order to maintain consistency and interoperability, metrics are usually exposed in a specific format, depending on the metric collection framework used. In this work, the Prometheus framework was chosen. Therefore, *Vanetza-NAP* uses the Prometheus-CPP library to generate and update the relevant metrics, and expose them via the creation of a webserver thread that centralised Prometheus server instances can periodically pull recent values from. The service exposes 2 key metrics: the number of messages received, and the processing latency observed while *Vanetza-NAP* performs message decoding and JSON generation tasks, or vice-versa. These metrics are calculated for each message type and message direction (received or transmitted).

These values can then be analysed either autonomously or manually by developers or administrators using complex PromQL queries that return trends in the values. From the first metric, the orchestrator can determine the current request rate to the given Vanetza instance, and if, for example, there has been an anomalous traffic situation that greatly increased traffic. From the seconds metric, the system can determine if the instance is currently managing to uphold timing guarantees. Fig. 3.5 shows an excerpt of the full list of metrics exposed by *Vanetza-NAP*.

```
observed_packets_count_total{direction="tx",message="spatem"}
observed_packets_count_total{direction="rx",message="spatem"}
observed_packets_count_total{direction="tx",message="vam"}
observed_packets_count_total{direction="rx",message="vam"}
...
# ---
observed_packets_latency_total{direction="tx",message="spatem"}
observed_packets_latency_total{direction="rx",message="spatem"}
observed_packets_latency_total{direction="tx",message="vam"}
observed_packets_latency_total{direction="rx",message="vam"}
...
```

Figure 3.5: Excerpt of the full list of metrics exposed by *Vanetza-NAP*

Configuration

Another important consideration when building orchestration-friendly applications is the inclusion of robust support for configuring app behaviour on the fly by using a configuration file or environment variables. In this way, services that are deployed multiple times and with slight differences in behaviour can be run from the same pre-packaged version and use configuration files to differentiate their mode of operation. This is the case for *Vanetza-NAP*, where typically, the same version of the pre-packaged Vanetza container or binary is deployed to all the edge nodes within a given infrastructure, but each one requires slightly different configuration parameters in order to work correctly, including, for example, their Station ID and MAC Address.

As such, Vanetza-NAP was designed to have a set of configurable attributes with the goal of allowing for fine-tuning its operation. These attributes are generally set in a configuration file that follows the INI schema, an industry standard configuration file format where properties are specified as key-value pairs. Using this approach, other applications can consult the Vanetza configuration file using available libraries built for any major programming language. Furthermore, automation tools like Ansible can also read and modify INI files automatically, which is useful for deployments.

However, for docker-compose deployments this would create a more difficult environment, as it would include several containers with heterogeneous configuration needs. In these situations, it would be required a separate config file for each container mounted as a volume. To solve this, *Vanetza-NAP* also accepts configuration via environment variables that can be set in the environment section of docker-compose.yml. Any values set via environment variables have priority over the ones found in the config file, thus making it possible to use the config file for common configurations and environment variables for values that are unique to each container (i.e. the aforementioned Station ID and MAC Address, etc).

Table 1 in Appendix 7.1 summarises the available configuration options. Each supported message type (CAM, DENM, CPM, VAM, SPATEM, MAPEM) has its own set of configurations, which are specified in Table 2 in Appendix 7.1 (using CAMs as an example).

3.5 Performance Evaluation

Considering the proposed approach, this section will focus on the evaluation of *Vanetza-NAP* when compared to a previous protocol stack, and an evaluation between several data brokers MQTT, ZeroMQ and DDS with *Vanetza-NAP*, to gather conclusions about which technology is more appropriate to real-time and scalable applications.

3.5.1 Vanetza-NAP vs Router Stack

The previous protocol stack is a Java-based ITS stack (known as Router) from the Alex Voronov's GeoNetworking library previously mentioned in section 3.2, modified to enable limited JSON and MQTT functionality.

A particularity of the previous protocol stack is that it is mainly implemented in Java but it uses Python scripts (one for each direction of communication) to convert the packets from/to the UDP layer to/from the layer 2 (802.11) (the blocks are named *802.11 to UDP* and *UDP to 802.11*). Given this dependency on Python scripts, higher degradation of the service's performance is expected when the edge device which runs the network stack experiences more extreme processing conditions.

The objective of this evaluation is to observe and analyse the difference in processing performance between these two stacks, and how it evolves when synthetic stress is applied to the system's CPU in order to simulate inter-process interference.

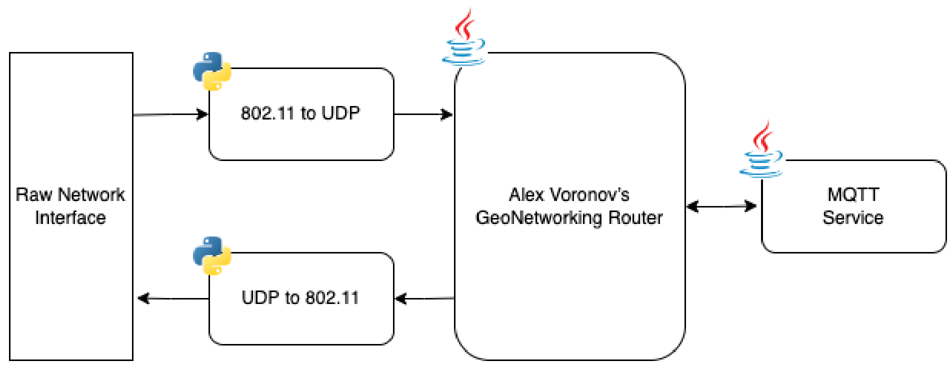


Figure 3.6: Router ITS Stack Architecture

Methodology

To reduce the number of variables between experiments, every test uses the exact same entry data, which was prepared in advance in the form of a `pcap` capture of the CAM messages exchanged between a testbed comprising RSU and OBU (11 messages per second). The `pcap` file was read and replayed by a python3 script which also collects the output data from the respective MQTT Topics into a `csv` file.

The CPU stress conditions are emulated using the $stress - ng$ package, which is configured to use all 4 cores at 70% (`stress-ng -c 4 -l 70`). Notably, the protocol stack services are in concurrence with the script required to produce this particular test, which has a negative impact on the performance results, in absolute terms. Nevertheless, the comparison is still deemed to be valid, since the environment is the same for both stacks.

The tests are performed using two PC Engines APU[4] with Debian 10 and kernel 5.7.10+, an OBU and an RSU. The OBU transmits CAMs through the IEEE 802.11p interface, and the testing service runs on the RSU side, measuring the following:

- The instant when the message is received by the RSU, at the IEEE 802.11p interface;
- The instant immediately before the JSON string is generated and sent to the MQTT broker.

Both ITS stacks generate the same JSON output, which includes those two timestamps. Therefore, the processing time of each stack is defined as the difference between these timestamps.

Service processing time

The first test establishes the baseline performance of both stacks without applying any stress on the CPU, with the transmission of more than 25k messages. Figure 3.7 shows the relevant results.

[4]`https://www.pcengines.ch/apu2.htm`

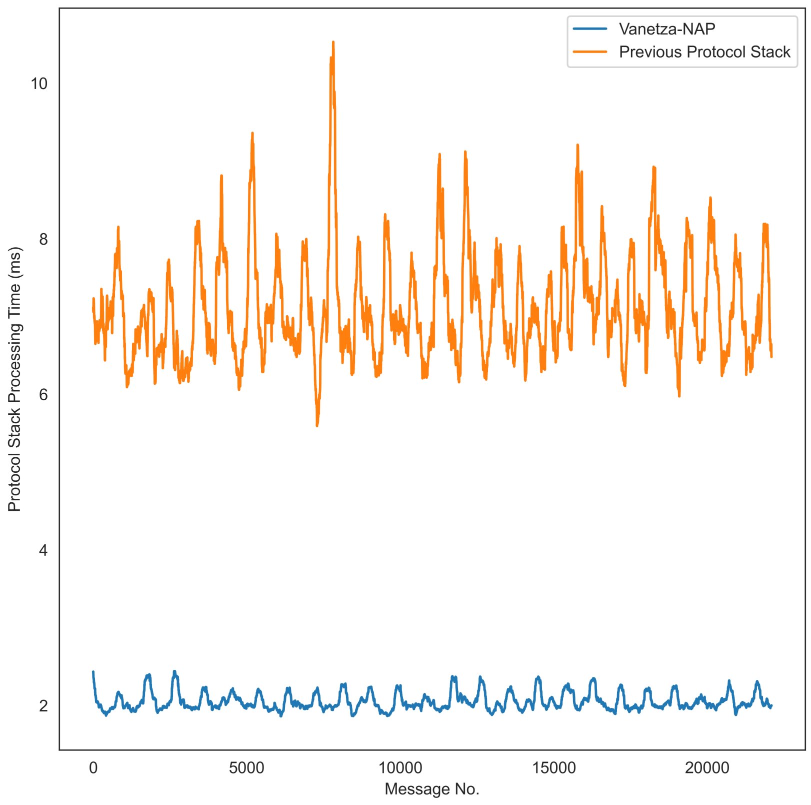


Figure 3.7: Service processing time – Vanetza-NAP vs Router

From this test, it is evident that the average delay of the *Vanetza-NAP* service is much lower than the one of the Router stack. Moreover, the variation of the delay is significantly larger on the latter, indicating that it is also much less stable.

These observations are explained, not only by the more efficient implementation of *Vanetza* and *Vanetza-NAP* through C++, but also by an additional important factor, the dependency of the Router stack on the aforementioned Python script that allows it to receive the ITS packets through the IEEE 802.11p interface. The reliance on the *802.11 to UDP* and *UDP to 802.11* blocks in both directions of the pipeline adds more delay to the overall response time of the service.

Service processing time under stress

Finally, the test is repeated while applying stress on the CPU, in order to evaluate the resilience of the network service to interference from concurrent processes. The *stress-ng* process uses all 4 cores at 70%. The same set of 25k messages is used. Figure 3.8 depicts the relevant results. The results of the first test are also included as baseline (blue and green curves). Note that the scale in the graph is logarithmic.

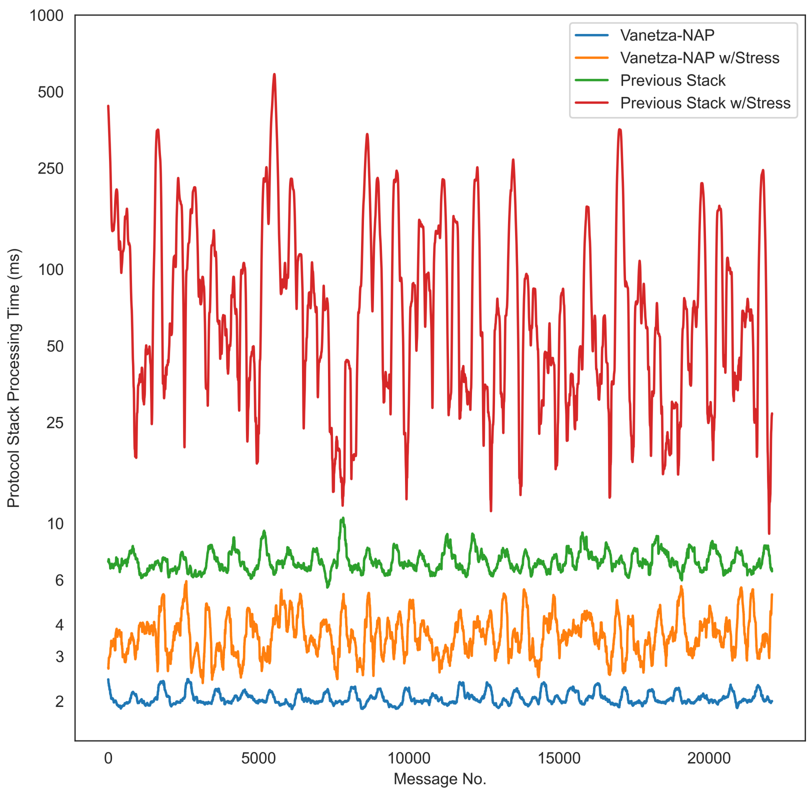


Figure 3.8: Service processing time – both stacks with and without concurrent processing stress

The results show that with stress applied, it becomes increasingly evident that the Router stack's processing time is highly unstable. In fact, its response time varies from a minimum of 10 ms up to 800 ms, which would be problematic for any latency-sensitive application that used the stack. Finally, the *Vanetza-NAP* stack, while under stress, still performs better than the Router stack's baseline, which is a good indication of its performance and stability.

In the next chapter, further evaluations will include the assignment of different CGroups to the *Vanetza-NAP* service, as well as the usage of the real-time scheduling and preemption capabilities of the Linux kernel, and also the containerisation of the service.

3.5.2 MQTT vs ZeroMQ vs DDS

Subsection 3.4.2 presented several communication strategies that can be used between *its* applications and the vehicular protocol stack, *Vanetza-NAP*, namely: MQTT, DDS and UDP sockets. Despite not being included in the final implementation, another messaging protocol/library, ZeroMQ, was considered during the development and initial evaluation phases. In these tests, the aim is to compare the performance of the three solutions: MQTT, ZeroMQ and DDS, which all implement a PubSub paradigm.

Methodology

This evaluation setup consists of a producer component and a consumer component. In all cases, both components were implemented in C++, in order to avoid any "overhead". Each group of tests includes the publication and receipt of 100k messages, which include the timestamp generated by the producer at the moment of publishing and extraneous padding in order to ensure that each message is exactly 800 bytes, so as to emulate the payload size of the JSON representation of a CAM message.

Upon receiving a message, the consumer calculates the time difference between its time of receipt and the timestamp included in the message, which represents the latency incurred by the protocol while transmitting the message. The tests are performed in a sole device, with the services implementing the publisher and the subscriber running in the same host.

Latency of different PubSub Messaging Protocols

This first evaluation is focused on measuring the average latency incurred by the messaging protocol between the publication and the receipt of the 100k messages. There are two groups of tests: a first where the communication is 1-to-1, i.e. one publisher to one subscriber; and a second test 1-to-5 where five consumers subscribe to the messages. Figure 3.9 presents the results of this evaluation.

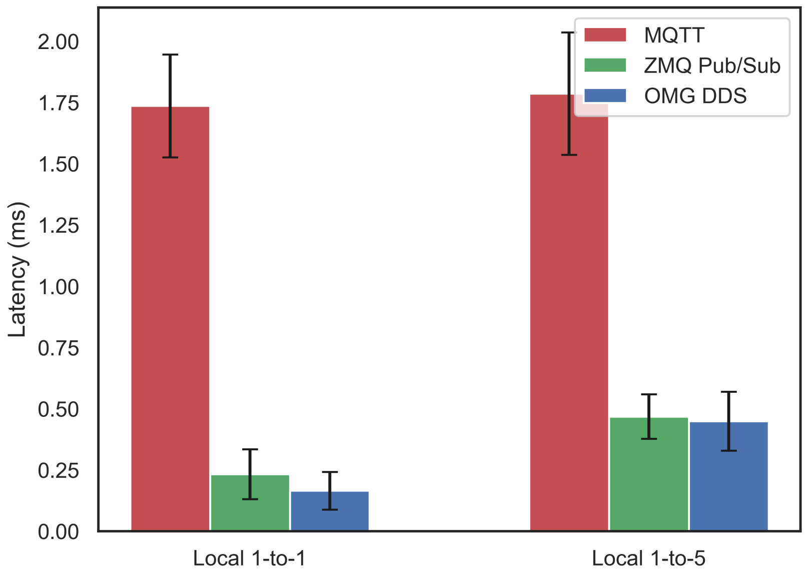


Figure 3.9: Latency of different types of PubSub communication protocols

These results show that the performance of both ZeroMQ and DDS is far superior to that of MQTT, which is attributable to their brokerless, decentralised, nature. This is an indication that the MQTT protocol should not be used in cases with increased sensitivity to latency. Moreover, although ZeroMQ and DDS have similar performance in this evaluation, DDS was selected as the most promising technology, due to its extensive set of configuration options for quality-of-service profiles and operation modes, envisioning some functionalities required for real-time applications.

Comparison between DDS' operating modes

Focusing on the DDS messaging protocol, this test compares different modes of operation to better understand their applicability in various use cases. The evaluated operation modes are: UDPv4, Shared Memory, Multicast, AutoThrottle and TurboMode. Figure 3.10 shows the results of this evaluation.

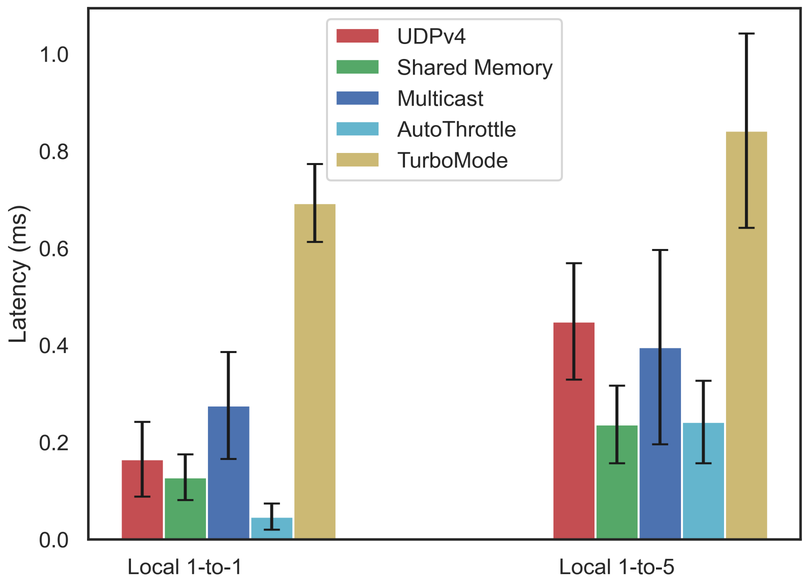


Figure 3.10: Latency incurred by DDS with different types of communications and different modes

The UDPv4 test corresponds to the same DDS results shown in the previous evaluation, serving as baseline for this comparison. The AutoThrottle and TurboMode modes also use UDPv4 as the transport layer. The default operation mode is SharedMemory, and it yields the lowest latency alongside AutoThrotle. However, SharedMemory cannot be considered if the services are containerised, due to that environment's inherent IPC namespace isolation. TurboMode's performance is the lowest, inline with the expected behaviour of this mode, since it optimizes the information batching for throughput performance over latency[42]. Since this work is focused in the study of deterministic and real-time services, for the next phase of evaluation tests, only the UDPv4 and Multicast modes are considered.

DDS containerised

One of DDS' interesting features is its scalability with multiple subscribers, despite its decentralised nature. This test measures the latency of 1-to-1 vs 1-to-5 when UDPv4 and Multicast modes are used, for both native and containerised versions of the service. Figure 3.10 depicts the results of this evaluation.

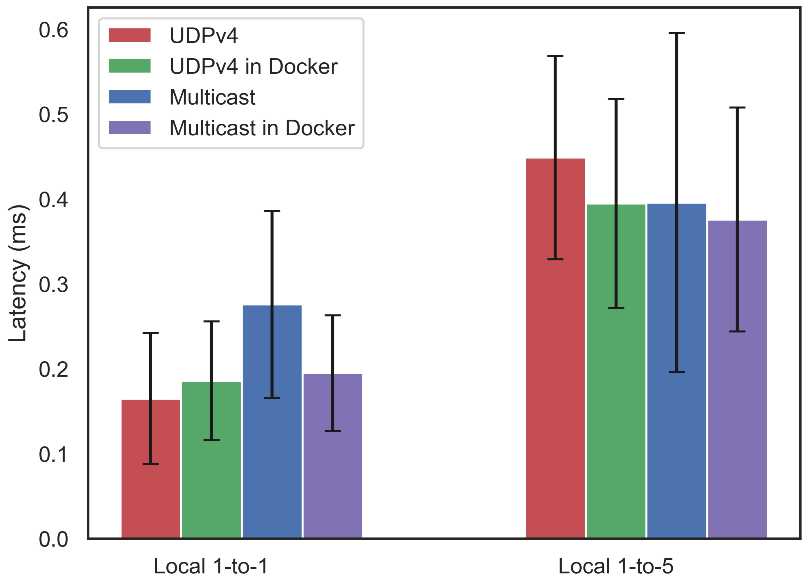


Figure 3.11: Latency of DDS with different types of communications and with containers

Although the Multicast mode shows more latency in the 1-to-1 case, mainly due to its overhead, when the number of subscribers is increased to five, the expected performance gains are immediately shown. Moreover, the utilisation of the Multicast mode in the containerised version shows promising latency results even in the 1-to-1 case. [42] performs is more extensive comparisons, although in a different hardware class. Nevertheless, the conclusions are very inline with the evaluation performed in this section.

3.6 Conclusions

This chapter presented the most common vehicular network architectures and discussed the differences between them. This analysis included open-source options such as the Router stack based on Voronov's GeoNetworking library, and the *Vanetza* project, which was ultimately used as the base for the implementation of the *Vanetza-NAP* protocol stack. This service was proposed to provide better support for MEC scenarios, with improved scalability, and envisioning time-sensitive applications. The chapter also contains an evaluation of *Vanetza-NAP* in comparison with the Router stack, and an evaluation between several data brokers MQTT, ZeroMQ and DDS, to gather conclusions about which messaging protocol is more appropriate to real-time, deterministic and scalable applications.

Notably, the *Vanetza-NAP* implementation presented in this chapter is already being used in other contexts and projects, including in production environments. More specifically:

- It integrates ongoing research work at the Network Architectures and Protocols (NAP) research group, including other Master's degree dissertations focusing on V2X use cases such as collective perception, vulnerable road users, and virtual traffic lights.
- It was already used in public demonstrations of V2X use cases with an autonomous shuttle during the Aveiro Tech Week 2022 (October 2022).

- It is currently deployed in the product environment of the ATCLL, comprising all the RSUs and OBUs within the infrastructure. It is a direct replacement of the previous protocol stack implementation in both RSUs and OBUs.
- Starting in the academic year of 2021/2022, it is included as an educational tool in the Autonomous Networks and Systems course of the Master's degree in Computer and Telematics Engineering at the University of Aveiro. Students use it to gain hands-on experience with VANETs and ETSI C-ITS messages, and also as part of their final project, if they choose to implement a V2X application.

CHAPTER 4

Deployment Strategies Evaluation

4.1 Motivation

As previously mentioned, vehicle-to-everything represents a major step in the evolution of ITS, by allowing vehicles and the infrastructure to share information seamlessly. The new set of sensors introduced by the next generation of vehicles, that can be autonomous or partially assisted, generate large volumes of data that, in some scenarios, may surpass the capabilities of available resource-constrained devices. Therefore, edge and fog concepts have been applied to vehicular networks under VEC, to perform computational offloading, content caching, data management, flexible network management, security, and others. Technologies like virtualization and containerization are the cornerstones when addressing service deployment on edge and cloud environments.

This chapter assesses different schemes of VEC service deployments, described in chapter 2, Section 2.4, to understand their use with the deterministic and reliable nature of some critical VEC services, such as an open-source implementation of the ETSI C-ITS protocol suite, called *Vanetza*. Service prioritization and preemption are explored in order to achieve an upper-bound latency in native and containerized services deployment.

This chapter is organized as follows. Section 4.2 presents the context of this work. Section 4.3 describes the different approaches and environment technologies, that will be deployed and tested. The respective results are depicted and discussed in Section 4.4. Finally, future approaches for deployment are introduced in Section 4.5, and the conclusions are presented in Section 4.6.

4.2 Context

New 5G verticals, such as connected and automated vehicles with V2X communication, represent a technological revolution in which the main focus is to remove/reduce human influence in all these processes [43]. To cope with this target, vehicles must be able to assess the environment conditions and quickly react to the dynamics of the environment and of its elements (i.e., using sensors like ADAS, LIDARs and cameras). This requires low latency communications, through V2X, to enable the transmission of the data between the different sensors and elements, and low processing time for data processing and computation, considering the strict timing requirements for the automated tasks between all elements. The first requirement is usually addressed through the network, using the 5G QoS identifier (5QI) [44], to provide low latency on the radio link layer (i.e., V2X 5QI). The second requirement is being addressed through egde computing, using VEC, providing application hosting from centralized data centers on the cloud closer to consumers, which also tries to meet the demanding key requirements of emergency applications, with new services on the edge, such as video processing and data fusion.

Although edge computing is justified with the need of low latency that in general is lower near the edge [45], current edge computing studies only focus on the unrealistic assumption that the localization and the huge data rates in the network is what is needed to support Ultra-Reliable and Low-Latency (URLLC) services. Therefore, the time-sensitive requirements in edge computing contexts have mostly been addressed from the network perspective [46]–[48]. Current research direction in the VEC topic only explores orchestration techniques in terms of service placement and service assignment [45]. However, it is imperative to also consider the processing latency for time-constrained applications, especially when using virtualised systems.

Moreover, on the proposal of future deployments with the aim to optimize the resources of devices on the edge, it is important to understand which deployment technologies are available [17], and how automatic deployment orchestration tools will be able to support latency-critical services. With this purpose in mind, this chapter studies different deployment configuration options in a real environment of VEC, considering the ones presented in Section 2.4, where edge computing is provided and used in a vehicular environment of sensed and connected vehicles and roads. This work considers deployment scenarios with vehicular services through the *Vanetza* protocol stack platform [41], with other services running in parallel on the vehicular edge. The test take into account the execution of both the *Vanetza* service and background processes in several different configurations, with/without RT priority to the service and/or to the background tasks, with and without containerized environments using Docker, and with and without the preemption of CPU time allocations.

4.3 Methodology

To evaluate the different approaches for service deployment (as described in Section 2.4), several rounds of tests were designed and centered around the processing latency of a high priority VEC service, *Vanetza-NAP*, an ETSI C-ITS protocol stack described in detail in the previous chapter.

This evaluation proposes a scenario in which an RSU with a running instance of the *Vanetza-NAP* stack is receiving CAM traffic originating from an OBU-equipped vehicle (Fig. 4.1). Upon the reception of a CAM, the Vanetza service deserializes the ASN.1 encoded payload and parses the message. Once those steps are complete, a JSON object containing

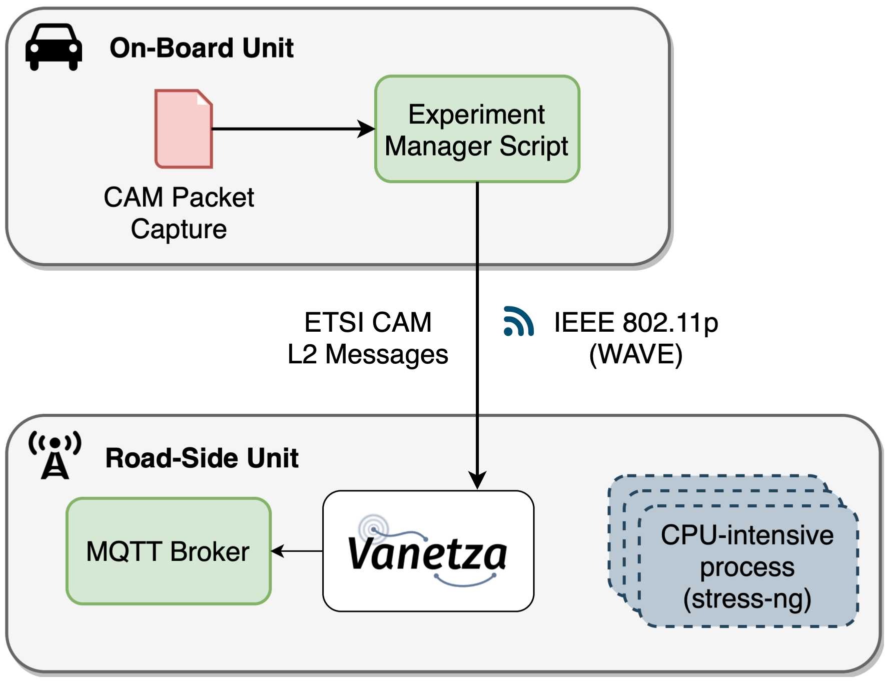


Figure 4.1: Experiment scenario: a RSU with Vanetza receiving CAM messages from an OBU equipped vehicle.

all of the relevant CAM fields is generated and published to an MQTT topic. This ensures that other edge and cloud applications within the ATCLL platform can use the information transmitted in the CAM.

In addition to the regular CAM fields, the JSON object also contains two separate UNIX timestamps:

- the *instant of message reception* in the RSU 802.11p interface;
- the *instant immediately before* the JSON string is published to the specified MQTT topic.

Therefore, for the purpose of this experiment, the stack's processing latency is defined *as the difference between* these timestamps, representing the total time elapsed while performing the decoding, parsing, and JSON generation tasks. In order to reduce the number of variables between experiments, every test used identical entry data, which consisted in a packet capture containing 10 minutes of CAM traffic exchanged between a RSU and OBU at a rate of 10 messages per second. This capture file was read and replayed by a Python script, which also collected the output data from the respective MQTT topics.

The experiment is performed using the PCEngines APU2 units of the ATCLL infrastructure, which are equipped with an AMD Embedded series 1GHz quad core CPU (GX-412TC) and 4GB of DDR3-1333 DRAM. At the time of the tests, the SBC was running Debian 10 with Linux kernel version 5.6.19, GCC version 8.3.0, and Docker Engine version 20.10.10. For the tests involving PREEMPT_RT, the version 5.6-rt12 of the patch was used to build a fully-preemptible variant of the 5.6.19 Linux kernel. *Vanetza-NAP*'s Docker image was built

Table 4.1: Cgroup scenarios description

Scenario	Description
SP	Both Vanetza and *stress-ng* running *in the same* Cgroup
LP	Vanetza in a hierarchically *inferior* Cgroup within the same top-level branch (System)
HP	Vanetza in a hierarchically *superior* Cgroup within the same top-level branch (System)
VHP	Vanetza in a *different top-level* Cgroup, isolated from any other processes
ND	Vanetza and *stress-ng* running *natively* but in a Cgroup configuration mirroring the default Docker Cgroup
Docker SP	Vanetza and *stress-ng* running in *Docker containers*, in the default Cgroup assigned by Docker
Docker HP	Containerized Vanetza running in *default* docker Cgroup, while *stress-ng* runs *natively* in the System Cgroup

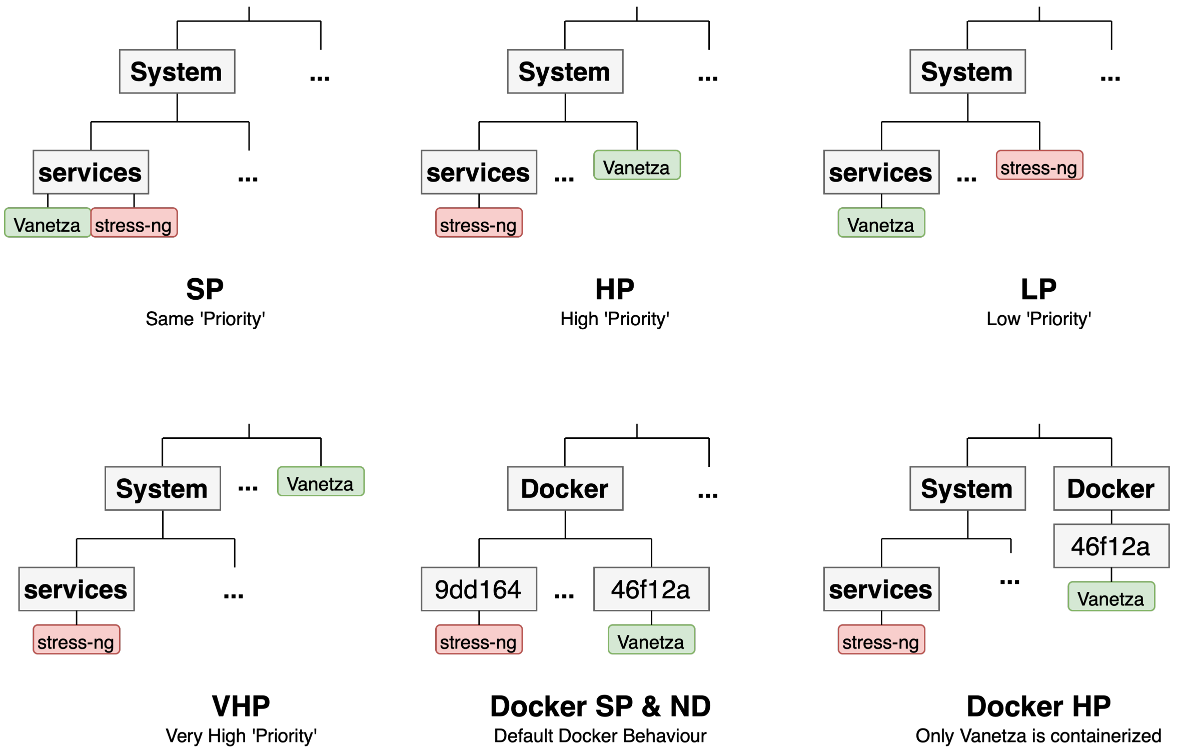


Figure 4.2: Cgroup test scenarios: SP; LP; HP; VHP; Docker SP and ND; and Docker HP.

using the same Debian version as the host SBC, and identical dependency versions whenever possible.

The main objective behind the experiment is to measure and analyse the impact of the different configurations, when compared to each other, and to the stack's performance baseline (BL). The experiments evaluate the impact:

- of running a service in parallel with a number of processes that synthetically increase the CPU load;

- of running both the service and the background tasks in several different Cgroup configurations;
- in the assignment of a real-time priority to the service;
- in the assignment of a real-time priority to the background tasks;
- of running the service in a containerized environment using Docker;
- of running the service in a PREEMPT_RT patched version of the Linux kernel.

To emulate low-priority CPU-intensive background tasks, multiple *stress-ng* processes were deployed, each one configured to use 2 CPU cores at 20% utilization. For the Cgroup component of the experiment, 7 different possible configurations are used, which are illustrated in Fig. 4.2 and specified in Table 4.1. Finally, the real time priority is assigned to the running processes using the *chrt* command (i.e. `chrt -r -p 98 PID`).

4.4 Results

This section presents the performance evaluation of several deployment configurations. Section 4.4.1 evaluates the impact of diverse services in concurrency (#Services and Native SP), and Section 4.4.2 presents the impact of running the services in different environments: Native, Docker and PREEMPT_RT. Section 4.4.3 measures the impact of several Cgroup configurations (HP, VHP, ND, LP). Section 4.4.4 evaluates the use of the PREEMPT_RT patch, Section 4.4.5 assesses the impact in concurrency with RT services, and lastly, Section 4.5 presents additional observations found in the experiments. The figures throughout the sections present the upper-bound values with the Complementary Cumulative Distribution Function (CCDF). Table 4.2 contains the results of each section, and it is used to get the numerical value of each case. To show the improvements in the upper-bound latency, the 99^{th} percentile is used.

Table 4.2: Complete version of the results in each deployment case

Runtime	CGroup	Stress RT	#Services	Mean		Median		Stdev		Min		Max		$P_{99\%}$	
				Vanetza RT											
				False	True	False	True	False	True	False	True	False	True	False	True
Native	SP	False	BL	0,56	0,57	0,48	0,49	0,23	0,20	0,28	0,29	5,88	3,14	1,96	1,31
			2	0,66	0,66	0,57	0,58	0,40	0,23	0,28	0,29	23,12	3,51	3,07	2,10
			5	0,78	0,76	0,72	0,73	0,72	0,28	0,29	0,28	46,00	4,35	10,32	2,78
			8	0,85	0,80	0,77	0,80	1,22	0,31	0,28	0,29	66,19	3,70	19,98	2,93
			10	0,88	0,85	0,80	0,85	1,30	0,32	0,28	0,29	43,51	4,18	21,37	2,94
	HP	False	BL	0,56	0,55	0,48	0,47	0,25	0,21	0,28	0,28	15,32	3,17	1,67	1,27
			2	0,66	0,64	0,54	0,54	0,47	0,24	0,29	0,29	20,83	3,51	4,13	1,70
			5	0,77	0,76	0,72	0,73	0,57	0,29	0,28	0,29	36,46	5,98	5,76	2,85
			8	0,81	0,81	0,78	0,80	0,56	0,31	0,28	0,29	31,10	4,03	5,27	3,08
			10	0,84	0,81	0,83	0,82	0,51	0,31	0,28	0,30	27,44	7,77	4,83	2,83
	HHP	False	BL	0,56	0,54	0,50	0,48	0,21	0,20	0,29	0,28	4,33	3,25	1,52	1,36
			2	0,64	0,64	0,54	0,54	0,32	0,24	0,29	0,29	22,17	3,95	2,18	1,73
			5	0,79	0,77	0,68	0,66	0,33	0,32	0,33	0,31	8,63	5,06	2,84	2,95
			8	0,83	0,82	0,77	0,74	0,36	0,34	0,31	0,29	9,42	3,61	3,32	3,02
			10	0,83	0,84	0,77	0,80	0,37	0,35	0,29	0,30	10,74	4,00	2,95	2,78
	LD	False	BL	0,56	0,54	0,50	0,48	0,21	0,20	0,29	0,28	4,18	3,35	1,76	1,55
			2	0,66	0,65	0,56	0,54	0,47	0,27	0,30	0,31	32,62	3,81	3,75	2,41
			5	0,80	0,76	0,68	0,64	0,68	0,33	0,31	0,32	24,40	3,68	10,72	2,88
			8	0,85	0,85	0,75	0,79	0,67	0,35	0,29	0,28	28,29	3,67	11,63	2,91
			10	0,88	0,86	0,80	0,84	0,71	0,34	0,29	0,30	23,06	3,86	13,54	3,17
	LP	False	2	0,66	0,66	0,56	0,57	0,29	0,24	0,28	0,28	9,38	3,27	3,04	1,73
			5	0,83	0,80	0,74	0,71	0,69	0,32	0,28	0,30	34,48	3,36	9,06	2,89
			8	0,79	0,79	0,76	0,79	0,54	0,29	0,27	0,28	30,17	3,41	7,45	2,72
			10	0,84	0,82	0,82	0,82	0,59	0,31	0,29	0,29	23,95	3,41	8,29	2,90
	SP	True	2	0,78	0,63	0,55	0,55	11,31	0,23	0,29	0,28	1093,19	6,80	4,51	2,42
			5	1,25	0,69	0,60	0,60	20,72	0,41	0,28	0,28	1207,08	34,40	29,80	2,82
			8	2,77	0,74	0.66	0,63	39,03	0,53	0,28	0,28	1408,48	48,00	693,43	2,98
			10	18,21	0,74	0,77	0,58	127,58	0,39	0,35	0,28	968,13	3,98	964,56	3,39
Docker	SP	False	BL	0,79	0,82	0,73	0,75	0,20	0,22	0,46	0,46	5,76	3,15	2,42	1,74
			2	1,01	0,87	0,82	0,79	0,74	0,26	0,48	0,44	20,67	3,46	10,02	3,06
			5	1,71	0,99	1,12	0,91	1,49	0,38	0,48	0,47	31,80	16,21	16,45	3,35
			8	2,13	1,10	1,69	1,06	1,97	0,35	0,49	0,52	62,93	8,01	22,92	3,42
			10	2,26	1,14	1,95	1,11	1,87	0,37	0,53	0,54	25,50	7,37	20,44	3,51
	HP	False	2	0,88	0,85	0,78	0,77	0,33	0,25	0,46	0,42	12,98	3,55	3,66	2,06
			5	1,21	1,01	0,96	0,92	0,76	0,33	0,49	0,48	12,56	3,86	6,27	3,29
			8	1,21	1,03	1,06	0,97	0,70	0,33	0,44	0,45	21,16	3,85	5,91	3,32
			10	1,18	1,04	1,07	1,01	0,61	0,34	0,48	0,42	10,55	8,14	5,82	3,13
Preempt	SP	False	BL	0,69	0,63	0,61	0,55	0,27	0,20	0,31	0,31	7,88	1,61	2,75	1,30
			2	0,79	0,69	0,68	0,62	0,58	0,24	0,29	0,31	27,92	2,23	7,00	1,69
			5	1,00	0,80	0,88	0,79	1,00	0,27	0,30	0,32	33,48	2,34	13,89	2,14
			8	1,06	0,86	0,91	0,87	1,18	0,28	0,31	0,33	28,89	2,71	14,90	2,30
			10	1,20	0,86	0,94	0,88	1,72	0,29	0,31	0,32	59,80	2,23	17,14	2,07
	HP	False	BL	0,68	0,61	0,59	0,51	0,33	0,21	0,31	0,30	13,17	1,72	2,62	1,29
			2	0,79	0,69	0,68	0,62	0,52	0,24	0,32	0,30	19,21	2,17	7,72	1,54
			5	0,90	0,81	0,81	0,80	0,64	0,26	0,29	0,34	23,34	2,40	8,74	2,13
			8	0,95	0,87	0,92	0,89	0,68	0,29	0,31	0,32	28,08	2,79	9,15	2,23
			10	1,01	0,89	0,97	0,91	0,74	0,30	0,32	0,33	30,70	2,53	9,07	2,24
	LP	False	2	0,81	0,77	0,68	0,65	0,47	0,29	0,31	0,32	17,88	2,40	5,93	1,80
			5	0,90	0,83	0,74	0,73	0,79	0,32	0,33	0,32	28,50	3,06	12,22	2,43
			8	0,92	0,84	0,82	0,82	0,62	0,30	0,31	0,34	19,03	2,70	9,49	2,25
			10	0,92	0,85	0,83	0,82	0,60	0,32	0,30	0,30	22,53	2,56	7,86	2,20
	SP	True	2	0,72	0,63	0,63	0,56	0,41	0,22	0,30	0,26	17,66	2,36	4,04	1,58
			5	1,02	0,67	0,69	0,58	9,06	0,53	0,29	0,29	629,49	50,29	18,16	2,40
			8	2,46	0,70	0,69	0,59	37,79	0,49	0,29	0,34	1713,25	32,40	843,88	3,20
			10	8,31	—	0,75	—	84,32	—	0,33	—	957,19	—	955,97	—

4.4.1 Concurrency in Services

This section presents the performance results of Vanetza, through CCDF and processing time, to analyse the impact of 0, 5 and 10 concurrent services in Normal and RT priority. Figure 4.3 presents these results, and the numerical values were presented previously in Table 4.2.

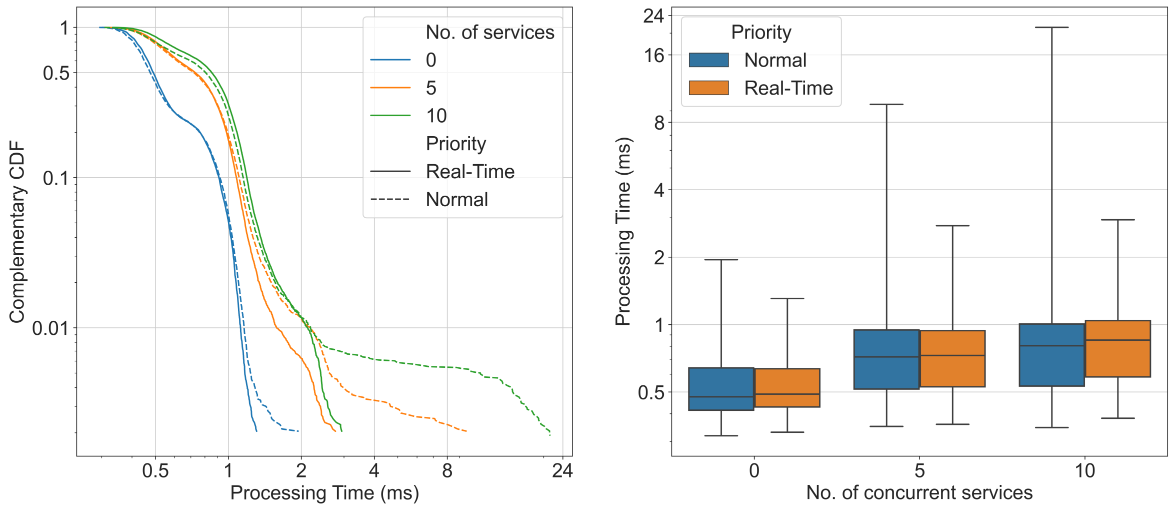


Figure 4.3: Concurrency in services with 0, 5 and 10 concurrent services in Normal and RT priority.

As the number of *stress-ng* services increases, the processing delay of the Vanetza service degrades both in terms of absolute value and consistency/stability. In fact, the test scenario with 10 parallel services, for $P_{99\%}$, shows an increase factor of 10.9 (i.e., 21.37/1.96 ms), and a 5.7 increase factor for the standard deviation, when compared with the BL (i.e., 1.30/0.23 ms). In extreme cases, this could potentially compromise any minimum response time guarantees/constraints.

The use of a real-time priority (*chrt*) greatly increases the predictability of Vanetza's processing delay, regardless of the number of concurrent *stress-ng* services currently in execution (i.e., from 21.37 ms to 2.94 ms, $P_{99\%}$). This benefit is also present when there are no *stress-ng* processes running, given that Vanetza is still in competition with miscellaneous Debian processes. Despite this, the median values consistently register a slight increase when RT priority is applied (i.e., from 0.56 ms to 0.57 ms, BL). This effect can also be observed in the CCDF graph, where a sizeable proportion of values obtained with RT priority are slightly higher than their non-RT counterparts.

This result is deemed to be due to the added scheduling and context switching overhead introduced by the usage of Linux' real-time scheduling features and the frequent preemption of lower-priority processes, which are discussed in [49]. Scheduling overhead models the overhead incurred by the clock interrupt handler which is responsible for scheduling, i.e. moving newly arrived and preempted jobs between the queues and selecting the running job. Context switch overhead represents the overhead associated with preempting the current job, saving its context, loading the context of the next job and resuming the next job. This reasoning is consistent with the fact that this discrepancy increases with the number of services, given that scheduling overhead scales linearly with the number of concurrent tasks, and that a higher number of parallel services increases the likelihood of preemption and the respective context switching operations. Resuming, the stability gains outweigh the slight overhead incurred.

4.4.2 Running Environments

This section presents the performance results of Vanetza in the overhead of Native, Docker and PREEMPT RT environments, in normal and with real-time tasks. Figure 4.4 presents these results, and the numerical values were presented previously in Table 4.2.

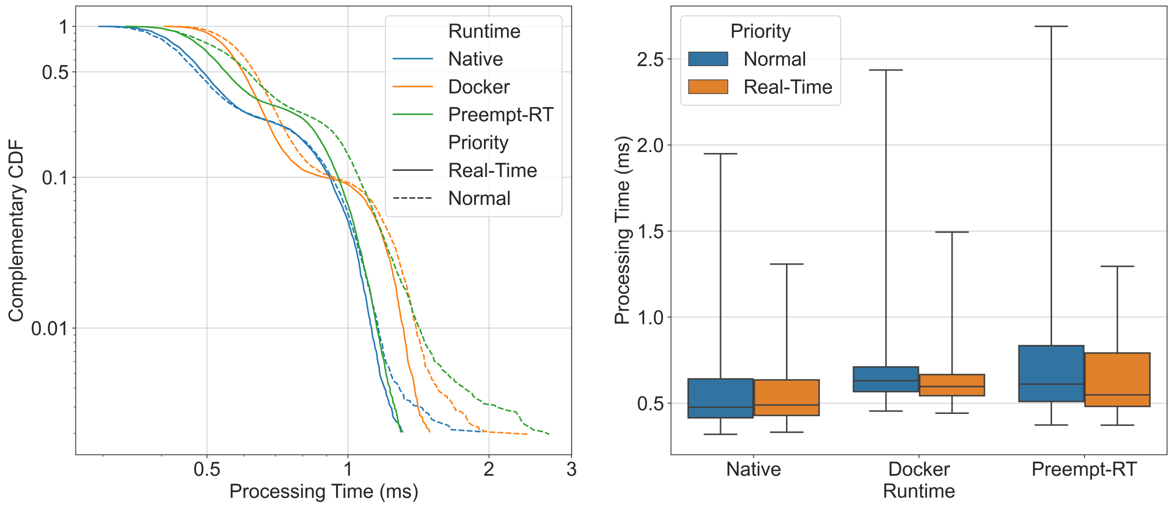


Figure 4.4: Runtime environments with Native, Docker and PREEMPT RT.

As discussed in [32], the PREEMPT_RT patch achieves its goal of low latencies and predictability of RT tasks at the expense of reducing the total system throughput, through an increased rate of context switches, resource contention, and priority inversion scenarios. As such, the significant decline in the Vanetza's BL performance without an RT priority is attributed to the added overhead (i.e., from 1.96 ms to 2.75 ms), while the subsequent application of an RT priority leads to a more pronounced improvement with a factor of 1.42 (i.e., 2.94/2.07 ms), when compared to a standard kernel. There is no benefit to using PREEMPT_RT in this specific situation where no CPU-intensive background tasks exist.

For this specific workload and hardware, Vanetza incurs an average performance penalty with a factor of 1.41 when running within a Docker container (i.e., 0.79/0.56 ms), which may be due to additional security features, OverlayFS latency, and other potential sources of Docker overhead. The application of an RT priority to the Vanetza container works as expected, and leads to a greater relative improvement, when compared to the native baseline.

4.4.3 CGroup Evaluation

This section presents the performance results of Vanetza with different Cgroup configurations, in normal and with real-time tasks. Figure 4.5 presents the results of the overall impact of Cgroup configurations, and the numerical values were presented previously in Table 4.2.

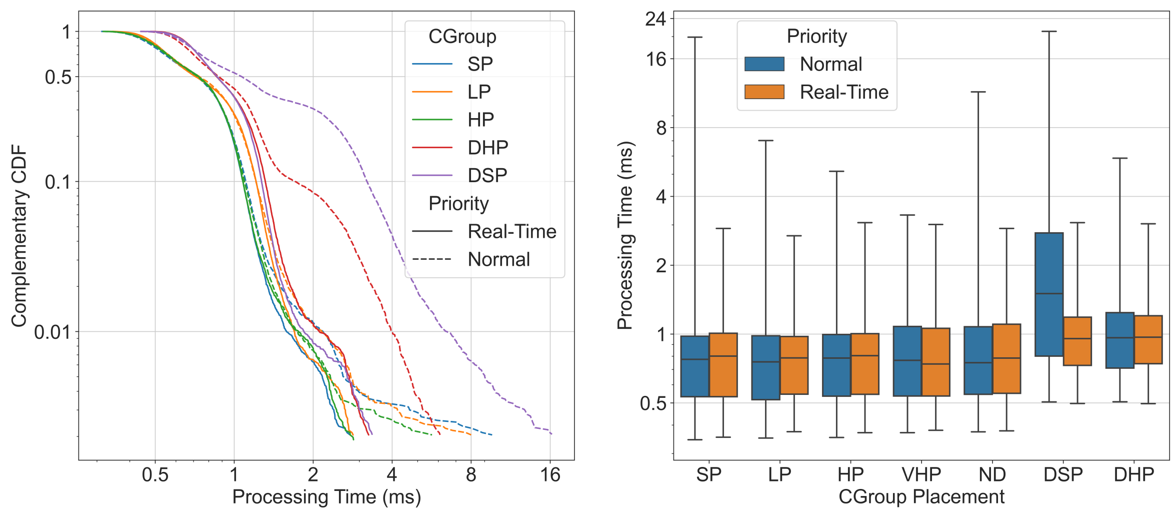


Figure 4.5: Different CGroup configurations.

The effects of different Cgroup configurations become more pronounced when there is a higher number of CPU-intensive concurrent tasks. This is due to the fact that a Cgroup's resource usage is only limited by the scheduler if the current CPU utilization is high (so as to guarantee that each Cgroup gets, at least, its minimum guaranteed CPU share allocation [50]). As specified in the Linux kernel documentation [51], the percentage of CPU time assigned to a Cgroup is equal *to the value of shares divided by the sum of all shares in every Cgroup in the same hierarchical level*. Given that each Cgroup has identical shares and usage quotas, the most beneficial configuration is to place critical services in their own isolated top-level Cgroup, thus maximizing the allotted CPU time.

Conversely, when there is a sufficient level of interference from CPU-intensive background tasks, the worst case scenario for critical services is sharing the same exact Cgroup as those tasks (SP scenario), which allows them to potentially use the majority of the allocated CPU time and starve the high priority processes. Configuring both HP and LP tasks in children Cgroup to the same parent is also discouraged, as they will each share the parent's allocation. Docker containers will, by default, use this configuration.

The obtained results are consistent with these observations: 1) isolating Vanetza as much as possible (VHP) yields the best performance (i.e., $P_{99\%}$: 2.95 ms); 2) configuring Vanetza in a higher hierarchical level within the same top-level parent (HP) returns better results than the reverse configuration, in which Vanetza is placed in LP (i.e., $P_{99\%}$: 8.29/4.83 ms); 3) running both types of processes within the SP results in the worst performance and stability.

Carefully considering the Cgroup placement of both critical services and background tasks is a worthwhile strategy for improving the performance and stability of high priority processes. This may be tuned further by adjusting the relative weights (shares) between Cgroup and the respective CPU usage quotas. Nevertheless, configuring a RT priority still supersedes any CFS tuning, thus yielding the best results in terms of stability. The Vanetza's Docker performance (SP) is more susceptible to interference from CPU-intensive background tasks than its native counterpart's. Isolating its Cgroup from the background tasks (Docker HP) results in a very significant improvement, but does not fully rectify this effect.

4.4.4 PREEMPT_RT Evaluation

This section presents the performance results of Vanetza with the PREEMPT_RT patch and concurrent services, in normal and with real-time tasks. Figure 4.6 presents the results of the overall impact of PREEMPT_RT, and the numerical values were presented previously in Table 4.2.

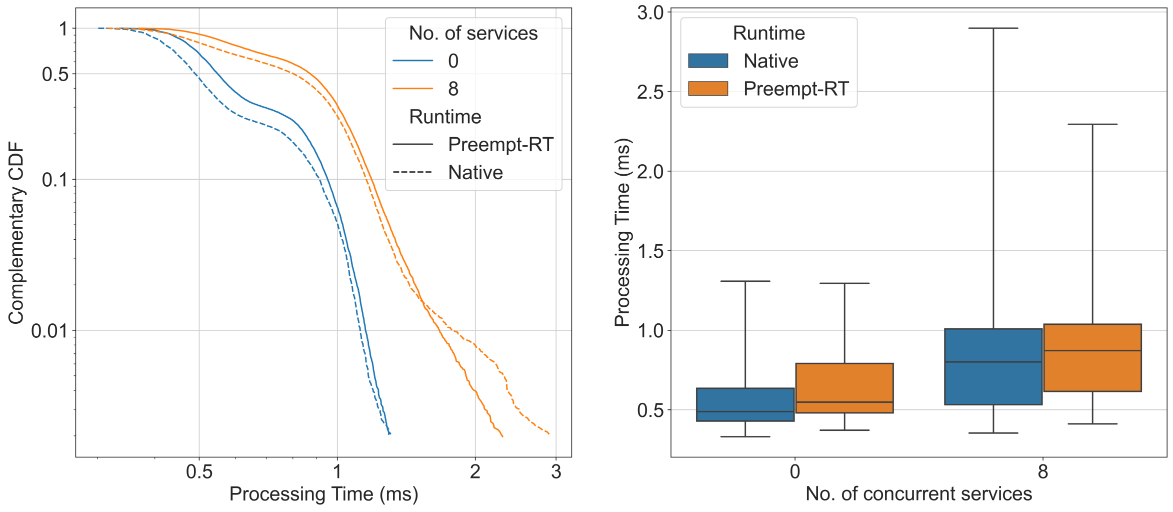


Figure 4.6: PREEMPT_RT with different concurrent services.

As previously discussed, the PREEMPT_RT patch leads to a slight increase in the average and median processing delay. However, it also results in positive effects for RT tasks, which become more evident as the system load increases. In fact, when running 10 parallel *stress-ng* services, a PREEMPT_RT patched kernel will improve the standard deviation and the $P_{99\%}$ value by factors of 1.08 and 1.42, respectively, when compared to the vanilla kernel. This may be a worthwhile tradeoff for critical services, but it is important to consider the effects on the overall system throughput.

4.4.5 Real-Time Service Priority

Finally, this section presents the performance results of Vanetza considering different concurrent RT services, in normal and with real-time tasks. Figure 4.7 presents the results of the overall impact of concurrent RT services, and the numerical values were presented previously in Table 4.2.

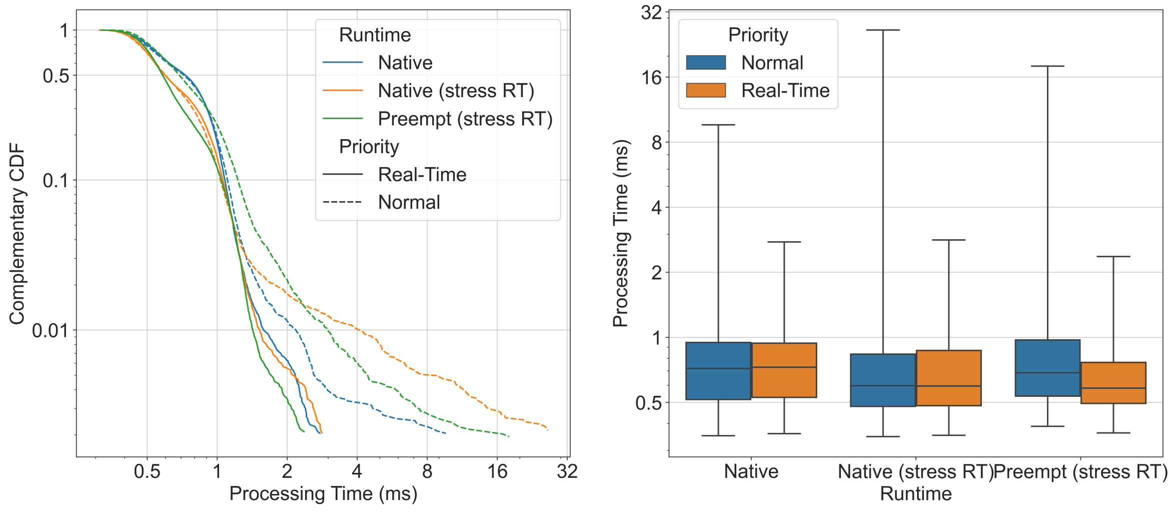


Figure 4.7: Concurrency of RT services with priority.

The execution of CPU intensive processes, configured with RT priority, has a very substantial negative impact for critical services running with a lower priority or without an RT priority altogether. In the case of the comparison between Native SP with and without stress RT, the $P_{99\%}$ for 8 services suffers a 32.44 increase factor (i.e., 693.32/21.37 ms), as evidenced by the horizontal growth of the CCDF's tail. Using a Linux kernel patched with PREEMPT_RT will yield, predicatively, higher average values but lower peaks and standard deviation, for the aforementioned reasons. When Vanetza is configured with a higher RT priority than the *stress-ng* processes, its performance returns to normal levels, which is expected, given that RT processes preempt all processes with a lower priority value.

4.5 Other Observations

While implementing and executing the testing strategy that was previously outlined, additional factors were discovered, which should be taken into consideration when setting up an edge computing environment.

The Kernel Address Sanitizer (KASAN) is an optional memory debugging tool included in the Linux kernel that detects dynamic memory access errors by running additional checks on every memory access operation. While this may not introduce any significant overhead in modern systems, it was found that it can drastically reduce the overall system throughput in older, more low-powered hardware such as the PCEngines APUs used in our tests. In fact, running a kernel compiled with the CONFIG_KASAN flag resulted in a significant performance penalty, with average and standard deviation values increasing by factors of 1.20 and 1.43. This is illustrated in Figure 4.8 and Table 4.3.

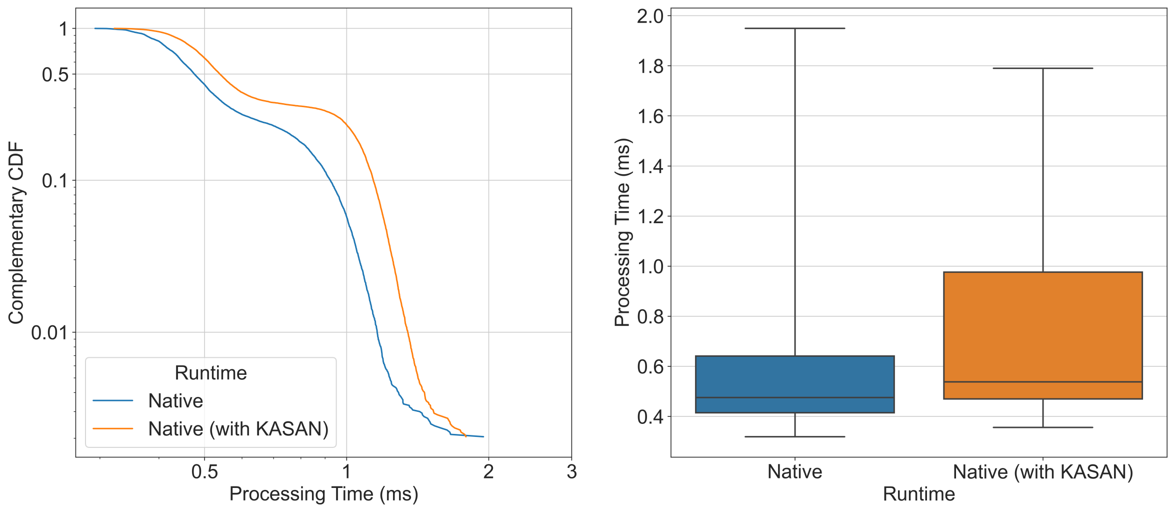


Figure 4.8: Evaluation of the impact of Kasan.

In order to improve security, Docker isolates containerized processes using SECCOMP profiles, which are a linux kernel security feature that restricts the system calls that are available within a given container. One of the measures included by default is the suppression of indirect branch prediction, which is also known as Single Thread Indirect Branch Predictors (STIBP), as a way to mitigate Spectre V2 side-channel attacks. Unfortunately, this introduces performance hits of up to 50% [52] on certain CPU-intensive workloads, which is why it is currently used only within SECCOMP jails, despite being originally intended to apply system-wide. This is illustrated in Figure 4.9 and Table 4.3.

By disabling the default SECCOMP profile on our Vanetza containers, the observed average and standard deviation values improved by factors of 1.15 and 1.03, respectively. As of version 5.16, the Linux kernel development team has decided to disable STIBP within SECCOMP jails, citing the "excessive impact on performance"[53].

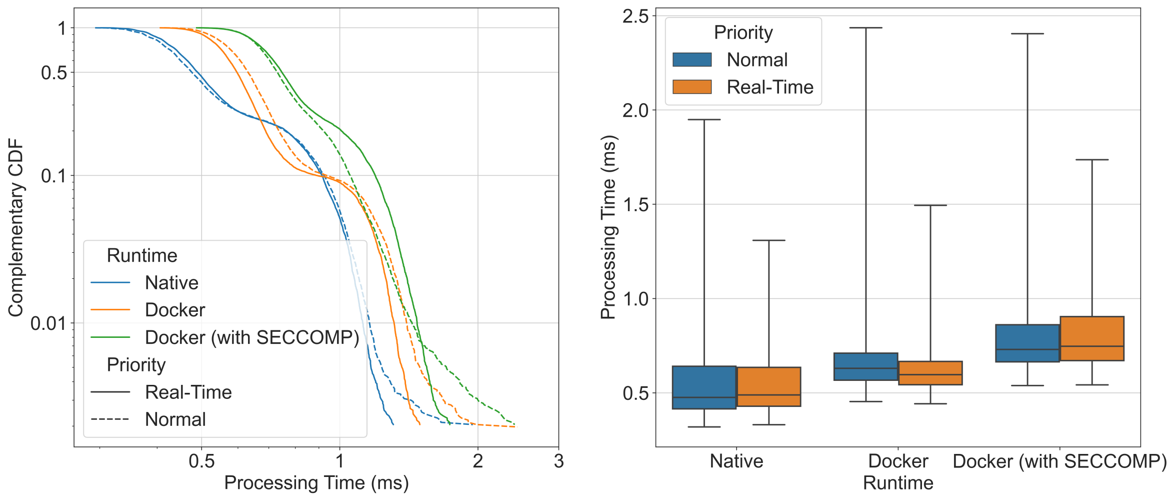


Figure 4.9: Evaluation of the impact of SECCOMP

Table 4.3: Vanetza running natively with a KASAN-enabled kernel and in Docker with SECCOMP (in milliseconds)

Configuration	Mean	Stdev
Native Vanetza with KASAN	0.68	0.29
Native Vanetza without KASAN	0.57	0.20
Docker Vanetza with SECCOMP	0.79	0.20
Docker Vanetza without SECCOMP	0.68	0.20

4.6 Conclusions

Time constrained services need to follow specific upper-bound latencies, that can be greatly impacted by different runtime environments and configurations. As shown in this chapter, poor configuration choices can lead to an increase of around 3200% for $P_{99\%}$, when a normal service is executed in parallel with services configured with RT. This work presented the impact of these service configurations in an example service, Vanetza, one of the aforementioned ETSI C-ITS protocol stacks. Our work explores the overhead of several runtime environments (Native, Docker and PREEMPT_RT), the impact of cgroup level, and the use of real-time prioritization with *chrt*.

The next chapter aims to integrate some of this knowledge in an orchestration tool. It envisions a system that, through custom scheduling plugins, selects the best worker node for a given workload based on, among other factors, the feasibility of assigning it a given real-time priority depending on other containers already running. Through a custom *Operator*, this system will closely monitor metrics exposed by the nodes and the applications, to adapt to any situation by dynamically changing real-time priority allocations and re-scheduling services if required.

CHAPTER 5

Intelligent Service Orchestration Proposal

5.1 Motivation

Chapter 3 highlighted the pivotal role of dependencies in microservice architectures, and the importance of minimizing link latency between dependant applications. However, most orchestration solutions do not measure the quality of network links between two nodes when attempting to schedule workloads close to their dependencies, opting instead for best-effort heuristics. Similarly, the algorithms responsible for load-balancing network requests between available replicas of a destination service typically prioritize absolute fairness regardless of the latency between the candidate replicas and the originating application. This may not constitute a problem in cloud environments where worker nodes are most likely located within the same building or even the same hypervisor host. However, in geographically dispersed Smart City edge-computing clusters, where worker nodes can conceivably be located in opposite ends of a city, this behaviour likely results in severe adverse effects on the overall end-to-end latency of the service, and essentially neutralises the advantages of using the specialised communication techniques discussed in chapter 3. In addition, chapter 3 also presented the benefits of exposing application-level metrics and the potential uses of that information within the orchestrator's control plane, such as complementing the aforementioned scheduling and load-balancing decisions so that the best or most available replicas are chosen. Unfortunately, none of the orchestration solutions that were contemplated currently implement such a feature. Finally, in chapter 4, it was shown that the impact of poor configurations in the service execution can be very strong, and that new approaches for the orchestration of services are required.

Given this context, the main objective laid out in this chapter is to apply the various lessons learned thus far in this dissertation to the creation of an orchestration framework that improves on the current state-of-the-art in several key areas, namely:

- Automatic configuration of real-time scheduling parameters for critical processes;
- Scheduling of workloads to nodes based on customised algorithms that take into account several factors such as the latency and application-level metrics of their dependencies;
- Automatic migration of workload allocations that become sub-optimal over time;
- Load balancing of network requests based on customised algorithms, that take into account the latency and application-level metrics of the destination replicas.

The convergence of all these additions, coupled with the use of the design principles discussed in chapter 3 and the knowledge acquired in chapter 4, should attain this dissertation's stated objective of substantially improving the stability of time-constrained Smart City services, regardless of the level of interference posed by other processes.

This chapter is organized as follows. Section 5.2 describes the requirements of the proposed system and its functionalities, while section 5.3 depicts the architecture, its systems, and describes its main modules. Section 5.4 is a extensive section that details the different operations and algorithms proposed to orchestrate all the services according to their requirements. Finally, section 6.5 presents a summary of the chapter and what can be achieved with this orchestration framework proposal.

5.2 Requirements

This work's goal is not to implement an entirely new orchestrator, since that would prove a time-consuming and ultimately ineffective process. Instead, the proposed functionalities and integrations should, whenever possible, be implemented as extensions of the various relevant components of existing orchestration solutions, either by following the plugin design pattern or by making extensive use of available control plane APIs and SDKs. In addition to serving as a proof-of-concept for the newly-added functionalities, the work described in this chapter is aimed to eventually be used in real infrastructures. In practice, this translates into a set of conscious efforts, as non-functional requirements, that should be reflected in most design and implementation decisions, namely:

- The design of workload specification files and other points of interface with developers should strive for ease-of-use and avoid overly complex schema that requires specialised knowledge. From a pragmatic standpoint, this can also be considered a net benefit for the stability of the cluster, since it reduces the probability that some developers will choose to deploy their applications manually in order to avoid an excessive learning curve, which would introduce extraneous resource usage not managed by the orchestrator;
- Despite the previous point, the pursuit of ease-of-use should not result in excessive assumptions that restrict the developer's freedom to implement a complex app or specify an advanced deployment, whenever possible. This principle also extends to the system administrators that will install and manage the system. Meaning that its various components should be loosely coupled to each other and to any modules of the base orchestration framework;

- The implementation should be as topology-agnostic as possible, in order to increase compatibility with different types of Smart City infrastructures and use cases. While the ATCLL project serves as an important source of inspiration and a likely first candidate for a future pilot test of the system, it does not restrict the scope of this implementation.

Besides these non-functional requirements, which take into account the usability of the orchestration tool in real infrastructures, the following subsections present the functional requirements that need to be supported.

5.2.1 Runtime Configuration

To make use of the advantages of the containerisation paradigm discussed in chapters 2 and 4, the system should be able to orchestrate deployments that are executed in containerised environments and packaged as images. However, the system should also retain the capability to deploy legacy applications that have not yet been migrated to containerised runtimes, or cannot feasibly run as containers for compatibility or operational reasons. This ensures that the orchestration process takes into account these types of applications, which enables more informed decisions, thus representing a net benefit to the infrastructure as a whole.

To leverage the performance and stability gains observed in chapter 4, the orchestrator must be able to autonomously apply real-time scheduling priorities to the processes that comprise a critical service, regardless if it is deployed as a legacy application or in a containerised environment. This feature should support the allocation of simple SCHED_FIFO priorities, as well as more advanced SCHED_DEADLINE attributes. Finally, it must have the capability to update the scheduling parameters of existing services to respond to new real-time deployments with varying priorities.

5.2.2 Deployment Specification

The system must provide a specification file format which developers can use to define a new service and its various deployment constraints. Most orchestration solutions already follow this paradigm and offer their own particular schema with varying levels of functionalities.

To ensure that developers have access to a robust set of options to fine-tune the deployment of their applications and how they are scheduled to the cluster's worker nodes, this proposal's specification file should include support for configuring the following base orchestration features:

- Container image and version tag
- Labels
- Mapped network ports
- Replication behaviour
- Affinity and anti-Affinity restrictions for worker nodes and application dependencies
- Resource requests and limits
- Security parameters
- Mapped volumes and/or configuration files
- Environment variables

- Liveness checks

However, several of the capabilities introduced in this proposal require additional, bespoke, configuration options that must also be specified by developers in order to be effective:

- Details on how to install, prepare, and execute the service as a legacy application;
- Real-time scheduling parameters for each RT process spawned in the container, such as the SCHED_FIFO priority or the SCHED_DEADLINE runtime, period, and deadline;
- A set of application-level metrics that should be collected and monitored by the orchestration framework.

5.2.3 Deployment Scheduling

The process of selecting a worker node for a given deployment should take into account the real-time scheduling attributes of both the candidate application and the services that are already in execution, if any. That way, RT services can be scheduled in a balanced manner that minimizes their interference on regular services. The scheduler should also compute the feasibility of assigning the deployment to a given node, in terms of quota utilisation, so as to prevent the over-allocation of RT workloads relative to the actual capacity of the node.

In scenarios where a deployment specifies one or more dependencies, the scheduling process should give preference to nodes that minimize the network latency between those services. This dependency-aware algorithm should also take into account the current performance level of each of the replicas of a dependency, thus preferring the most-performing replicas and maximising the quality of service.

Finally, the allocation of new deployments to nodes where an existing higher-priority service is currently in a degraded state should be blocked.

5.2.4 State Monitoring

When considering kubernetes, Pods are scheduled to the most optimal spot at the time. Since the scheduling algorithm takes into account complex factors, over different nodes, this has to be included in the scheduling approach. However, most orchestration solutions do not re-evaluate a deployment's placement after the initial scheduling process. This means that the cluster will inevitably deteriorate into a sub-optimal state.

To address this, the system should continuously monitor active deployments and proactively migrate them to other nodes that increase their performance and/or the overall cluster balance. Furthermore, in cases where the performance of a high priority deployment enters a degraded state (as measured through its application-level metrics), the system should autonomously begin evicting lower priority services from the worker node in question.

Finally, in mobile and moving environments the orchestration needs to be location-aware and follow the mobility of the nodes and the requirements of services.

5.2.5 Load-Balancing

In order to take advantage of the dependency-aware scheduling algorithm, the system should include an intelligent load-balancing algorithm that gives precedence to destinations that incur the lowest amount of latency when forwarding network requests to replicas. Additionally, the current performance levels of the replicas should also be taken into account, so as to increase the overall stability and quality of service.

5.3 Proposed architecture

This section presents an architectural proposal for a framework that extends existing orchestration solutions in order to fulfil the requirements specified above. An overview of this architecture is depicted in figure 5.1, followed by a high-level description of each component and its purpose.

Given its extensional nature, the proposal includes a mix between components that introduce new features envisioned for the purposes of this dissertation, and industry-standard components that already have several implementations available for use (such as the orchestrator itself). In order to distinguish between the two types of modules more clearly, the former are pictured as highlighted blocks in figure 5.1.

This section introduces each module in a generic manner, without restricting to any specific existing implementation (ie: Kubernetes, Docker Swarm, etc). In fact, the solutions

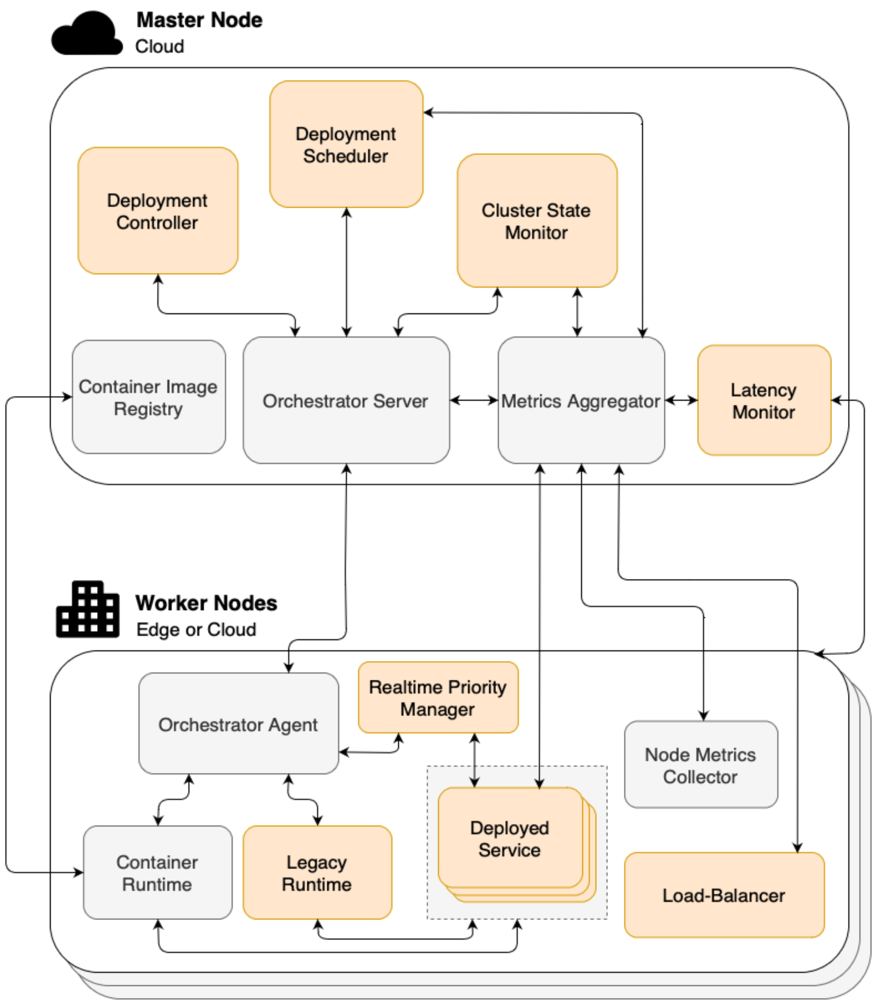


Figure 5.1: Overview of the proposed architecture

presented in this architecture are applicable to most orchestration frameworks. However, it is important to note that finished implementations may present slight differences depending on the underlying orchestrator that is chosen, since each one provides distinct avenues for extensions and integrations. For instance, in a Kubernetes-based implementation, the Deployment Scheduler module can be developed as a set of plugins to the existing scheduler instead of as a separate component. Similarly, a Nomad-based implementation will not require the Legacy Runtime module, since the orchestrator includes native support for that feature.

From a functionality standpoint, the architecture can be divided into 3 distinct groups, which are described in the following subsections.

5.3.1 Orchestration

This includes all the components that contribute to the service deployment pipeline, from the moment that it is requested, to when the service is successfully executing in a worker node with the desired real-time priority applied, if applicable:

- **Orchestrator Server** - Manages and monitors all deployments and worker nodes on the cluster. It exposes an Application Programming Interface (API) that services can use to request a deployment and consult its state.
- **Deployment Controller (new)** - Interprets the customised Deployment Specification File and converts it into a set of actions that the orchestrator server must perform in order to successfully complete the deployment.
- **Deployment Scheduler (new)** - Each time a new deployment is queued, this component analyses its requirements and the current state of the cluster in order to select the most appropriate worker node to allocate it to. This selection is done based on several factors, including the current value of crucial node metrics, the RT feasibility of the node's task set, and the node's proximity to the deployment's dependencies.
- **Orchestrator Agent** - Interfaces with the Orchestrator Server and with the available runtimes in order to create, manage, and monitor local deployments.
- **Container Runtime** - Manages the complete lifecycle of a containerised deployment, including downloading the respective image, creating and executing the container, and exposing its logs and current status to the orchestrator.
- **Container Image Registry** - Provides centralised storage of container images for every deployment. Images are uploaded by developers and administrators, and downloaded by the Worker nodes after a new deployment is scheduled.
- **Legacy Runtime (new)** - Manages the complete lifecycle of a legacy deployment, including downloading and installing dependencies, executing the relevant processes, and exposing their logs and current status to the orchestrator.
- **Realtime Priority Manager (new)** - Interfaces with the worker node's operating system in order to apply the desired realtime priority or SCHED_DEADLINE attributes to the processes of a newly assigned deployment. It is also capable of dynamically re-assigning the priorities of existing deployments, if necessary.

5.3.2 Metrics Aggregation

This includes the components that comprise the metric collection, persistence, and querying architecture of the proposed system:

- **Metrics Aggregator** - Continuously pulls the most recent values from the various metric sources and persists them in a database, thus allowing the remaining architectural components to perform complex queries for both current values and historical trends.
- **Deployed Service (new)** - When applicable, it exposes application-level metrics that provide insight into the performance of the application and allow the orchestrator to perform corrective actions when it enters a degraded state.
- **Node Metrics Collector** - Exposes a vast array of detailed metrics regarding the state of the worker node and the utilisation of its resources (CPU, Random Access Memory (RAM), Storage, Networking, among others).
- **Latency Monitor (new)** - Continuously collects and exposes the current ping averages for every node in the cluster, thereby enabling other components to make informed decisions regarding the state of the network.

5.3.3 Monitoring and Load-Balancing

This includes other components that do not fit in the previous groups, but play a critical role in assuring quality of service and stability over time:

- **Cluster State Monitor (new)** - Continuously monitors the current cluster state in order to correct deployment allocations that have gradually become sub-optimal by evicting the respective deployments, thereby forcing their re-scheduling into the most suitable node.
- **Load-Balancer (new)** - Balances network requests to a service between all the running replicas of that service. When possible, this process takes into account the current network link latency between the client and the potential destination, and additionally, the application-level metrics exposed by the replicas in order to give precedence to the replicas which will provide the best Quality of Service.

5.3.4 Metric Monitoring Infrastructure

The metric collection and persistence infrastructure is a critical component of the proposed architecture, since, as mentioned in the requirements section, the system's scheduler and load-balancer implementations are expected to take application-level metrics into account.

To accomplish this goal, this implementation makes use of one of the several metric aggregation software stacks that are freely available for use. In the case of this implementation, the Prometheus stack was deemed the most suitable option due to its ease of use and widespread industry adoption, which ensures the existence of metric exporter libraries for nearly every programming language. This is crucial, since it allows developers to integrate metrics into their applications without being constricted in their programming language selection.

Additionally, selecting the Prometheus stack also provides the opportunity to leverage third-party community tools that are capable of integrating directly with some internal Kubernetes mechanisms such as the HorizontalPodAutoscaler, as described later in this chapter. In order to greatly simplify its deployment and configuration, the Prometheus stack is installed within the cluster itself through the use of the prometheus-operated helm chart, which provides the following relevant components:

- **Prometheus Server**: Keeps a time-series record of node and application-level metrics throughout the entire cluster; it responds to queries by Kubernetes components/clients.
- **Alertmanager Server**: Manages and executes the configured metric-based alert notifications.
- **Prometheus Operator**: Automatically reconfigures the prometheus server to scrape metrics from new application-level exporters that declare a ServiceMonitor Custom Resource.
- **NodeExporter DaemonSet**: Deploys to each node a node-level metrics exporter that exposes a very comprehensive set of system metrics (CPU, RAM, Networking, ...).

For the purposes of this implementation, the Prometheus Server, Alertmanager, and Operator components are configured to run on a cloud-based worker or master node, so as to avoid burdening a resource-constricted edge computing node with unnecessary processing and network overhead. Furthermore, this increases their proximity to other critical components of the implementation which will be the major sources of metric queries. Namely, the scheduler and FogService Operator.

5.4 Operation and Algorithms

This section details how this architecture and its various components operate in practice. Each subsection describes the technologies used, as well as the algorithms proposed together with the advantages, disadvantages, and the mitigation approaches.

5.4.1 Orchestrator

The first major design decision that was undertaken during the implementation process was the choice of the underlying orchestration system. This decision is crucial, since it informs the different APIs that will be available for the integration of the other modules described in the architecture, as well as the overall feature-set and advanced algorithms that the system will support.

As mentioned in the beginning of this chapter, this implementation is not meant as a proof-of-concept for the proposed innovations. Instead, its final objective is to eventually be deployed in production Smart City environments with complex requirements and multiple partner entities creating deployments. As such, the orchestrator selection process is focused primarily on identifying projects that:

- Provide extensive APIs and other mechanisms for the friction-less extension of default behaviours with custom domain-specific logic;

- Implement advanced features across several domains such as workload scheduling, monitoring, autoscaling, load-balancing, among others, thus increasing the overall ability of the system to maintain service availability and workload balance, while reducing the effort required to implement the proposed additions;
- Can feasibly be used in resource-constrained environments without introducing excessive undue overhead;
- Are widely known and used and, ideally provide intuitive interfaces for deployment specification, so as to minimise the learning curve and overall barrier to entry.

Based on the analysis made in section 2.5.4, the Kubernetes project was the one selected, because it is widely used and extensible, with advanced features and tooling. As a drawback, it is not very lightweight and does not support legacy applications. Since Kubernetes was developed primarily for cloud computing environments containing much more powerful hardware, its official release is not very suited for its use in low-powered devices. Instead, those types of deployments should use alternative distributions such as MicroK8s or Rancher k3s, which include several optimisations in terms of overall resource overhead. For the purposes of this implementation, Rancher k3s was considered more favourable since, unlike MicroK8s, it also supports ARM32 devices like older Raspberry Pi models (1, 2, and 3).

5.4.2 FogService Custom Resource Definition

As discussed previously, one of the significant issues that hinder Kubernetes' adoption as a service orchestration solution is its complexity and steep learning curve. In order to successfully deploy a service, developers must first specify all of its configuration parameters using a predefined schema. Furthermore, if the service requires features like configuration files, volume mounts, or even inter-service communication capabilities, all these must be declared as separate Kubernetes resources that each have their own schema.

Given that smart city projects are frequently built by a consortium of numerous different partners such as companies, research institutions, and other entities, it would be unreasonable to expect every stakeholder to learn all the intricacies of Kubernetes before being able to deploy a simple service to the infrastructure. A possible solution would be for developers to hand off the deployment of their services to a specialised deployment/infrastructure team. However, this approach would introduce inefficiencies in regards to inter-team communication, due to the fact that the developer team would not be aware of deployment restrictions while developing the service, and the deployment team would not be aware of all of the service's intricacies while deploying it.

The second issue arises from the fact that the additions proposed in this dissertation would require configuration parameters that are not supported by the current Kubernetes API.

Instead, the proposed architecture sets out to solve both problems simultaneously by creating a new type of service descriptor file that abstracts the usual Kubernetes complexity behind a single file that supports both the most commonly used Kubernetes configurations, and all the additions required by the new features implemented in this dissertation. To deploy most services, developers only need to learn and use this single descriptor file, which follows a custom schema that is also registered in the Kubernetes instance, so that its API can know what to expect, and therefore reject any mis-specified deployments. The descriptor file is presented in Annex 7.1.

5.4.3 FogService Operator

Kubernetes is only capable of natively orchestrating the deployments of built-in resources such as Pods, Volumes, Config Maps, etc. Therefore, when a custom resource (i.e. FogService) is specified and declared by a developer, the orchestrator requires custom domain-specific logic in order to interpret the schema and actually perform the corresponding actions. To that end, the proposed architecture includes the FogService Controller, which implements the Kubernetes Operator Pattern discussed in chapter 2 in order to watch and orchestrate FogService resources. When a new FogService resource is declared, the controller parses each of its sections and autonomously creates and declares all the different built-in Kubernetes resources that the service requires, thus allowing Kubernetes to proceed with the orchestration of the deployment, and fulfilling FogService's role as an abstraction layer for the default resources.

In cloud-based cluster environments, the existence of multiple worker nodes is primarily used to provide horizontal scaling (and, thus, high-availability, load-balancing, and increased performance) to generic applications. In some specific cases, the worker nodes can also be geographically dispersed, in order to increase the proximity of end clients to an available replica of a given service. Therefore, the replication controllers used by Kubernetes' native resources are optimised for the requirements of these specific use cases.

As previously discussed, edge-computing clusters are also capable of fulfilling these use cases, but normally have the advantage of providing even greater proximity between an application and its clients. However, in the context of a Smart City, each node typically has the possibility of being equipped with a set of sensors and communication technologies that allow it to collect and process data relevant to its specific location, and interface with the real world. As such, these types of clusters can also implement a slightly different usage pattern where a given service is deployed to several different nodes, but each one is conceptually treated as a distinct stateful application whose scope of operation is limited to that specific location.

Examples of this type of usage pattern include, for instance:

- An object detection service using video frames obtained in real-time from an Internet Protocol (IP)-enabled camera.
- A local database that records all of the data points generated at a given location, so that other services can access them.
- A vehicular network Road-Side Unit service using *Vanetza-NAP* to send and receive ITS-G5 messages.

The examples constitute services that are meant to be deployed to multiple distinct locations/nodes, but do not fit the archetypical pattern of being replicas of a single cluster-scoped application. Instead, their deployment is implemented as a set of location-scoped applications that utilise location-specific resources and produce/consume location-specific data. Following this approach, each location-scoped service can then have its own custom configuration, which is important since a given application may require different configuration options depending on the location it is allocated to, as each worker node can typically have a distinct set of sensors and/or communication interfaces that best suit its specific location and use cases. Furthermore, a given location-scoped service can also have its own set of replicas which keep the original's scope. This enables the operator to perform differential scaling of replicas on each location-scoped deployment, allowing it to react to higher utilisation levels or performance issues in a specific location. Notably, these replica Pods can potentially be scheduled on different nodes due to scheduling constraints, while still keeping their original location scope (provided that the application does not require access to specialised resources).

In order to support a wider range of possible cluster topologies, this implementation assumes that each location can be composed of multiple nodes, thus forming what is known in Kubernetes' terminology as a Topology Zone. Therefore, services that follow this paradigm should be deployed as set of zone-scoped applications, which would preferably be allocated to one of the nodes that comprise the Zone.

Since Kubernetes does not natively implement this pattern, the necessary logic must be instead included in the FogService Operator. The controller also reacts when an existing FogService is updated by the developer, in which case it re-computes the underlying Kubernetes built-in resources and determines if any of them needs to be updated from its current state.

Additionally, the operator continuously monitors the state of the existing FogService resources by periodically collecting the current value of the declared application-level metrics in each replica, and storing them within the Status section of the respective FogService resource. This approach greatly reduces the network congestion by ensuring that only the controller performs expensive queries for metric values; other components that require that information can access the summarised version by performing a single Kubernetes API request to get the aforementioned Status field of the FogService resource.

The following subsections describe the various Kubernetes resources that are created by the Operator, and their respective functions.

StatefulSet

This constitutes the most crucial resource created by the operator, since it is responsible for the actual deployment of the Pod(s), whereas the remaining resources are mostly auxiliary in nature. Kubernetes provides several different types of built-in resources to accomplish this, known collectively as Workload Resources. The two most suitable for the purposes of this operator are Deployments and StatefulSets, which differ mainly in their approach to Pod replication. The Deployment resource is aimed for stateless applications where every replica is identical in its function and operation, and shares the same persistent data volume. In contrast, the StatefulSet resource treats each replica as a separate entity that receives a predictable name and Domain Name System (DNS) domain, as well as an individual persistent data volume. As such, it is typically used for more complex stateful applications, such as database management systems, which require advanced processes such as leadership elections and automatic fail-over.

In this implementation, the FogService Operator uses the StatefulSet as the underlying workload resource in order to ensure that developers have the freedom to implement their applications in either a stateless or stateful manner, without being restricted by the framework itself.

To accomplish the aforementioned zone-scoped deployments, the Operator first evaluates the list of desired Zones for the deployment of a given application, via the respective field in the FogService specification. In the case that no Zones are specified, the application is treated as a normal cluster-scoped service, which requires the creation of a single StatefulSet resource, as previously described. However, if one or more Zones are configured, the controller creates a distinct StatefulSet resource for each Zone, whose name is derived from the FogService's name with the addition of the Zone as a prefix. Each StatefulSet also includes NodeAffinity rules that instruct to allocate its Pods to the respective Zone, either on a preferential or mandatory basis, depending on if the application requires the use of specialised hardware (as configured in the FogService specification).

This resource is also used to configure other features defined in the FogService schema, such as Resource Requests and Limits, Liveness Checks, Security Context, Environment Variables, among others.

Service

This resource enables external clients and other applications in the cluster to communicate with a given Zone-scoped application, through the exposed TCP and/or UDP ports that were specified in the respective FogService resource. Conceptually, the Service operates as the single point of entry for any network requests destined for the application, and balances their load between existing replicas.

Since this resource is also Zone-scoped, the Operator creates a separate one for each generated StatefulSet.

ConfigMap

This resource includes the contents of all of the configuration files that the application requires in order to function correctly in the given zone, which are sourced from the relevant fields in the FogService specification.

Since this resource is also Zone-scoped, the Operator creates a separate one for each generated StatefulSet.

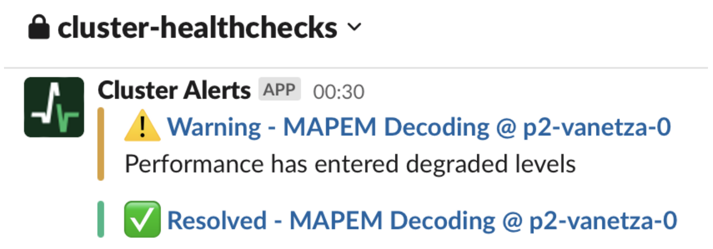
ServiceMonitor

This resource configures the Operated Prometheus stack described in section 5.3.4 to periodically pull exposed metric values from the given Zone-scoped application via the port configured on the respective FogService specification. This allows the centralised metrics server to collect information on the application's performance and provide it to the Operator and other components of the framework described in this chapter.

Since this resource references the Service resource, which is Zone-scoped, the Operator creates a separate one for each generated StatefulSet.

PrometheusRule

This resource configures the Operated Prometheus stack described in section 5.3.4 (namely, the AlertManager component) to automatically trigger alerts when the application's metrics reach the degradation thresholds specified in the respective FogService resource. Within the scope of this dissertation, the Prometheus stack is configured to also send these alerts to a predefined Slack channel, so that the cluster administrators and application developers can be kept informed of the state of the cluster and its deployments.


cluster-healthchecks

Cluster Alerts APP 00:30

⚠️ Warning - MAPEM Decoding @ p2-vanetza-0
Performance has entered degraded levels

✅ Resolved - MAPEM Decoding @ p2-vanetza-0

Figure 5.2: Example of a degraded performance alert observed in the replica number 0 of the Vanetza FogService in the P2 Zone

Since this resource applies to the FogService as a whole, the Operator only creates one instance.

HorizontalPodAutoscaler

This resource enables Kubernetes' HorizontalPodAutoscaling mechanism and configures it to connect to the Prometheus Metric Server and take into account the custom application-level metrics defined in the respective FogService specification. This allows Kubernetes to continuously monitor the average performance of a given Zone-scoped service across all of its active replicas, and automatically scale the number of Pods either up or down when the metric values begin to stray from the target set by the developers in the FogService schema. Additionally, the HorizontalPodAutoscaler establishes upper and lower bounds for the total number of replicas that the application can have, as configured in FogService.

Since this resource is also Zone-scoped, the Operator creates a separate one for each generated StatefulSet.

PodDisruptionBudget

This resource configures Kubernetes Eviction API constraints for the minimum available and/or maximum unavailable number (or percentage) of replicas that the Zone-scoped application must have at a given time, as configured in the respective FogService specification. This configuration effectively establishes a buffer of replicas that cannot be simultaneously evicted, thus safeguarding the availability of the service. Within the scope of this dissertation, this effect is important, since it curtails any potentially excessive corrections from the Cluster State Monitor component of this implementation, described later in this chapter.

Since this resource is also Zone-scoped, the Operator creates a separate one for each generated StatefulSet.

5.4.4 Pod Scheduler Plugins

In order to bring extensibility to the scheduler, the Deployment Scheduler module can be developed as a set of plugins to the existing scheduler instead of as a separate component.

One possibility would be to build an entirely new, bespoke, scheduler designed with RT-awareness and the other requirements in mind. This approach would allow for near-complete control of scheduling decisions, without having to conform with any pre-established Kubernetes scheduling paradigms. However, it is important to consider that the default Kubernetes scheduler is a mature framework built over the course of several years and with a diverse set of features and advanced algorithms, which the new scheduler would not be able to use. Re-implementing this base feature-set would not be worth the cost in the vast majority of scenarios. On the other side, using a scheduler without those features would break several Kubernetes patterns and behaviours that developers and infrastructure specialists expect, especially in a context such as this, where multiple partner entities could be expected to make deployments.

With that in mind, the custom scheduler proposed in the previous section was instead implemented as a set of four plugins which augment the default scheduling framework with new capabilities, while keeping all the existing tried-and-true features that developers expect. These plugins are registered as Filter, PostFilter, and/or Score extension points within the Pod scheduling context, as discussed in chapter 2.

NodeMetrics

By default, Kubernetes scheduling attempts to minimize the load on each node by taking into account its current resource consumption and the resource requests specified by developers on prospective Pods. However instead of obtaining the current load from the node's operating system, Kubernetes computes it as a function of the resource requests of all the currently deployed Pods on that node. This means that the scheduler is unaware of any external factors that may be taxing the CPU and/or memory such as OS tasks, daemons, and even services that are deployed manually or by means other than Kubernetes.

While this may not pose a problem in cloud environments where each Kubernetes worker runs on a purpose-built VM, it can become an issue in Edge Computing and Smart City environments where worker nodes are often not virtualised or of single purpose.

To address this issue, the *NodeMetrics* plugin is registered as a Score extension point and ranks nodes by their CPU and RAM utilisation scores as measured by the operating system itself, preferring nodes with lower utilisation. As discussed previously, the metrics are collected by Prometheus exporters placed in each of the nodes as Pods, and periodically polled by the centralised Prometheus metric server. In order to access the most current values, the plugin issues PromQL queries each time a new Pod triggers Kubernetes' scheduling pipeline. The algorithm for the node metrics sorting is depicted in algorithm 1.

Each node's score is calculated as the weighted sum of the current utilisation of the CPU and RAM resources, as a percentage of their total installed amount. By default, the CPU and RAM resources are seen as equally important in determining the best node for a given Pod and, as such, both weights are set at 50%. Nevertheless, these values are fully configurable by the system administrators of each cluster that makes use of the plugin. The choice of the weighted sum algorithm is also a future-proofing effort, since it simplifies the inclusion of other metrics with varying levels of relevance.

While implementing the plugin, one of the main design considerations was the trade-off between ensuring an acceptable response time to changing conditions, and simultaneously avoiding potential overreactions to ephemeral anomalies. Therefore, the plugin uses the 5 minute average of each of the metrics instead of their most recent value. This was deemed an acceptable compromise to ensure that the system responds mostly to sustained trends as opposed to sporadic anomalies.

Algorithm 1 NodeMetrics Sorting

```
function SCORE($Pod, Node$)
    $cpu \leftarrow$ GETCPUUTILISATION(Node, 5mins)
    $ram\_a \leftarrow$ GETRAMAVAILABLE(Node, 5mins)
    $ram\_t \leftarrow$ GETRAMTOTAL(Node)
    $ram \leftarrow (ram\_t - ram\_a)/ram\_t$
    $score \leftarrow (cpu * 0.5) + (ram * 0.5)$
    return $(1 - score) * 100$
end function
function NORMALIZESCORE($Pod, Scores$)
    $highestScore \leftarrow 0$
    for each score in Scores do
        if $score > highestScore$ then
            $highestScore \leftarrow score$
        end if
    end for
    for each score in Scores do
        $score \leftarrow (MaxPossibleScore * score)/highestScore$
    end for
end function
```

Realtime

Given that the default Kubernetes scheduler was not designed to take a Pod's Realtime attributes into account, Realtime Pods are scheduled using the exact same criteria as any other Pod. In terms of resource balancing, this means that Pods are preferably allocated to the node with the smallest percentage of resources claimed by existing Pods, and no effort is made to minimise the CPU utilisation of Realtime processes. As a result, some nodes will inevitably exhibit a higher number of Realtime Pods than others. However, since developers are expected to include the CPU usage of Realtime processes in their Pods' overall resource requests, this imbalance will not prevent the algorithm from successfully maintaining similar levels of resource usage across all of the cluster's nodes.

Nevertheless, as discussed in chapter 4, the analysis of experimental results reveals that Realtime priority tasks can have a significant impact on the performance of regular processes even when there is CPU headroom available, by virtue of their precedence in terms of CPU time allocation, and the preemption mechanism that interrupts other tasks. The same observations also apply to Realtime Pods themselves, in cases where one or more higher-priority Pods exist.

To address these issues, this architecture includes the *Realtime* scheduler plugin, which implements the Score extension point in order to rank candidate nodes by their overall share of CPU capacity being allocated to Realtime tasks. To accomplish this, the first design challenge is on how to accurately determine this value. The RT utilisation of a Pod is not necessarily equal to its total CPU resource requests, since Pods can spawn a mix of both Realtime and regular processes. Instead, the value must be obtained and/or calculated from other fields included in each process section of the FogService specification.

In the case of RT processes that use the SCHED_DEADLINE scheduler, it is possible to derive their CPU utilisation per unit of time directly from the ratio between their *runtime* and *period* attributes, in accordance with the documentation for the Linux kernel's implementation of the Earliest Deadline First scheduling algorithm:

$$U = \sum_{i=1}^{n} \frac{runtime_i}{period_i}$$

where U represents the overall utilisation and n represents the number of SCHED_DEADLINE processes running on the worker node.

In contrast, RT processes that use the SCHED_FIFO scheduler only require a priority attribute that does not convey any information regarding its resource usage. To address this, the FogService schema includes an extra field where developers can specify the amount of CPU allocation that the process requires, in the same format as a typical CPU resource request (*requests*). The plugin, with its algorithm presented in algorithm 2, determines a node's Realtime CPU utilisation by performing Kubernetes API queries to obtain the FogService resource associated with each of the Pods running on it, and adding up the CPU time that is requested by all of their k RT processes, following an adapted version of the previous formula:

$$U = \sum_{i=1}^{n} \frac{runtime_i}{period_i} + \sum_{i=1}^{k} requests_i$$

where k represents the number of SCHED_FIFO processes running on the worker node.

This value can then be used to allocate a score to the node, thus fulfilling the plugin's main objective. However, while this approach successfully minimizes the overall Realtime interference on each node, it does not address the interference that higher-priority RT tasks have on their lower-priority counterparts. As such, when the prospective Pod contains processes with SCHED_FIFO attributes, the plugin also calculates the combined utilisation of all the existing processes that have a higher priority in the Linux kernel's schedulers. This includes any SCHED_FIFO task with a higher priority and all of the SCHED_DEADLINE tasks that may exist, since the EDF scheduler assumes precedence over all others. When scheduling new SCHED_DEADLINE workloads, this algorithm includes the utilisation of all of the existing SCHED_DEADLINE tasks, given that they lack discrete priority levels which would allow for a direct comparison. In fact, any SCHED_DEADLINE task can potentially become a source of interference for any other tasks, depending on runtime circumstances that are not trivial to predict at the Kubernetes Scheduler level, especially for multi-core systems.

By including this information in the process of determining a final node score, the *Realtime* plugin becomes capable of also favouring nodes that have fewer or less-demanding higher-priority processes, thus reducing interference caused by other RT Pods. Nevertheless, when considered in isolation, this strategy has the potential to re-introduce undesirable imbalanced situations where a large amount of RT Pods with identical priorities are allocated to the same node in order to avoid nodes with a higher priority Pod. Therefore, for the purposes of this dissertation, this approach is deemed as much less valuable than the main objective of ensuring balanced RT utilisation levels. As such, the final score is calculated as the weighted sum of the results of each strategy, where the former is allocated a weight of 80%, and the latter 20%. These weights are considered as variables in the algorithm, $RTUtilizationWeight$, so that they can be configured to optimize the process of final score calculation. This ensures that the latter strategy is used essentially as a tie breaker in situations where two nodes have similar overall RT utilisation, greatly reducing its harmful potential while retaining some of its advantageous behaviour.

One potential drawback of this plugin is that it may result in situations where a Realtime Pod and its Realtime dependencies are separated on different nodes, which would have an adverse effect on the performance of that service due to increased communication latency. However, this is deemed to be an acceptable compromise, since maximizing the performance of any single workload is not generally worth severely jeopardising other services and the overall cluster balance. Nevertheless, this behaviour can still be tweaked by the system administrators of each production environment using the scheduling weights of the *Realtime* and *Dependencies* plugins.

Finally, this plugin also addresses the filtering requirement mentioned in section 5.2, since it is important to ensure that the compute resources of the cluster's worker nodes do not become over-allocated. By default, the Kubernetes scheduler accomplishes this by calculating the total amount of resources requested by the various workloads running on a given node and comparing it to that node's available capacity. Any node that does not possess sufficient available capacity to accommodate the prospective Pod's resource requests is automatically excluded from consideration in the scoring process.

However, it is important to note that the available capacity for regular processes and RT-enabled processes are not necessarily identical. In fact, Linux systems have the option to set a limit on the quota of the overall processing time that can be allocated to Realtime tasks, which is typically set at 95% by default. This is intended as a safeguard to ensure that RT-enabled processes cannot completely starve out critical Kernel and Operating System functions.

Therefore, by itself, Kubernetes' default approach is not entirely suitable in this context, since it could result in the allocation of a Realtime Pod to a node with sufficient overall capacity but insufficient Realtime capacity. Such an allocation would seriously jeopardise the ability of lower-priority RT Pods to complete their tasks within the expected time frames. In fact, some RT processes could actually assume less precedence over the allocation of CPU time than some of the system's regular tasks. Furthermore, in cases where both the prospective Pod and the existing Realtime Pods consist of SCHED_DEADLINE processes, the Linux kernel would detect the over-allocation and automatically reject the allocation of RT parameters to the prospective Pod's processes. In this scenario, the affected processes would continue to run as regular CFS-scheduled tasks, which could potentially have catastrophic effects on their performance.

To address this issue, the Realtime plugin implements the Filter extension point in order to perform a utilisation analysis of each node and preemptively exclude any nodes where the prospective Realtime Pod would exceed the remaining capacity for RT-enabled tasks. The aforementioned Realtime utilisation quota is set on a per-node basis and, as such, may not be identical for every node, especially in heterogeneous clusters with distinct hardware constraints. However, Kubernetes does not provide any native mechanism for the control plane to obtain each node's configured quota. For this reason, the implementation presented in this dissertation requires each worker node to include the values of its *sched_rt_period_us* and *sched_rt_runtime_us* kernel parameters as labels within the node resource.

This algorithm, presented in algorithm 3, can also be presented in mathematical terms, using the following formula from the Linux kernel documentation as a starting point:

$$\sum_{i=1}^{n} \frac{runtime_i}{period_i} \leq M * G$$

where n represents the number of SCHED_DEADLINE processes running on the node, M represents the number of processor cores installed on the host, and G the aforementioned maximum quota of processor time dedicated to Realtime tasks.

This inequation represents the condition that the Linux Kernel enforces when new processes attempt to register SCHED_DEADLINE parameters. As discussed earlier in this section, it can be extended with the sum of all the CPU requests made for SCHED_FIFO processes, as well as the sum of requests made by the Realtime tasks of the prospective Pod. Therefore, the final formula can be written as:

$$\sum_{i=1}^{n} \frac{runtime_i}{period_i} + \sum_{i=1}^{k} requests_i + \sum_{i=1}^{j} \frac{runtime_i}{period_i} + \sum_{i=1}^{w} requests_i \leq M * \frac{sched\_rt\_runtime\_us}{sched\_rt\_period\_us}$$

where k represents the number of SCHED_FIFO processes running on the worker node, j represents the number of SCHED_DEADLINE processes spawned by the prospective Pod, and w represents the number of SCHED_FIFO processes spawned by the prospective Pod.

In extreme scenarios where there are no available nodes with sufficient resources for a given candidate Pod, the plugin will remove every node from consideration, thus marking the Pod as *Unschedulable* even if there are lower-priority RT Pods currently running. To avoid this, the plugin also implements the PostFilter extension point in order to perform a preemption analysis of all the worker nodes and evict the lowest priority Pod(s) that will free sufficient resources for the allocation of the higher-priority candidate Pod.

Algorithm 2 Realtime Scoring

```
$RTUtilisationWeight \leftarrow$ GetRTWeightFromPluginConfig()
$HigherPriorityRTUtilisationWeight \leftarrow$ GetHigherRTWeightFromPluginConfig()
function GetRTUtilisationForNode($Pod, Node$)
    $existingPods \leftarrow$ GetPodsInNode(Node)
    $nodeRTUtilisation, nodeHigherPriorityRTUtilisation \leftarrow 0$
    $candidateFogService \leftarrow$ GetFogServiceForPod(Pod)
    $candidatePriority \leftarrow$ GetPriorityForFogService(candidateFogService)
    for each existingPod in existingPods do
        $podRTUtilisation, podHigherPriorityRTUtilisation \leftarrow 0$
        $existingFogService \leftarrow$ GetFogServiceForPod(existingPod)
        $existingPodPriority \leftarrow$ GetPriorityForFogService(existingFogService)
        $realtimeProcesses \leftarrow$ GetRTProcessesForFogService(existingFogService)
        for each realtimeProcess in realtimeProcesses do
            if isSCHED_FIFO(realtimeProcess) then
                $podRTUtilisation \leftarrow podRTUtilisation + realtimeProcess.CPURequests$
                if $existingPodPriority > candidatePriority$ then
                    $podHigherPriorityRTUtilisation \leftarrow podHigherPriorityRTUtilisation + realtimeProcess.CPURequests$
                end if
            else if isSCHED_DEADLINE(realtimeProcess) then
                $processRTUtilisation \leftarrow realtimeProcess.runtime/realtimeProcess.period$
                $podRTUtilisation \leftarrow podRTUtilisation + processRTUtilisation$
                $podHigherPriorityRTUtilisation \leftarrow podHigherPriorityRTUtilisation + processRTUtilisation$
            end if
        end for
        $nodeRTUtilisation \leftarrow nodeRTUtilisation + podRTUtilisation$
        $nodeHigherPriorityRTUtilisation \leftarrow nodeHigherPriorityRTUtilisation + podHigherPriorityRTUtilisation$
    end for
    return $(nodeRTUtilisation, nodeHigherPriorityRTUtilisation)$
end function
function Score($Pod, Node$)
    $nodeResults \leftarrow$ GetRTUtilisationForNode(Pod, Node)
    return $(nodeResults.RTUtilisation * RTUtilisationWeight) + (nodeResults.HigherPriorityRTUtilisation * HigherPriorityRTUtilisationWeight)$
end function
function NormalizeScore($Pod, Scores$)
    $highestScore \leftarrow 0$
    for each score in Scores do
        if $score > highestScore$ then
            $highestScore \leftarrow score$
        end if
    end for
    for each score in Scores do
        $score \leftarrow (MaxPossibleScore * score)/highestScore$
    end for
end function
```

Algorithm 3 Realtime Filtering

```
function FILTER($Pod, Node$)
    $nodeResults \leftarrow$ GETRTUTILISATIONFORNODE(Pod, Node)
    $nodeMaximumRTUtilisation \leftarrow$ GETMAXUTILISATIONFORNODE(Node)
    $PodRTUtilisation \leftarrow 0$
    $candidateFogService \leftarrow$ GETFOGSERVICEFORPOD(Pod)
    $realtimeProcesses \leftarrow$ GETRTPROCESSESFORFOGSERVICE(candidateFogService)
    for each realtimeProcess in realtimeProcesses do
        if ISSCHED_FIFO(realtimeProcess) then
            $podRTUtilisation \leftarrow podRTUtilisation + realtimeProcess.CPURequests$
        else if ISSCHED_DEADLINE(realtimeProcess) then
            $processRTUtilisation \leftarrow realtimeProcess.runtime/realtimeProcess.period$
            $podRTUtilisation \leftarrow podRTUtilisation + processRTUtilisation$
        end if
    end for
    $finalNodeRTUtilisation \leftarrow nodeResults.RTUtilisation + podRTUtilisation$
    if $finalNodeRTUtilisation <= nodeMaximumRTUtilisation$ then
        return $Schedulable$
    end if
    return $Unschedulable$
end function
```

Dependencies

Kubernetes includes native support to establish a dependency relation between two services by including a PodAffinity section within the deployment resource. When this relation exists, the scheduler automatically favours nodes that possess a replica of the service on which the prospective Pod depends. In cases where this is not possible, it attempts to allocate the Pod to a node belonging to the same topology zone as nodes where dependencies are located, thus leveraging the fact that zones frequently represent sets of nodes in geographical proximity, which should therefore be able to communicate with lower latency.

By favouring the exact nodes where dependencies are running or, when this fails, nodes within the same topology zones, this PodAffinity support should, in most cases, minimize communication latency between them. Nevertheless, this dissertation identifies two potential avenues of improvement, namely:

- In situations where it is not possible to allocate a Pod to any node within the same topology zone as a dependency, Kubernetes will not be able to make any further attempts to minimize latency, as it will lack the context and data necessary to do so. In this scenario, the nodes will be ranked based on all the other considerations taken into account by the scheduler's default plugins. From the standpoint of achieving closeness to dependencies, the chosen node will be, in effect, arbitrary. Ideally, the scheduler should still be able to prioritise nodes that demonstrate a lower communication latency to the node running the dependency, despite not belonging to the same topology zone.
- In scenarios where there are multiple replicas of the same dependency running on separate nodes, the scheduler should also be able to account for differences in performance, as

discussed in section 5.2. Ideally, Pods should be scheduled as close as possible to the least-used and/or best-performing replicas of a dependency (as determined by one or more predefined metrics), instead of any arbitrary replica.

To that end, this plugin implements the Score extension point and ranks nodes by their proximity (in latency terms) to replicas of a service identified as a dependency of the prospective Pod, and also by the current values of those replica's application-level metrics.

The customised load-balancing strategy described later in this section further amplifies the positive effects of this approach, since it ensures that the network traffic exchanged between the Pod and the dependency also favours the replicas that present the least latency and best metric values.

As can be seen in algorithm 4, the plugin determines the individual scores of the candidate worker nodes by first considering the quality of a potential link between them and each existing replica of the Pod's dependencies. In accordance with what was presented in algorithm 1, the current value of the various metrics exposed by the application is pre-computed and continuously updated by the FogService Operator into a single normalised value for each replica, that represents their respective performance. To obtain these values, the plugin performs Kubernetes API queries for the Status section of each dependency's FogService resource.

This type of pre-processing pipeline only exists for the exposed metrics, and does not apply to information provided by the latency monitor component. Instead, the plugin performs PromQL queries to the centralised Prometheus Metric Server, which replies with the current absolute latency values for each link, expressed in milliseconds. That information is then normalised in a similar fashion to the metric data, so that each node's latency score is relative to the node that exhibits the lowest latency, which receives a score of 100.

Since different applications exhibit distinct behaviours and bottlenecks, some dependencies may benefit from scheduling decisions that favour application metrics over the network latency, or vice-versa. To accomplish this, the developers of each dependency can use the FogService specification schema to define the weight that should be allocated to each of the two factors by this plugin. As such, the heuristic used by the algorithm to define the quality of a potential link between the prospective node and a given replica consists of a weighted sum between the replica's normalised metric score and the normalised latency score. This scoring-based approach also helps to ensure that the plugin can be easily extended in the future to account for additional factors.

Before the final node score can be calculated, the scores of each dependency's replicas are aggregated into a single value that represents the overall quality of the communication experience between the prospective Pod and the dependency, on the node being considered. This calculation could be feasibly accomplished using a simple heuristic like an average of its replicas' scores. However, such an approach would not be an accurate representation of real-world behaviour since, in this proposed architecture, the customised load-balancer implementation ensures that the highest ranked replicas receive the majority of requests.

As will be discussed later in subsection 5.4.7, the behaviour of this load-balancing algorithm is based on a Markov chain where each state represents a candidate replica, and the state transition probabilities reflect the differences in metric values and link latency. Each time a new network request is received, the load balancer uses the chain to select a suitable destination replica. Therefore, in order to provide the best possible approximation of real-world behaviour, the *Dependencies* plugin builds its own representation of this Markov chain's probability matrix for every dependency, and then calculates the average percentage of time spent in each state (through the respective stationary distribution vector), which effectively represents the percentage of requests that the load-balancer will send to each replica based on the current conditions. By combining this information with the quality score allocated to each replica/state in the previous step, the algorithm can accurately determine the average quality score experienced in a given node's communications with a dependency. This calculation is performed as the weighted average of the previously obtained scores, where the relative percentages of requests are used as weights.

One drawback of this approach is that it restricts the use of this plugin exclusively to clusters that include the aforementioned load balancer. This constraint was deemed acceptable, given that the plugin's usefulness is greatly reduced when paired with other load balancer implementations that do not favour sending network traffic to the closest replicas.

Up until this point, each of the dependencies associated with a given service has been treated as equally important to the others. However, in real scenarios, some dependencies can have a larger impact on the performance of an application (or represent a more significant bottleneck to its operation) than others. For that reason, developers also have the ability to specify a weight for each of the dependencies defined in the FogService resource. Using that information, the final node score is calculated as the weighted average of the various dependency scores.

While evaluating the suitability of this plugin's design for use in production environments, one of the main concerns that became apparent was the risk of over-allocation of Pods to nodes with lower-latency network links or in close proximity to particularly well-performing replicas. Such a scenario would, naturally, result in severe imbalances in resource usage throughout the different worker nodes, and reduced overall performance.

However, it is important to keep in mind the following factors:

- As previously mentioned, the Kubernetes Scheduler runs all of its configured plugins concurrently, and performs a weighed average of their individual scores in order to select the best node for a given Pod. Therefore, the use of the *Dependencies* plugin does not preclude the default workload balancing logic included in Kubernetes. As such, any tendencies for over-population of a given node introduced by this plugin will eventually be counterbalanced by Kubernetes' balancing plugins, especially when the Pod's specification includes resource requests. By tweaking the scheduling weight of the *Dependencies* plugin, system administrators can configure their ideal balance between maximizing performance and maintaining cluster balance.
- If an excessive amount of Pods is scheduled in close proximity to a particularly well-

performing replica of a dependency, the metrics exposed by both the nodes and the replica will most likely be adversely affected. This will, in turn, affect the scores generated by the *NodeMetrics* and *Dependencies* plugins, discouraging similar allocations in the future. Furthermore, this effect is also acted upon by other components of this architecture. The customised load-balancing algorithm will shift network traffic to less-used replicas and the cluster state monitoring system will migrate some of the Pods to other worker nodes. Additionally, if the average values of the dependency's metrics degrade beyond the specified expected operating levels, the Horizontal Auto-scaling mechanism provided by Kubernetes and configured in the respective FogService resource will automatically spawn more replicas. As a consequence, all of the aforementioned mitigation mechanisms will be reinforced, since the newly-created replicas provide additional targets for future allocations, network traffic load-balancing, and the migration of existing Pods.

Given these attenuating factors, this drawback was deemed to be adequately addressed.

Algorithm 4 Dependencies Scoring

```
function NormalizeScore($Pod, Nodes$)
    $fogService \leftarrow$ GetFogServiceForPod(Pod)
    $dependencies \leftarrow$ GetNormalisedDependencyMetrics(fogService)
    $dependencyWeights \leftarrow$ GetDependencyWeightsForFogService(fogService)
    $nodeScores \leftarrow \{\}$
    for each node in Nodes do
        $nodeScore \leftarrow 0$
        $latencies \leftarrow$ GetNormalisedLatencies(Node, dependencies)
        if isEmpty(dependencies) then
            $nodeScore \leftarrow 1$
        else
            for each dependency in dependencies do
                $dependencyFogService \leftarrow$ GetFogServiceForDependency(dependency)
                $dependencyLatencyWeight \leftarrow$ GetLatencyWeight(dependencyFogService)
                $dependencyMetricsWeight \leftarrow$ GetMetricsWeight(dependencyFogService)
                $scores \leftarrow []$
                for each dependencyPod in dependencyFogService do
                    $podScore \leftarrow (latencies[node][dependencyPod] * dependencyLatencyWeight) + (dependencies[dependencyPod] * dependencyMetricsWeight)$
                    $scores \leftarrow$ append$(scores, podScore)$
                end for
                $matrix \leftarrow$ GetMarkovProbabilityMatrix$(scores, dependencyFogService)$
                $distributionVector \leftarrow$ GetDistributionVectorForMarkovChain$(matrix)$
                $weightedProbabilities \leftarrow$ matrixMultiplication$(scores, distributionVector)$
                $averageScoreValue \leftarrow$ matrixAddition$(weightedProbabilities)$
                $nodeScore \leftarrow nodeScore + (averageScoreValue * dependencyWeights[dependency])$
            end for
        end if
        $nodeScores[node] \leftarrow nodeScore$
    end for

    $highestScore \leftarrow 0$
    for each score in nodeScores do
        if $score > highestScore$ then
            $highestScore \leftarrow score$
        end if
    end for
    for each score in nodeScores do
        $score \leftarrow (MaxPossibleScore * score)/highestScore$
    end for
    return $nodeScores$
end function
```

DegradedPerformance

This plugin is designed to aid in the recovery of Pods that enter a degraded state by preventing the allocation of new sources of interference to the affected worker node. To accomplish this, the plugin registers as a Filter extension point, and evaluates each available node to determine if any of its existing Pods are currently marked by the FogService Operator as having degraded levels of performance. When that is the case, the degraded performance filtering algorithm, algorithm 5, only marks the node as suitable if the prospective Pod has a higher priority than all of the degraded Pods. Otherwise, the application is prevented from being allocated to that node.

As such, the plugin effectively quarantines worker nodes that are currently experiencing problems so that they are afforded the best chance of recovering rapidly. It also supports efforts by other components with similar objectives that will be discussed in later sections.

Considered in a vacuum, this strategy is problematic due to the fact that, while the degraded state persists, lower-priority Pods may be allocated to sub-optimal nodes that do not promote the best possible application performance and/or cluster balance. This produces a long lasting adverse effect on the cluster, since the Pods remain on their respective nodes until they are manually removed or the node itself suffers an outage. Nevertheless, when the plugin is evaluated in conjunction with the overall architecture presented in this chapter, this issue is fully addressed by the Cluster State Monitor, which starts migrating Pods to their preferred node as soon as the degraded state is resolved. This tool is discussed in further detail in the next subsection.

The basic effects of this *DegradedPerformance* plugin could conceivably be attained using just the *NodeMetrics* plugin, since any undue increase in CPU and memory usage would lead the scheduler to preferably allocate Pods to other nodes. However, it is important to consider that applications can expose a vast set of metrics, and not all degraded states will directly correlate with increased CPU and memory usage. Furthermore, some applications may not have other nodes that satisfy their geographical location requirements or complex scheduling constraints. In these cases, being allocated a low score by *NodeMetrics* will not prevent the node from being selected by the scheduler, given that it is the only option available.

Finally, this implementation also presents some noteworthy drawbacks. Firstly, developers must consider the metrics that their applications expose and the respective degradation thresholds very judiciously, so as to prevent spurious incidents that cause unduly interference on the cluster. The aforementioned alerting mechanisms can aid in that tuning process, since developers are immediately notified when incidents occur, and can then verify if the situation is truly problematic or if the detection threshold should be adjusted. Secondly, the plugin may cause situations in which one or more low-priority Pods are unable to be scheduled, and therefore remain in that unscheduled state until the degradation incident is resolved. This happens in cases where some low-priority Pods, by virtue of their complex scheduling constraints, can only be scheduled to the specific node that hosts the degraded higher-priority application.

Algorithm 5 DegradedPerformance Filtering

```
function FILTER($Pod, Node$)
    $newFogService$ ← GETFOGSERVICEFORPOD(Pod)
    $newPriority$ ← GETPRIORITY(newFogService)
    $existingPods$ ← GETPODSINNODE(Node)
    for each existingPod in existingPods do
        $existingFogService$ ← GETFOGSERVICEFORPOD(existingPod)
        $existingPriority$ ← GETPRIORITY(existingFogService)
        $degraded$ ← ISDEGRADED(existingFogService, existingPod)
        if $degraded$ AND $existingPriority > newPriority$ then
            return $Unschedulable$
        end if
    end for
    return $Schedulable$
end function
```

5.4.5 Cluster State Monitor

As discussed in 5.3, continuously monitoring the current cluster state is required in order to ensure that the allocations remain optimal, which may require the re-scheduling of active Pods into the most suitable node.

This problem becomes especially relevant within the context of this implementation, as scheduling decisions are influenced by, among others, communication latency and application performance, which constitute notoriously volatile metrics. After the initial deployment of a Pod, as time passes, the conditions that lead to the allocation of the corresponding node will inevitably change, potentially leaving the cluster in a sub-optimal, or even degraded, state for long periods of time.

As an attempt to address some of these issues, Kubernetes' Special Interest Groups initiative introduced an open-source Golang project named Descheduler[1], that is currently still in active development. This tool integrates with the Kubernetes API to periodically iterate through every Pod in the cluster and determine which Pods should be moved from their respective worker nodes, if any. This determination can take into account several factors, depending on configurations made by the cluster administrators. These include, for instance, attempting to minimize the resource usage of each worker node, and detecting Pod allocations that violate node affinity, Pod anti-affinity, or topology spread constraints.

When a Pod is deemed to not be running in the most optimal node, the Descheduler, as implemented, does not allocate it to a new node. Instead, the process is handled through the Eviction API, which removes the Pod from the current node and adds it to the Kubernetes Scheduler's queue, so that it can, hopefully, be re-scheduled to a better node.

Given these functionalities, this tool could successfully be used to help maintain the long-term health of the cluster and its deployments, especially in the case of unequivocal situations where Pods violate a strict scheduling constraint, as opposed to a preference. However, this implementation also presents a major drawback to its use in production environments, since

[1]https://pkg.go.dev/sigs.k8s.io/descheduler

Pods are evicted solely based on the decisions of the Descheduler's limited algorithms, without confirming that the Scheduler can, in fact, allocate them to a more optimal node. This can result in an increased instability due to frequent situations where the Scheduler's more complex algorithm determines that a Pod should be re-allocated to the node that it was evicted from.

Furthermore, an ideal cluster monitoring solution should also take into account additional properties introduced by this dissertation, such as maintaining the balance of Realtime workloads between the available worker nodes, and responding to significant variations in network link latency or the application-level metrics of dependencies.

Fortunately, the Descheduler project supports the friction-less extension of its core behaviour through the use of plugins that add support for new factors to be taken into consideration. Therefore, the following plugins were developed in order to increase the suitability of the project for this dissertation's objectives.

BetterNode

To improve the decisions of the best nodes to include the services and replicas, one possible proposal would be to develop a descheduler plugin for each of these additional considerations. However, unlike the Kubernetes Scheduler, the Descheduler project does not provide any mechanism for achieving a consensus between different plugins, meaning that each plugin has full autonomy in eviction decisions. Due to this fact, this proposal would not be able to guarantee that the plugins would accurately predict if the scheduler would ultimately act in accordance with the logic that lead to an eviction, allocating the Pod to a different node.

Instead, the *BetterNode* plugin solves both problems simultaneously by submitting each existing Pod to a simulated scheduling process in order to determine if it is running on the most optimal node. To achieve this, the plugin includes a modified version of OpenShift's *capacity-analysis* open-source project[2], that provides a dummy version of the Kubernetes API, and a Scheduler implementation loaded with the custom scheduling plugins described in 5.4.4.

For each of the active Pods in the cluster, the algorithm starts with building a local representation of the cluster state, by synchronizing the dummy API with its real counterpart. Crucially, the Pod being evaluated is excluded from this local representation, given that the objective is to ascertain the worker node where it would be scheduled to, if it was not already deployed in the cluster. If the Pod was not removed, the worker node where it is currently running would be unfairly penalised in the simulated scheduling.

After the local API is ready, the *BetterNode* plugin uses it to simulate the re-scheduling of the Pod by running the scheduler algorithm in a dry-run configuration that does not apply any changes to the actual cluster. By comparing the simulation's result with the current worker where the Pod is located, the plugin can accurately determine if there is a more optimal node, and, therefore, if the Pod should be evicted.

However, despite being a crucial resource for maintaining a balanced cluster state over the long term, Pod evictions constitute a destructive process that negatively impacts the

[2]https://docs.openshift.com/container-platform/3.11/admin_guide/cluster_capacity.html

availability of a given service by introducing unscheduled downtime while the migration is performed, and by discarding the application's runtime context and variables, since the Pod is restarted from its base image.

To mitigate this, the plugin employs a set of strategies that aim to ensure that each eviction is judiciously considered. Specifically:

- As can be observed in algorithm 6, a Pod is only considered for eviction once it has been running for a certain amount of time, which by default is set at 120 seconds. This effectively establishes a grace period in which the application has time to perform potentially expensive start-up preparations without being penalised, allowing it to first reach stable levels of both performance and exposed metric values.
- Likewise, Pods are only evicted if they have not been the target of a previous eviction within a certain timeframe, set at 20 minutes. This backoff strategy prevents the system from adversely impacting application availability by reacting too frequently to changing cluster conditions.
- Upon completion of each simulated scheduling, the *BetterNode* plugin receives feedback indicating the optimal worker node that was selected. Unfortunately, there is no mechanism through which the plugin could receive more detailed information such as the final score allocated to each node. Therefore, it is not able to determine the degree to which the selected node is considered better than the current one. This is especially problematic in cases where the best nodes share a nearly identical score, including the node that currently hosts the Pod. In such a scenario, the node that is ultimately selected by the scheduler can seem arbitrary, and repeated runs may not always produce the same result due to very small score variations that naturally occur. This behaviour is not desirable, since it results in unnecessary evictions that have an adverse effect on the overall stability of the cluster. As a mitigating strategy, *BetterNode*'s copy of the custom Kubernetes Scheduler includes an extra scheduling plugin named *MigrationAvoidance* that attributes the maximum score to the current node, and a score of zero to all other nodes. This plugin is configured with a scheduling weight that is lower than all others by a factor of 10, which is crucial, since its objective is to introduce a small bias toward preventing evictions that is only expressed in tie-breaker scenarios.
- To minimize the negative impacts on service availability, developers of applications that include two or more replicas can limit the number of replicas that are allowed to be evicted at any given time, either in absolute or as a percentage. Alternatively, this effect can also be attained from the opposite perspective by establishing a minimum number or portion of replicas that must be kept available at all times. Both these types of constraints are achieved with the declaration of a PodDisruptionBudget within the corresponding FogService resource. By using the Eviction API, the Descheduler framework automatically respects any declared PodDisruptionBudgets, since Kubernetes blocks requests that would violate them.
- Some applications perform functions so critical that not interrupting their execution becomes more important than ensuring the most optimal worker node placement at all

times. In these cases, developers can mark their applications as *not deschedulable* in their respective FogService specification, which will prevent this plugin from considering their eviction.

- Finally, the Descheduler framework itself can also maintain stability by enforcing a maximum number of Pods that can be evicted from a given node within a single cycle. This amount is normally set to 10.

In summary, this implementation enables the descheduler to make eviction decisions that take into account complex scheduling constraints and domain-specific logic. This, in turn, ensures that the orchestration framework is able to detect and respond to changes in the performance of services and network links that occur naturally over time. This way, the system maintains optimal deployment placement and a balanced cluster state. Additionally, this approach eliminates nearly all false positives by confirming that the scheduler will not simply reallocate the Pod to the same node. This is achieved without burdening the cluster's scheduler with a large number of additional operations, since a separate, local, copy is used.

Despite these advantages, the plugin's design also presents some drawbacks like, for instance, the imperfect process used to simulate that a given Pod has not yet been scheduled. In fact, while the Pod is successfully removed from the local representation of the cluster state, the system is not able to exclude the Pod's influence on the resource usage and performance metrics exposed by the worker node and other applications running on it. Left unchecked, this constraint may, in certain cases, favour the incorrect conclusion that the Pod should be moved to a different node. However, the aforementioned *MigrationAvoidance* scheduling plugin also serves to counterbalance this effect to some degree.

Algorithm 6 BetterNode

```
for each node in nodes do
    Pods ← GetPodsOnNode(Node)
    for each Pod in Pods do
        podStartTime ← GetPodStartTime(Pod)
        resultingNode ← SimulateScheduling(Pod)
        now ← GetCurrentTime()
        if $resultingNode \neq node$ AND $now - podStartTime > 120$ then
            lastBackoffTime ← backoff[Pod]
            if now − lastBackoffTime> 1200 then
                EvictPod(Pod)
                backoff[Pod] ← now
            end if
        end if
    end for
end for
```

DegradedNode

This plugin's objective is to improve and stabilise the performance of critical services whose application-level metrics have entered degraded levels. This is achieved by incrementally evicting lower-priority Pods from the worker node, thus reducing interference and improving resource headroom. Given the results discussed in chapter 4, this is especially relevant when the degraded service does not possess real-time scheduling attributes, increasing the effects of interference caused by other running tasks.

Notably, this plugin is designed to work in tandem with the *DegradedPerformance* scheduling plugin presented in 5.4.4, since it prevents evicted Pods from being re-scheduled to the same worker node while the critical application remains in a degraded state. If this was not the case, Pod evictions would result in an exacerbation of the situation, since the returning Pods could potentially cause an even greater degree of interference as part of their start-up procedures.

Similarly to the previous plugin, it is important to carefully consider the balance between the speed at which an incident is corrected, and the adverse effects of excessive evictions on the stability of the cluster and the availability of the respective services. This is presented in algorithm 7. This algorithm starts by searching for the degraded pods and sequences them through their priority. Then, it removes the pod with lower priority. The *DegradedNode* plugin also implements a backoff strategy that prevents the removal of a new Pod on a given node within 5 minutes of the previous eviction. This ensures that, after each eviction, there is enough time for the change to be reflected on the performance of the degraded Pod and its metrics, so that the system only removes exactly as many lower-priority Pods as are absolutely necessary for the critical Pod to recover.

Algorithm 7 DegradedNode

```
for each node in nodes do
    minPri ← $MaxIntegerValue$
    maxPri ← 0
    hasDegradedPods ← $false$
    $Pods$ ← GetPodsOnNode(Node)
    for each Pod in Pods do
        $fogService$ ← GetFogServiceForPod(Pod)
        $servicePriority$ ← GetPriority(fogService)
        $degraded$ ← IsDegraded(fogService, Pod)
        if $servicePriority < minPri$ then
            minPri ← $servicePriority$
            minPod ← $Pod$
        end if
        if $degraded$ AND $servicePriority > maxPri$ then
            hasDegradedPods ← $true$
            maxPri ← $servicePriority$
        end if
    end for

    now ← GetCurrentTime()
    if $hasDegradedPods$ AND $maxPri > minPri$ then
        lastBackoffTime ← backoff[node]
        if now-$lastBackoffTime > 300$ then
            EvictPod(minPod)
            backoff[node] ← $now$
        end if
    end if
end for
```

5.4.6 Container Runtime Interface Shim

As previously discussed, supporting the orchestration of legacy apps is an important requirement for the system described in this chapter, since some services may not yet have been ported to run on containers, or may even be entirely incompatible with containerised environments due to, for instance, the lack of system access or excessive overhead. However, unlike alternatives such as the aforementioned Nomad orchestration framework by Hashicorp[3], Kubernetes was designed exclusively to support containerised applications and does not natively support other runtime environments.

Therefore, an integration of this type requires a new implementation, which involves a number of factors that must first be considered. Specifically:

- How are legacy applications declared and presented within the Kubernetes API?

  Should they be represented with a new, bespoke, resource, or use the same *Pod* resource as containerised services?
- How are the legacy applications managed within the worker node?

[3]https://www.nomadproject.io/

Should this new system manage the execution and state of processes itself, or should it simply integrate with existing service management utilities?

- How are the Kubernetes API and the application management system integrated?

  How should the Kubernetes control plane interface with the system in order to start or stop applications, periodically get information on their state, and get application-level logs?

- How to download the binaries/code and dependencies?

  How should the system distribute each applications' binaries (or source code in the case of software developed using interpreted languages) and guarantee the installation of required dependencies?

Possible approaches

After performing a search for existing implementations of such an integration, the option that fulfilled the most requirements was SystemK[4], an open-source project developed by the virtual-kubelet team. This project's main objetive was to enable Kubernetes worker nodes to run Pods as Systemd services instead of containers. To achieve this, the project implements a custom version of the Kubernetes Kubelet that interfaces directly with the host's Systemd API in order to create and manage Pods, instead of using the Container Runtime Interface (CRI).

Within this architecture, both containerised and legacy deployments are declared as normal Pod resources that include scheduling constraints which restrict their allocation to a subset of worker nodes. Legacy services may only be scheduled to nodes managed by the SystemK Kubelet, and containerised applications are restricted to the remaining workers.

Using the Pod resource to represent legacy deployments is not conceptually or semantically appropriate, since it implies using container-specific fields for purposes other than what they were original intended for. Instead, the most correct approach would be to extend the base Pod resource with a new optional section containing bespoke configuration options. However, such a design would require making changes to base Kubernetes components. Alternatively, an entirely new purpose-built resource could be introduced. Despite being technically possible, this would require an even greater refactoring effort to the Scheduler and other core Kubernetes components, which would fall outside the scope of this dissertation.

Given these constraints, on balance, using the Pod resource is deemed to be the most sensible approach. As discussed previously, any missing fields can be specified in auxiliary Custom Resource Definitions, such as FogService.

By using Systemd instead of managing application processes itself, the implementation of SystemK is greatly simplified, since the complexity of executing and monitoring processes is abstracted into straightforward API calls. Nevertheless, it also restricts potential worker nodes to those that run a Systemd-enabled Linux distribution.

Finally, SystemK uses the Pod's Image field to specify the name of the Debian package that corresponds to the application, thereby ensuring the installation of the necessary binaries and their dependencies.

[4]https://github.com/virtual-kubelet/systemk

While this project does fulfil all of the required features set out in section 5.2, the fact that it is implemented as a custom Kubelet means that each worker node needs to be configured with either the default or the customised Kubelet version. As such, the cluster would be comprised of a set of container-only nodes and a set of legacy-only nodes. Alternatively, the worker nodes could be configured to run both instances of the Kubelet concurrently. However, this approach would increase the processing overhead and also result in each physical host being represented within the Kubernetes cluster as two entirely separate worker nodes. This proposal is unsuitable for production environments, since it would inevitably lead to a severely imbalanced cluster state. Either solution ultimately leads to a sub-optimal usage of valuable compute resources. Therefore, this architecture is not compatible with the purposes of this dissertation.

During the aforementioned search, some alternative projects were also found. Namely, KubeVirt[5] and Kata-containers[6], which grant Kubernetes the capability to orchestrate entire Virtual Machines, concurrently with the normal containerised deployments. While these projects do not add legacy application support, they can still provide valuable insight into potential strategies for the integration of alternative runtimes within the Kubernetes architecture.

The Kata-containers project declares Virtual Machines as Pod resources and integrates with Kubernetes via a *containerd* runtime plugin. Requests from the Kubernetes control plane are sent to *containerd* through the Kubelet as usual, with the notable exception that the Pod specification makes use of Kubernetes' *runtimeClass* field. Using this field, *containerd* ascertains the correct plugin to use in order to fulfil the request, which can either be its default container handler, or Kata-containers' custom implementation that spawns and manages Virtual machines.

Following this architecture, a similar *containerd* plugin could be developed to run Pods that present a specific *runtimeClass* as legacy applications. Compared to SystemK, this potential solution has the advantage of conforming to an extension paradigm instead of requiring changes to core Kubernetes components, as previously discussed.

However, this solution would still be sub-optimal, since it would restrict worker nodes to exclusively use *containerd*, whereas normally, each node can run whichever container runtime is most suitable for its particular use case. In fact, the container runtime is typically completely abstracted from the control plane's perspective, since all runtimes communicate with the Kubelet using the same protocol, the CRI.

Furthermore, while some alternatives to *containerd*, such as CRI-O, also provide support for their own plugin system, others, like the Docker Engine, do not. This would be especially limiting for Compute Unified Device Architecture (CUDA)-enabled applications running on Graphics Processing Unit (GPU)-equipped worker nodes (such as the NVIDIA Jetson SBCs used in the ATCLL project's infrastructure), since they generaly require the use of the NVIDIA Container Toolkit, which extends the Docker Engine with the necessary capabilities

[5]https://kubevirt.io/

[6]https://katacontainers.io/

to allow containers to interface with NVIDIA GPUs. Using *containerd* on these systems would preclude containerised services from accessing the node's specialised compute resources.

Given this limitation, this proposal was deemed to be too restrictive. The ideal solution, then, would be an extension-based, runtime-agnostic architecture that allows each node to select the most suitable container runtime for its purposes, but also runs an entirely separate, parallel, legacy runtime. Developers would then select the desired environment for their application using the aforementioned *runtimeClass* field.

This is not possible to implement using just the current vanilla Kubernetes components, since the default Kubelet implementation only supports configuring one single container runtime and the *runtimeClass* field is likewise only intended to discriminate between that runtime's plugins, if they exist.

To address this issue, the Atrio Inc. team developed Multi-CRI[7], an open-source Golang project that acts as a proxy layer between the Kubelet and one or more runtimes. In order to accomplish this, the Kubelet is configured to treat Multi-CRI as its container runtime, thus sending all CRI requests directly to it. The tool then selectively forwards these requests to the correct underlying runtime based on the *runtimeClass* field associated with each Pod, and the values specified in its configuration file.

Proposed approach

This dissertation's implementation uses a custom version of this Multi-CRI tool and a purpose-built CRI Shim that converts the Kubelet's requests into *Systemd* API calls to fulfil the requirements for orchestrating legacy applications set forth in section 5.2, and the architecture is depicted in Figure 5.3. The shim was implemented in Golang and is named "Systemd Adapter", so as to conform to the Multi-CRI project's existing terminology for integrations of its nature.

With this implementation, the Kubernetes control plane, as well as the developers and system administrators that use it, treat legacy applications exactly as regular Pods, (with the exception of the *runtimeClass* field), while in the worker nodes themselves each Pod is, in reality, executed as a *Systemd* unit/service.

Using *Systemd* as the underlying service management layer leverages all the advantages mentioned when discussing the SystemK project. Since *Systemd* has become fairly ubiquitous within the Linux world, this integration was not deemed overly restrictive. Any Pod without the legacy *runtimeClass* field is still treated as a normal containerised workload, as Multi-CRI forwards its CRI traffic to *containerd* or whichever container runtime was configured for that particular worker node.

[7]https://github.com/atrioinc/multi-cri

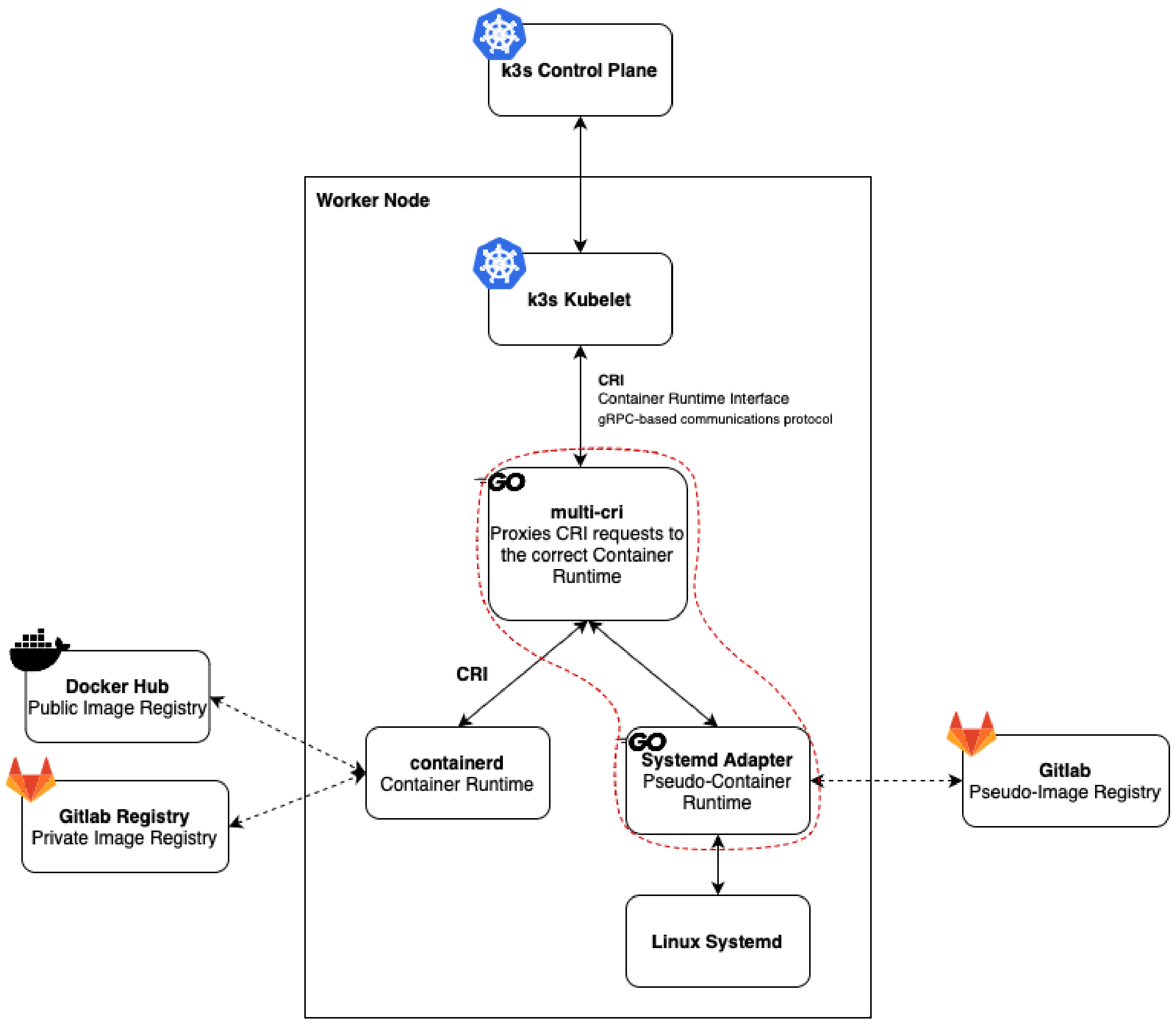


Figure 5.3: Overview of the system's runtime architecture

Within the scope of this dissertation, this *Systemd* Adapter was not envisioned as an advanced implementation that fully maps every possible CRI request into its *Systemd* API counterpart, but rather as a proof-of-concept that focuses on implementing the set of critical features required for the orchestration of most legacy applications without any issues, namely:

- **Pulling and installing images:** implements the *PullImage* CRI Google Remote Procedure Call (gRPC) request using a methodology that is described further in this section.
- **Setting CGroup hierarchy level and resource limits:** implements the *Create-Container* CRI gRPC request by altering the underlying unit file in order to configure the application to run within a child to the top-level "Legacy" parent CGroup, and additionally, to set the CGroup's resource limits according to the values specified within the Pod specification.
- **Starting and stopping services:** Implements the *StartContainer* and *StopContainer* CRI gRPC requests by performing *Systemd* API calls to start and stop the underlying unit/service, respectively.

- **Getting the service's current status:** Implements the *ContainerStatus* CRI gRPC request by performing *Systemd* API calls to get the current state and properties of the underlying unit/service.

Kubernetes State field	**Systemd Unit property**
State	SubState
StartedAt	ExecMainStartTimestamp
FinishedAt	ExecMainExitTimestamp
ExitCode	ExecMainStatus
Reason	SubState

- **Getting the latest application-level logs:** When starting new Pods, the Kubernetes Kubelet instructs the container runtime to write all application output to a specific file that is unique for each Pod. When a Pod's logs are requested by developers or system administrators, the Kubelet returns the contents of the respective file.

  Kubernetes also performs log rotation daily, or if the log file grows beyond 10MB in size. When this rotation happens, the container runtime is instructed to redirect output to the newly created file.

  By default, *Systemd* uses its own logging architecture based on the $systemd-journald$ service. However, it can also be configured to append new entries to specific files, on a service per service basis.

  This crucial *Systemd* Adapter feature implements the *ReopenContainerLog* CRI gRPC request by first altering the underlying unit file to replace the *StandardOutput* and *StandardError* fields with the new log file path indicated by the Kubelet, and then reloading the *Systemd* daemon to apply the changes.

Notably, the current limitations of the system include the lack of network isolation and Container Network Interface (CNI) integration. In fact, all legacy applications use Host networking mode, and the FogService Operator ensures that their Pod specification reflects this. In the future, these features could be introduced through the use of Linux network namespaces.

As previously mentioned, one of the most relevant design challenges for tools like *Systemd* Adapter is how to download and distribute the application binaries (or source code) and their respective dependencies using a methodology that is reliable and repeatable. While the SystemK implementation of using the Pod's image field to specify the Uniform Resource Locator (URL) to a *.deb* file would be a promising solution, it also restricts the system to be used exclusively on Debian-based Linux distributions. More importantly, some applications may require complex multi-step setup procedures that can involve, for instance, the installation of dependencies from multiple package managers, the creation of directory structures, or the re-configuration of system resources such as *sysctls* or network interfaces.

Since automatically predicting these complex requirements for each application is not feasible, the *Systemd* Adapter places the burden of explicitly specifying each step with the developers of the service, who are expected to provide an installation script that serves as an analogue for the container image. Based on the same logic, the adapter does not automatically create *Systemd* unit files for each Pod, unlike SystemK. Instead, the installation script must include the preparation and distribution of the application's *Systemd* Unit file, which must bear the same name as the respective FogService. This way, developers have the freedom to use advanced configurations that would not otherwise be possible. The installation script is then downloaded by the worker nodes from the respective project repository on the organisation's version control platform, which is itself an analogue of a registry for these pseudo-images.

Over time, as new versions of the applications are developed, their installation scripts will need to be re-executed to ensure that the latest changes are applied, mirroring the update process of container images. However, this repeated execution introduces the potential for failures, particularly in the case of poorly designed scripts that include commands that can only be successfully ran once. Therefore, it is crucial to develop scripts that follow the idempotency principle.

Given that fully idempotent scripts are not trivial to implement using typical Portable Operating System Interface (POSIX) scripting languages (i.e: bash), applications that use the *Systemd* Adapter must instead provide their installation setup steps in the form of Ansible playbooks. This approach incurs a slight learning curve for developers, but greatly increases readability and the robustness of the process. This is depicted in Figure 5.4 and described in the following steps:

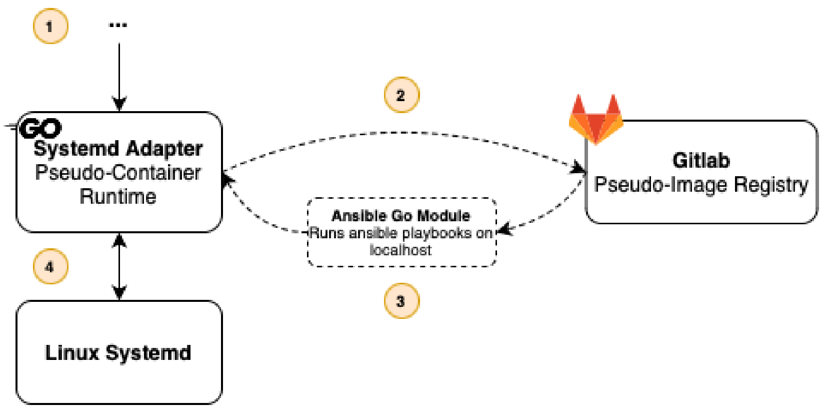


Figure 5.4: Overview of the Systemd Adapter's pseudo-image pull implementation

- **Step 1 -** The developer specifies a gitlab project ID, the Ansible playbook's file path and the desired version/branch tag. This information is saved within the respective FogService Custom Resource (CR) specification.
- **Step 2 -** The *Systemd* Adapter uses the Gitlab API to download the file from the remote repository into a pre-defined directory within the worker node's local storage.
- **Step 3 -** The *Systemd* Adapter uses the Ansible API to execute the Ansible playbook in order to download and/or install the binaries, source code, dependencies, and the *Systemd* unit file.

- **Step 4 -** The *Systemd* Adapter interfaces with *Systemd* using the *dbus* API to manage and monitor the service's state.

Automatic management of Real-Time priorities and attributes

To fulfil the requirements laid out for the automatic management of real-time priorities in section 5.2, one promising implementation would be for the FogService Operator to determine the correct absolute priority value to apply to each process, and then include a Lifecycle Hook section within the respective Pod's specification. In this context, the Operator could apply a PostStart Lifecycle Hook with a bash payload that automatically executes the relevant *chrt* command(s) in the container's shell as soon as the Pod becomes ready. Since the Lifecycle hook can only be specified and executed once without restarting the Pod, subsequent priority changes (if any) could be accomplished by the FogService Operator, using Kubernetes' Exec API to send new *chrt* commands to the containers while they are running.

An advantage of this proposal is the fact that the priority management is accomplished entirely by the cluster's control plane (specifically, the FogService Operator), and therefore it does not require any extra processing on the part of the worker nodes. However, the solution also presents drawbacks which ultimately make it unsuitable for use in production environments.

Firstly, since this proposed implementation executes commands on the container itself, every container that requires a Real-Time priority needs to include the *chrt* utility. This can not be taken for granted, especially since Alpine, a lightweight Linux distribution popular for container images, does not include it by default.

And secondly, each container also needs to be granted the SYS_NICE capability in order to successfully alter the scheduling attributes of its processes. This weakens the containers' isolation and introduces a significant risk that rogue processes can abuse this capability in order to heighten their priority indiscriminately and up to potentially dangerous levels. In extreme cases, CPU-intensive rogue or faulty processes could conceivably starve all other processes on the system, crashing it until the host can be physically rebooted.

To fulfil the requirements and address the issues with the first proposal, the implemented solution shifts the priority management logic to the worker nodes themselves, where it is executed each time a new Pod is allocated to them. In order to avoid the creation of another daemon service, and to leverage the fact that Multi-CRI already executes some logic each time a new Pod is deployed, this Real-Time priority management logic is integrated within the customised version of the Multi-CRI used by the system.

As can be observed in algorithm 8, this logic accesses the real-time priority information for each FogService resource, and ranks (in ascending order) all the processes that request a real-time priority within the node, first by the priority of their respective Pods, then by their QoS class, and finally by the relative priority of each process within the Pod. The processes are then granted an absolute priority value within Linux's First-In First-Out (FIFO) real-time scheduler according to their rank. By design, this value may coincide with other processes that were assigned the same relative priority, in cases where both processes belong to the same Pod or to Pods with identical priorities and QoS classes. Notably, the two highest absolute priority values, 98 and 99, are reserved for any critical tasks that system administrators may deem necessary, and as such, this management system never allocates those values to processes.

Since the absolute priority value scale only ranges from 1 to 97, there may also be cases where there are more distinct process ranks than possible absolute priority values. In these scenarios, the lowest priority processes are all assigned the priority value 1 until the number of ranks remaining reaches 96. This behaviour effectively neutralises the differences in rank for lower priority tasks in order to maintain the hierarchy for the maximum number possible of higher priority tasks.

To assign the priority value to each process, the real-time priority management module calls the *chrt* utility on the worker node itself. However, as seen in the FogService resource reference, real-time processes are specified either by their PID within the container, or by the process name (or a substring of it) that can be used to find it. As a consequence, the *chrt* utility cannot be used directly, given that it requires the process' PID on the host namespace. To solve this problem, the module starts by using the *nsenter* utility to enter the PID namespace of the container, while retaining the ability to use the host's *chrt* binary.

In cases where one or more of a Pod's processes have a delayed activation relative to the Pod's start time, the system will be unable to find them (and therefore set their priority) when the Pod is created. In these scenarios, the affected processes are registered to a pending queue which is revisited every 30 seconds until all the processes have successfully been allocated their priority.

Finally, the module is also capable of allocating SCHED_DEADLINE real-time scheduling parameters to processes. That process is significantly simpler than the one that was just described, since there is no need to rank processes by their priority. The system merely applies the attributes as specified in the corresponding FogService resource.

When compared to the first proposal, this approach presents significant gains in robustness and security. In fact, both of the former's main drawbacks are successfully addressed, since the new system performs the priority assignments on the worker node itself instead of the container environment, thereby precluding the need for the SYS_NICE capability and the inclusion of the *chrt* utility in every container image. Furthermore, its deep integration with the Multi-CRI service enables it to act instantly as part of the Pod creation pipeline, whereas if it had been implemented as a separate service, it would likely have to adopt a polling paradigm, where it periodically determines if new Pods have been allocated, therefore potentially incurring significant delays. In addition, this Multi-CRI integration also results in network bandwidth and latency savings, since the algorithm can get almost all of the information it requires about the Pod directly from the Kubelet's CRI requests, instead of querying Kubernetes' control plane.

A final design challenge to take into consideration is how to ensure that the CPU usage limits specified by developers are also enforced for the processes that use Linux's Realtime scheduling features. Normally, Kubernetes delegates this responsibility to the underlying container runtime, which in turn, applies the limits as CFS quotas in the configuration of the container's cgroup. These parameters only apply to processes that are scheduled using CFS, and as such, they will have no effect on Realtime tasks. This raises very serious stability issues, since a single rogue or malfunctioning Realtime process can potentially starve a large number of other workloads running on the host, including lower-priority RT tasks and any regular processes. Fortunately, the Linux kernel prevents this effect from causing catastrophic consequences (such as the complete crash of the Operating System) through the aforementioned cap on the total allocation of CPU time to RT processes, which is typically set at 95% by default. Nevertheless, setting per-Pod RT resource limits still constitutes a critical requirement.

Some container runtimes (such as, for example, the Docker Engine) have already introduced support for this feature by configuring the *cpu.rt_period_us* and *cpu.rt_runtime_us cgroup* parameters, where the latter represents the maximum number of microseconds of CPU time that can be allocated to the *cgroup*'s tasks during each period. Developers can specify these limits through special fields when describing their container deployment. However, this has not yet been included in Kubernetes' Pod specification or in the CRI request schema, meaning that, even if the underlying runtime supports these configurations, Kubernetes currently has no mechanism with which to instruct it to apply them.

To address this, the Realtime priority allocation logic included in the customised Multi-CRI service uses the *cgset* API to set the two parameters manually on the respective *cgroup*. In terms of determining which values to set, the simplest approach would be to use the CPU resource limit specified by the developer in the Pod (or FogService) resource, and convert it to the required units. For Pods that are entirely composed of Realtime tasks, this would work exactly as intended. However, in cases where Pods contain a mix of both Realtime and regular processes, this solution would effectively double the Pod's usage limit, since both types of workloads would have identical restrictions that would be accounted separately. Therefore, for the purposes of this implementation, the regular CPU limits apply exclusively to processes that use CFS (as usual), and the FogService resource includes a separate field to specify Realtime limits.

Algorithm 8 Realtime Priority Attribution Logic

```
function ApplyPrioritiesToAll($Node$)        ▷ Called whenever a new Pod is scheduled
    $Pods \leftarrow$ GetPodsOnNode(Node)
    $priorityMap \leftarrow \{\}$
    $priorityList \leftarrow []$
    $pendingProcesses \leftarrow \{\}$
    for each Pod in Pods do
        $fogService \leftarrow$ GetFogServiceForPod(Pod)
        $podPriority \leftarrow$ GetPodPriority($fogService$)
        $QoSClass \leftarrow$ GetQoSClass($fogService$)
        processList $\leftarrow$ GetProcessList($fogService$)
        for each process in processList do
            relativePriority $\leftarrow$ GetRelativePriority($process$)
            priorityBucket $\leftarrow \{podPriority, QoSClass, relativePriority\}$
            priorityMap[priorityBucket] $\leftarrow$ Append($priorityMap[priorityBucket], process$)
        end for
    end for
    for each bucket in priorityMap do
        priorityList $\leftarrow$ Append($priorityList, bucket$)
    end for
    $priorityList \leftarrow$ SortByPriorityAscending(priorityList)
    $i \leftarrow 1$
    $len \leftarrow$ GetLength($priorityList$)
    for $j = 1$, $j{+}{+}$, while $j < len$ do
        bucket $\leftarrow priorityList[j]$
        processes $\leftarrow priorityMap[bucket]$
        for each process in processes do
            result $\leftarrow$ ApplyPriority($process, i$)
            if $result \neq Successfull$ then
                pendingProcesses[i] $\leftarrow$ Append(pendingProcesses[i], process)
            end if
        end for
        if $len - j \leq 98$ then
            i $\leftarrow i + 1$
        end if
    end for
    return $pendingProcesses$
end function
```

5.4.7 Load Balancer

One of the most important features of most orchestration systems is the support for the duplication of a running application into multiple replicas running the same processes, typically on different worker nodes. This technique ensures added resiliency to failures and potentially higher overall performance through the load-balancing of the networking traffic between the available instances.

To make this possible, the orchestrator provides a virtual network endpoint that works, conceptually, as a single point of entry for traffic that needs to reach a replica of an application. This functionality includes logic to track all the available replicas and load-balance the network traffic between them. When the orchestrator detects that a given replica (or the node it was running on) becomes unavailable, this forwarding logic removes it from consideration, in order to prevent unanswered packets, thus preserving the availability of the service from the perspective of its clients.

In Kubernetes' architecture, this virtual endpoint is implemented by a native resource named Service that developers and system administrators must explicitly declare for each application that requires it. In this Service declaration, developers specify the name of the underlying deployment and a list of ports and transport protocols that the service should expose. Kubernetes then provides this Service with a network IP address and a predictable DNS name within its virtual network. Client applications can use these to reach a replica through the load-balancer.

When evaluating a feature of this nature, it is also important to consider how this virtual single point of entry is implemented and its effect on the total round-trip time of inter-Pod traffic. In fact, the overall latency of client-server communications could potentially be greatly affected by a poorly designed architecture, such as, for instance, if the Virtual IP address directs traffic to a master node where the load-balancing logic was actually performed, thus introducing superfluous hops to the traffic's path.

Kubernetes cleverly avoids this issue by including the load-balancing logic in its Kube-Proxy tool that runs on every worker node, meaning that the traffic's ultimate destination is determined by logic present within the node where it originates.

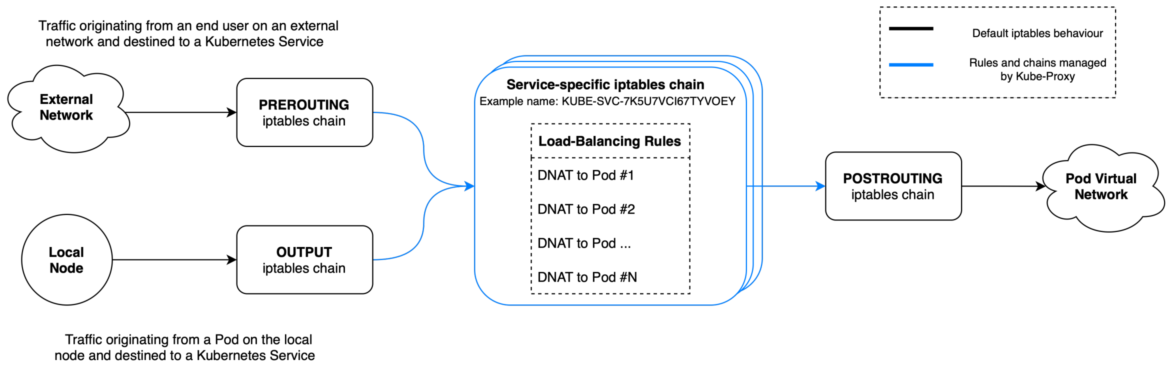

Traffic originating from an end user on an external network and destined to a Kubernetes Service
External Network
PREROUTING iptables chain
Local Node
OUTPUT iptables chain
Traffic originating from a Pod on the local node and destined to a Kubernetes Service
Service-specific iptables chain
Example name: KUBE-SVC-7K5U7VCI67TYVOEY
Load-Balancing Rules
DNAT to Pod #1
DNAT to Pod #2
DNAT to Pod ...
DNAT to Pod #N
POSTROUTING iptables chain
Pod Virtual Network
Default iptables behaviour
Rules and chains managed by Kube-Proxy

Figure 5.5: *Iptables* chain pipeline

Kube-proxy accomplishes this at the Linux kernel firewall level by creating new *iptables* chains and rules with the goal of performing Destination Network Address Translation (DNAT) on any traffic destined to the Virtual IP address of a Kubernetes Service resource. As depicted in figure 5.5, this traffic can originate from two separate types of sources. Packets sent by an end user client on a network that is external to Kubernetes are handled by the PREROUTING chain of *iptables* rules, whereas those that are sent by Pods running on the node itself are directed to the OUTPUT chain.

In both these chains, Kube-Proxy adds one new rule for each exposed port declared by each existing Service in the cluster. This rule identifies the relevant traffic and directs it to another chain that is created specifically for that port and service, and is identified by the hash of a string consisting of the service's name and a tuple composed of the exposed port and its respective transport protocol.

In turn, each service-specific chain contains one DNAT rule for every available replica of the service. Each rule has an associated probability value, which leads *iptables* to select the wining rule for each packet based on the Markov chain formed from the set of rules and their respective probabilities, instead of always selecting the first rule on the chain. All these new chains and rules are continuously updated in order to reflect the additions and removals of Pods and Services at the cluster level.

Another decisive factor to consider is the nature of the load-balancing algorithm itself, or, in other words, the criteria used to divide network requests among the available destinations. In the case of Kube-proxy's implementation, the algorithm strives for maximum fairness, meaning that the aforementioned probability values are identical for every replica of a service. Therefore, on average, each replica should receive the same amount of network requests as every other, $1/N^{th}$.

While this strategy ensures a balanced resource usage by every Pod (in cases where every network request requires identical resources to process), it will frequently result in sub-optimal performance in terms of overall request round-trip time. As discussed in the requirements, a delay-sensitive service orchestration framework for smart cities should include a load balancing algorithm that takes into account the network topology and the current performance of its links. In fact, since most edge computing clusters in smart city contexts are spread out geographically, and Kubernetes automatically places replicas in as many different topology zones as possible to increases resilience, some replicas will invariably be much farther away from a given client application, in network latency terms, than others. In these cases, using a fair algorithm like the one employed by Kube-Proxy means that the farthest Pods will receive the same amount of requests as the closest ones, which will dramatically increase the maximum Round-Trip Time (RTT) values and negatively impact the system's ability to comply with performance guarantees.

Furthermore, in real production deployments some replicas will frequently have uneven performance and resource headroom due to being allocated on nodes that are more resource-constrained or that are host to more processes that are prone to cause interference. As such, an ideal system should leverage metrics exposed by the worker nodes and the applications themselves, both in live or in historical trend formats, to ascertain the replicas that are most likely to provide the fastest response time.

Using the default Kube-Proxy behaviour as a base, this dissertation proposes an approach where the aforementioned integrations can be accomplished by dynamically changing the rule probability values in each service-specific *iptables* chain. While this could conceivably be achieved by developing a customised version of Kube-Proxy, such an approach would have the same drawbacks as similar proposals discussed in the precious section, namely, the fact that updating to future versions of Kube-Proxy would become more difficult due to potential merge conflicts. Additionally, the custom version would be exclusive to Rancher k3s, since each Kubernetes distribution has small differences in the way that Kube-Proxy is implemented, packaged, and deployed.

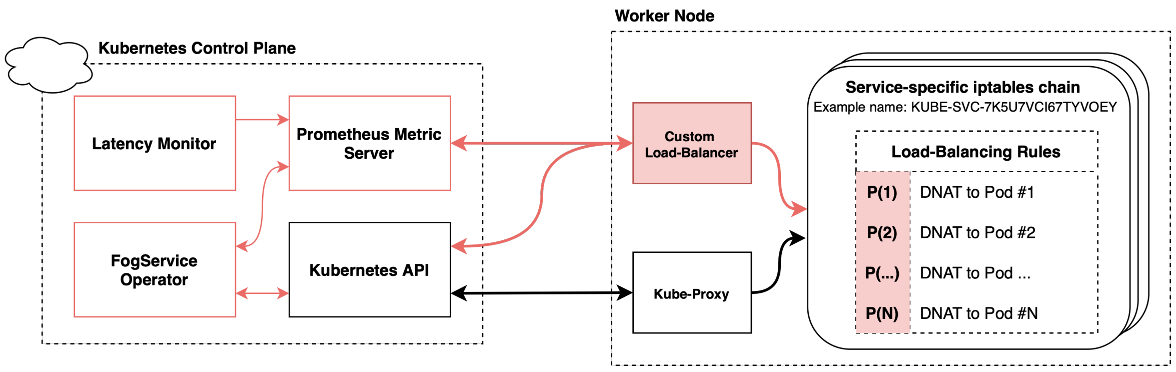


Figure 5.6: Load balancing architecture

Instead, the proposed approach (depicted in figure 5.6) consists of an entirely new Golang service that runs in the background on each worker node, and periodically overwrites the probability values in order to reflect the current state of the latencies and metrics throughout the cluster.

However, this process requires that every worker node performs expensive queries to collect the current value of the declared application-level metrics in each replica, and for each FogService resource. This would result in a substantial amount of network congestion and possibly processing delay within the metric server itself. Instead, as discussed in 5.4.3, the FogService operator continuously performs these operations itself and allocates a normalised score to each replica. The scores are then stored within the Status section of the respective FogService resource. As such, only one component performs the expensive queries, and the load-balancer services that require that information can access the summarised version by performing a single Kubernetes API request, greatly reducing congestion.

As can be observed in algorithm 9, the Load-Balancer service starts each cycle by iterating over the list of Service resources declared in the cluster and obtaining the relevant latency and metric information. Each replica i of the respective FogService is then allocated a score based on the following formula:

$$s_i = (mv_i * mw_i) + (lv_i * lw_i)$$

where mv_i represents the normalised and aggregated metric score of the given Pod, and lv_i represents the normalised latency score of the node where that Pod is located. In turn, mw_i and lw_i represent the weight that was specified on the FogService resource by the developer for mv_i and lv_i, respectively.

Using these scores as weights, the load-balancer daemon can successfully determine the percentage of the network traffic that should be directed to each replica. However, as previously discussed, the *iptables* DNAT rule probabilities cannot be updated directly with these weight values since the kernel executes rules in sequential order, forming a Markov chain. Is this Markov chain, P_i is the probability of staying in each state (probability of no transition to the next state). This probability, presented below, depends on the probabilities of no transition in the previous states, and the one of N is 1, since it cannot transition to a next state. Therefore, the replicas must first be sorted by score in ascending order, and the following formula must be used in order to accurately calculate the probability of a given rule being selected when it is considered by the kernel:

$$P_i = \begin{cases} s_1, & \text{if } i\text{=1} \\ s_i * \frac{1}{\prod_{j=1}^{i-1}(1-P_j)}, & \text{if } 1 < i < N \\ 1, & \text{if } i\text{=N} \end{cases}$$

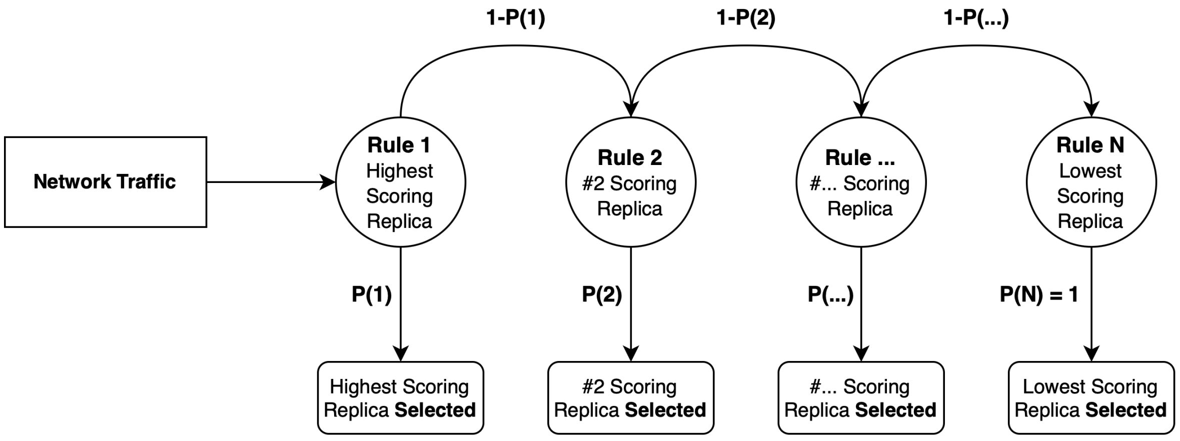

1-P(1)
1-P(2)
1-P(...)
Network Traffic
Rule 1 Highest Scoring Replica
Rule 2 #2 Scoring Replica
Rule ... #... Scoring Replica
Rule N Lowest Scoring Replica
P(1)
P(2)
P(...)
P(N) = 1
Highest Scoring Replica Selected
#2 Scoring Replica Selected
#... Scoring Replica Selected
Lowest Scoring Replica Selected

Figure 5.7: Markov chain of execution orders

After this process is complete, the load-balancer updates the existing rules in reverse order, so that the highest scoring replica is the first to be considered, as seen in figure 5.7. By default, this algorithm is executed once every 30 seconds, as a balance between guaranteeing responsiveness to changing cluster conditions, and not generating excessive queries and network congestion.

One potential drawback of this implementation is its decentralised architecture where different worker nodes do not coordinate amongst themselves to avoid situations where certain replicas start receiving excessive amounts of traffic. However, when applications expose meaningful performance metrics, the system can automatically self-correct this behavior in real-time, by detecting the fact that certain replicas' metric values are worsening, and adjusting the Markov chain probabilities accordingly.

This methodology is inspired by related work done in [54], and expands upon it by including the application metrics as an additional decision factor, since the original work focused only on the latency of network links.

Within the scope of the broader orchestration framework envisioned in this dissertation, this component is crucial since, beyond the reasons that have already been mentioned, the *Dependencies* scheduler plugin and, by extension, part of the *BetterNode* cluster state monitoring plugin, would be rendered almost entirely ineffective without it. In fact, scheduling applications as close as possible to replicas of their dependencies would not have nearly the same effect if worker nodes used the standard Kube-Proxy load-balancer implementation, where the farthest replicas receive the same amount of traffic as the closest ones.

Algorithm 9 Load Balancer

```
while true do
    endpoints ← GetKubernetesServices()
    for each service in services do
        fogService ← getFogServiceByService($service$)
        ports ← getPortsByService($service$)
        Pods ← getPodsByService($service$)
        metricWeight ← getMetricWeight($fogService$)
        latencyWeight ← getLatencyWeight($fogService$)
        metricValues ← getNormalisedMetricValues($fogService$)
        latencyValues ← getNormalisedLatencyValues()
        podScores ← {}
        podProbabilities ← {}
        podList ← []

        for each Pod in Pods do
            score ← $(metricValues[Pod] * metricWeight) + (latencyValues[Pod] * latencyWeight)$
            podScores[Pod] ← $score$
            podList ← Append($podList, Pod$)
        end for
        podList ← SortByAscendingScore($podList$)
        $len$ ← GetLength($podList$)
        for $i = 0$, $i$++, while $i < len$ do
            Pod ← $podList[i]$
            if $i == (len - 1)$ then
                podProbabilities[Pod] ← 1
            else
                prod ← 1
                for $j = 0$, $j$++, while $j \leq i - 1$ do
                    prevPod ← $podList[j]$
                    prod ← $prod * (1 - podProbabilities[prevPod])$
                end for
                prob ← $podScores[Pod] + (1/prod)$
                podProbabilities[Pod] ← $prob$
            end if
        end for

        for each port in ports do
            for each Pod in podList do
                UpdateRule(podProbabilities[Pod])
            end for
        end for
    end for
    Sleep(30$secs$)
end while
```

5.5 Conclusions

This chapter proposed an orchestration framework that is able to perform automatic configuration of real-time scheduling parameters for critical processes depending on the nodes and network conditions, and the priority of the running services. This approach performs the scheduling of workloads to nodes based on customised algorithms that take into account several factors such as latency and application-level metrics of their dependencies. Moreover, it also supports the automatic migration of workload allocations that become sub-optimal over time. Finally, a load balancing algorithm for network requests is proposed, which takes into account the latency and application-level metrics of the destination replicas.

The objective of the proposed approach and its algorithms is to significantly improve the stability of time-constrained services, regardless of the level of interference posed by other processes. This will be assessed through a set of experiments that will be described and discussed in the next chapter.

CHAPTER 6

Experimental Study and Evaluation of Service Orchestration

Building upon the previous chapter, that proposed several approaches for the orchestration of services in the network and computation platform, this chapter presents the evaluation setup, the tests, and functional and performance results of the proposed approaches.

The chapter is organized as follows. Section 6.1 presents the experimental setup of the framework to test the proposed approaches. Then, section 6.2 presents results of the scheduler-based approaches, section 6.3 presents results related to the scheduler approaches, and section 6.4 presents the load-balancer results. Finally, section 6.5 presents the conclusions of this chapter.

6.1 Experimental Setup

The functional and performance evaluation tests are conducted in a virtualised cluster framework presented in figure 6.1, which is deployed as a set of virtual machines running within a Proxmox Hypervisor. When compared to a platform using real devices instead of virtual machines, this approach enables deployments with a larger number of nodes and offers more flexibility regarding the cluster topology, networking architecture, and dynamic control of network link performance using Traffic Control (TC)[1]. The cluster contains 1 master node and 10 worker nodes belonging to 5 different topology zones. The 8 worker nodes that comprise Zones $P1$ through $P4$ consist of Docker containers created from a customised Ubuntu image with $k3s$ and support for *systemd.* The same is true for the Server node. In contrast, the 2 worker nodes of Zone $P5$ consist of full Ubuntu Virtual Machines, in order to ensure greater resource isolation compared to their containerised counterparts, which is relevant for some of the tests performed in this chapter.

[1]https://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html

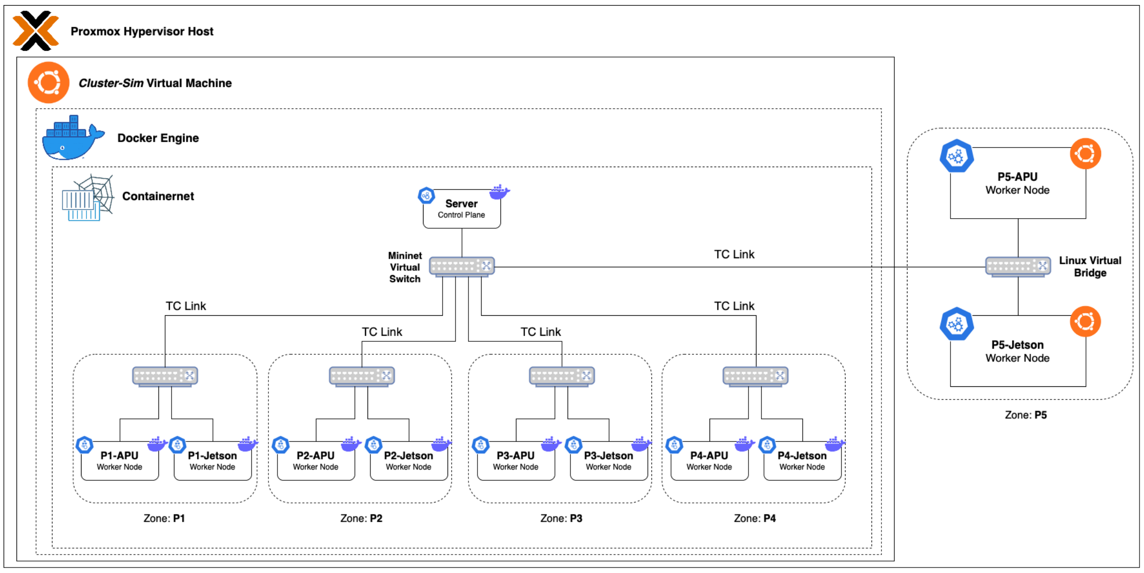


Figure 6.1: Overview of the evaluation setup, inspired by ATCLL, of the virtualised cluster

The test environment is implemented inside *Containernet* (based on Mininet)[2] to allow the creation of a dynamic number of containers, networked using a similar architecture as the one of the ATCLL. The performance of each link is configurable using TC, which is useful to test scenarios where the links between the zone switches and the core switch have different latency values.

Furthermore, the evaluations explored in this chapter are mostly functional in nature, and aim to demonstrate and validate the behaviour of the implemented components in different scenarios. As such, the results mainly depend on the chosen cluster topology, network link performance, and characteristics inherent to the specific scenarios themselves, and not on the performance of the underlying worker nodes. Chapter 4 provides the necessary context regarding the performance of an Edge Computing SBC in several of the runtimes, priorities, and configurations used by this orchestration system. Tables 6.1 and 6.2 present the specifications of the Proxmox hypervisor host and of the virtual machines, respectively. Table 6.3 presents the relevant software in the worker nodes.

Table 6.1: Proxmox Hypervisor Host Specifications

Proxmox Hypervisor Host	
Model	Minisforum EliteMini HX90
Processor	AMD Ryzen 9 5900HX (8c / 16t @ 3.3GHz)
Memory	64 GB (LPDDR4 3200 MT/s)
Storage	1 TB SSD (NVMe)
Proxmox-VE Version	7.2-3
Linux Kernel	5.15.35-1-pve

[2]https://containernet.github.io/

Table 6.2: Virtual Machine Specifications

Virtual Machines			
Name	Cluster-Sim	P5-APU	P5-Jetson
Processor	10 vCPU	4 vCPU	4 vCPU
Memory	42 GB	4 GB	4 GB
Storage	128 GB	64 GB	64 GB
Docker Engine Version	20.10.17	N/A	N/A
Ubuntu Version	20.04.4	22.04	22.04
Linux Kernel	5.4.0-126	5.15.0-25	5.15.0-25

Table 6.3: Versions of relevant software present in worker nodes

Worker Node Containers/Virtual Machines	
Ubuntu Version	22.04
Rancher k3s Version	1.23.8+k3s2
containerd Version	1.5.9-0ubuntu3
sched_rt_runtime_us	950000
sched_rt_period_us	1000000
Containernet Version	3.0

6.2 Scheduling Analysis

This section presents the tests and results of the scheduler-based plugins: node metrics, dependencies and real-time plugins.

6.2.1 NodeMetrics Plugin

The goal of the following test is to evaluate how both scheduler implementations (default and custom) respond in situations where one or more nodes are subject to different levels of CPU utilisation caused by external processes.

Methodology:

In this scenario, 30 identical pods are declared at the same time; their deployment is restricted on two nodes, *p5-APU* and *p5-jetson* in Figure 6.1. For the first round of testing, both nodes present an approximate 0% CPU utilisation, with no interference by external processes. In subsequent rounds, the overall CPU utilisation of the *p5-jetson* node is increased in increments of 10% until 100% is reached. To achieve this, a *stress-ng* deployment is created with the appropriate parameters in order to synthetically generate the desired utilisation in each round. Throughout the testing, *p5-APU* retains no external interference. Each round consists of two separate tests, one for each of the schedulers under evaluation.

This experiment is managed autonomously by a bash script that interacts with the Kubernetes Command Line Interface (CLI) to perform the deployments of the test pods and the *stress-ng* pod. This script starts by deploying the *stress-ng* pod and holding until the node's CPU load metric reflects the desired result, at which point all 30 test pods are deployed simultaneously. After every pod has been successfully scheduled, the current test results are appended to a Comma-Separated Values (CSV) file and the deployments are deleted. This process loops until all 20 tests have been performed.

Results:

Figure 6.2 presents the number of pods allocated to each of the two nodes in each test, as a function of the total CPU utilisation taken up by *stress-ng* processes on *p5-jetson*. Results on the use of Kubernetes' default scheduler show that each node receives an allocation of exactly half of the queued pods (15) in every test, regardless of the stress level imposed on the *p5-jetson* Não!. This behaviour confirms that, natively, Kubernetes is not provided with relevant context on the true state of each node's system resources utilisation, and therefore, it cannot take it into account when scheduling pods.

In contrast, when using a custom scheduler equipped with the NodeMetrics plugin, results show that the number of pods allocated to *p5-jetson* decreases as the CPU utilisation on that node is increased. Conversely, *p5-APU* receives an increasingly larger amount of pods. In fact, when *stress-ng* is configured to use 20% of the available CPU allocation, 11 pods are scheduled to *p5-jetson*, compared to *p5-APU*'s 19 pods. As the utilisation increases to 50%, *p5-jetson* receives only 5 pods, whereas the remaining 25 are allocated to *p5-APU*.

Another important observation that can be made from the results shown in Figure 6.2 is the fact that, when *p5-jetson* reaches utilisation levels of 80% or above, the scheduler does not allocate any pods to that node, leaving all 30 pods to *p5-APU*. This behaviour is explained by the fact that, at such high CPU load levels, the difference in the scores attributed by the NodeMetrics algorithm described in section 5.4.4 to the two nodes is so large, that it becomes highly unlikely that Kubernetes will ever select *p5-jetson*.

This effect could be reduced by applying small tweaks to the algorithm, thereby increasing the number of pods allocated to nodes experiencing high levels of CPU load. However, such a change would be detrimental to the performance of both newly-scheduled and pre-existing applications allocated to these nodes, and would hamper efforts to return them to normal operating conditions. The current implementation enables a better balance.

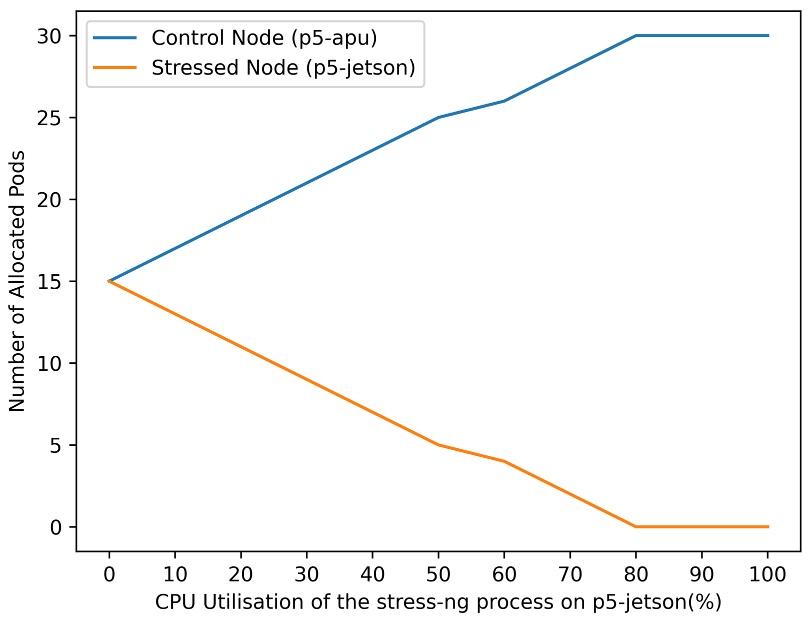


Figure 6.2: Number of pods allocated to each of the two nodes in each test, as a function of the total CPU utilisation taken up by *stress-ng* processes on *p5-jetson*

6.2.2 Dependencies Plugin

The goal of this test is to validate the Dependency plugin's capability to influence the Pod Scheduling process to prioritize the nodes, in order to provide the lowest amount of communication latency between the Pod and the best performing and/or most available replicas of its dependencies.

Methodology:

In this scenario, a dependency application has already been deployed to the cluster in the form of 2 replicas. Figure 6.3 presents the locations of these dependency replicas, as determined by the scheduler. Table 6.4 indicates the latency values for the network links between each zone switch and the core switch. These latency values are induced on each link

Table 6.4: Latency values configured in each network link

Network Link	TC configured latency
p1	0.5 ms
p2	0.8 ms
p3	1 ms
p4	1.2 ms

Table 6.5: Internal metric values for each destination node

Dependency Replica	Static Metric Value
dependency-0	5.0
dependency-1	1.0

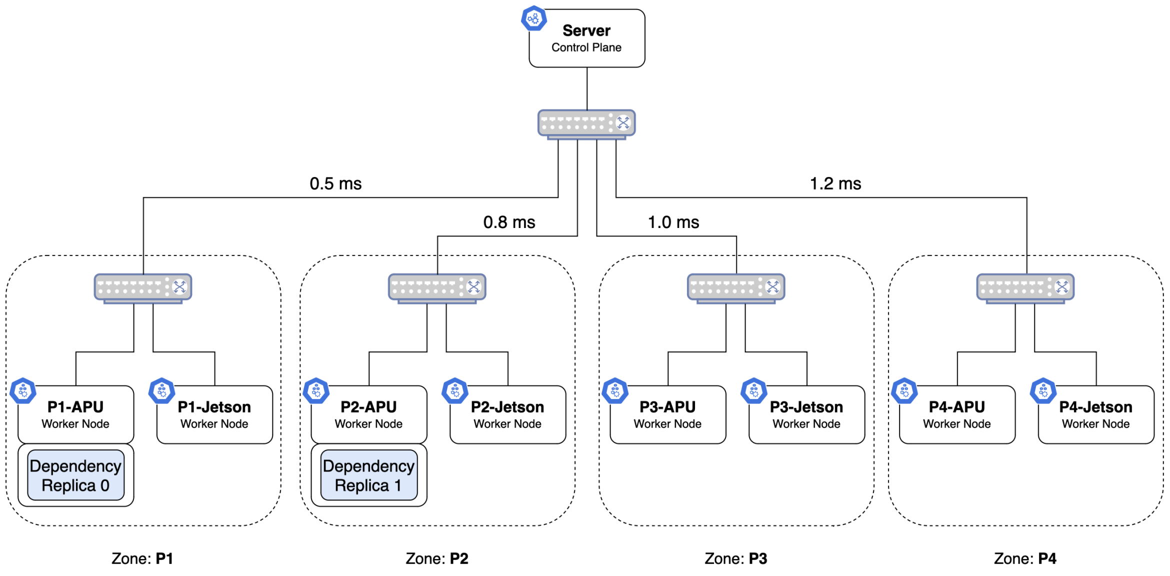


Figure 6.3: Locations of the dependency replicas within the cluster topology

using containernet's TC functionality. Additionally, the dependency application exposes an internal metric that the orchestrator can use to determine the current performance level of each replica. For the purposes of this experiment, the exposed values are statically defined and a lower value denotes a more suitable replica. These values can be consulted in table 6.5.

The experiment is conducted by a bash script that deploys the candidate Pod to the cluster, and then registers the node that was selected by the Scheduler, before removing the deployment to clear the cluster state. This procedure is repeated 200 times in order to ensure statistically relevant results. Finally, each round of tests is performed in 3 different scenarios:

- Using the default Kubernetes Scheduler without any indication of a dependency relation so as to establish a baseline behaviour;
- Informing the default Scheduler of the dependency relation using the PodAffinity specification with Zones as the topologyKey;
- Using a custom Scheduler that includes the Dependencies plugin.

It is also important to note that the candidate Pod's image was pulled to each worker node in advance, so as to neutralize the effect of Kubernetes' *ImageLocality* scheduling strategy, which favours nodes that already possess the required image.

Results:

Figure 6.4 presents the resulting distribution of the 200 scheduling attempts throughout the cluster's worker nodes, for each Scheduler configuration. The default Scheduler results demonstrate an overall trend for the equal (on average) distribution of allocations between the available nodes, which is consistent with the fact that the Scheduler does not possess any information concerning the dependency relation. However, it is important to note the absence of any allocations to nodes *p1-APU* and *p2-APU*. This is attributed to Kubernetes' *NodeResourcesBalancedAllocation* scheduling strategy, which disfavours those nodes due to

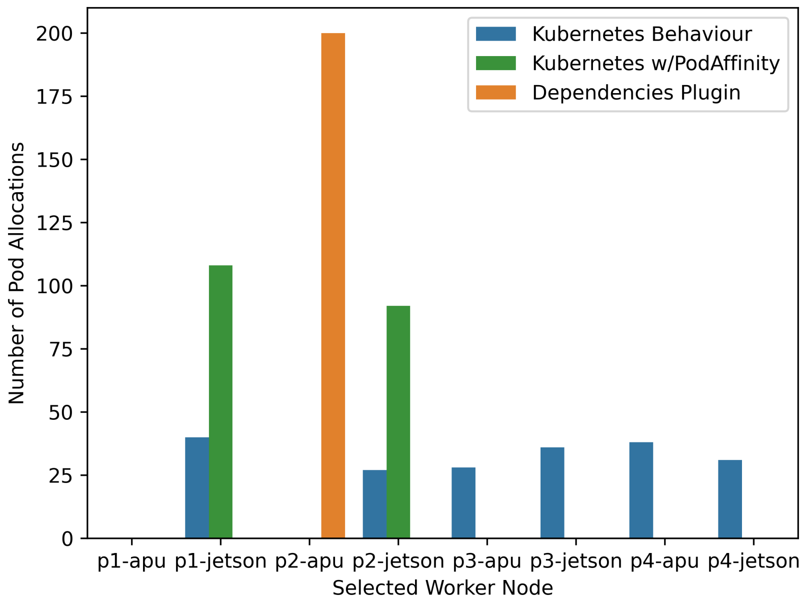


Figure 6.4: Distribution of scheduling results for each Scheduler configuration

the fact that they each already host one Pod (the replicas of the dependency service), whereas the remaining nodes do not.

Compared to this first approach, the inclusion of the PodAffinity specification is a significant improvement, since the Scheduler becomes aware of the dependency constraint and applies a preference for nodes that are located within the same topology zone as a node with a running replica of the dependency. Nevertheless, results show that nodes *p1-APU* and *p2-APU* are still disregarded due to the same reasons as before, despite being the ideal choices. Furthermore, the plugin remains unable to distinguish potential nodes through the application-level metrics of the closest dependency replicas, trending instead towards an equal distribution of allocations between the nodes belonging to suitable topology zones.

The dependencies plugin resolves all of these issues by correctly identifying the *p2-APU* node as the optimal choice for the allocation of the candidate Pod, and consistently selecting it on all of the scheduling attempts, thus maximising the quality of the link between the candidate Pod and the service on which it depends.

6.2.3 Realtime Plugin

The goal of the test is to evaluate how both scheduler implementations take into account the real-time attributes of pods, and how well the final allocation balances these pods throughout the cluster.

Methodology:

In this scenario, 120 pods are declared at the same time. Of those, 80 consist of regular pods, and the remaining 40 pods possess real-time priority attributes; their deployment is restricted to the 8 nodes belonging to zones P1, P2, P3, and P4.

The second test's objective is focused on the allocation of pods with SCHED_DEADLINE attributes. Both schedulers are evaluated for how many nodes, if any, are allocated unfeasible RT task sets, and also the overall balance of SCHED_DEADLINE utilisation between worker nodes. In this scenario, 8 high-utilisation RT pods and 8 low-utilisation RT pods are deployed. High-utilisation is defined as a pod that requires 60% of the total capacity available for RT tasks, whereas low-utilisation pods require 20%. As such, if 2 high-utilisation pods are allocated to the same node, their combined utilisation will total 120% of the nodes capacity, thus constituting an unfeasible task set. As before, the deployments are restricted to the 8 nodes belonging to zones P1, P2, P3, and P4.

Both experiments are managed autonomously by a bash script which interacts with the Kubernetes CLI to perform the deployments of the test pods, wait until every pod has been scheduled, record the resulting allocation in a CSV file, and delete the deployments in preparation for the next test run. This process loops until all 20 test runs have been performed (for each of the two tests).

Results:

Figure 6.5 presents the number of regular pods and real-time pods allocated to each of the eight nodes. The first subplot depicts the results achieved by using the default Kubernetes scheduler, whereas the data for the second subplot is obtained using the custom implementation. In order to facilitate the visual presentation of the data and the subsequent discussion, only one test run result is presented for each of the schedulers. An analysis of the remaining 38 test runs reveals that, while the precise values vary between runs, the trends and behaviours identified and discussed in this section are present in every one.

Results on the use of Kubernetes' default scheduler show that Kubernetes attempts to divide the load equally amongst the available nodes, with each receiving an average allocation of approximately 15 pods. Despite this, since the default scheduler does not possess the necessary context, it does not discriminate pods by their real-time priority or lack thereof. This leads to a pronounced imbalance in the number of real-time pods allocated to each of the nodes, which has implications for the performance of both types of pods, but especially for those with regular priority, as discussed in chapter 4.

Results in the second subplot demonstrate that a custom scheduler equipped with the Realtime plugin is capable of distinguishing between the types of pods and balancing real-time pods evenly throughout the cluster. In this case, the initial 40 RT pods are distributed equally as 5 pods for each of the 8 worker nodes. This minimizes the amount of CPU allocation that is devoted to RT tasks on each node, which in turn increases the performance of both types of pods. However, the analysis of the results also shows that the number of regular pods per node becomes severely imbalanced during this process, which is an undesired consequence that occurs due to a convergence of factors.

As presented in section 5.4.4, beyond the capabilities discussed thus far, the Realtime plugin also influences the scheduling of regular pods by favouring nodes with the lowest level of interference by RT tasks, as measured by the number of RT pods and their respective

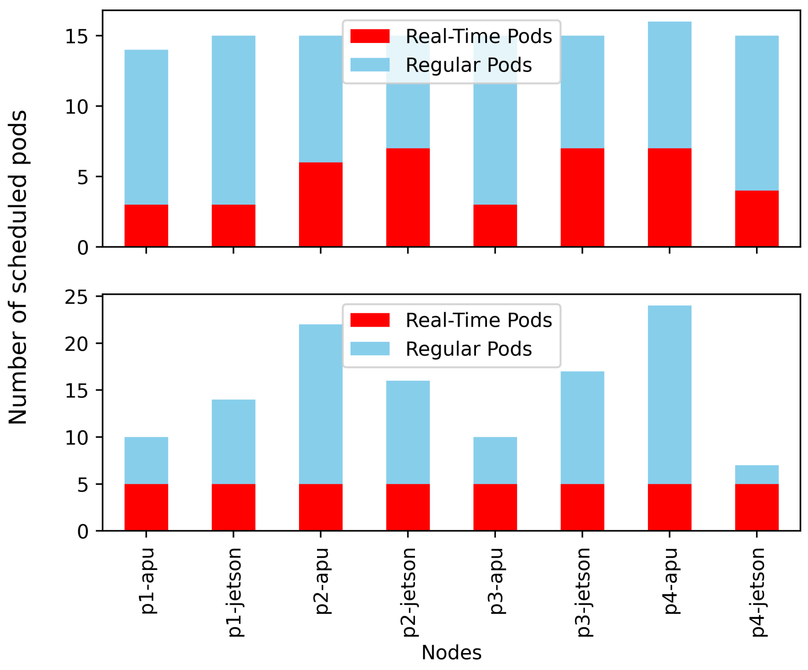


Figure 6.5: Number of regular pods and real-time pods allocated to each of the eight nodes for test suite number 1: default and proposed approach

resource requests. Since the scheduling of pods is performed sequentially and the experiment queues pods in an arbitrary order irrespectively of their RT status, there is a high probability that there are multiple moments during the scheduling process of the 120 pods at which there is at least one node that, at that point in time, has a lower number of RT pods than the rest, as not all RT pods will have been scheduled yet. At these moments, if one or more regular pods enter the scheduling process, the Realtime plugin will assign scores that favour the aforementioned node or nodes. Over time, this leads to the imbalance in regular pod placement observed in the final result.

Therefore, given enough entropy in the queueing of new pods, this undesired consequence will always be present, to a certain degree. Unfortunately, even though the cluster becomes clearly unbalanced, Kubernetes is unable to natively resolve this issue, since the placement of pods is never re-evaluated once they have been scheduled, except in very specific situations. In normal circumstances, this side-effect would call into question the usefulness of the Realtime plugin. However, as presented in section 5.4.5, this dissertation's architecture includes a Cluster State Monitor that will gradually solve this issue, as will be demonstrated in section 6.3.

As a final note, the imbalance that occurs in this specific experiment could also be fixed in an alternative way by implementing a custom scheduler plugin for the Sort extension point, which would sort all queued pods so that RT pods are given precedence in scheduling. However, this solution would only work in scenarios where pods are queued in bulk, which is exceedingly rare. Additionally, this sorting mechanism would be inferior to the default in other respects. For instance, there could conceivably exist a critical service that might

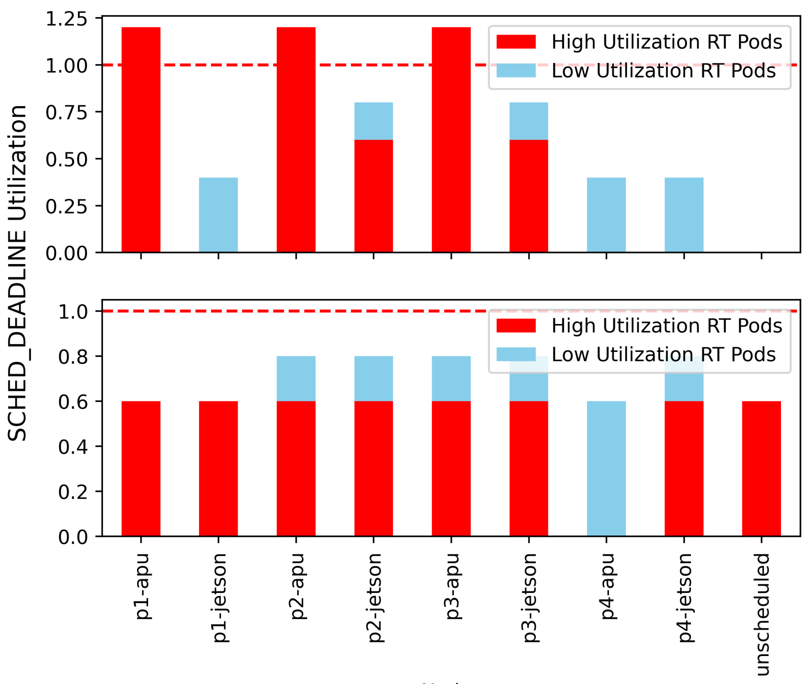


Figure 6.6: Total share of CPU time used by the Linux SCHED_DEADLINE EDF scheduler in each node for test suite number 2: default and proposed approach

not require a real-time CPU allocation priority but should still be scheduled ahead of all other queued pods. As such, the correct sorting mechanism for the context presented in this dissertation is by priorityClass.

Figure 6.6 presents the total share of CPU time used by the Linux SCHED_DEADLINE EDF scheduler in each node. The first subplot depicts the results achieved by using the default Kubernetes scheduler, whereas the data for the second subplot is obtained using the custom implementation. As before, only one test run result is shown for each of the schedulers. An analysis of the remaining 38 test runs also revealed results consistent with the ones presented.

Results on the use of Kubernetes' default scheduler show that Kubernetes attempts to divide the load equally amongst the available nodes in terms of the number of pods, with each receiving an exact allocation of 2 pods. Despite this, since the scheduler does not possess the necessary context, it does not discriminate pods by their real-time scheduling attributes. As a result, 2 high utilisation RT pods are assigned to the same node at least once. Since their combined utilisation surpasses their node's maximum available quota, Linux will not grant the requested RT attributes to the pod scheduled last, which will greatly impact its performance.

Results in the second subplot demonstrate that a custom scheduler equipped with the Realtime plugin can leverage its knowledge of the RT attributes declared for each pod within their FogService CRs to calculate their EDF CPU utilisation, and compare it against the maximum value for each prospective node, taking into account the pods that have already been scheduled to each one. All nodes present similar final utilisation values, to the extent that such is possible, due to the fact that the plugin also strives to minimize that value in every node, so as to increase headroom and improve performance for other types of pods. This validates the plugin's Sort extension point component.

Despite this, the analysis of the second subplot also reveals, as before, an unintended consequence caused by a similar set of factors. Due to the aforementioned arbitrary queue order and the sequential nature of the scheduler, 3 low utilisation RT pods were allocated for the *p4-APU* node before the final high utilisation RT pod could be scheduled. Due to this sub-optimal configuration, every node in the cluster had an utilisation of at least 0.6, which made it impossible to schedule that final pod. This does, however, demonstrate that the Filter extension point component of the plugin worked as intended, by filtering out any nodes where it would not be possible to schedule the pod.

Ideally, this scenario should result in each node executing one high utilisation pod and one low utilisation pod, which would result in a full deployment without any pods being deemed unschedulable. Unlike the previous tests, this issue cannot be fixed using the Cluster State Monitor as currently implemented, since this situation would require a new monitor plugin with purpose-designed algorithms. However, it is possible to achieve the optimal allocation by using the preemption capabilities of the Realtime plugin. To accomplish this, high utilisation pods would need to be declared with a higher priorityClass in their respective FogService CRs. With that change, the scheduler would be able to preempt 2 of the existing low utilisation RT pods on the *p4-APU node* in order to place the final high utilisation pod, after which the preempted pods would be re-scheduled to other nodes.

6.3 Cluster State Monitoring Analysis

This section presents the tests and results of the cluster-based plugins to allocate the services to the best nodes, and to improve the performance of degraded nodes.

6.3.1 BetterNode Plugin

The goal of this test is to evaluate if the Cluster State Monitor is capable of gradually resolving the imbalance observed in section 6.2.3, and how the number of pods allocated to each node evolves over that period of time.

Methodology:

This test starts with the allocation of the same 120 pods (80 regular pods and 40 real-time pods) described in the aforementioned test. Results are collected using a bash script that interacts with the Kubernetes CLI to obtain the current allocation of pods at a rate of 1Hz, and append that information to a CSV file.

Results:

Figure 6.7 presents the number of scheduled real-time pods and regular pods to each of the 8 nodes. The first subplot depicts the initial cluster state, whereas the second subplot presents the cluster state after 380 seconds have elapsed. Figure 6.8 presents the evolution of the total number of pods (of both types) that were allocated to each node over time.

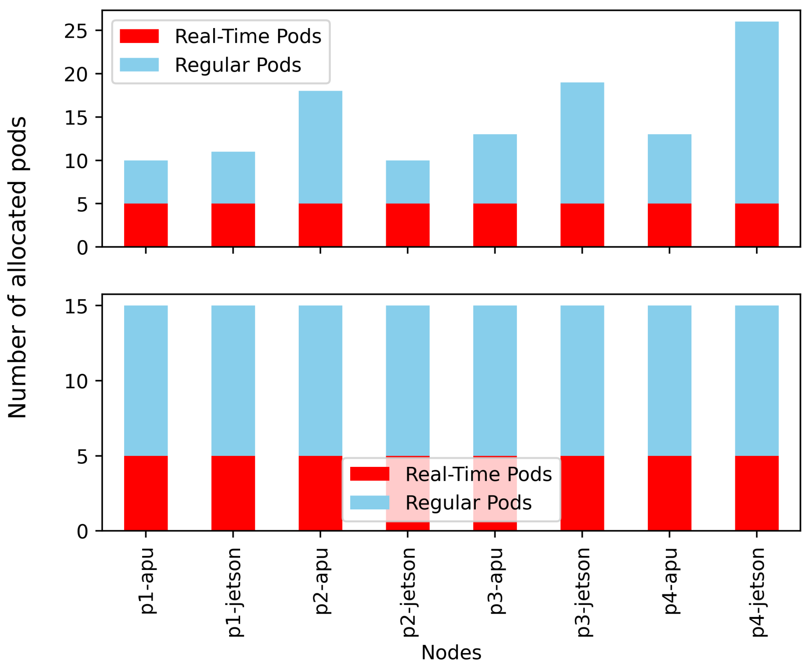


Figure 6.7: Number of scheduled real-time pods and regular pods to each of the 8 nodes: initial state and after 380 seconds

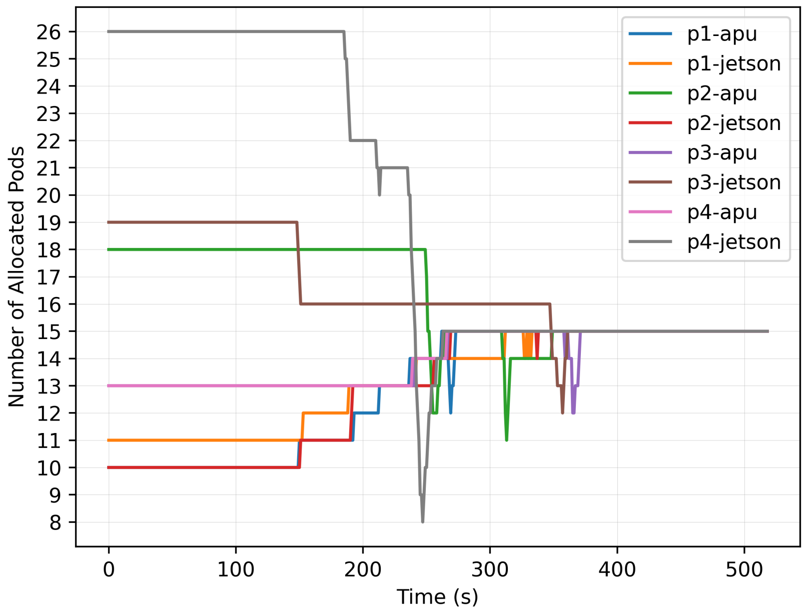


Figure 6.8: Evolution of the total number of pods (of both types) that were allocated to each node over time

Results in the second subplot of figure 6.7 show that, during the 380 second period considered, the Cluster State Monitoring was able to completely re-balance the pod allocation of each node, in order to achieve an optimal state where both real-time and regular pods are perfectly balanced, with each node running exactly 5 and 10 pods of each type, respectively, totalling 15 per node. This result validates that this tool is able to successfully solve the issue presented by the Realtime scheduling plugin in section 6.2.3.

By analysing figure 6.8, one can clearly discern the gradual nature of this process and the evolution of the result. Initially, no preemption events occur, since the entire set of pods was created at the same time, and the 120 second grace period is therefore still in effect. Once that time elapses, the cluster state monitor's descheduler loop determines, using its BetterNode plugin, that it must evict podes currently located in nodes that have been over-allocated, in comparison with others, namely *p2-APU*, *p3-jetson* and *p4-jetson*.

Notice, however, that there are moments in which some nodes, like, for instance, *p3-jetson* and *p4-jetson*, present an excessive number of evictions compared to what was actually necessary, lowering the total number of pods on those nodes below the final value of 15. This behaviour occurs due to the fact that this tool operates by evaluating the current placement of each pod in the same sequential manner as the scheduler, as opposed to pre-computing the optimal sequence of evictions to obtain the ideal result in the least number of steps.

At those instants in time, a number of evicted pods had not yet been re-scheduled, which lead the descheduler to attempt further balancing corrections that were ultimately not necessary. A pre-computed approach would yield more efficient results but at a greater computational cost, and likely requiring a very substantial increase in Kubernetes API requests and Prometheus queries, which could present challenges in terms of scalability. Nonetheless, the Cluster State Monitor strives to minimize evictions as much as possible by using the backoff mechanisms described in section 5.4.5.

Furthermore, these inefficiencies are most evident in scenarios similar to this one, where pods are added in bulk, which does not represent most real use cases, where sources of entropy that may require corrective action, as the addition of new pods and cluster events like node reboots, mostly happen in small, gradual increments.

After all nodes are allocated 15 pods each, there are no further pod evictions, which shows the effectiveness of the BetterNode plugins' algorithm and of the mechanisms presented to prevent spurious evictions that impact service availability and performance, as well as overall cluster stability.

6.3.2 DegradedNode Plugin

The goal of this test is to validate the Cluster State Monitor's capability to aid the recovery of pods with degraded performance levels by evicting lower priority pods running on the same node.

Methodology:

This test measures the performance of a Vanetza protocol stack instance that is initially running in isolation on worker node *p5-APU* and without any Real-Time scheduling attributes. In this particular case, the stack's performance is defined as the processing latency incurred by the service while it decodes incoming MAPEM messages. These messages are produced at a rate of 10Hz by another Vanetza instance on the same network, and were chosen due to the fact that their decoding process is significantly more computationally expensive than other message types, and as such, more susceptible to interference.

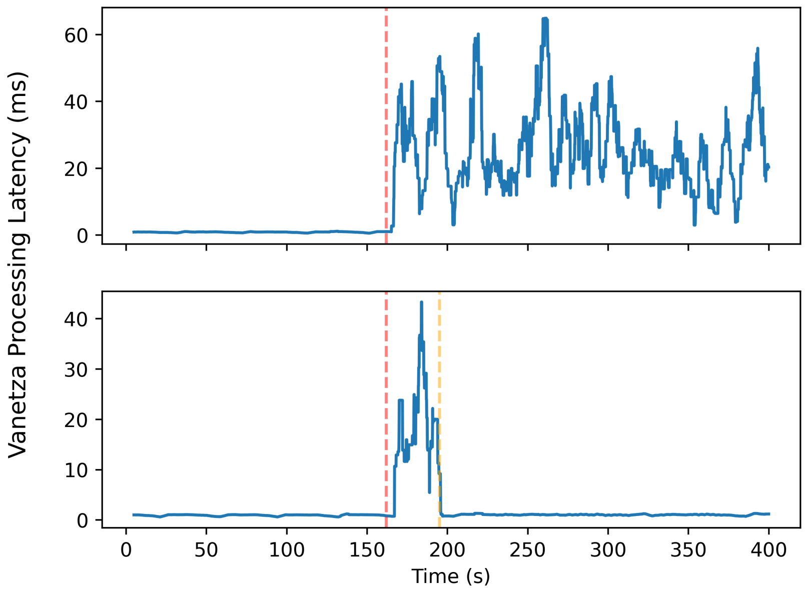


Figure 6.9: Evolution of the processing latency of Vanetza's MAPEM decoding process when faced with inter-workload interference: default behavior and proposed eviction

This initial period helps establish a baseline for the expected performance levels in this scenario. After approximately 160 seconds, an additional, lower-priority, workload is scheduled to the same node, consisting of a *stress-ng* instance that is configured to use 99% of all 4 cores on the worker node. This addition is meant to emulate the introduction of one or more CPU-intensive processes that cause substantial interference to Vanetza's performance, inducing a degraded state. For the purposes of this experiment, the degradation threshold of this metric is set at 5 milliseconds or above.

The test runs for a total of 400 seconds and is performed at two separate times, in order to generate results both with and without the presence of the Cluster State Monitor. These results are collected using a Python script that consumes the Vanetza's output via the relevant MQTT topic, and saves the respective latency information on a CSV file.

Results:

Figure 6.9 presents the evolution of the aforementioned processing latency values throughout the test period. The first subplot depicts Kubernetes' default behaviour, whereas the second subplot reflects the intervention of the custom eviction logic on the Cluster State Monitor's DegradedNode plugin.

In both cases, the latency values in the beginning assume an average of approximately 1.5 milliseconds, which remains stable until the *stress-ng* workload is added at the 160 seconds mark. Following this event, which is denoted in the figure by a red vertical line annotation, Vanetza's performance suffers a very substantial degradation, both in absolute value and in terms of stability.

In the case of the first subplot, this interference continues until the end of the experiment, since Kubernetes does not have any native mechanisms to monitor application-level metrics and perform dynamic changes based on that information. Therefore, the *stress-ng* pod is allowed to continue running on the *p5-APU* node indefinitely.

Conversely, in the second experiment, the DegradedNode plugin eventually determines that the *stress-ng* workload must be evicted in order to restore the higher-priority Vanetza workload to acceptable performance levels. This event occurs approximately 195 seconds from the start of the experiment, and is denoted with an orange vertical line annotation. Following this correction, the *stress-ng* Pod is allocated to one of the other available nodes, and the stack's processing latency rapidly returns to its previous baseline values.

6.4 Request Load-Balancing Analysis

The goal of this test is to evaluate how both load-balancing approaches (default and custom) take into account the network link latency between a client service and the server's various replicas, and the impact that those decisions have on the round-trip time of those network requests.

Methodology:

In this scenario, a client application sends Hypertext Transfer Protocol (HTTP) requests at a rate of 10Hz to a server which has 5 replicas spread throughout zones P1, P2, P3, and P4. Figure 6.10 presents the locations of the client pod and server replicas, as determined by the scheduler, and table 6.6 indicates the average link latency between the node hosting the client and the ones hosting each server replica, at the time of the test. These latency values are induced on each link using containernet's TC functionality.

Each server replica maintains a count of the number of requests it has received, and the client application records the round-trip time of every request in a CSV file.

Table 6.6: Average RTT values for each destination node

Destination Node	Average ping RTT
p1-APU	0.043 ms
p1-Jetson	0.063 ms
p2-APU	2.763 ms
p3-Jetson	3.165 ms
p4-Jetson	3.578 ms

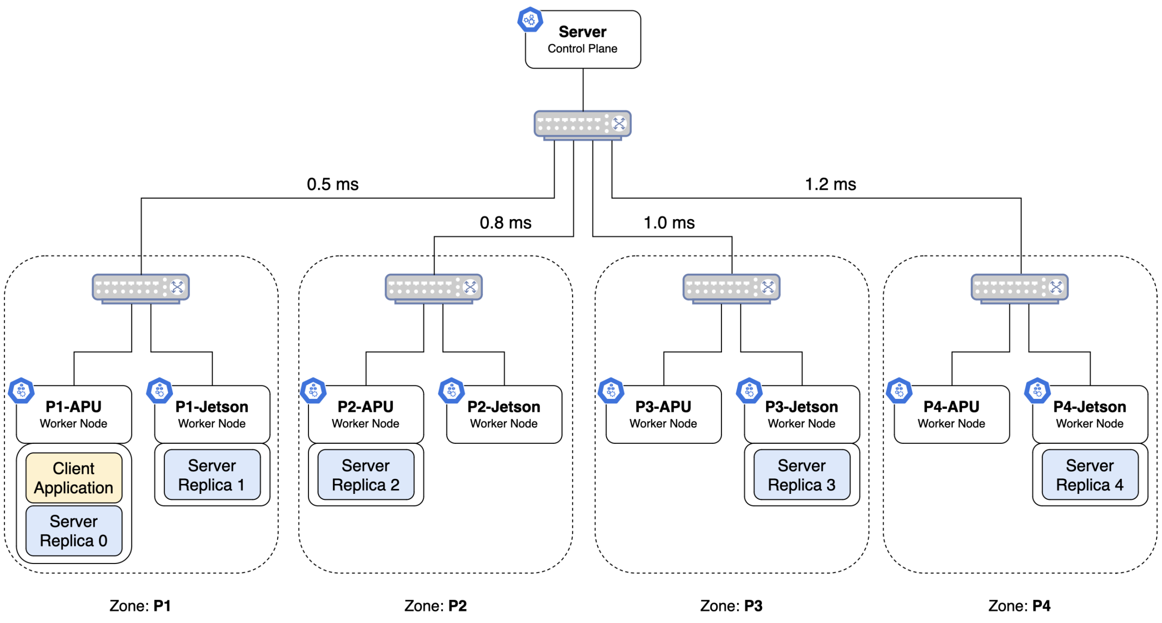


Figure 6.10: Locations of the client pod and the 5 server replicas within the cluster topology

Results

Figure 6.11 presents the number of network requests received by each of the five server replicas, in both tests. Figure 6.12 presents the cumulative distribution function of the round-trip time measurements taken by the client application, also in both tests. Results on the use of Kubernetes' default load-balancing solution show that Kube-Proxy attempts to divide the load equally amongst the available replicas of the destination service in terms of the number of pods, with each receiving an average of approximately 2000 requests. Since the load-balancer does not possess the necessary context, it does not take into account the network link latency between the client and each of the replicas. As a result, both the average and standard deviation values for the round-trip time are high.

In contrast, results for the custom load-balancing solution demonstrate that it directs the vast majority of requests, 7000, towards the closest replica, which in this case resides within the same worker node. Other replicas receive increasingly fewer requests the farther they are located from the node executing the client application, in latency terms. As a result, average and standard deviation values are much improved, as evidenced by the respective cumulative distribution function. Note, however, that the custom solution still presents high maximum values, which correspond to the portion of requests answered by the farthest replica.

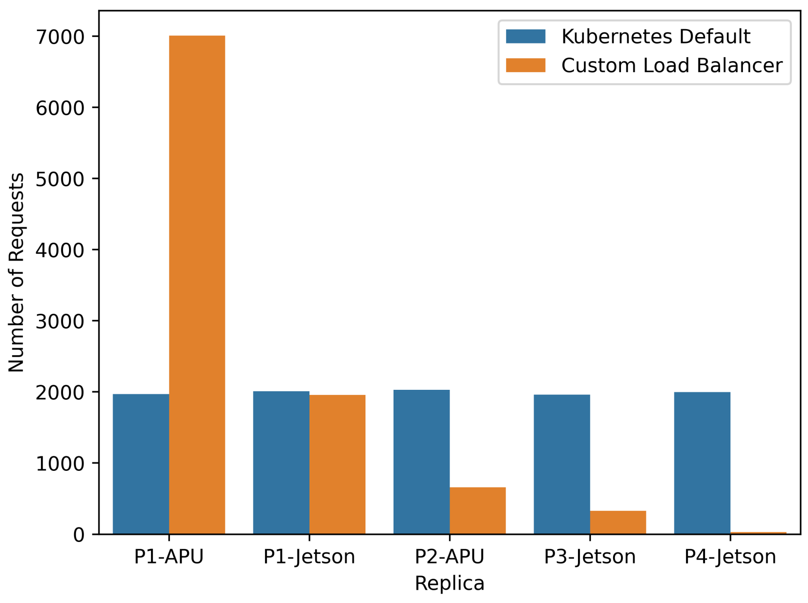


Figure 6.11: Number of network requests received by each of the five server replicas, in both tests

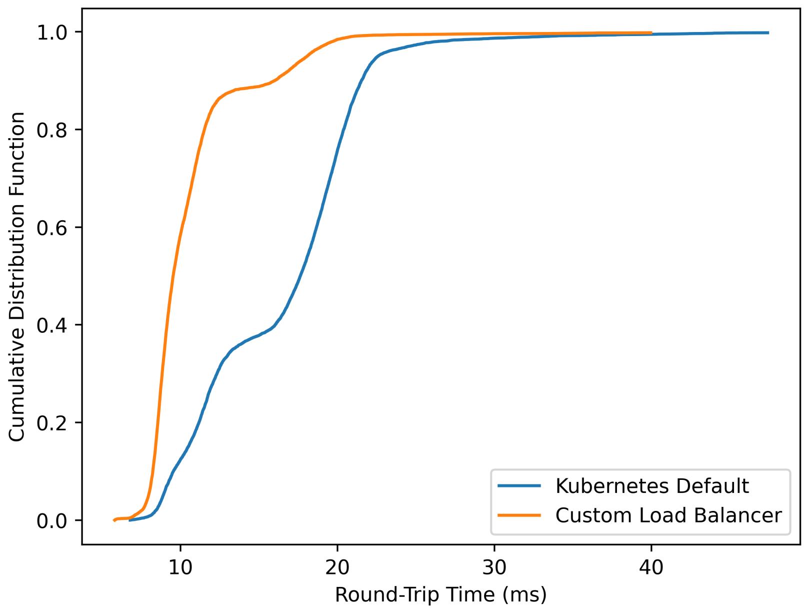


Figure 6.12: Cumulative distribution function of the round-trip time measurements taken by the client application, in both tests

6.5 Conclusions

This chapter discussed several tests and results to assess the proposal contributions of the algorithms and plugins proposed in chapter 5 in a cluster-based infrastructure with nodes with different characteristics. From the obtained results, it was shown that: (1) the proposed allocation of the services to the nodes is dynamically performed, according to the load on the nodes and the respective proposed scores; (2) the allocation is able to prioritize the nodes and leverage on the real-time requirements of the services, without interfering between different types of services; (3) the proposed approach is able to evolve over time according to the conditions of the nodes and services, and recover degraded nodes and pods by rescheduling the pods to the services; and finally, (4) it is able to load balance the requests and change the default Kubernetes algorithm in a seamless approach through plugins. Several inefficiencies of the proposed approach have been identified, and they will be the purpose of the future work of this Thesis.

CHAPTER 7

Conclusion and Future Work

This dissertation proposed orchestration strategies for smart city services with time-constrained requirements in a cloud-edge-based approach, and that are able to be automated and fault-tolerant. To reach this final result, several steps had to be addressed, which started with the study of available strategies, and the evaluation of a proposed ITS service with these approaches, and culminated on the proposal of a dynamic orchestration framework that addresses these and other constraints.

In tackling its main objective, this dissertation identified three central areas which must be considered. Namely:

- Designing applications using best practices that increase orchestration efficiency and minimize processing and communication latency.
- Executing applications using specialised configurations that mitigate interference by other processes, such as Real-Time scheduling.
- Extending existing orchestration solutions with domain-specific knowledge and new functionalities that promote a more optimal placement of services in worker nodes and the long-term stability of the cluster.

The first part of this work, in Chapter 3, provided a discussion of possible service architectures and their advantages and drawbacks when used in orchestration environments. As a case study, a new ETSI C-ITS protocol stack, *Vanetza-NAP*, was implemented using a microservice architecture that represents a paradigm shift in the way that C-V2X applications are designed, in how they interact with each other and with the VANET, and also in how they can be orchestrated in an efficient manner. The advantages of this implementation have since been demonstrated by the fact that it is being actively used in C-V2X research and demonstrations; in the production environment of ATCLL (a VANET-enabled Smart City infrastructure); and in educational settings within the University of Aveiro.

This work also consisted on the evaluation of the advantages that an orchestrator could have on the overall efficiency of a cluster and the performance of the workloads running on it, particularly when paired with orchestration-friendly software design principles. It also demonstrated that these architectural patterns can be applied successfully to critical Smart

City use cases such as Vehicular Communications, despite their traditionally monolithic nature. More specifically, it was concluded that the performance of both ZeroMQ and DDS is far superior to that of MQTT. Moreover, comparing DDS with AutoThrottle and TurboMode modes, the latency of TurboMode is the slowest, inline with the expected behaviour of this mode, since it optimizes the information batching for throughput performance over latency. the use of DDS in the containerised version shows very promising latency results, which are even emphasised in multicast communications.

Later, chapter 4 introduced some of the reasons why the cloud-focused design of existing orchestrators prevents them from attaining optimal levels of efficiency in edge-computing environments, as well as ensuring that critical services meet their upper-bound latency requirements. To address this area, a comprehensive experimental evaluation was performed in order to study the effects of several different runtime configurations on the performance of *Vanetza-NAP* in an edge computing node. The experimental results showed that time constrained services can be greatly impacted by different runtime environments and configurations. As shown, poor configuration choices can lead to an increase of around 3200% for $P_{99\%}$, when a normal service is executed in parallel with services configured with RT. The experiments also outlined that the added overhead incurred through the use of containerisation technologies in resource-constrained hardware, while not ideal, is not significant enough to outweigh their various benefits. More importantly, the results proved the overwhelming effectiveness of Real-Time scheduling strategies in guaranteeing the stability of critical tasks, even when applied to processes in containerised environments. The results showed that assigning a Real-Time priority to critical services was, in general, the most effective strategy for increasing stability and minimising interference from other processes.

Finally, this dissertation proposed, implemented and tested, in Chapters 5 and 6, an orchestration solution that leverages Kubernetes' extensibility in order to create a set of custom tools that implement domain-specific logic to enable new functionality, namely:

- **Operator -** Abstracts Kubernetes complexity from application developers by allowing them to use a single deployment descriptor file with simplified syntax; it also enables the specification of new fields concerning metrics, realtime priorities, and others.
- **Scheduler -** Adds metric awareness (both at the node and application levels), realtime priority awareness, and link latency awareness to the scheduling logic to ensure that the RT allocation of each node is minimised, and that each application is placed as close as possible to the replicas of their dependencies that are exhibiting the best performance.
- **State Monitor -** Dynamically re-schedules pods in order to correct imbalances that occur naturally over time, and to react to changes in application performance and link latency.
- **CRI Shim -** Automatically applies the correct real-time priority to processes based on the attributes specified by their developer. Additionally, this component also enables Kubernetes to manage *Systemd* services, allowing for the inclusion of older legacy applications in the orchestration process.

- **Load Balancer -** Improves the average RTT of network requests between services by dynamically shifting the load-balancing weights of each possible destination replica in order to grant precedence to replicas that are performing better, and exhibit a lower total network latency to the origin of the network request.

In short, the objective of the proposed approach and algorithms is to significantly improve the stability of time-constrained services, regardless of the level of interference posed by other processes. To validate this approach, each component was individually tested on a virtualised cluster using bespoke methodologies for each one. Results show that all the components performed within expectations and, in general, exhibited significant improvements relative to Kubernetes's default behaviour. The obtained results show that: (1) Kubernetes' Pod scheduling process is successfully extended in order to introduce new algorithms that take into account new factors such as node and application-level metrics, among others; (2) the orchestration solution is able to leverage Linux's real-time scheduling capabilities to help fulfil the time-constraints of critical services, while minimising inter-process interference; (3) the proposed approach is able to respond to imbalances that occur over time, and recover degraded nodes and pods by rescheduling sub-optimal allocations; and finally, (4) it is able to influence Kubernetes' load balancing of network requests in order to prioritise the replicas that will provide the best experience and response times, using a seamless plugin-based approach.

Therefore, the proposed orchestration approach is considered ready to be integrated in the ATCLL platform running as a production environment.

7.1 Future Work

Although this work can be considered an important step towards dynamic orchestration of services in dynamic infrastructures with changing users and services with strict requirements, there are several aspects that are missing or that can be improved. In the following, some of these aspects are considered for future work:

- **Integrate Time-Sensitive Networking technologies via CNI plugins:** Such an integration would allow for a much greater degree of control over the Quality of Service experienced by the network requests that critical applications perform.
- **Implement support for live migrations using container checkpoint and restore:** By extending the CRI shim component, the system could leverage containerd's checkpointing support to create a full copy of a running container and its current execution state. This would have the potential to greatly reduce the downtime experienced by applications when the Cluster State Monitor initiates a migration.
- **Integrate or implement latency-sensitive and highly-available storage:** Using the native classes of Kubernetes volumes, Pods can either minimize latency by reading and writing to local worker node storage, or maximize high-availability by reading and writing from a centralised storage location using protocols such as NFS. Ideally, this orchestration solution should provide a storage architecture capable of automatically

replicating local storage to other nodes in the background, thus striking a better balance between the two goals.

- **Introduce mobility of the users and services:** The mobility of the users can be a challenge, but also an opportunity to evolve to a paradigm where the services can follow the users' mobility and be deployed in the new nodes proactively. This requires that the plugins and algorithms take also into account this mobility for the allocation decisions.
- **Deploy the system in a production cluster such as ATCLL:** Such a deployment would provide further opportunities for validating the implementation and performing extensive performance and scalability tests.

References

[1] Y. Zhang, F. Lyu, P. Yang, W. Wu, and J. Gao, “Iot intelligence empowered by end-edge-cloud orchestration,” *China Communications*, vol. 19, no. 7, pp. 152–156, 2022. DOI: `10.23919/JCC.2022.9837843`.

[2] L. Mendiboure, M.-A. Chalouf, and F. Krief, “Edge computing based applications in vehicular environments: Comparative study and main issues,” *Journal of Computer Science and Technology*, vol. 34, no. 4, pp. 869–886, Jul. 2019, ISSN: 1860-4749. DOI: `10.1007/s11390-019-1947-3`. [Online]. Available: `https://doi.org/10.1007/s11390-019-1947-3`.

[3] T. S. J. Darwish and K. Abu Bakar, “Fog based intelligent transportation big data analytics in the internet of vehicles environment: Motivations, architecture, challenges, and critical issues,” *IEEE Access*, vol. 6, pp. 15 679–15 701, 2018. DOI: `10.1109/ACCESS.2018.2815989`.

[4] D. Loghin, L. Ramapantulu, and Y. M. Teo, “Towards analyzing the performance of hybrid edge-cloud processing,” in *2019 IEEE International Conference on Edge Computing (EDGE)*, 2019, pp. 87–94. DOI: `10.1109/EDGE.2019.00029`.

[5] Y. Zhang, “Mobile edge computing,” in *Mobile Edge Computing*. Cham: Springer International Publishing, 2022, pp. 9–21, ISBN: 978-3-030-83944-4. DOI: `10.1007/978-3-030-83944-4_2`. [Online]. Available: `https://doi.org/10.1007/978-3-030-83944-4_2`.

[6] Y. Lai, F. Yang, J. Su, *et al.*, “Fog-based two-phase event monitoring and data gathering in vehicular sensor networks,” *Sensors*, vol. 18, no. 1, 2018, ISSN: 1424-8220. DOI: `10.3390/s18010082`. [Online]. Available: `https://www.mdpi.com/1424-8220/18/1/82`.

[7] B. Hibat Allah and I. Abdellah, “Mec towards 5g: A survey of concepts, use cases, location tradeoffs,” *Transactions on Engineering and Computing Sciences*, vol. 5, no. 4, Sep. 2017. DOI: `10.14738/tmlai.54.3215`. [Online]. Available: `https://journals.scholarpublishing.org/index.php/TMLAI/article/view/3215`.

[8] J. Pisarov and G. Mester, “Ipsi tar july 2020 - the impact of 5g technology on life in the 21st century,” vol. 16, pp. 11–14, Jul. 2020.

[9] P. Rito, A. Almeida, A. Figueiredo, *et al.*, “Aveiro tech city living lab: A communication, sensing and computing platform for city environments,” *IEEE Internet of Things Journal*, pp. 1–1, 2023. DOI: `10.1109/JIOT.2023.3262627`.

[10] L. Liu, C. Chen, Q. Pei, S. Maharjan, and Y. Zhang, “Vehicular edge computing and networking: A survey,” *Mob. Netw. Appl.*, vol. 26, no. 3, pp. 1145–1168, Jun. 2021, ISSN: 1383-469X. DOI: `10.1007/s11036-020-01624-1`. [Online]. Available: `https://doi.org/10.1007/s11036-020-01624-1`.

[11] 5. A. Association, “C-V2X Use Cases and Service Level Requirements Volume I,” vol. I, 2021. [Online]. Available: `https://5gaa.org/wp-content/uploads/2021/01/5GAA_T-200116_TR_C-V2X_Use_Cases_and_Service_Level_Requirements_Vol_II_V2.1.pdf`.

[12] 5GAA, “C-V2X Use Cases Volume II: Examples and Service Level Requirements 5GAA Automotive Association White Paper Contents,” vol. II, 2020. [Online]. Available: `www.5gaa.org`.

[13] F. Pereira, A. S. Oliveira, N. Borges Carvalho, *et al.*, “When Backscatter Communication Meets Vehicular Networks: Boosting Crosswalk Awareness,” *IEEE Access*, vol. 8, pp. 34 507–34 521, 2020, ISSN: 21693536. DOI: `10.1109/ACCESS.2020.2974214`.

[14] M. Jutila, J. Scholliers, M. Valta, and K. Kujanpää, "ITS-G5 performance improvement and evaluation for vulnerable road user safety services," *IET Intelligent Transport Systems*, vol. 11, no. 3, pp. 126–133, 2017. DOI: `https://doi.org/10.1049/iet-its.2016.0025`. eprint: `https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-its.2016.0025`. [Online]. Available: `https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-its.2016.0025`.

[15] A. Figueiredo, P. Rito, M. Luís, and S. Sargento, "Mobility sensing and V2X communication for Emergency Services," *Mob. Netw. Appl.*, 2022, Accepted, Oct 2022. [Online]. Available: `https://www.dropbox.com/s/ris89aec3p6xu4t`.

[16] A. Voronov, *ETSI ITS G5 Geonetworking Stack*, `https://github.com/alexvoronov/geonetworking`, 2022.

[17] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-time containers: A survey," *OpenAccess Series in Informatics*, vol. 80, no. 7, pp. 1–7, 2020, ISSN: 21906807. DOI: `10.4230/OASIcs.Fog-IoT.2020.7`.

[18] A. Madej, N. Wang, N. Athanasopoulos, R. Ranjan, and B. Varghese, "Priority-based Fair Scheduling in Edge Computing," *Proceedings - 4th IEEE International Conference on Fog and Edge Computing, ICFEC 2020*, no. February, pp. 39–48, 2020. DOI: `10.1109/ICFEC50348.2020.00012`. eprint: `2001.09070`.

[19] C. Scordino, I. M. Savino, L. Cuomo, *et al.*, "Real-time virtualization for industrial automation," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2020, pp. 353–360. DOI: `10.1109/ETFA46521.2020.9211890`.

[20] F. Reghenzani, G. Massari, and W. Fornaciari, "The Real-Time Linux Kernel: A Survey on PREEMPT_RT," *ACM Comput. Surv.*, vol. 52, no. 1, Feb. 2019, ISSN: 0360-0300. DOI: `10.1145/3297714`.

[21] M. Sollfrank, F. Loch, S. Denteneer, and B. Vogel-Heuser, "Evaluating Docker for Lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 5, pp. 1–1, 2020, ISSN: 1551-3203. DOI: `10.1109/tii.2020.3022843`.

[22] B. Liu, Z. Luo, H. Chen, and C. Li, "A survey of state-of-the-art on edge computing: Theoretical models, technologies, directions, and development paths," *IEEE Access*, vol. 10, pp. 54 038–54 063, 2022. DOI: `10.1109/ACCESS.2022.3176106`.

[23] D. Matotek, J. Turnbull, and P. Lieverdink, "Startup and services," in *Pro Linux System Administration: Learn to Build Systems for Your Business Using Free and Open Source Software*. Berkeley, CA: Apress, 2017, pp. 181–216, ISBN: 978-1-4842-2008-5. DOI: `10.1007/978-1-4842-2008-5_6`. [Online]. Available: `https://doi.org/10.1007/978-1-4842-2008-5_6`.

[24] C. Bernardos, A. Rahman, J. Zuniga, L. Contreras, P. Aranda, and P. Lynch, "Network virtualization research challenges," RFC Editor, RFC 8568, Apr. 2019.

[25] S. Kaiser, M. S. Haq, A. S. Tosun, and T. Korkmaz, "Container Technologies For ARM Architecture: A Comprehensive Survey Of The State-of-the-art," *IEEE Access*, vol. 10, no. July, pp. 84 853–84 881, 2022, ISSN: 21693536. DOI: `10.1109/ACCESS.2022.3197151`.

[26] Michael Kerrisk, *Namespaces - Overview of Linux namespaces*, Accessed Sep. 23, 2022. [Online]. Available: `https://man7.org/linux/man-pages/man7/namespaces.7.html`.

[27] Michael Kerrisk, *Cgroup Namespaces: Overview of Linux cgroup namespaces*, Accessed Sep. 23, 2022. [Online]. Available: `https://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html`.

[28] Michael Kerrisk, *capabilities - Overview of Linux capabilities*, Accessed Sep. 26, 2022. [Online]. Available: `https://man7.org/linux/man-pages/man7/capabilities.7.html`.

[29] Z. Kang, R. Canady, A. Dubey, A. Gokhale, S. Shekhar, and M. Sedlacek, "A study of publish/subscribe middleware under different iot traffic conditions," in *Proceedings of the International Workshop on Middleware and Applications for the Internet of Things*, ser. M4IoT'20, Delft, Netherlands: Association for Computing Machinery, 2021, pp. 7–12, ISBN: 9781450382052. DOI: `10.1145/3429881.3430109`. [Online]. Available: `https://doi.org/10.1145/3429881.3430109`.

[30] B. Cheng, A. Rostami, and M. Gruteser, "Experience: Accurate simulation of dense scenarios with hundreds of vehicular transmitters," in *Proceedings of the 22nd Annual International Conference on*

Mobile Computing and Networking, ser. MobiCom '16, New York City, New York: Association for Computing Machinery, 2016, pp. 271–279, ISBN: 9781450342261. DOI: 10.1145/2973750.2973779. [Online]. Available: https://doi.org/10.1145/2973750.2973779.

[31] M. N. Tahir and M. Katz, "Performance evaluation of ieee 802.11p, lte and 5g in connected vehicles for cooperative awareness," *Engineering Reports*, vol. 4, no. 4, e12467, 2022. DOI: https://doi.org/10.1002/eng2.12467. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/eng2.12467. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/eng2.12467.

[32] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on preempt_rt," *ACM Computing Surveys*, vol. 52, pp. 1–36, Feb. 2019. DOI: 10.1145/3297714.

[33] R. Queiroz, T. Cruz, and P. Simões, "Testing the limits of general-purpose hypervisors for real-time control systems," *Microprocessors and Microsystems*, vol. 99, p. 104 848, 2023, ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2023.104848. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933123000947.

[34] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, pp. 351–359. DOI: 10.1109/NETSOFT.2019.8806671.

[35] A. Javed, A. Malhi, and K. Främling, "Edge computing-based fault-tolerant framework: A case study on vehicular networks," in *2020 International Wireless Communications and Mobile Computing (IWCMC)*, 2020, pp. 1541–1548. DOI: 10.1109/IWCMC48107.2020.9148269.

[36] S. Böhm and G. Wirtz, "Cloud-edge orchestration for smart cities: A review of kubernetes-based orchestration architectures," *EAI Endorsed Transactions on Smart Cities*, vol. 6, May 2022. DOI: 10.4108/eetsc.v6i18.1197.

[37] L. Liu, C. Chen, Q. Pei, S. Maharjan, and Y. Zhang, "Vehicular edge computing and networking: A survey," *Mob. Netw. Appl.*, vol. 26, no. 3, pp. 1145–1168, Jun. 2021, ISSN: 1383-469X. DOI: 10.1007/s11036-020-01624-1. [Online]. Available: https://doi.org/10.1007/s11036-020-01624-1.

[38] Cohda Wireless, *V2X-stack*, Accessed Sep. 23, 2022. [Online]. Available: https://www.cohdawireless.com/solutions/v2x-stack/.

[39] Unex, *V2X on-board unit, its-G5 stack, v2xcast*, Aug. 2022. [Online]. Available: https://www.unex.com.tw/obu-301e-sheet/.

[40] Unex, *V2X on-board unit, its-G5 stack, v2xcast2*, Aug. 2022. [Online]. Available: https://www.unex.com.tw/som-301e-sheet/.

[41] Raphael Rieb, *Vanetza*, Accessed Sep. 23, 2022. [Online]. Available: https://github.com/riebl/vanetza.

[42] Z. Kang, R. Canady, A. Dubey, A. Gokhale, S. Shekhar, and M. Sedlacek, *http://www.dre.vanderbilt.edu/ gokhale/WWW/papers/M4IoT2020.pdf*, http://www.dre.vanderbilt.edu/~gokhale/WWW/papers/M4IoT2020.pdf, Accessed: 2022-12-15.

[43] C. R. Storck and F. Duarte-Figueiredo, "A Survey of 5G Technology Evolution, Standards, and Infrastructure Associated With Vehicle-to-Everything Communications by Internet of Vehicles," *IEEE Access*, vol. 8, pp. 117 593–117 614, 2020. DOI: 10.1109/ACCESS.2020.3004779. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.3004779.

[44] 3GPP 5G-ETSI, "3GPP Technical Report 26.925," no. Release 16, 2020. [Online]. Available: https://www.3gpp.org/DynaReport/26925.htm.

[45] R. Meneguette, R. De Grande, J. Ueyama, G. P. R. Filho, and E. Madeira, "Vehicular edge computing: Architecture, resource management, security, and challenges," *ACM Comput. Surv.*, vol. 55, no. 1, Nov. 2021, ISSN: 0360-0300. DOI: 10.1145/3485129. [Online]. Available: https://doi.org/10.1145/3485129.

[46] V. Charpentier, N. Slamnik-Krijestorac, and J. Marquez-Barja, "Latency-aware c-its application for improving the road safety with cam messages on the smart highway testbed," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2022, pp. 1–6. DOI: 10.1109/INFOCOMWKSHPS54753.2022.9798350.

[47] Y. Zhao, W. Wang, Y. Li, C. Colman Meixner, M. Tornatore, and J. Zhang, "Edge Computing and Networking: A Survey on Infrastructures and Applications," *IEEE Access*, vol. 7, no. August, pp. 101 213–101 230, 2019, ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2927538.

[48] L. Liu, C. Chen, Q. Pei, S. Maharjan, and Y. Zhang, "Vehicular edge computing and networking: A survey," *Mob. Netw. Appl.*, vol. 26, no. 3, pp. 1145–1168, Jun. 2021, ISSN: 1383-469X. DOI: 10.1007/s11036-020-01624-1. [Online]. Available: https://doi.org/10.1007/s11036-020-01624-1.

[49] M. Holenderski, "Real-time system overheads: A literature overview," *Computer Science Report*, vol. 8, p. 26, 2008.

[50] *Cgroups - linux programmer's manual*, https://man7.org/linux/man-pages/man7/cgroups.7.html, Accessed: 2022-05-30.

[51] *Cpu controllers - linux kernel documentation*, https://kernel.googlesource.com/pub/scm/linux/kernel/git/glommer/memcg/+/cpu_stat/Documentation/cgroups/cpu.txt, Accessed: 2022-05-30.

[52] Yusuke Endoh, *CPU-intensive Ruby/Python code runs slower on default-configured Docker*, Accessed May. 23, 2022. [Online]. Available: http://mamememo.blogspot.com/2020/05/cpu-intensive-rubypython-code-runs.html.

[53] Michael Larabel, *Linux 5.16 Loosens The Spectre Defaults Around SSBD / STIBP*, Accessed Oct. 21, 2022. [Online]. Available: https://www.phoronix.com/news/Linux-5.16-Spectre-SECCOMP-To-P.

[54] A. Fahs, "Proximity-aware replicas management in geo-distributed fog computing platforms," *Operating Systems [cs.OS]. Université de Rennes*, vol. 1, 2020.

Appendix A

This appendix provides a concise listing of all the various configuration options supported by the *Vanetza-NAP* vehicular network protocol stack service developed within the scope of this dissertation and introduced in section 3.4, as well as a brief description of their functions. These options are grouped into two separate categories: Table 1 summarises the general configuration options for the service as a whole, while Table 2 contains options that are specific to each C-ITS message type (CAM, DENM, CPM, VAM, SPATEM, MAPEM), using CAMs as an example.

.ini file key	**Description**
general.interface	Network interface where the ETSI messages are exchanged
general.local_mqtt_broker	Local MQTT Broker's IP address or DNS name
general.local_mqtt_port	Local MQTT Broker's Port
general.remote_mqtt_broker	Remote MQTT Broker's IP address or DNS name
general.remote_mqtt_port	Remote MQTT Broker's Port
general.remote_mqtt_username	Remote MQTT Broker's Auth Username
general.remote_mqtt_password	Remote MQTT Broker's Auth Password
general.gpsd_host	Global Positioning System (GPS) position provider host
general.gpsd_port	GPS position provider port
general.prometheus_port	Port on which Vanetza exposes metrics
general.rssi_enabled	Enable discovering the Received Signal Strength Indication (RSSI) value associated with each inbound ITS-G5 message by interfacing with the kernel
general.ignore_own_messages	Don't capture or decode messages originating from the station itself
general.ignore_rsu_messages	Ignore messages from RSUs - Usually set on RSUs
general.to_dds_key	Unix System V (SysV) Message Queue Key which Vanetza uses to send JSON to be published in DDS topics
general.from_dds_key	SysV Message Queue Key which Vanetza uses to receive JSON from DDS topics
general.enable_json_prints	Print a JSON representation of incoming messages on the terminal/logs
station.id	ETSI Station ID field
station.type	ETSI Station Type field
station.mac_address	Virtual Mac Address used as the source on L2 ethernet headers
station.beacons_enabled	Send GeoNetworking Beacons every 3 seconds
station.use_hardcoded_gps	Use hardcoded GPS coordinates instead of information provided by a GPS module
station.latitude	Hardcoded GPS latitude - Usually set on static RSUs
station.longitude	Hardcoded GPS longitude - Usually set on static RSUs
station.length	Vehicle lenght in meters
station.width	Vehicle width in meters
cam.full_topic_in	MQTT/DDS topic from which Vanetza receives JSON CAMs in the full ETSI spec format
cam.full_topic_out	MQTT/DDS topic to which Vanetza sends JSON CAMs in the full ETSI spec format
vam.full_topic_in	MQTT/DDS topic from which Vanetza receives JSON VRU Awareness Message (VAM)s in the full ETSI spec format
vam.full_topic_out	MQTT/DDS topic to which Vanetza sends JSON VAMs in the full ETSI spec format
cam.own_topic_out	MQTT/DDS topic to which Vanetza sends a JSON representation of the hardcoded CAMs in the simple format
cam.own_full_topic_out	MQTT/DDS topic to which Vanetza sends a JSON representation of the hardcoded CAMs in the full ETSI spec format

Table 1: Available configuration options for Vanetza-NAP

.ini file key	**Description**
cam.enabled	Enable the CAM module
cam.mqtt_enabled	Enable publishing and subscribing to MQTT topics
cam.mqtt_time_enabled	Enable publishing to the respective time topic for performance measurement purposes
cam.dds_enabled	Enable publishing and subscribing to DDS topics
cam.periodicity	Periodicity with which to send the default CAM, in milliseconds
cam.topic_in	MQTT/DDS topic from which Vanetza receives JSON CAMs to encode and send
cam.topic_out	MQTT/DDS topic to which Vanetza sends JSON CAMs that were received and decoded
cam.topic_time	MQTT/DDS topic to which Vanetza sends JSON CAMs that were received in vanetza/in/cam and sent through the WAVE interface
cam.udp_out_addr	Address of the User Datagram Protocol (UDP) server to which Vanetza sends decoded JSON CAMs, in order to minimize communication latency
cam.udp_out_port	Port of the UDP server to which Vanetza sends decoded JSON CAMs, in order to minimize communication latency

Table 2: Available configuration options for each message type (CAM in this example)

Appendix B

FogService Custom Resource Reference

This appendix provides a comprehensive reference to the schema and capabilities of the FogService Custom Resource Definition developed within the scope of this dissertation and introduced in section 5.4.2. The following tables list all of the fields that comprise the schema, as well as their type, requiredness, and detailed description of their function, including any relevant examples.

FogServiceSpec

Field	Description
image *string* *Optional*	Name of the image used for the container. Follows the industry standard notation and supports any publicly accessible container image registry. If no registry URL is specified, Docker Hub is assumed. *Examples:* • nginx:v3.0 • code.nap.av.it.pt/mobility-networks/vanetza:latest
service *FogServiceSystemdSpec* *Optional*	Specification of the repository containing the legacy application to run as a Systemd service, and the respective installation playbook
hostNetwork *bool* *Optional*	Configures the Pod to use Host Networking Mode instead of the default virtual interface generated by the CNI provider. This enables applications to access and use the host's network interfaces, which allows for a greater degree of communication with external hosts and services when compared with simple port-forwarding. Nevertheless, this mode should be used judiciously, as it does not provide as much security and isolation to the application and the overall system.

	Default value: false *Use Case Examples:* • Applications that exchange L2 traffic with other hosts, such as Vanetza-NAP. • Applications that monitor or sniff traffic on an interface.
hostSharedMem *bool* *Optional*	Configures the Pod to use the Host's IPC namespace in order to enable direct communication between processes using non-IP based Unix IPC mechanisms such as Message Queues. This feature should increase the communication performance of two applications running within the same node. However, this mode introduces a significant security risk by decreasing the level of isolation between applications and the overall system. Therefore, it should only be used in very specific, controlled, circumstances.
ports *[]FogServicePortsSpec* *Optional*	Specification of the network ports used by the application, and their respective port forwarding mappings, if applicable.
podLabels *[]string* *Optional*	List of labels applied to all pods/replicas of the application, so as to help identify and group similar types of services, and simplify the specification of inter-pod affinity contraints.
loadBalancing *FogServiceLoadBalancingSpec* *Optional*	Specification of parameters to tune the behaviour of the network request load balancing algorithm to the specific needs of the application.
scheduling *FogServiceSchedulingSpec* *Optional*	Specification of application-specific constraints and/or preferences when allocating pods to worker nodes.
realTimePriority *[]FogServiceRealTimePrioritySpec* *Optional*	Specification of real-time scheduling attributes to one or more critical sub-processes.
resources *FogServiceResourcesSpec* *Optional*	Specification of expected resource usage levels and corresponding limits, when applicable.
security	Configuration of security exceptions for applications that

FogServiceSecuritySpec *Optional*	require special access.
metrics *FogServiceMetricsSpec* *Optional*	Specification of application-level metrics and their expected range of values, enabling comprehensive monitoring of the service's real-time performance and historical trends.
liveness *FogServiceLivenessSpec* *Optional*	Specification of continuous application-level liveness checks for effective monitoring of service state.
volumes *FogServiceVolumesSpec* *Optional*	Mapping of host directories to volumes within application containers, enabling persistent data storage and increased sharing.
config *FogServiceConfigSpec* *Optional*	Specification of configuration files and environment variables in order to fine-tune application behaviour.

FogServiceSystemdSpec

Field	**Description**
ProjectID *int* *Mandatory*	Numerical identifier of the GitLab project where the service is stored. *Example:* • 485
Playbook *string* *Mandatory*	Relative path within the repository's directory structure to the Ansible playbook that must be downloaded and executed in order to install the service an its dependencies. *Example:* • deployment/installation.yml
Reference *string* *Mandatory*	Reference to the Git branch, tag, or commit which contains the desired version of the service and its installation playbook. *Examples:* • master • v3.0 • 79473e2

FogServicePortsSpec

Field	Description
Protocol *string* *Mandatory*	Transport-layer protocol used when exchanging network traffic. *Permitted Values:* • Transmission Control Protocol (TCP) • UDP
Port *string* *Mandatory*	Can be used as a simple declaration of (one of) the network port(s) used for inter-service communications within Kubernetes' virtual Local Area Network (LAN) by the application. Or, alternatively, to specify a port-forwarding mapping from a port on the host to one on the respective container. *Examples:* • 1883 # Simple declaration • 1701:7117 # Mapping of the host's port 1701 to port 7117 on the container

FogServiceLoadBalancingSpec

Field	Description
LatencyWeight *string* *Mandatory*	Relative weight that should be attributed to the latency between nodes when load-balancing network requests between the application and its dependencies. *Permitted Values:* 0.0 to 1.0 The sum of both weights should equal 1.
MetricsWeight *string* *Mandatory*	Relative weight that should be attributed to the current performance of each destination replica, as measured by their exposed application-level metrics, when load-balancing network requests between the application and its dependencies. *Permitted Values:* 0.0 to 1.0 The sum of both weights should equal 1.

FogServiceSchedulingSpec

Field	Description
Priority *int* *Mandatory*	Relative priority of the application in relation to others. Applications with higher priorities are given precedence in the pod scheduling queue and may preempt lower-priority pods that are already running, if necessary. *Permitted Values:* 1 to 1000000000 Higher values represent higher priority
Deschedulable *bool* *Optional*	Indicates that the Cluster State Monitoring tool can evict pods of the application in order to achieve a balanced state. This option should be disabled in the case of applications that can not be migrated after they have started their execution. *Default Value:* true
Zones *FogServiceZoneSpec* *Optional*	Specification of the topology zones where the application should be running, and associated constraints.
Replicas *FogServiceReplicasSpec* *Optional*	Specification of constraints regarding the number of replicas that should be instantiated for the application.
Regions *FogServiceRegionSpec* *Optional*	Specification of the topology regions (sets of zones) where the application should be running, and associated constraints.
PreferNodesWith *[]FogServiceAffinitySpec* *Optional*	Specification of preferred worker nodes for the application through the use of node labels and associated constraints.
Dependencies *[]FogServiceDependenciesSpec* *Optional*	Specification of other services on which the application depends in order to perform its function, and associated constraints.
AvoidPodsWith *[]FogServiceAffinitySpec* *Optional*	Specification of other services which should not be running within the same worker node as the application, for performance and/or functional reasons, among others.

FogServiceZoneSpec

Field	Description
List *[]string* *Optional*	List of topology zones that should each run an instance of the application. *Example:* • ["p1", "p2", "p3"]
All *bool* *Optional*	Indicates that every available topology zone should run an instance of the application, as opposed to explicitly listing each one using the previous field. *Default value:* false
Strict *bool* *Optional*	Indicates that each instance of the application should run exclusively on nodes belonging to its respective topology zone. This option should be used for services which can not fulfil their function if they are executed outside their intended topology zone, and therefore should remained in an unschedulable state until it can be allocated to a suitable node. *Default value:* false

FogServiceReplicasSpec

Field	Description
Minimum *int* *Mandatory*	Minimum number of replicas of each instance of the application that should be instantiated. This guarantees that the HorizontalPodAutoscaler will not scale down the deployment beyond this pre-specified point.
Maximum *int* *Mandatory*	Maximum number of replicas of each instance of the application that should be instantiated. This guarantees that the HorizontalPodAutoscaler will not scale up the deployment beyond this pre-specified point.
MinAvailable *int* or *string* *Optional*	Minimum amount of replicas of each instance of the application that should remain active at all times. This guarantees that tools like the Cluster State Monitor are not able to negatively affect the availability of the service by migrating an excessive number of replicas simultaneously.

	Examples: • 3 • "15%"
MaxUnavailable *int* or *string* *Optional*	Maximum amount of replicas of each instance of the application that can be in an inactive state at any given time. This guarantees that tools like the Cluster State Monitor are not able to negatively affect the availability of the service by migrating an excessive number of replicas simultaneously. *Examples:* • 7 • "85%"

FogServiceRegionSpec

Field	**Description**
List *[]string* *Mandatory*	List of topology regions that should each run an instance of the application. *Example:* • ["downtown", "central-park"]
Strict *bool* *Optional*	Indicates that each instance of the application should run exclusively on nodes belonging to its respective topology region. This option should be used for services which can not fulfil their function if they are executed outside their intended topology region, and therefore should remain in an unschedulable state until it can be allocated to a suitable node. *Default value:* false

FogServiceAffinitySpec

Field	**Description**
Key *string* *Mandatory*	Name of the label whose values will be constrained *Example:* • beta.kubernetes.io/arch
Values	List of possible label values that fulfil the desired affinity

[]string *Mandatory*	constraint. *Example:* • ["amd64", "arm64"]
Required *bool* *Optional*	Indicates that this label value constraint represents a mandatory affinity restriction that must be satisfied in order for the application to be executed, as opposed to representing a mere preference that informs the pod scheduling process. In the case of *PreferNodesWith*, this field will ensure that only nodes with the selected labels will be considered for pod allocation. In the case of *AvoidPodsWith*, nodes that run pods with the selected labels will be filtered out of the scheduling process. *Default value:* false

FogServiceDependenciesSpec

Field	**Description**
Name *string* *Mandatory*	Name of the service on which the application depends. If both applications are deployed as zone/region-scoped instances, the FogService operator will automatically select the appropriate instance of the dependency. *Example:* • vanetza-nap
RequiredOnSameNode *bool* *Optional*	Indicates that this dependency relation represents a mandatory restriction that must be satisfied in order for the application to be executed, as opposed to representing a mere preference that informs the pod scheduling process. *Default value:* false
Weight *string* *Mandatory*	Weight attributed to a particular dependency when comparing the suitability of a Pod allocation in scheduler and descheduler plugins. *Permitted Values:* 0.0 to 1.0 The sum of all weights should equal 1.

FogServiceRealTimePrioritySpec

Field	Description
ProcessID *int* *Optional*	PID of the desired process within the PID namespace of the container. Since this namespace is isolated from the host system, this value remains consistent between runs and on different hosts.
RelativePriority *int* *Optional*	Relative priority value of the processes in relation to the other real-time processes in the application. *Permitted Values:* 1 to 1000000000 Higher values represent higher priority
Runtime *int* *Optional*	Specifies the runtime parameter for SCHED_DEADLINE policies. Represents the number of microseconds that the process requires in order to complete its task in each period timeframe.
Period *int* *Optional*	Specifies the period parameter for SCHED_DEADLINE policies. Represents the number of microseconds until the processes' function needs to be repeated.
Deadline *int* *Optional*	Specifies the deadline parameter for SCHED_DEADLINE policies. Represents the maximum number of microseconds by which the process needs to complete its task, in each period timeframe.
ProcessName *string* *Optional*	Name of the desired process. In cases where the it is is impractical or infeasible to specify an exact PID, the system is capable of identifying the processes by their names. This field can state either the complete process name, or a subset of it. *Example:* • python3

FogServiceResourcesSpec

Field	Description
Requests *FogServiceRequestsSpec*	Describes the minimum amount of compute resources required to successfully run the application. This

Optional	information is taken into account during the scheduling process and during execution, in order to guarantee that the node can always allocate at least that amount of resources to the service.
Limits *FogServiceRequestsSpec* *Optional*	Describes the maximum amount of compute resources that the application should be allowed to use. Kubernetes enforces these limits to avoid the starvation of other lower-priority tasks.

FogServiceRequestsSpec

Field	Description
Memory *string* *Optional*	Amount of system memory usage, measured in bytes. *Example Values:* • 128974848 • 129M • 2.4Gi
CPU *string* *Optional*	Amount of CPU allocation per unit of time to processes that do not have realtime scheduling priorities, measured in Kubernetes CPU units. Using this metric, 1.0 is equivalent to 1 physical CPU core. *Example Values:* • 1.0 (one physical CPU core) • 0.2 (20% of a physical CPU core) • 100m (one hundred millicores)
RealtimeCPU *string* *Optional*	Amount of CPU allocation per unit of time specifically to realtime-enabled processes, measured in Kubernetes CPU units. Using this metric, 1.0 is equivalent to 1 physical CPU core. *Example Values:* • 1.0 (one physical CPU core) • 0.2 (20% of a physical CPU core) • 100m (one hundred millicores)

FogServiceSecuritySpec

Field	Description
Capabilities *[]string* *Optional*	List of additional system capabilities that the container requires and that are not included in the default security context. This field is crucial for applications that require specialised permissions to access and make changes to system resources such as, for example, network interfaces. *Example Values:* • CAP_SYS_ADMIN • CAP_SYS_NICE • CAP_NET_RAW
Sysctls *[]FogServiceSysctlsSpec* *Optional*	List of sysctl configurations that apply to the Pod. The sysctl interface allows an administrator to modify kernel parameters at runtime and within the Pod's namespace/scope.

FogServiceSysctlsSpec

Field	Description
Name *string* *Mandatory*	Name of the particular kernel parameter that should be configured. *Example Values:* • net.ipv4.ip_forward • net.ipv4.tcp_fin_timeout • vm.max_map_count • kernel.msgmax
Value *string* *Mandatory*	Respective value that should be configured within the Pod's namespace/scope

FogServiceMetricsSpec

Field	Description
Port *int* *Mandatory*	Internal port number where the Pod exposes its current metric values in Prometheus format.

List *[]FogServiceMetricsListSpec* *Mandatory*	List of relevant metrics that should be monitored by the orchestrator, and their respective normalcy thresholds.

FogServiceMetricsListSpec

Field	Description
Query *string* *Mandatory*	Query that returns the desired metric value from Prometheus. This query is specified in PromQL format (as opposed to simply specifying the metric name), so that developers have the freedom to configure complex queries that take into account time ranges and other advanced PromQL operations. *Note*: The query must include at least one metric/label selector (ie: *{}*) so that the orchestrator can identify where to insert a filter for the pod names of each replica. This is required even when the query itself does not otherwise need it, in which case the label selector should be left empty. *Example Values:* • frames_per_second{} • num_of_received_requests{endpoint="process"} • histogram_quantile(0.9, rate(http_request_duration _seconds[10m])) • rate(observed_packets_latency_total{message="mapem" ,direction="rx"}[2m]) / rate(observed_packets_count_total {message="mapem",direction="rx"}[2m])
AlertName *string* *Optional*	Title of the alert (and corresponding Slack message) that is triggered by the Prometheus AlertManager when the metric value enters degradation levels.
Weight *string* *Optional*	Weight attributed to a particular metric when comparing the health and suitability of a set of replicas in scheduler and descheduler plugins. *Permitted Values:* 0.0 to 1.0 The sum of all weights should equal 1.
HigherIsBetter *bool*	Indication that higher values are considered better (ie: indicative of higher performance or lower utilisation)

Optional	for this particular metric. By default, the orchestrator and its scheduler plugins expect the opposite behaviour, where lower metric values are considered better. *Default value:* false
Autoscale *FogServiceAutoscaleSpec* *Optional*	Configuration of how the autoscaling behaviour should react to changes to values of this particular metric.
Thresholds *FogServiceThresholdsSpec* *Mandatory*	Configuration of degradation thresholds for metric values.

FogServiceAutoscaleSpec

Field	**Description**
TargetValue *string* *Mandatory*	Average value that Kubernetes' autoscaling mechanism should target for this particular metric. If this value degrades the orchestrator will automatically spawn additional replicas of the service, provided that the resulting number still meets the minimum and maximum constraints specified in the scheduling section of the *FogService*. Conversely, if the value improves, some of the existing replicas will gradually be evicted. This field is placed within FogServiceAutoscaleSpec for readability and future-proofing purposes.

FogServiceThresholdsSpec

Field	**Description**
DegradedPerformance *FogServicePerformanceSpec* *Optional*	Metric value threshold after which the performance of a given replica is considered degraded by the orchestrator. This information is used to generate alerts and to influence the operation of the Scheduler and Cluster State Monitor.
CriticalPerformance *FogServicePerformanceSpec*	Metric value threshold after which the performance of a given replica is considered critically degraded by the orchestrator.

Optional	Within the scope of this dissertation, this information is used only to generate alerts, but may, in the future, also be used to influence the operation of the Scheduler and Cluster State Monitor.
CriticalFailure *FogServicePerformanceSpec* *Optional*	Metric value threshold after which the a given replica is considered to have hit an unrecoverable error or state. This information is used to generate alerts and to automatically evict the affected pod, if desired. This mechanism essentially serves as an additional liveness check, used to detect and correct situations where an unforeseen error prevents the normal execution of the service but does not cause the Pod's main process to exit, thus avoiding detection by Kubernetes.

FogServicePerformanceSpec

Field	Description
Alert *string* *Optional*	Additional context that can be included in the body of the respective Alertmanager/Slack alert.
Value *string* *Mandatory*	Respective threshold value, in absolute terms.

FogServiceFailureSpec

Field	Description
Alert *string* *Optional*	Additional context that can be included in the body of the respective Alertmanager/Slack alert.
Restart *bool* *Optional*	Indication if the orchestrator should evict the corresponding Pod when the CriticalFailure event is triggered, thus effecting a restart of the replica/service. *Default value:* false

Value *string* *Mandatory*	Respective threshold value, in absolute terms.

FogServiceLivenessSpec

Field	Description
InitialDelaySeconds *int* *Mandatory*	Number of seconds to wait for before conducting the first liveness check after the Pod is initiated, so as to grant sufficient time for all the processes to start up and stabilise.
PeriodSeconds *int* *Mandatory*	Periodicity between liveness checks, in seconds.
HTTP *FogServiceHTTPSpec* *Optional*	Configuration options for an HTTP-based liveness check, which validates that an API endpoint within the Pod successfully establishes an HTTP session and returns a 200 status code.
TCP *FogServiceTCPSpec* *Optional*	Configuration options for a Socket-based liveness check, which validates that a TCP server within the Pod successfully establishes a connection.
Exec *FogServiceExecSpec* *Optional*	Configuration options for a Shell-based liveness check, which validates that a given command successfully executes inside the Pod's environment, with a return code of *0*.

FogServiceHTTPSpec

Field	Description
Path *string* *Mandatory*	URL path of the desired API endpoint. *Example Value:* • "/status"
Port	Port which the HTTP server uses to listen to incoming

int *Mandatory*	connections.

FogServiceTCPSpec

Field	Description
Port *int* *Mandatory*	Port which the TCP server uses to listen to incoming connections.

FogServiceExecSpec

Field	Description
Command *[]string* *Mandatory*	Shell command that should be executed within the Pod's environment. *Example Value:* • ["/bin/pidof", "python3", ">", "/dev/null"]

FogServiceVolumesSpec

Field	Description
Mapped *[]string* *Optional*	List of mappings between the container and the host's filesystems. This field uses the typical Docker format, which first specifies the container path followed by the host path, separated by a *:* character. This field is placed within FogServiceVolumesSpec for future-proofing purposes, since the Future Work section of this dissertation identifies other potential backends for volumes. *Example Value:* • ["/output:/home/user/service-output"]

FogServiceConfigSpec

Field	Description
Global	Environment variables and configuration files that apply to

FogServiceGlobalSpec *Optional*	every zone, unless overridden by specific configurations.
Specific *[]FogServiceSpecificSpec* *Optional*	Environment variables and configuration files that apply to specific zones, overriding any overlapping global configurations.

FogServiceGlobalSpec

Field	**Description**
Environment *[]string* *Optional*	List of environment variables that should be made available to the Pod. This field uses the typical Docker format, which first specifies the variable name path followed by its desired value, separated by a = character. *Example Value:* • ["VANETZA_LOCAL_MQTT_PORT=1883"]
Files *[]FogServiceFilesSpec* *Optional*	List of configuration files that should be passed through to the container's filesystem, and their respective contents.

FogServiceSpecificSpec

Field	**Description**
Name *string* *Mandatory*	Name of the specific Topology Zone to which the environment variables and configuration files should be applied.
Environment *[]string* *Optional*	List of environment variables that should be made available to the Pod. This field uses the typical Docker format, which first specifies the variable name path followed by its desired value, separated by a = character. *Example Value:* • ["VANETZA_LOCAL_MQTT_PORT=1883"]
Files *[]FogServiceFilesSpec* *Optional*	List of configuration files that should be passed through to the container's filesystem, and their respective contents.

FogServiceFilesSpec

Field	**Description**
Path *string* *Mandatory*	Absolute path where the configuration file should be placed within the container's filesystem.
Contents *string* *Mandatory*	Multi-line string containing the raw contents of the respective configuration file (typically in yaml, json, or ini formats).