FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Vahid Mohsseni

DESIGN AND IMPLEMENTATION OF A NEXT-GENERATION TASK ORCHESTRATION PLATFORM FOR EDGE COMPUTING WITH RUST LANGUAGE

Master's Thesis
Degree Programme in Computer Science and Engineering
June 2023

Mohsseni V. (2023) Design and Implementation of a Next-Generation Task Orchestration Platform for Edge Computing with Rust Language. University of Oulu, Degree Programme in Computer Science and Engineering, 67 p.

ABSTRACT

Edge computing has evolved a favorable paradigm for processing data nearer to the point of its origin, enabling low-latency and real-time applications in various domains. However, existing orchestration platforms, such as Kubernetes, face limitations when applied to edge computing scenarios due to the unique challenges posed by resource-constrained and dynamic edge environments. This thesis focuses on addressing these limitations and developing an alternative solution, a specialized orchestration platform for edge computing, named the Resilient On-demand Distributed Systems (RODS). Key research questions that drive this study contain exploring the boundaries of state-of-the-art orchestration systems and alternatives, designing and implementing RODS, and addressing its challenges. Through an extensive review of related work and the utilization of the Rust language, the methodology chapter presents the design and architecture view, implementation details, fault tolerance mechanisms, and potential future enhancements of RODS. The findings highlight the effectiveness of RODS in addressing the limitations of existing orchestration platforms, providing enhanced resource allocation, fault tolerance, and scalability in edge environments. Additionally, the study explores the generalizability of RODS in cloud environments by proposing adapting container-based technologies for isolation. The thesis concludes with a discussion of the overall impact and contribution of the study, emphasizing how RODS fills the gaps in knowledge, advances edge computing research and practice, and offers practical implications for future development and deployment.

Keywords: RODS, Raft, Consensus, Kubernetes, edgeIO, Tokio Library, Socket, RocksDB

Mohsseni V. (2023) Reunalaskentaan soveltuva seuraavan sukupolven Rust-pohjainen resurssien orkestrointialusta. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 67 s.

TIIVISTELMÄ

Reunalaskennassa tietojen käsittely suoritetaan lähellä tiedon tuottajaa mahdollistaen viiveettömät ja reaaliaikaiset sovellukset eri aloilla. Olemassaolevat resurssien orkestrointiin suunnitellut alustat, kuten Kubernetes, eivät toimi optimaalisesti dynaamisissa reunaympäristöissä. Tämä opinnäytetyö keskittyy analysoimaan reunalaskennan aiheuttamia haasteita orkestroinnille ajantasaisilla työkaluilla. Analyysin perusteella työssä ehdotetaan vaihtoehtoista, reunalaskentaan erikoistunutta orkestrointiratkaisua, Resilient On-demand Distributed Systems (RODS). Työn metodologiaosuudessa esitetään uuden ratkaisun suunnittelun lähtökohdat sekä kehitetty arkkitehtuuri. Lisäksi analysoidaan toteutuksen ratkaisut sekä vikasietomekanismit. Työssä analysoidaan myös toteutetun ratkaisun skaalautuvuutta. Ratkaisu toteutettiin Rust-kielellä. Työn validointiosuudessa osoitetaan RODSin tehokkuus vasten olemassaolevia orkestrointiratkaisuja resurssien allokoinnin, vikasietoisuuden ja skaalautuvuuden suhteen. Tutkimuksessa selvitetään RODS:in yleistettävyysominaisuuksia sekä eristettävyyttä konttipohjaisilla teknologioilla pilviympäristöissä. Lopuksi työssä analysoidaan tutkimuksen vaikuttavuutta sekä kontribuutioita tieteen ja tekniikan tilaan, Työn kirjallinen osuus tuo tuoretta tietoa reunalaskennan järjestelmien analysointiin. Työssä on suunniteltu ja toteutettu uusi ratkaisu reunan resurssien optimointiin, mahdollistaen selkeitä parannuksia reunalaskennan dynaamisten arkkitehtuurien suunnitteluun ja toteutukseen.

Avainsanat: resurssien orkestrointialusta, Raft, konsensus, skaalautuvuus, Kubernetes, Rust ohjelmointikieli

TABLE OF CONTENTS

ABSTRACT
TIIVISTELMÄ
TABLE OF CONTENTS
FOREWORD
LIST OF ABBREVIATIONS AND SYMBOLS
1. INTRODUCTION 7
2. RELATED WORKS 15
   2.1. Edge Computing 15
   2.2. Distributed Systems: Evolution and Development 18
   2.3. Fog Computing and Orchestration 19
   2.4. Orchestration Challenges 20
   2.5. Container Orchestration Tools 21
   2.6. Related Academic Works 23
   2.7. Summary 28
3. METHODOLOGY 29
   3.1. Design and Architecture Overview 29
   3.2. Rust Programming Language 31
   3.3. Summary 37
4. IMPLEMENTATION 38
   4.1. Components 38
   4.2. Nodes Formation 41
   4.3. Fault Tolerance 42
   4.4. Summary 44
5. DISCUSSION 45
   5.1. Brief Comparison with State-Of-The-Art 45
   5.2. Address to Practical Challenges 46
   5.3. Limitations and Future Enhancements 49
      5.3.1. Consensus 49
      5.3.2. Scheduling Policies 50
      5.3.3. Security Enhancement 51
      5.3.4. A New Framework for Worker Nodes 52
   5.4. Generalizability and Applicability 52
   5.5. Summary 54
6. CONCLUSION 55
   6.1. Impact and Contribution 55
   6.2. Summary 56
7. REFERENCES 58
8. APPENDICES 63

FOREWORD

I am delighted to present this thesis which aims to address the challenges and limitations of edge computing orchestration by developing a specialized platform, the Resilient On-demand Distributed Systems (RODS).

Throughout the various stages of this research, the aim was to explore the value and benefits of adopting RODS in edge computing environments. The thesis delves into developing a platform, resiliency and reliability mechanisms, scalable operations, and high availability approaches within network and hardware constraints.

I would like to express my sincere appreciation to my supervisor, Dr. Susanna Pirttikangas, for her invaluable support throughout the entire research process. Her expertise, insightful feedback, and unwavering commitment to academic excellence have been instrumental in shaping this thesis. I am also grateful to Dr. Lauri Loven, my second supervisor, for his valuable contributions and constructive inputs.

Additionally, I would like to unfold my heartfelt gratitude to my wife and family for their tireless encouragement, understanding, and support. Their belief in my abilities and their constant motivation have been the driving force behind the successful completion of this thesis.

I would also like to acknowledge the FRACTAL project (FRACTAL ECSEL JU (grant 877056), funded by Horizon Europe and Business Finland) and its partners for their funding, which provided the necessary resources and opportunities to undertake this research.

Finally, I extend my thanks to all the individuals who have played a part in this thesis, including colleagues and friends. Their contributions and participation have greatly enriched the research and its outcomes.

Oulu, June 15th, 2023

Vahid Mohsseni

LIST OF ABBREVIATIONS AND SYMBOLS

API	Application Programming Interface
AR	Augmented Reality
AWS	Amazon Web Service
CA	Certificate Authority
CAP	Consistency, Availability, Partition tolerance
CDN	Content Delivery Network
CLI	Command-Line Interface
CNCF	Cloud Native Computing Foundation
DAG	Directed Acyclic Graph
DB	DataBase
DoS	Denial of Service
EF	Edge Function
EN	Edge Node
FIFO	First-In, First-Out
HTTP	Hyper Text Transfer Protocol
IoT	Internet of Things
K8s	Kubernetes
NAT	Network Address Translation
RSU	RoadSide Units
RODS	Resilient On-demand Distributed Systems
RPC	Remote Procedure Call
SJF	Shortest Job First
SLA	Service Layer Agreement
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
TLS	Transport Layer Security
VR	Virtual Reality

1. INTRODUCTION

Edge computing has revolutionized the field of distributed computing by bringing computation and data storage closer to the network edge. Unlike traditional cloud computing, which centralizes processing and data storage in remote data centers, edge computing leverages the resources available at the edge of the network, in close proximity to where data is generated and consumed [1, 2]. This paradigm shift has significant implications for a wide range of industries, offering reduced latency, improved scalability, and enhanced efficiency. The transformative nature of edge computing lies in its ability to reshape the way computation and data storage are approached. By processing and analyzing data at the edge, organizations can benefit from real-time insights and faster decision-making. This proximity to data sources minimizes latency and network congestion, enabling quicker response times and enhanced user experiences. With edge computing, industries can unlock new possibilities and reap the benefits of distributed computing in ways not previously feasible.

The potential impact of edge computing extends across numerous industries. In the realm of the Internet of Things (IoT), edge computing plays a vital role in addressing the challenges associated with managing and processing massive amounts of data generated by interconnected devices. Organizations may lessen the load on centralized cloud resources, reduce network traffic, and allow speedier and more accurate choices by processing IoT data at the edge [3]. This has significant implications for industries such as manufacturing, transportation, healthcare, and agriculture, where IoT devices are deployed at scale.

Smart cities, too, stand to benefit greatly from the adoption of edge computing. By leveraging distributed edge nodes, cities can efficiently collect and process data from various sources, such as sensors, cameras, and connected infrastructure. This allows for real-time monitoring of traffic conditions, waste management systems, energy consumption, and public safety, enabling more efficient resource allocation, improved urban planning, and enhanced citizen services. The entertainment and media industry is another domain where edge computing has gained traction. With the proliferation of high-definition video streaming, augmented reality (AR), and virtual reality (VR) applications, the demand for low-latency, high-quality content delivery is paramount [4, 5]. Edge computing enables content caching, processing, and delivery at the edge of the network, closer to end-users. This reduces latency, ensures smoother streaming experiences, and enables immersive AR/VR applications with minimal lag, enhancing user satisfaction and engagement. Besides, industries such as healthcare, finance, and retail can leverage edge computing to enhance their operations. In healthcare, for example, edge computing enables real-time monitoring of patient vitals, analysis of medical imaging data, and prompt delivery of critical information in life-saving situations. Financial institutions can utilize edge computing for real-time fraud detection, personalized customer experiences, and secure transactions. Retailers can leverage edge computing to deliver personalized offers, optimize inventory management, and enhance the in-store shopping experience.

The transformative nature of edge computing lies in its ability to bring computation, data storage, and processing capabilities closer to the point of data generation and consumption. By doing so, edge computing enables faster insights, reduced network

traffic, improved scalability, and enhanced user experiences. The impact of this paradigm shift extends to various industries, unlocking new opportunities, improving operational efficiency, and enabling innovative applications.

Edge computing faces a unique set of challenges that differentiate it from traditional cloud-based approaches. These challenges arise from the nature of the edge environment, characterized by limited computing resources, unreliable network connectivity, and stringent latency requirements. One of the primary challenges of edge computing is the limited computing resources available at the network edge. Edge devices such as sensors, gateways, and IoT devices often have constrained processing power, memory, and storage capabilities. Unlike the vast resources available in cloud data centers, edge devices must operate within these limitations while still performing their intended tasks. This necessitates the development of efficient algorithms, lightweight software, and resource optimization techniques to ensure optimal utilization of the available resources. Unreliable network connectivity is another significant challenge in edge computing. Unlike centralized cloud environments with robust network infrastructure, the edge environment is characterized by intermittent connectivity, limited bandwidth, and potential network disruptions. Edge devices may be deployed in remote locations or mobile environments, where network connectivity can be unpredictable and prone to fluctuations. This poses challenges for real-time data transmission, synchronization, and coordination between edge devices and cloud services. To address these challenges, edge computing solutions must incorporate mechanisms for offline operation, local processing, and intelligent data caching to mitigate the impact of network disruptions.

Stringent latency requirements further complicate edge computing deployments. Many edge applications, such as real-time monitoring, autonomous systems, and responsive user experiences, demand low-latency data processing and decision-making [6]. For example, in applications like autonomous vehicles or industrial control systems, even slight delays in data processing and response times can have severe consequences. This necessitates the need for edge computing solutions that can provide rapid and efficient data processing at the edge, minimizing the time required to transmit data to and from remote cloud data centers. Strategies such as data filtering, local analytics, and distributed processing are essential to meet these stringent latency requirements. The challenges faced by edge computing require tailored solutions that differ from traditional cloud-based approaches. Edge computing platforms must be designed to operate within the constraints of limited computing resources, adapt to unreliable network connectivity, and meet stringent latency requirements. These solutions involve deploying intelligent algorithms and distributed processing techniques to optimize resource usage, implement local decision-making capabilities, and enable efficient data synchronization when network connectivity is available. Furthermore, edge computing architectures need to incorporate fault tolerance mechanisms to handle device failures and ensure the resilience and reliability of edge deployments. By addressing these challenges, edge computing can unlock the full potential of distributed computing at the network edge, enabling real-time processing, improved scalability, and enhanced efficiency in various industries and applications. The development of tailored solutions that consider the unique requirements and constraints of edge environments is crucial for maximizing the benefits of edge computing.

The CAP theorem, also known as Brewer's theorem [7], serves as a fundamental principle in distributed systems, asserting that it is impossible for a distributed system to simultaneously guarantee consistency (C), availability (A), and partition tolerance (P). Consistency ensures that all nodes in a distributed system perceive the same data simultaneously. At the same time, availability denotes the system's ability to respond consistently to client requests. Partition tolerance refers to the system's capability to endure network failures or partitioning without ceasing operation. When applied to edge computing, the implications of the CAP theorem become particularly significant. Edge environments involve geographically dispersed edge devices and nodes interconnected by potentially unreliable networks. In this context, striking a balance between consistency, availability, and partition tolerance is paramount when designing an effective orchestration platforms. The nature of edge computing introduces inherent constraints that make achieving strong consistency across all edge nodes in real-time challenging. Factors such as network latency, limited bandwidth, and intermittent connectivity can impede instant data synchronization, leading to eventual consistency or temporary inconsistencies among edge nodes. Nevertheless, in edge computing, prioritizing high availability and partition tolerance remains critical since the system must remain operational and responsive despite network failures or node unavailability.

In the context of edge computing, Kubernetes [8] has emerged as the most prominent container orchestration platform. Originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF) [9, 10], Kubernetes has garnered widespread adoption in the cloud computing domain. Its robust features for managing containerized applications, scaling workloads, and ensuring high availability have made it a go-to solution for many cloud-based deployments. However, when it comes to edge computing, the suitability of Kubernetes as an orchestration tool becomes a topic of discussion. The unique challenges posed by edge environments, such as limited computing resources, unreliable network connectivity, and strict latency requirements, demand tailored solutions that differ from traditional cloud-based approaches. While Kubernetes excels in managing large-scale deployments in centralized cloud environments, it faces limitations when applied directly to the edge computing context. By investigating the limitations of Kubernetes when applied to edge computing, it is possible to shed light on the potential drawbacks and inefficiencies that arise in edge deployments. One such limitation is the reliance on centralized control and coordination mechanisms, which may not align well with the distributed nature of edge environments. Edge deployments often consist of numerous geographically dispersed devices and heterogeneous computing resources. Employing a centralized orchestration framework like Kubernetes in such scenarios can lead to increased network traffic, performance bottlenecks, and reduced efficiency.

There are also other tools and platforms that exists in the context of orchestration, which will be discussed shortly as the following list:

1. Other variation of Kubernetes: The k0s [11] is a minimalistic Kubernetes distribution for edge computing, IoT, and other resource-limited environments. It focuses on simplicity, lightweight footprint, and ease of deployment. The k0s eliminates the need for external dependencies and enables running Kubernetes clusters with minimal resources, making it suitable for edge devices with

limited processing power and memory. It provides essential features for edge computing, including orchestration, container management, and workload scheduling, while minimizing the overhead typically associated with running Kubernetes. K3s [12] is another lightweight Kubernetes distribution tailored for edge computing and IoT deployments. It is designed to be efficient and resource-friendly, enabling Kubernetes clusters to run on low-power ARM-based devices and other edge hardware. K3s simplifies the installation and operation of Kubernetes, providing a minimalistic footprint while retaining core functionality. It incorporates only essential components, reducing the memory and disk footprint, and optimizes for limited network bandwidth and intermittent connectivity often encountered in edge environments. MicroK8s [13], as the name suggests, is a lightweight, single-node Kubernetes distribution suitable for local development, edge computing, and IoT use cases. It is designed for easy installation and quick setup on a single machine or edge device. It enables developers to prototype and experiment with Kubernetes-based applications in resource-constrained environments. MicroK8s offers a compact footprint, fast startup times, and simplified operations, making it well-suited for edge scenarios where efficient resource utilization and ease of management are critical.

2. Docker Swarm [14]: Docker Swarm is a container orchestration platform that manages containerized applications across multiple nodes. It provides features for service discovery, load balancing, and scalability. However, Docker Swarm's centralized control and coordination mechanisms may need to be better suited for edge computing environments with limited resources and unreliable network connectivity.

3. Apache Mesos [15]: Apache Mesos is a distributed systems kernel that enables the management and execution of applications across clusters of machines. It provides resource isolation, scalability, and fault tolerance. While Mesos offers flexibility and scalability, its centralized architecture may introduce overhead and performance bottlenecks in edge computing scenarios.

4. Nomad [16]: Nomad is an open-source cluster scheduler and orchestrator that supports a variety of workloads, including containers and non-containerized applications. It offers features like task scheduling, health monitoring, and scalability. However, similar to other centralized platforms, Nomad's architecture may need to be optimized for edge computing, and it may suffer from increased network traffic and limited offline operation capabilities.

5. AWS IoT Greengrass [17]: AWS IoT Greengrass is a service provided by Amazon Web Services (AWS) that extends AWS capabilities to edge devices. It enables local data processing, device management, and cloud integration. While Greengrass offers integration with cloud services, it relies heavily on the AWS ecosystem, which may limit its portability and flexibility for organizations using different cloud providers or hybrid cloud architectures.

6. Oakestra [18]: Oakestra is an orchestration framework specifically designed for edge computing environments. It focuses on optimizing the execution of distributed applications across edge devices by considering resource constraints,

network variability, and application requirements. Oakestra leverages techniques such as dynamic load balancing, adaptive resource allocation, and network-aware scheduling to improve the performance and reliability of edge applications. This framework emphasizes the unique characteristics of edge computing, including limited resources, intermittent connectivity, and latency-sensitive workloads.

7. MAPE-K [19]: MAPE-K (Monitor-Analyze-Plan-Execute over a feedback loop) is an autonomic management framework proposed in academia. It focuses on self-management capabilities for edge computing environments, including orchestration. MAPE-K employs a closed feedback loop approach, continuously monitoring the state of the system, analyzing it, planning appropriate actions, and executing them to optimize the performance and resource allocation in edge deployments. This framework highlights the importance of autonomic management and adaptive orchestration in dynamically changing edge environments.

Unikernel technology, such as Unikraft [20] and Nanos [21], is a specialized approach to operating systems that focuses on creating lightweight and optimized runtime environments for running single applications or services. Unikernels are designed to be minimalistic, efficient, and secure, offering benefits such as reduced resource requirements, fast boot times, and enhanced isolation [22]. While unikernel technology is an interesting and relevant topic in the field of distributed systems and edge computing, it is important to note that it is not one of the specific objectives of this thesis. The primary focus of this thesis is on exploring the limitations of Kubernetes in edge computing and proposing a specialized orchestration solution. Therefore, the discussion will primarily revolve around Kubernetes and its suitability for edge environments, rather than delving into the intricacies of unikernels. However, it is worth acknowledging the potential of unikernel technology in the context of edge computing. Unikernels offer an alternative approach to optimizing resource utilization and enhancing security in resource-constrained edge scenarios. Their lightweight and tailored nature can contribute to improved performance and efficiency. While beyond the scope of this thesis, further exploration and research on unikernel technology could provide valuable insights for future developments in the field of edge computing.

So, Kubernetes's resource-intensive nature and dependencies on cloud-based services can pose challenges in edge environments, where computing resources are constrained, and network connectivity is unreliable. In edge computing scenarios, resources are often constrained due to the limited processing power, memory, and storage capacity of edge devices or edge nodes. These devices may have lower specifications compared to cloud servers, making it challenging to run Kubernetes effectively. The resource-intensive nature of Kubernetes can strain the limited resources available in edge environments, potentially leading to performance degradation and inefficiencies. Kubernetes relies on various cloud-based services for its operation. These services include features like load balancing, network routing, storage management, and service discovery, among others. However, in edge computing, network connectivity is often unreliable due to factors such as intermittent connections, limited bandwidth, and high latency. The dependency on cloud-based services assumes a reliable and high-bandwidth network connection, which may not

be readily available in edge environments [23]. This can result in disruptions, delays, or even complete failures in Kubernetes operations when the underlying network is not stable or robust. The resource requirements and overhead associated with deploying and managing Kubernetes clusters may outweigh the benefits, leading to suboptimal resource utilization and increased operational costs. In edge computing, where efficiency and cost-effectiveness are crucial, it becomes essential to explore alternative orchestration platforms that can provide lightweight, decentralized, and resource-efficient management of edge resources. Addressing the limitations of Kubernetes in the context of edge computing is essential for unlocking the full potential of distributed computing at the network edge. It covers the way for the development of a specialized orchestration platform that can seamlessly handle the unique challenges posed by edge environments. This platform should provide efficient resource allocation, decentralized control, offline operation capabilities, and enhanced adaptability to varying network conditions. By leveraging such a specialized platform, organizations can harness the benefits of edge computing while ensuring optimal performance, scalability, and reliability.

The research questions guiding this study aim to explore the challenges and limitations of existing orchestration platforms, particularly Kubernetes, in the context of edge computing. By addressing these questions, the thesis seeks to advance the understanding and practical implementation of edge computing orchestration, ultimately contributing to filling existing gaps in knowledge in this rapidly evolving field.

- **What are the limitations of Kubernetes and other alternatives when applied to edge computing scenarios?** The first research question focuses on the limitations of Kubernetes and alternatives when applied to edge computing scenarios. By thoroughly examining these limitations, identifying the specific challenges that arise when deploying Kubernetes in edge environments is possible. Understanding these challenges is crucial as it helps us identify the gaps between the requirements of edge computing and the capabilities of Kubernetes, highlighting the need for a specialized orchestration platform tailored to the unique demands of edge environments.

- **How can a specialized orchestration platform be developed to address the identified limitations of Kubernetes (and state-of-the-art platforms discussed in subsequent chapters) and provide an alternative solution better suited to the challenges posed by edge environments?** The second research question centers around the development of a specialized orchestration platform for edge computing. By investigating and proposing a new platform, the thesis aims to address the identified limitations of Kubernetes (and state of the art platforms which will be discussed in the following sections) and provide an alternative solution that is better suited to the challenges posed by edge environments. This research question is essential as it enables us to explore innovative approaches, algorithms, and architectural designs that can optimize resource allocation, handle intermittent network connectivity, and meet the stringent latency requirements of edge computing.

- **What are the practical challenges in the implementation and development of the proposed orchestration platform?** The third research question focuses on the practical challenges in implementation and development of the proposed orchestration platform. By conducting extensive experiments and evaluations, the thesis aims to assess the availability, scalability, and reliability of the platform in real-world edge computing scenarios. This empirical analysis is crucial as it provides insights into the practical feasibility and effectiveness of the proposed solution, enabling practitioners and researchers to understand its strengths, limitations, and potential areas for improvement.

By addressing these research questions, contribution to the field of edge computing in several ways. Firstly, we enhance the understanding of the limitations of existing orchestration platforms, when applied to edge computing. This knowledge empowers researchers and practitioners to make informed decisions regarding the selection and customization of orchestration solutions in edge deployments. Secondly, by proposing a specialized orchestration platform for edge computing, we fill existing gaps in knowledge by providing a tailored solution that can overcome the limitations of Kubernetes and better address the challenges of edge environments. This contributes to the practical implementation of edge computing by offering a platform specifically designed to meet the unique requirements of distributed computing at the network edge; a new platform named Resilient On-demand Distributed Systems (RODS) is proposed. RODS aims to provide a holistic approach to orchestrating tasks in edge computing, accommodating the heterogeneity of computing resources prevalent at the edge. Lastly, through the practical complexities in implementation and development of the proposed orchestration platform, we provide valuable insights into its consistency, scalability, and reliability. This empirical evaluation helps refine the platform, identify potential areas for optimization, and facilitates its adoption by organizations seeking to leverage the benefits of edge computing. Addressing the research questions outlined in this study contributes to filling existing gaps in knowledge and improving the understanding and practical implementation of edge computing orchestration. By examining the limitations of existing platforms, proposing a specialized solution, and evaluating its performance, the thesis advances the field and paves the way for more efficient and effective orchestration of edge computing resources.

RODS emphasizes availability and responsiveness more than strong consistency within edge environments. By leveraging techniques like eventual consistency and optimistic replication, RODS enables decentralized decision-making and data management across edge nodes. This design acknowledges the challenges of achieving strong consistency in highly distributed and dynamically changing edge environments while ensuring the system remains available and responsive to user requests. During the design of RODS, careful consideration is given to the trade-offs presented by the CAP theorem. RODS adopts a partition-tolerant design, ensuring reliable operation in the face of network failures or intermittent connectivity. Its focus lies in maintaining availability, allowing edge nodes to continue providing services to clients even in the presence of network partitions. These carefully considered trade-offs between consistency, availability, and partition tolerance in RODS aim to strike a balance that aligns with the unique requirements of edge computing. By prioritizing availability and partition tolerance while embracing eventual consistency,

RODS provides a practical and efficient orchestration platform for edge environments. This design approach recognizes the realities of edge computing, where ensuring constant availability and responsiveness, even in the face of network disruptions, is crucial for delivering efficient and reliable services. The implications of the CAP theorem for edge computing underscore the need to navigate the trade-offs between consistency, availability, and partition tolerance. The design of an orchestration platform like RODS shifts its focus towards ensuring availability and responsiveness while acknowledging the challenges associated with achieving strong consistency in highly distributed edge environments. Through its embrace of partition tolerance and adoption of techniques such as eventual consistency, RODS strives to deliver a reliable and efficient orchestration solution for edge computing, wherein consistent availability and resilience in the face of network failures remain fundamental considerations.

Subsequent chapters will delve into the background and related works in the field of edge computing, providing a comprehensive overview of existing literature and approaches. Following chapters will then present the details of the RODS implementation, highlighting the architectural design, algorithms, and techniques employed. Following that, an evaluation of RODS performance in edge environments will be conducted, considering factors such as scalability, resource utilization, and fault tolerance. Subsequently, a thorough discussion will be undertaken to analyze the findings and implications of the research.

This thesis aims to provide a comprehensive understanding of the challenges and opportunities in edge computing orchestration, proposing a novel platform that addresses the limitations of existing solutions. By enhancing the capabilities of orchestration in edge environments, RODS strives to pave the way for more efficient and reliable edge computing deployments. The thesis will conclude with a summary of the research outcomes, implications, and suggestions for future research directions.

2. RELATED WORKS

The related work chapter comprehensively overviews prior research and literature in edge computing orchestration. This chapter explores various topics, including edge computing, distributed systems, fog computing, orchestration challenges, container orchestration tools, and relevant academic works. By reviewing and analyzing existing research, this chapter lays the groundwork for developing and validating the RODS platform. Scope and Organization: This chapter is organized into several sections, each focusing on a specific topic related to edge computing orchestration. The sections include

- an exploration of edge computing and its key characteristics (2.1),
- the evolution and development of distributed systems (2.2),
- specific challenges in orchestration (2.4),
- an overview of container orchestration tools (2.5), and
- a review of relevant academic works (2.6) such as Oakestra and HYDRA.

This structured approach allows for a systematic examination of the relevant literature. The insights gained from the review of prior research are crucial for informing and guiding this thesis' investigation. By examining different approaches, methodologies, and findings, it is possible to gain leverage the strengths and lessons learned from previous studies to develop and validate the RODS platform. The identified gaps in the existing literature will guide us in addressing the limitations of current orchestration platforms and contribute to advancing edge computing research and practice. The related work chapter sets the stage for the subsequent chapters, where the methodology, implementation, and evaluation of the RODS platform are presented. It forms a solid foundation for this research, enabling us to build upon existing knowledge and significantly contribute to edge computing orchestration.

2.1. Edge Computing

Edge computing is a distinct computing paradigm that operates at the edge of the network, aiming to bring computational capabilities closer to the data source. Researchers have proposed various definitions of edge computing. One definition describes it as a mode of network edge execution where the downlink data represents cloud services, the uplink data represents the Internet of Everything, and the edge of edge computing refers to the computing and network resources between the data source and the path of cloud computing centers [24, 25]. Another definition characterizes edge computing as a computing model that deploys computing and storage resources closer to mobile devices or sensors at the network edge [26]. A unified definition suggests that edge computing unifies resources close to the user geographically or in terms of network distance to provide computing, storage, and network services for applications [27]. Additionally, some industrial alliance defines edge computing as an open platform that integrates networking, computing, storage, and applications and

provides edge intelligent services nearby to meet industry requirements in connection, real-time business, data optimization, application intelligence, security, and privacy [28, 25].

In essence, edge computing aims to provide services and perform computations at the edge of the network and data generation [25]. By migrating the network, computing, and storage capabilities of cloud computing to the network edge, edge computing offers intelligent services that meet critical IT industry needs, including agile linking, real-time business, data optimization, application intelligence, security, privacy, low latency, and high bandwidth requirements. While edge computing has gained significant attention as a research topic, it is important to note that it does not replace cloud computing. Instead, the two paradigms should coexist, complement each other, and develop in a coordinated manner to facilitate industry digital transformation. Data generated on edge nodes must still be summarized in the cloud for in-depth analysis and meaningful insights. Cloud computing continues to play a crucial role in developing increasingly intelligent IoT devices [25, 29].

The emergence of edge computing addresses the challenges posed by traditional cloud computing, which involves transferring all data to centralized cloud computing centers for computation and storage. With the proliferation of IoT devices and the generation of vast amounts of data, cloud computing's network bandwidth often falls short of meeting the requirements of time-sensitive and real-time systems [30]. Cloud computing also exhibits deficiencies in terms of load, real-time performance, transmission bandwidth, energy consumption, and data security and privacy protection.

In the context of IoT, edge computing serves as a means to alleviate the load on cloud computing by handling tasks locally at the network edge. Edge computing shares the burden of the cloud. It is responsible for tasks within its scope, ensuring that data is not lost even if issues arise with edge computing. Some data processed by edge computing may still need to be sent back to the cloud for in-depth analysis, data mining, and sharing. The collaboration between cloud computing and edge computing provides stability to connected IoT devices. The working method often involves cloud computing performing big data analysis and output, passing it to the edge for processing and execution. This cooperative approach has found application in various domains such as intelligent manufacturing, energy, security, privacy protection, and smart homes. Cloud computing controls the overall process, while edge computing provides real-time detection and timely problem resolution. In smart homes, edge computing nodes handle heterogeneous data from different devices, which is then uploaded to the cloud for processing, enabling edge nodes' access and control from the cloud. Both cloud computing and edge computing leverage their respective strengths to cater to the needs of IoT devices, and their joint development continuously drives IoT progress [31].

Edge computing extends cloud computing but possesses distinct characteristics. Cloud computing excels in comprehensive data processing, conducting in-depth analysis, and serving non-real-time data processing applications such as business decision-making. On the other hand, edge computing focuses on local operations and performs better in small-scale, real-time intelligent analysis, meeting the immediate needs of local businesses. In terms of network resources, edge computing handles data closer to the information source, enabling local storage and processing without uploading all data to the cloud. This reduction in network burden significantly

enhances network bandwidth utilization efficiency. Cloud computing and edge computing play pivotal roles in the future development of intelligent IoT applications. The main differences between the two paradigms are summarized in Table 1. In addition to the Table 1, Table 2 is presented to complement it.

Table 1. Short comparison between Edge and Cloud Computing presented in [25]

	Applicable Situation	**Network Bandwidth Pressure**	**Real-time**	**Calculation Mode**
Cloud Computing	Global	More	High	Large-scale centralized processing
Edge Computing	Local	Less	Low	Small-scale intelligent analysis

Table 2. Extension of the Table 1 in this thesis

	Data Processing Location	**Data Storage Location**	**Architecture of the Devices/Servers**
Cloud Computing	Centralized	Centralized	Homogenous
Edge Computing	Distributed	Distributed	Heterogenous

The edge computing model stores and processes data on edge devices without uploading them to a cloud computing platform. This characteristic offers several advantages:

1. **Fast data processing and real-time analysis**: Compared to traditional cloud computing, edge computing excels in response speed and real-time performance. Being closer to the data source, edge computing enables data storage and computation at the edge node, eliminating the need for intermediate data transmission. This proximity to users leads to improved data transmission performance, ensures real-time processing, and reduces latency. Edge computing provides users with various responsive services, particularly in fields like autonomous driving, intelligent manufacturing, and video monitoring, where location awareness and rapid feedback are crucial.

2. **Enhanced security**: Traditional cloud computing requires uploading all data to the cloud for centralized processing, which poses risks such as data loss and leakage, compromising security and privacy. In contrast, edge computing limits its responsibilities to tasks within its scope, processing data locally and avoiding the risks associated with network transmission. This approach guarantees data security, as attacks only impact local data rather than the entire dataset.

3. **Cost-effectiveness, energy efficiency, and reduced bandwidth requirements**: Edge computing minimizes the need for network bandwidth by processing data locally without uploading it to the cloud. This reduction in network load not only enhances network bandwidth utilization but also significantly reduces energy

consumption in edge devices. Edge computing's smaller scale allows companies to reduce data processing costs on local equipment. Consequently, edge computing reduces data transmission volume, lowers transmission costs and network bandwidth pressure, minimizes energy consumption in local equipment, and improves overall computing efficiency.

2.2. Distributed Systems: Evolution and Development

The development and evolution of distributed systems have been driven by the increasing demand for applications capable of handling many users, providing real-time services, and efficiently utilizing network resources [32]. To address these challenges, various architectural paradigms have emerged over time. This section provides a comprehensive overview of the taxonomy of distributed system evolution, highlighting the key features and advancements in each architecture.

Client-Server Model: The client-server model represents one of the earliest and fundamental architectures for distributed systems. In this model, clients initiate requests to a centralized processing side known as the server, which provides the requested services. The primary focus of this architecture was to facilitate communication and enable users to access services from servers. Communication in this model typically relied on Remote Procedure Calls (RPC) for interaction [33].

Mobile Agents: The mobile agents architecture aimed to overcome the limitations imposed by slow networks by moving computation to the server side. Agents, comprising code, data, and state, were capable of migrating between hosts to accomplish specific tasks. This architecture was designed to handle complex systems that required continuous connection with information sources (servers) to track updates [34, 35]. However, as communication links improved and businesses demanded more structured software systems, this paradigm received less attention. Moreover, security issues in mobile agents in distributed systems can pose significant challenges and risks including Denial of Service (DoS) Attacks, trust and authentication issues, and information leakage.

Service-Oriented Architecture (SOA): SOA emerged as a successful extension of the client-server architecture, providing additional business value. Its objective was to offer reusable and loosely coupled services, enabling integration across diverse organizations and operational systems. SOA introduced the concept of explicit boundaries between autonomous services located on different servers. Communication in SOA relied on protocols such as SOAP (Simple Object Access Protocol) and aimed to leverage services across the web. However, SOA still relied on monolithic systems, which highlighted the need for more resilient, scalable, and resource-efficient architectures [36].

Microservices Architecture: The microservices architecture evolved to address the limitations of previous architectures and meet the requirements of modern applications. It emphasizes breaking down applications into smaller, independent services that can be developed and deployed separately. Each microservice focuses on a specific functional operation and can be designed to be independent or further divided into smaller services. Communication between microservices relies on lightweight protocols, facilitating agility and efficient resource utilization [37]. The elimination

of a centralized service bus enhances decoupling between services, enabling the distribution of control and intelligence. The benefits of microservices architecture include scalability, resilience, frequent updates, and dynamic user experiences. Microservices can be organized in various ways, such as shared databases as services and utilizing lightweight communication protocols.

2.3. Fog Computing and Orchestration

Edge computing refers to the decentralized data processing at or near the network edge, where devices generate and consume data. It aims to alleviate bandwidth constraints and reduce latency by performing computation closer to the data source. In contrast, Fog computing extends the cloud infrastructure to the network edge, leveraging intermediate computing nodes called fog nodes or fog servers. These fog nodes act as intermediaries between the cloud and edge devices, providing computational resources and storage capabilities. While edge computing focuses on pushing computation to the network edge, Fog computing encompasses a broader architecture that includes edge devices, fog nodes, and cloud resources [38, 39]. Fog computing offers a hierarchical approach, with fog nodes acting as intermediaries between the edge and the cloud. This hierarchical structure enables scalable and distributed data processing, analysis, and storage, resulting in improved performance and reduced dependency on centralized cloud infrastructure. Fog computing complements edge computing by providing a more comprehensive framework for managing resources and orchestrating distributed systems.

The convergence of edge and Fog computing opens up new possibilities for developing effective orchestration systems. With its distributed architecture and hierarchical structure, Fog computing enables efficient resource management, workload distribution, and service coordination across edge devices and fog nodes. Orchestration systems leveraging Fog computing can intelligently allocate computational tasks, dynamically adjust resource utilization, and ensure optimal performance for data-intensive applications. Fog computing's proximity to edge devices allows for real-time data processing, enabling rapid decision-making and response. This capability is precious in applications such as industrial automation, smart cities, healthcare monitoring, and real-time analytics [40]. By combining the strengths of edge computing and Fog computing, orchestration systems can leverage the scalability, low latency, and distributed nature of Fog infrastructure to provide reliable and efficient services in dynamic and heterogeneous environments.

The following instance emphasizes the need for an orchestration system that effectively integrates cloud, edge, and fog computing. This integration is crucial to harness the combined capabilities of these computing paradigms and maximize their potential for distributed systems.

For instance, consider a smart city scenario where various sensors and devices are deployed throughout the urban landscape to monitor environmental conditions, traffic flow, and energy consumption. These devices generate massive amounts of data that must be processed and analyzed in real-time to enable intelligent decision-making and optimize resource utilization. In such a scenario, more than cloud computing is required to handle the sheer volume and low-latency requirements of

data processing. While edge computing can alleviate the bandwidth constraints by performing computation closer to the data source, it may still face scalability and resource availability limitations. This is where fog computing comes into play. By extending the cloud infrastructure to the network edge through fog nodes or servers strategically located within the smart city, fog computing enables localized data processing, analysis, and storage. The fog nodes act as intermediaries between the edge devices and the cloud, facilitating efficient resource management and workload distribution. An orchestration system is needed to effectively orchestrate this complex ecosystem of cloud, edge, and fog resources. This system would dynamically allocate computational tasks, optimize resource utilization, and coordinate interactions between different computing layers.

For example, the orchestration system could intelligently distribute data processing tasks, offloading computationally intensive operations to the cloud when resources are available and leveraging edge and fog computing for real-time and localized processing. It could also manage data storage and synchronization between the cloud, edge devices, and fog nodes, ensuring data consistency and availability across the entire system. By integrating cloud, edge, and fog computing in a cohesive orchestration system, the smart city scenario can benefit from the scalability and computational power of the cloud, the low-latency processing of edge computing, and the localized intelligence of fog computing. This integration enables efficient resource management, real-time analytics, and improved decision-making capabilities, leading to enhanced services, optimized resource utilization, and a better quality of life for city residents.

2.4. Orchestration Challenges

The orchestration of virtualized environments poses significant challenges due to the expansive scale, diverse resource types, and heterogeneity of the underlying cloud environment. Managing these challenges becomes even more complex due to uncertainties stemming from various factors, such as the fluctuating demand for resource capacities (e.g., bandwidth and memory), potential failures (e.g., network link disruptions), user access patterns (e.g., user numbers and locations), and the lifecycle activities of applications [41]. The intricate nature of cloud resource orchestration is of particular concern, as applications are composed of multiple software and hardware resources that often exhibit diverse characteristics, leading to intricate integration and interoperation dependencies [42]. These complexities in virtualized environment orchestration demand novel solutions that address the intricacies arising from the scale, heterogeneity, and uncertainties of the cloud environment. A comprehensive orchestration framework can enable efficient deployment and operation of applications by effectively managing and coordinating the allocation, provisioning, and utilization of resources. Such a framework should provide robust mechanisms for resource allocation, taking into account the dynamic resource demand, potential failures, and various integration dependencies within applications.

The process of orchestration within the Cloud-to-Edge continuum introduces additional complexities and poses unique challenges. With the advent of the cloud-to-thing era, applications, and storage are now distributed across geographical locations.

Consequently, there is a need to restructure applications to distribute their logic across the network while also decentralizing storage. This shift in architecture gives rise to novel concerns regarding reliability and data integrity, particularly in the context of broadly decentralized networks. In this paradigm, cloud servers assume the role of control nodes, overseeing the operations of intelligent edge devices. They handle summary analytics tasks while leaving real-time decision-making capabilities to the edge servers. This distribution of responsibilities allows for efficient data processing and analysis at the edge, enabling timely and context-aware decision-making. However, it introduces challenges related to maintaining data integrity, ensuring reliable communication, and managing the coordination between cloud and edge resources. Effectively navigating these complexities requires innovative approaches to address the unique requirements of the Cloud-to-Edge continuum [31, 43]. It involves designing architectures that distribute application logic and decentralize storage while ensuring reliable and secure data transmission. Furthermore, integrating cloud and edge resources necessitates robust coordination mechanisms to optimize resource utilization and facilitate seamless communication between the two tiers. Understanding and addressing the challenges associated with orchestration in the Cloud-to-Edge continuum is crucial for enabling the efficient and reliable operation of applications in this distributed environment.

2.5. Container Orchestration Tools

Kubernetes [8] has emerged as a leading container orchestration platform in the realm of cloud-native computing with a wide range of features and functionalities that simplify the deployment, scaling, and management procedures, and it offers a reliable and expandable infrastructure for managing containerized applications. At its core, The deployment, scaling, and management of containerized applications across node clusters are all automated by Kubernetes. It offers a declarative approach, allowing users to define the desired state of their applications and letting Kubernetes handle the necessary operations to achieve and maintain that state. Key features of Kubernetes include its ability to automatically distribute containers across nodes, handle container scheduling and resource allocation, monitor application health, and provide self-healing capabilities. It also supports horizontal scaling, allowing applications to adjust their resource usage based on demand dynamically. Moreover, Kubernetes incorporates powerful networking and service discovery mechanisms, enabling seamless communication between containers and services within the cluster. It provides load balancing, service discovery, and DNS-based name resolution, facilitating the creation of robust and interconnected application architectures. By utilizing Kubernetes, organizations can benefit from improved resource utilization, simplified deployment workflows, enhanced scalability, and high availability of their applications. Due to its open-source nature, broad community support, and interoperability with a variety of cloud providers and infrastructure platforms, Kubernetes has experienced substantial growth in popularity.

At the core of Kubernetes is its support for containerization. Kubernetes enables portability, consistency, and scalability by encapsulating applications and their dependencies into containers. Containers provide an isolated and lightweight

environment, ensuring applications run consistently across different infrastructures. Kubernetes also embraces the microservices architecture, a fundamental tenet of cloud-native computing. With Kubernetes, complex applications can be decomposed into more minor, loosely coupled services, each running in its own container. This modular approach enables agility, as each microservice can be developed, deployed, and scaled independently. Scalability is a critical requirement in cloud-native environments, and Kubernetes addresses this aspect effectively. It provides mechanisms for automatic scaling based on metrics such as CPU utilization or incoming requests. Kubernetes dynamically adjusts the number of replicas for each service, ensuring optimal resource utilization and efficient handling of varying workloads. Declarative configuration is a fundamental principle of Kubernetes, allowing users to define the desired state of their applications and infrastructure. Kubernetes takes care of the actual deployment and management, continuously working to converge the system state through the desired configuration. This declarative approach simplifies operations, promotes consistency, and reduces manual intervention. With its robust container orchestration capabilities, Kubernetes enables deploying and managing complex, distributed applications. It coordinates the scheduling, scaling, and monitoring of containers across clusters of nodes, ensuring high availability, fault tolerance, and efficient resource utilization. Kubernetes also provides advanced features like service discovery, load balancing, and rolling updates, making it an ideal choice for cloud-native environments.

While Kubernetes has proven to be a powerful tool in cloud-native environments, its application to edge computing scenarios presents unique challenges. The characteristics of edge environments, such as limited computing resources, unreliable network connectivity, and stringent latency requirements, demand tailored solutions beyond traditional cloud-based approaches. In edge computing, computing resources are often constrained due to the limited hardware capabilities of edge devices. With its resource-intensive nature, Kubernetes may struggle to operate efficiently in such resource-constrained environments. The overhead of running Kubernetes components on edge devices can strain the available computational resources, impacting the overall performance and scalability. Unreliable network connectivity is another critical challenge in edge environments. Edge devices may experience intermittent or limited network connectivity, unlike cloud environments with stable and high-bandwidth connections. Kubernetes relies heavily on network communication for inter-container communication, service discovery, and coordination. The assumption of reliable network connectivity in Kubernetes may not hold in edge environments, leading to potential disruptions in application functionality. Edge computing requires strict latency requirements due to real-time data processing and decision-making at the edge. However, Kubernetes, primarily designed for cloud-based scenarios, may not provide the necessary optimizations for achieving ultra-low latency in edge environments. The inherent architectural design and distributed nature of Kubernetes can introduce additional latency, impacting the responsiveness and real-time capabilities expected in edge computing applications. Considering these limitations, it becomes evident that while Kubernetes excels in cloud environments, there may be more suitable tools for edge computing and orchestration tasks. Edge computing requires specialized solutions optimized for resource-constrained environments, intermittent network connectivity, and low-latency requirements. Recognizing the unique challenges of edge computing, researchers and industry practitioners have been exploring alternative

orchestration platforms and approaches that can better address the specific needs of edge environments.

In addition to Kubernetes, several other container orchestration tools commonly used in the industry offer alternatives to address the challenges of edge computing. These tools, including k0s, K3s, MicroK8s, Docker Swarm, Apache Mesos, and HashiCorp's Nomad, provide unique features and functionalities that may be more suitable for edge computing environments. A comparison table highlighting their key features, functionalities, and how they compare to Kubernetes in the context of edge computing is presented in Table 3.

Table 3. Alternative tools and platforms to Kubernetes

Tool	Key Features and Functionalities	Advantages	Limitations
k0s [11]	Lightweight and minimal Kubernetes distribution	Reduced resource footprint, suitable for edge nodes	Limited community support
K3s [12]	Lightweight and secure Kubernetes distribution	Simple installation, lower resource requirements	Fewer advanced features compared to full Kubernetes
MicroK8s [13]	Lightweight and easy-to-install Kubernetes	Quick deployment, ideal for edge and IoT scenarios	Limited scalability for large deployments
Docker Swarm [14]	Native orchestration tool for Docker containers	Simplicity and seamless integration with Docker	Fewer advanced features compared to Kubernetes
Apache Mesos [15]	Distributed systems kernel for resource management	Robust resource allocation and isolation capabilities	Steeper learning curve compared to Kubernetes
HashiCorp Nomad [16]	Flexible and lightweight workload orchestrator	Simple configuration and cross-platform support	Limited ecosystem compared to Kubernetes

Each tool has its strengths and limitations when applied to edge computing. For instance, Docker Swarm offers simplicity and tight integration with Docker, making it an attractive option for organizations already utilizing Docker containers. Conversely, Mesos provides advanced resource allocation and isolation capabilities, ensuring efficient utilization of edge resources. However, it is important to note that these tools may have fewer advanced features than Kubernetes and may have a smaller ecosystem of supporting tools and applications.

2.6. Related Academic Works

This section will explore a variety of academic publications that have contributed to edge computing and container orchestration. These publications cover a wide range of various topics, including algorithms, prototype platforms, and theoretical foundations. The information will be about the difficulties, breakthroughs, and real-world applications of container orchestration by looking at these research

contributions. It may be possible to make links, spot gaps, and situate this study within the scholarly debate thanks to this exploration. The scholarly publications that inform and direct this thesis's investigation into container orchestration and edge computing are briefly summarized in this section.

The study by Bartolomeo presents Oakestra [18], a hierarchical orchestration framework specifically designed to address the challenges of edge computing. The framework focuses on supporting service operation over heterogeneous edge infrastructures while considering strict quality-of-service requirements, resource heterogeneity, and network fluctuations. Oakestra offers efficient service management and computation offloading capabilities tailored for edge environments, unlike traditional cloud-native technologies such as Kubernetes. Oakestra introduces several notable contributions. Firstly, it employs a two-tier logical hierarchical orchestration of computing resources. Local cluster orchestrators manage individual clusters, coordinating with the root orchestrator to provide aggregated resource utilization and service operation statistics. Application developers interact with the root orchestrator to deploy services, specifying high-level service level agreements (SLAs) and deployment descriptors. Secondly, Oakestra adopts a delegated service scheduling principle, decentralizing task placement to find optimal deployments quickly.

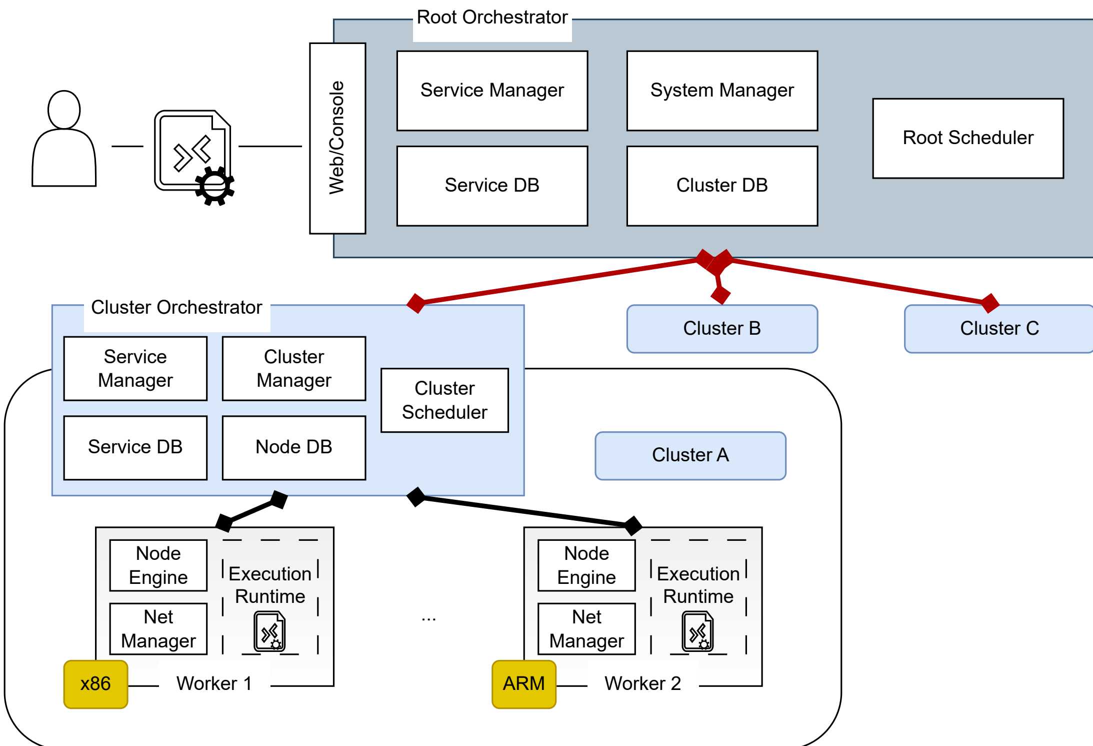


Figure 1. Architectural view of Oakestra framework adapted from [18].

As Figure 1 illustrates, the root orchestrator calculates the most suitable clusters for service requests based on requirements and offloads the requests to the cluster scheduler for efficient resource deployment. This approach significantly reduces the complexity and cost of service scheduling at the edge. Oakestra utilizes semantic IP addresses and load-balancing techniques to support flexible and transparent load

balancing. The framework enables service mobility and handles hardware constraints without requiring code adaptation. It establishes an overlay network for service-to-service communication, allowing traversal of network address translation (NAT) and firewalls. The study demonstrates Oakestra's capabilities through a live demonstration in an edge-cloud infrastructure using a latency-critical augmented reality (AR) application [44]. The demonstration showcases the deployment of the AR application using Oakestra's APIs, handling spikes in application load through rescheduling and replication, and transparently managing resource and service failures at the edge. The application's performance and the infrastructure's operational load are monitored and observed during the demonstration. The research conducted by Bartolomeo et. al. in Oakestra aligns closely with the objectives of this thesis. It addresses the limitations of Kubernetes in edge computing scenarios and proposes a specialized orchestration framework for edge environments. While Bartolomeo et. al. focuses on hierarchical orchestration, service scheduling, and load balancing, this thesis aims to explore the challenges and opportunities of orchestrating containerized applications in edge computing. By incorporating the insights and findings from Bartolomeo's work, this thesis builds upon the existing body of knowledge and contributes to the advancement of edge computing orchestration research. Oakestra framework provides valuable insights into edge computing orchestration and offer a complementary perspective to the objectives of this thesis. By combining these findings with the specific focus of this research, a comprehensive understanding of the orchestration landscape in edge environments can be achieved, enabling the identification of further opportunities and potential improvements.

The academic work by Jiménez and Schelén introduces HYDRA, a decentralized orchestrator designed specifically for containerized microservice applications. HYDRA operates as both a location-aware and location-agnostic orchestrator, providing functionalities for efficient resource and application management [45]. Figure 2 shows an example of the HYDRA in which it illustrates two control types for application management. The application consists of 9 services. In the first type (Type 1, left side of the figure), the HYDRA orchestrator operates without location awareness, resulting in non-location-based service placement. A single layer of management is conducted by the root controllers. On the other hand, in the second type (Type 2, shown on the right), HYDRA is location-aware, leading to service deployment based on location in different regions (R1 to R3). The root controllers handle global, lightweight management, while the leaf controllers perform local, active management.

The authors' research contributions in this paper include:

1. **ID-based Identifier Design**: HYDRA utilizes unique identifiers (IDs) to differentiate nodes, applications, and resources. By incorporating geographical mapping, HYDRA achieves location awareness when necessary.

2. **Node Discovery**: HYDRA employs a node discovery mechanism based on Kademlia's [46] distributed hash table and node lookup algorithm to enable decentralized orchestration. Each node maintains a routing table, facilitating the discovery of other nodes within the overlay network.

3. **Application Management**: HYDRA enables the deployment and control of containerized microservice applications. Application owners provide relevant

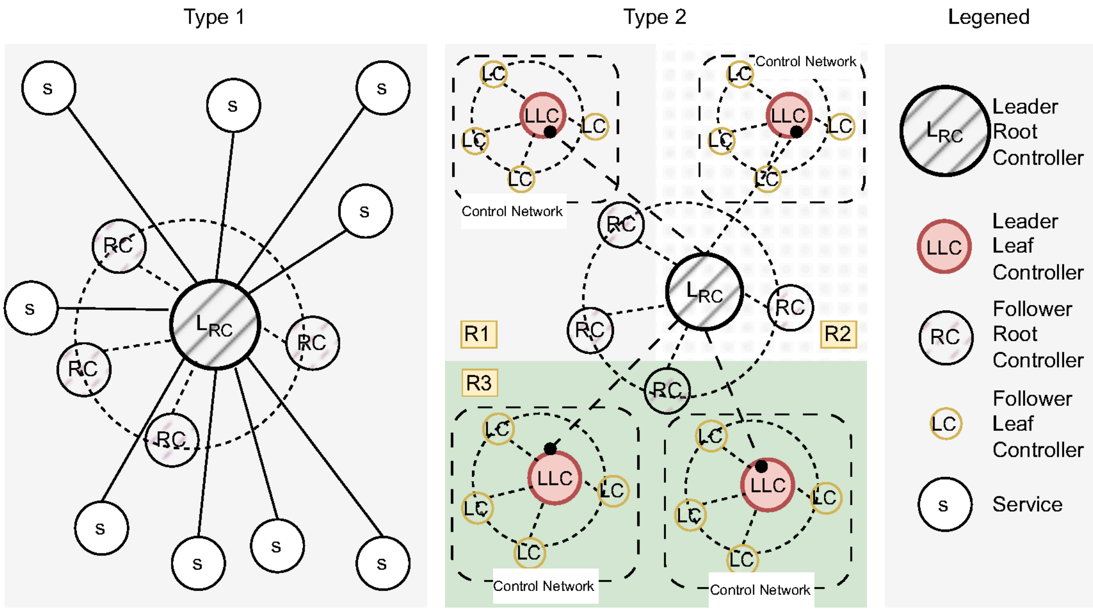


Figure 2. Two different types of acquiring HYDRA's orchestrator. Adapted from [45].

information, including service definitions, configuration details, and deployment specifications. HYDRA supports dynamic roles for application orchestration and diverse forms of application control.

4. **Resource Search**: HYDRA incorporates a resource search algorithm to efficiently locate and allocate the necessary resources for deploying application services.

5. **Distributed Consensus**: HYDRA utilizes distributed consensus algorithms to ensure replication and data consistency across nodes. This robust mechanism allows for effective application control and leader election during failure scenarios.

The paper also introduces the concept of location-aware nodes in HYDRA, which are identified by both location-agnostic IDs and location-aware IDs associated with the physical location of the nodes. This location awareness feature empowers applications to be deployed based on specific location requirements or run anywhere within the network, depending on the orchestrator's mode of operation. Furthermore, the paper presents a hierarchical ID design scheme reminiscent of IP addressing to enable fine-grained or coarse-grained network designs. This design incorporates regions, areas, and node addresses while also facilitating low-latency communication through control networks. Applications in HYDRA are defined as collections of loosely coupled containerized services, with each application uniquely identified by a location-agnostic root ID generated through hashing deployment data. The management of applications encompasses various roles, such as entry roles for handling application requests, controller roles for supervision, and service host roles for hosting application services. In relation to this research, HYDRA's decentralized and location-aware orchestration framework offers valuable insights. Its focus on efficient resource management

and location awareness aligns with the objectives of this thesis. However, further exploration is needed to identify synergies and differences between HYDRA and this research, particularly within the context of edge computing and its specific challenges. By incorporating the methodologies and findings presented in the HYDRA paper, this thesis aims to build upon and extend the existing knowledge in the field of container orchestration. It seeks to address the unique requirements and constraints of edge computing environments while leveraging the benefits of decentralized orchestration approaches.

The article by Cozzolino et. al., introduces ECCO, a framework for a roadside infrastructure that integrates smart vehicles, roadside units (RSUs), and cloud servers to improve road safety and alleviate congestion. ECCO enables the development and deployment of applications by leveraging a distributed, edge-cloud computational model. The framework focuses on use cases such as car crash detection, road hazard detection, and smart parking. It utilizes edge functions (EFs) and multi-node execution pipelines to process inputs and deliver outputs [47]. The Figure 3 illustrates the modules of the ECCO framework. The deployment of pipelines is managed by both the cloud and the edge, with the cloud defining the deployment plan and the edge making local scheduling decisions. The architecture employs edge computing to provide real-time information to nearby vehicles and enhance context awareness. The ECCO architecture revolves around edge-cloud pipelines, which involve multiple edge nodes (ENs) like RSUs and other roadside devices. The pipelines consist of inputs, EFs for processing, and outputs for enabling services. The deployment is controlled by the cloud, with the cloud nodes defining the high-level deployment plan while the edge nodes make local scheduling decisions. This control aspect highlights the centralized role of the cloud nodes in ECCO's architecture. The pipeline execution follows a directed acyclic graph (DAG) structure, and EFs are chained together to form an execution sequence. Each EF performs a specific role and is hosted on an EN. The ENs strategically place EFs based on available data sources, load status, and geographic position. The system leverages edge computing to deliver real-time information to nearby vehicles, enhancing their context awareness. ECCO provides insights into the orchestration of edge resources and the utilization of multi-node execution pipelines even though the main focus of the work is presenting cloud nodes controlling over the edge nodes.

Above all the frameworks and prototypes introduced earlier, many articles focus on the algorithmic view of the orchestration problem. As an example, Mehar et. al.,'s [48] works try to optimize the roadside units (RSU) on vehicular networks. Vo et. al., in the article published in 2017, alongside with Castellano et. al. emphasize an optimal allocation algorithm for high-performance video streaming in 5G networks [49, 50]. For the sake of resource partitioning DRAGON [51], as a prototype and algorithm, seeks optimal partitioning of shared resources between different applications running over a standard edge infrastructure in their resource orchestration with guarantee distributed systems.

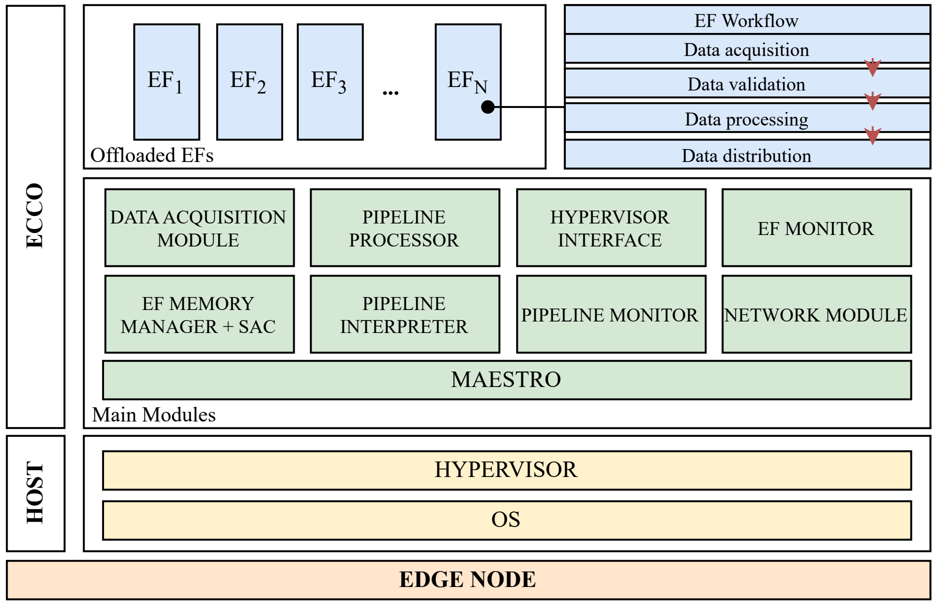


Figure 3. Overview of ECCO Modules, adapted from [47].

2.7. Summary

In this chapter, a thorough review of the relevant research in the areas of container orchestration and edge computing was presented to answer the first research question of this thesis. It has explored various topics, including the concept of edge computing, the limitations of Kubernetes in edge environments, alternative orchestration tools, and significant academic works in the field. Through this review, we have gained a deeper understanding of the existing approaches, challenges, and advancements in container orchestration for edge computing. The insights obtained from these works will serve as a foundation for this thesis and the development of an effective and efficient orchestration platform for edge environments. With this solid understanding of the current state of the field, we are well-equipped to proceed with own investigation and contribute to the advancement of container orchestration in edge computing.

3. METHODOLOGY

In the methodology chapter, a roadmap is served for implementing and detailing the proposed solution. This chapter outlines the design and architecture view of the system, providing insights into the overall structure and organization. The utilization of the Rust language is explored as it provides unique advantages for performance, memory safety, and concurrency.

3.1. Design and Architecture Overview

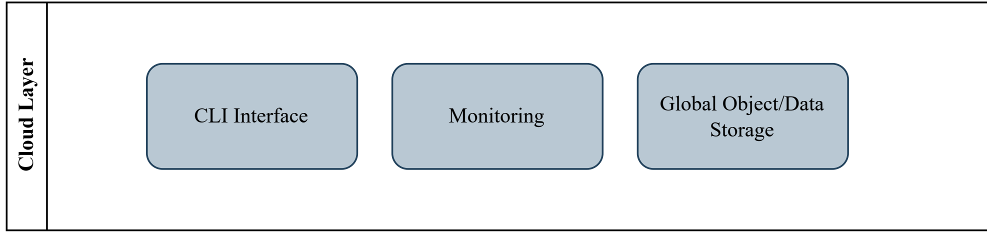
The design and architecture of the Resilient On-demand Distributed Systems (RODS) platform is carefully crafted to enable efficient and reliable task distribution based on a modified version of the RAFT algorithm, ensuring seamless operation in edge computing environments. RODS encompasses three distinct categories of nodes, each playing a specific role in the system. As Figure 4 illustrates an overview of the RODS platform, in the following, the details of each category will be explained.

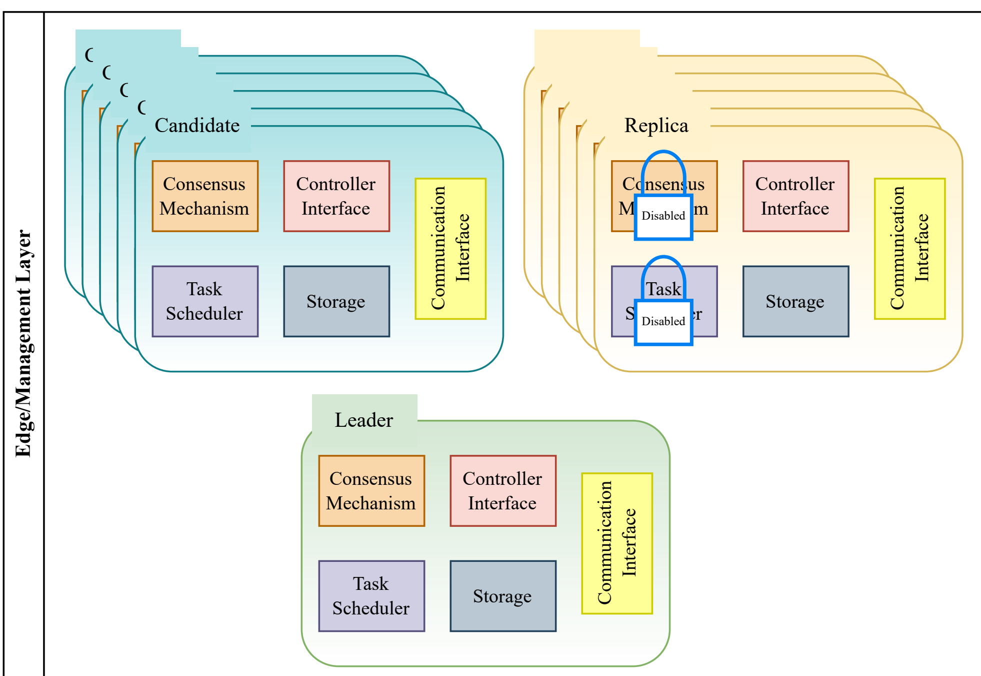
The first category consists of cloud nodes, which are primarily responsible for enhancing the usability of the system. Two components are present within this category: monitoring and global data/object storage. The monitoring component enables efficient system monitoring and management. In contrast, the global data/object storage component provides a centralized repository for storing and retrieving data and objects. Although the cloud nodes do not actively participate in system management, their presence significantly simplifies the utilization of the RODS platform.

The second category is dedicated to edge nodes, which are specifically designed to operate at the network edge. The edge nodes act as the central hub of the system, facilitating seamless connectivity between the monitoring and worker nodes. Within the edge node category, three types of nodes coexist: Leader, Candidate, and Replica. The system operates with a single leader at any given time, which is initially selected based on a consensus using the modified RAFT algorithm. The leader assumes the role of coordinating the distributed tasks and ensuring their timely execution. Candidate nodes, on the other hand, are capable of detecting leader failures, initiating leader elections, and running autonomous components such as task scheduling within their locally interconnected worker nodes. Furthermore, candidates maintain a replica of the database, ensuring data availability and fault tolerance in the system. Candidates can establish connections with other candidates or directly with the leader, fostering a resilient and distributed network topology.

The third category encompasses worker nodes, also known as low-end nodes. These nodes are equipped with the runtime environment for supported languages (currently python), allowing them to execute tasks or binary files. Worker nodes are capable of efficiently handling computational workloads and contribute to the overall task distribution and execution within the RODS platform.

The connectivity between the different categories is crucial for the seamless operation of RODS. Edge nodes serve as the central point of connection, facilitating communication between the monitoring and worker nodes. This interconnected network of edge nodes promotes efficient task distribution and collaboration among





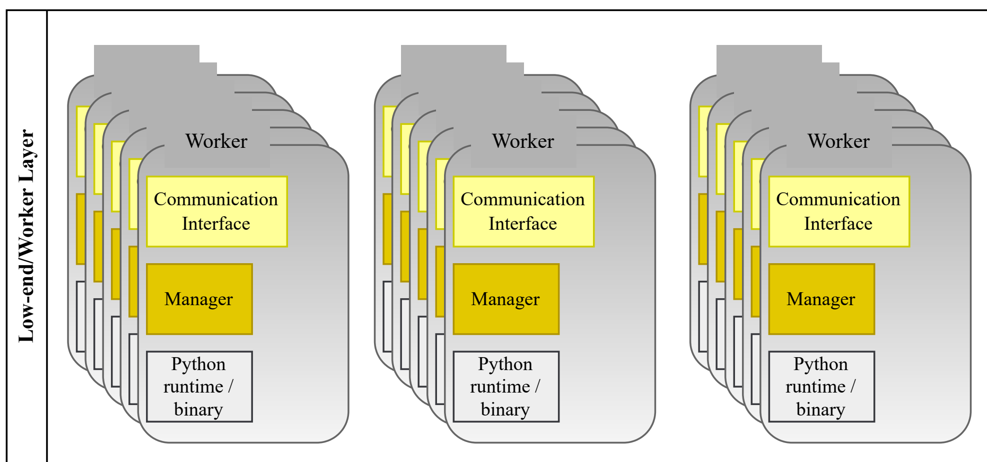


Figure 4. RODS Logical Architectural Model.

the nodes. It is important to note that the leader selection process in RODS is based on a modified version of the RAFT algorithm, tailored to support single-node operation. This modification removes the minimum requirement of three nodes for consensus, enabling the system to function effectively even with a limited number of nodes.

The task scheduling mechanism employed by RODS follows a simple First-In, First-Out (FIFO) approach. Tasks are queued based on their arrival time, ensuring fair and efficient execution. This simplicity in scheduling allows for easy adoption and scalability within edge computing environments. To join the RODS platform, a new node of any type must obtain a unique secure token called a secret. This token is generated by one of the candidate nodes or the leader and is securely shared with the new node using the `rodsctl` command-line application. By acquiring this token, the new node can establish secure connections and seamlessly integrate into the RODS cluster. The details of the joining node is explained in the following sections.

Communication within the RODS platform is facilitated through a socket-based protocol, which supports Transport Layer Security (TLS) for encryption. The implementation of TLS ensures the confidentiality and integrity of the exchanged messages, enhancing the overall security of the system. Furthermore, the system mandates that the certificate of any server, or master, is signed by a trusted certificate authority. This requirement ensures that only legitimate and authorized connections are accepted by the RODS platform.

In addition to the previously mentioned aspects, it is worth highlighting that the RODS platform is designed to operate seamlessly regardless of the operating system or CPU architecture of the host nodes. This platform-agnostic approach ensures that RODS can be deployed in heterogeneous computing environments, enabling compatibility and interoperability across a wide range of hardware and software configurations. This flexibility further enhances the versatility and accessibility of the RODS platform in various edge computing scenarios.

3.2. Rust Programming Language

Rust is a modern and innovative programming language that combines the performance of low-level systems languages with the safety and expressiveness of high-level languages [52]. Designed with a focus on memory safety, concurrency, and reliability, Rust provides developers with a powerful toolset for building robust and efficient software systems. Its unique features, including a strict ownership model, a borrow checker for preventing memory errors, and lightweight, thread-safe abstractions for concurrent programming, make Rust an ideal choice for applications that demand high performance, reliability, and security. With a growing ecosystem of libraries and tools, Rust offers a solid foundation for developing complex software solutions across diverse domains.

Rust's exceptional performance capabilities make it a compelling choice for implementing the RODS platform, especially in the context of edge computing. The language's design philosophy of controlling system resources allows developers to write highly optimized code that efficiently utilizes available hardware. Rust's low-level control over memory and efficient abstractions enable the RODS platform to achieve high-performance task distribution and management, ensuring effective

utilization of computing resources in edge computing environments. When comparing Rust's performance to C, another popular systems programming language, both languages offer similar capabilities in terms of low-level control and efficient code execution. However, Rust brings additional benefits to the table. While C provides manual memory management and control, it lacks built-in safety mechanisms. This can lead to common programming errors such as null pointer dereferences, buffer overflows, and memory leaks, which can negatively impact performance and introduce security vulnerabilities. Rust, on the other hand, combines the low-level control of C with advanced memory safety features. Rust's ownership model and borrow checker provide compile-time guarantees that prevent common memory-related errors. Rust ensures memory safety without sacrificing performance by enforcing strict rules and analysis at compile-time. This significantly reduces the likelihood of crashes, data corruption, and security vulnerabilities that could impact the stability and integrity of the RODS platform. Rust's focus on zero-cost abstractions allows developers to write high-level code with minimal runtime overhead. This means that developers can benefit from expressive and safe abstractions without sacrificing performance. Rust's borrow checker helps eliminate data races, ensuring safe concurrent programming and efficient utilization of multiple cores in edge computing scenarios.

Algorithm 1. An example of a Rust code

```rust
fn main() {
    let mut data = vec![0; 1000000];
    for i in 0..1000000 {
        data[i] = i;
    }
    println!("Data: {:?}", data);
}
```

Algorithm 2. Same example of Algorithm 1 in C language

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int* data = (int*)malloc(sizeof(int) * 1000000);
    for (int i = 0; i < 1000000; i++) {
        data[i] = i;
    }
    printf("Data: ");
    for (int i = 0; i < 1000000; i++) {
        printf("%d ", data[i]);
    }
    free(data);
    return 0;
}
```

The Algorithm 1 and 2 show the same example in Rust and C, respectively. In this example, the algorithm allocates an array of 1 million integers and populate it with

values from 0 to 999999. The Rust code uses a vector (`Vec`) to manage the dynamic array. In contrast, the C code manually allocates and frees memory using `malloc` and `free`. The critical difference between the two is how they handle memory safety. In the Rust code, the vector ensures memory safety by tracking the length and capacity of the array and automatically resizing it when needed. It also enforces strict ownership and borrowing rules simultaneously, preventing common memory-related bugs like buffer overflows and memory leaks. On the other hand, the C code relies on manual memory management, which is prone to errors. Suppose an accidental access memory beyond the allocated range or forgetting to free the memory. In that case, it can lead to undefined behavior, crashes, or memory leaks. It is worth mentioning that both codes have roughly the same performance.

Rust's memory safety features, including its ownership model and borrow checker, play a crucial role in enhancing the stability and security of the implementation in the context of the RODS platform. Rust mitigates common memory-related errors by enforcing strict rules at compile-time, such as null pointer dereferences and buffer overflows. The ownership model ensures that each piece of data has a unique owner, preventing data races and memory leaks. The borrow checker enforces strict rules for borrowing and mutation, guaranteeing that references to data remain valid and preventing dangling pointers or use-after-free errors. In the case of the RODS platform, these memory safety features provide significant advantages. They reduce the likelihood of crashes, data corruption, and security vulnerabilities that could compromise the stability and integrity of the system. By catching potential memory errors at compile-time, Rust enables developers to write code with higher confidence in its correctness. This is very important in edge computing environments, where reliability and security are critical.

The Algorithm 3 shows how the ownership mechanism works in commented (`///`) snippet. In the main function, the vector data is created and then passed as an argument to both the `print_vector` and `add_one` functions. However, when the developer passes a value as an argument to a function in Rust, it transfers ownership of that value to the function. So, in this case, the ownership of the vector data is moved to the `print_vector` function, and then it is moved again to the `add_one` function. This results in a compilation error because Rust enforces the principle of single ownership, meaning that two or more owners can not be given the same variable without (`&`) reference at a time. To solve this, the developers can use references (`&`) instead of passing the vectors by value. By passing a reference to the vector, the developers allow the functions to borrow the vector without taking ownership.

Algorithm 3. Memory safety in Rust implemented by ownership concept

```
fn main() {
    let data = vec![1, 2, 3];  // Create a vector

    // Pass the reference of the vector
    // to two separate functions
    print_vector(&data);
    add_one(&data);
    /// WRONG WAY WHICH GENERATES ERROR
    /// Pass the vector and code won't be compiled
    /// print_vector(data);
    /// add_one(data);



}
/// the arguments only accepts the vector
/// fn print_vector(vector: Vec<i32>) {
fn print_vector(vector: &Vec<i32>) {
    for element in vector {
        // Print each element in the vector
        println!("{}", element);
    }
}

/// the arguments only accepts the vector
/// fn add_one(vector: Vec<i32>) {
fn add_one(vector: &Vec<i32>) {
    for element in vector {
        // Print each element + 1 in the vector
        println!{"+1 is: {}", element + 1};
    }
}
```

Rust provides excellent support for concurrent programming, offering lightweight and thread-safe abstractions that enable efficient parallel execution and synchronization of tasks. This concurrency support in Rust is particularly beneficial for the RODS platform, as it allows optimal CPU and memory utilization and scalability in edge computing environments. One of the key features in Rust's concurrency model is the concept of lightweight threads, known as "async tasks." These tasks are managed by an asynchronous runtime, such as Tokio, widely used in the Rust ecosystem. Async tasks allow for the concurrent execution of multiple operations without the need for traditional threads, reducing overhead and improving performance. By leveraging async Rust and Tokio, the RODS platform can handle multiple concurrent operations, such as processing incoming requests, managing resource allocation, and orchestrating edge nodes' activities. This concurrency enables the RODS platform to efficiently utilize available resources, maximizing

system throughput and responsiveness. Async programming in Rust brings additional benefits beyond traditional concurrency. It allows developers to write non-blocking code that can efficiently handle I/O operations, such as network communication or disk access, without blocking the execution of other tasks. This non-blocking behavior is achieved through asynchronous functions and "await" expressions, which suspend the execution of a task until the awaited operation completes. With async programming, the RODS platform can handle many concurrent requests without being limited by the number of available threads. This scalability is crucial for edge computing scenarios where numerous devices and clients interact with the system simultaneously. The RODS platform can effectively handle concurrent operations, distribute tasks across edge nodes, and ensure smooth component coordination.

Rust's expressive type system ensures robust software development within the RODS platform. The language's strong static typing provides several benefits, including early error detection, improved code maintainability, and enhanced reliability and stability. Other key advantage of Rust's type system is its ability to catch errors at compile-time. By requiring variables and functions to be explicitly typed and enforcing strict type-checking, Rust eliminates a wide range of common programming mistakes before the code is even executed. This early error detection significantly reduces the likelihood of runtime failures. Additionally, the compiler's type inference capabilities alleviate the need for excessive type annotations, allowing for cleaner and more concise code. These aspects contribute to improved readability, making it easier to maintain and update the codebase as the RODS platform evolves over time. Rust's type system enables the detection of many common programming errors related to data manipulation and memory management. The ownership model and borrow checker in Rust enforce strict rules for memory safety, preventing issues like null pointer dereferences, data races, and buffer overflows. By catching these errors at compile time, Rust enhances the reliability and stability of the RODS platform, reducing the risk of crashes, data corruption, and security vulnerabilities. The expressive type system in Rust also aids in catching logic errors and promoting code correctness. Using algebraic data types (enums) and pattern matching allows developers to model complex data structures and capture all possible cases explicitly. This approach minimizes the potential for logic errors and helps ensure that all code paths are handled correctly. Additionally, Rust's type system supports the concept of "ownership," which facilitates resource management and prevents common pitfalls like dangling pointers or resource leaks.

One notable advantage of implementing RODS in Rust is the ease of portability. Written code in Rust can be compiled and run on various systems, thanks to the language's focus on platform compatibility. Most of the components in RODS are built using the standard library, which ensures portability across different platforms. Additionally, some components rely on the core library, allowing them to run without an operating system. This flexibility enables the RODS platform to be deployed on a wide range of edge devices, regardless of their operating system or CPU architecture.

Rust stands out as a sustainable programming language, exemplified by its comparative energy consumption in the Table 4. Pereira et. al., positioned as the second language with a value of 1.03, closely trailing the baseline of C at 1.00, Rust demonstrates its efficiency and eco-friendly nature [53]. This finding highlights Rust's ability to deliver optimal performance while minimizing environmental impact.

With its emphasis on performance optimization and meticulous resource management, Rust empowers developers to craft sustainable code that is both high-performing and reliable. By leveraging Rust, projects and applications can align with sustainable development principles, positively contributing to a greener future.

Table 4. A comparison between different programming languages, sorted by Energy consumption, the full table available in [53]

Programming Language	Energy ↑	Execution Time	Memory (Mb)
C	1.00	1.00	1.17 (Ranked 3rd)
Rust	1.03	1.04	1.54
C++	1.34	1.56	1.34
Ada	1.70	1.85	1.47
Java	1.98	1.89	6.01
...			
...			
Ruby	69.91	59.34	3.97
Python	75.88	71.90	2.80
Perl	79.58	65.79	6.62

To conclude this section, Table 5 gives a short overview of comparing Rust with other popular programming languages. The Go language is excluded from this comparison because the author does not have a deep understanding of the Go language. It is worth mentioning that the performance row is the author's grasp while experiencing those languages.

Table 5. A brief comparison between different possible options for implementing the RODS platform

Aspect	**Rust**	**C**	**Java**	**Python**
Performance				
Memory safety	Strong memory safety	Manual memory management	Garbage collection	Garbage collection
Concurrency	Safe and efficient concurrency	Manual synchronization	Built-in concurrency support	Global Interpreter Lock
Type System	Strong	Strong	Strong	Dynamic
Portability	HIGH	HIGH	Moderate	Moderate

3.3. Summary

The methodology chapter presented the detailed procedures and techniques employed in this thesis to address the research questions and achieve the stated objectives. By carefully designing the experiments and selecting appropriate methods, the necessary data to investigate and analyze the problem are gathered at hand. The implementation of the RODS platform, combined with the utilization of the Rust programming language, a solid foundation is built for a resilient and efficient orchestration platform for edge computing.

4. IMPLEMENTATION

The chapter delves into the intricate details of the implementation, including the components involved and the formation of nodes. The aspect of fault tolerance is also addressed to ensure the system's resilience in the face of failures. By following this methodology, a comprehensive understanding of the system's design and its practical implications can be achieved. Now the reasons for selecting Rust language for the implementation of RODS platform are explained in the previous chapter; this chapter will provide deep insights into the RODS platform and its components.

4.1. Components

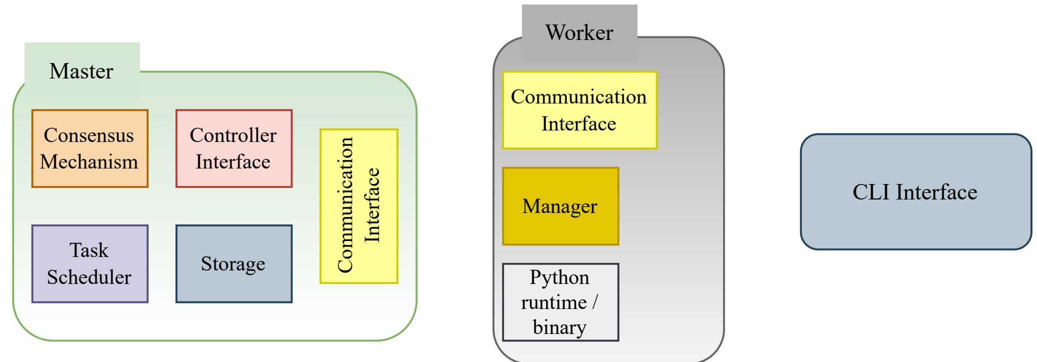


Figure 5. A Close look at the nodes and components.

In the RODS platform, various components have been implemented to enable efficient task distribution and management. As described in section 3.1, the architecture of RODS consists of different categories of nodes. Within each category, there are specific components that play essential roles. For simplicity, all nodes in the Edge/management layer will be referred to as "master nodes," as depicted in Figure 5.

Figure 5 illustrates the components within each type of node. These components include the Consensus Mechanism, Communication Interface, Task Scheduler, Storage, and Controller Interface for master nodes, as well as the Communication Interface, Manager, and Runtime for Python/binary executor in worker nodes. Additionally, the Command-Line Interface (CLI) component is independent and can be located and executed on any device to perform actions or retrieve status information from the nodes.

The Communication Interface[1], implemented alongside the Storage [2] component, is publicly available on GitHub [3] and can be utilized as a standalone library in separate projects. The Communication Interface employs raw sockets and implements its own message transfer protocol which is secured by TLS. The secure channel would not be established if the certificate of the private key is not signed by a certificate

[1]The project is called `async-socket`.
[2]The project is called `rust-rocksdb`.
[3]The projects are publicly available in https://github.com/vahidmohsseni/async-socket and https://github.com/vahidmohsseni/rust-rocksdb, respectively, with MIT open-source license.

authority (CA), and if it is a self-signed certificate the root certificate of CA must be presented to the other nodes. Notably, the entire communication interface is designed asynchronously to efficiently handle IO operations in the network. When a connection is established between two nodes, the node that accepts the connection runs an infinite asynchronous task to receive messages. Once a message is fully received, it is passed to the responsible task handler through channels in Rust. In RODS, the main controller of the platform serves as this task handler. On the other hand, the sending end of the established connection remains passive, waiting for incoming messages to be sent to the other node. To keep the channel open and avoid an idle state, the other node periodically sends "bit" characters every 1000 milliseconds.

The Storage component in the RODS platform, developed as an open-source project by the author, is a critical module responsible for efficient and reliable data storage. Leveraging the proven database architecture of RocksDB [54], the Storage component offers a robust foundation for managing data in edge computing environments. One of the key design considerations in developing the Storage component was to provide a minimal and intuitive application programming interface (API) that facilitates seamless integration with other modules of the RODS platform. The API abstracts away the complexities of the underlying storage implementation, allowing developers to interact with the storage component straightforwardly and efficiently. This simplicity promotes ease of use and accelerates the development process for applications leveraging the RODS platform. The Storage component incorporates two main storage strategies to optimize data access and persistence: in-memory and disk persistence. By utilizing an in-memory storage approach, the component employs a binary search tree (BST) algorithm to achieve fast data retrieval. This strategy is particularly useful for scenarios where low-latency access to frequently accessed data is crucial, enabling applications to retrieve information with minimal delay. The BST algorithm efficiently organizes and indexes the data in memory, facilitating quick search and retrieval operations. To ensure durability and reliability, the Storage component also supports disk persistence using a log-structured merge (LSM) [55] data structure. The LSM data structure efficiently handles write operations by utilizing a sequence of sorted on-disk data files and an in-memory write buffer. This approach minimizes disk `I/O` operations and optimizes write performance, making it suitable for scenarios where data durability and consistency are of utmost importance. The LSM data structure ensures that data modifications are efficiently logged and periodically merged to maintain a compact and consistent on-disk representation. Combining the benefits of in-memory storage and disk persistence, the Storage component provides a comprehensive storage solution for edge computing environments. It offers the flexibility to accommodate diverse data access patterns and requirements, optimizing both read and write operations. Whether applications need rapid access to frequently accessed data or require persistent and reliable storage, the Storage component within the RODS platform can adapt and deliver the necessary performance and durability. As the Algorithm 4 shows the APIs of the component, it is totally observable that all the complexities are behind the abstractions.

Algorithm 4. The public interfaces of the Database Engine in storage component

```
use std::{sync::{Arc, Mutex}, path::PathBuf, io};

use crate::{db::Db, entry::Entry};

#[derive(Clone)]
pub struct DBEngine {
    pub database: Arc<Mutex<Db>>,
}

impl DBEngine {
    pub fn new(dir: PathBuf) -> io::Result<Self>{
        Ok(Self { database: Arc::new(Mutex::new(Db::init_from_existing(dir)?)) })
    }

    pub fn set(&mut self, key: &[u8], value: &[u8]) -> io::Result<()> {
        let mut db = self.database.lock().unwrap();
        db.set(key, value)?;
        Ok(())
    }

    pub fn instant_set(&mut self, entry: &mut Entry) -> io::Result<()> {
        let mut db = self.database.lock().unwrap();
        db.instant_set(entry)?;
        Ok(())
    }

    pub fn get(&mut self, key: &[u8]) -> Option<Entry> {
        let mut db = self.database.lock().unwrap();
        db.get(key)
    }

    pub fn get_keys_with_pattern(&mut self, pattern: &[u8]) -> Vec<Entry> {
        let mut db = self.database.lock().unwrap();
        db.get_keys_with_pattern(pattern)
    }

    pub fn delete(&mut self, key: &[u8]) -> io::Result<()> {
        let mut db = self.database.lock().unwrap();
        db.delete(key)
    }

    pub fn get_snapshot(&mut self) -> Vec<u8> {
        let mut db = self.database.lock().unwrap();
        db.get_snapshot()
    }

    pub fn set_snapshot(&mut self, raw_data: Vec<u8>) -> io::Result<()> {
        let mut db = self.database.lock().unwrap();
        db.set_snapshot(raw_data)
    }

    pub fn purge_database(&mut self) -> io::Result<()> {
        let mut db = self.database.lock().unwrap();
        db.purge_database()
    }
}
```

The Controller Interface serves as the core of the RODS platform. It is a stateful component that receives structured messages from nodes or external sources and determines the appropriate actions and responses based on the message source and type. For instance, if the leader node fails, a candidate's controller interface may initiate an election with other candidate nodes to select a new leader in the system. Additionally, the controller interface triggers the scheduler component when a new task is introduced into the system.

The current consensus mechanism utilized in RODS is based on a modified version of the RAFT algorithm. This component is designed as an independent module to provide flexibility to the RODS platform, allowing the configuration of different consensus approaches if needed.

The Task Distribution Mechanism utilizes the RODS scheduler to find a suitable node for task execution. The current scheduling policy follows a First-In-First-Served approach. As this component is designed independently without dependencies on other components, alternative scheduling algorithms can be implemented and configured within RODS to cater to specific use cases.

Within the worker node, the communication interface functions similarly to the master node, but with the additional capability of sending a request to join the cluster. The manager component handles node management and task status monitoring, communicating with the cluster to keep the controller interface updated.

The CLI interface, known as `rodsctl`, is an independent component that can be executed from any accessible point within or outside the cluster to facilitate user interaction. It avoids the use of common protocols like Hyper Text Transfer Protocol (HTTP)[4] and instead utilizes a specialized protocol, aligning with the communication interface's design choice. This selection minimizes dependencies on external libraries, resulting in a small overall binary size for the entire RODS project.

4.2. Nodes Formation

The formation and connectivity of nodes within the RODS platform play a crucial role in establishing a distributed and resilient system. In this subsection, we will explore the process of connecting nodes and the mechanisms that enable effective communication and coordination among them.

To form a cluster, RODS employs a token-based approach. When a new node, whether it belongs to the cloud, edge, or worker category, intends to join an existing cluster, it requires a unique secure token known as the "secret." One of the existing candidate or leader nodes within the cluster generates this secret using `rodsctl`. The node seeking to join the cluster can acquire this secret through the `rodsctl join` command. Once a fresh node possesses the secret, it can initiate the connection process with the cluster. Using the secure token, the node establishes a connection to a master node, which serves as the entry point to the RODS cluster. The master node verifies the authenticity of the secret and, upon successful verification, grants the joining node access to the cluster. This process ensures secure and controlled access to the RODS system, preventing unauthorized nodes from joining. The connectivity between different categories of nodes follows a hierarchical structure. The edge nodes have distinct types: Leader, Candidate, and Replica. Only one leader node exists within the system at any given time. The leader is determined through a consensus algorithm, such as the modified version of the RAFT algorithm employed in RODS, or alternatively, it is the first node and needs to be initialized with `rodsctl init` command. Notably, RODS allows for the operation of a single leader, eliminating the

[4]At the early stages of the development, RODS employed tide library to make a web-server-based interface for the CLI. However, in the first release of the platform, lots of dependencies appeared, making the final size of the binary large.

minimum requirement of three nodes typically associated with consensus algorithms. Figure 6 demonstrates a couple of possibilities for the formation. This figure eliminates the components inside different types of nodes for the sake of clarity.

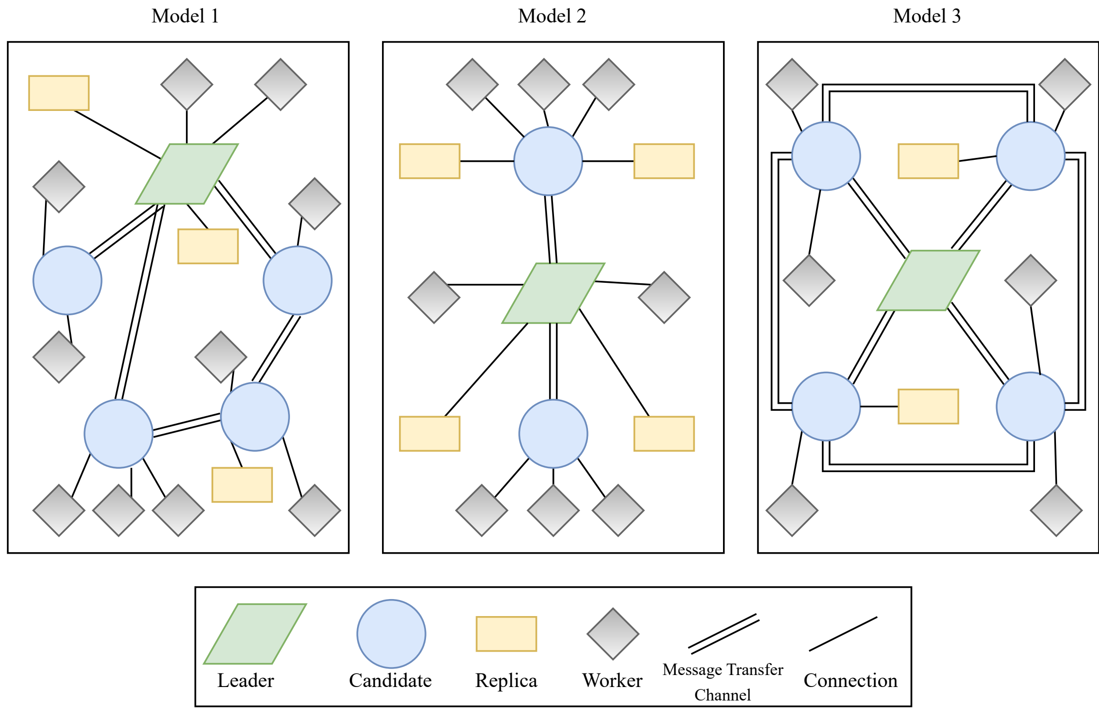


Figure 6. Different RODS cluster formation. The platform is flexible enough to shape any kind of hierarchy between different types of nodes.

Candidate nodes, another type within the edge category, play crucial roles in detecting leader failures, initiating leader elections, and running autonomous components like scheduling within their interconnected worker nodes. Candidate nodes can establish connections with other candidates or directly with the leader node. This flexibility allows for effective communication and coordination within the RODS cluster. The third type of edge node is the Replica, which primarily serves as a replication of the storage component. Although, Replicas can also transform into Candidate nodes by interacting with the CLI interface. This capability further enhances the fault tolerance and adaptability of the RODS system. The communication between nodes in the RODS cluster is facilitated through a message-passing protocol implemented over sockets. This protocol supports secure communication via TLS encryption. To ensure secure connections, the certificate of each server, also referred to as a master node, must be signed by a trusted CA. This requirement ensures that the RODS system only accepts incoming connections from authenticated sources, bolstering the overall security of the cluster.

4.3. Fault Tolerance

Fault tolerance is a crucial aspect of the RODS platform, ensuring the system's resilience in the face of node failures and maintaining uninterrupted operation. In

this subsection, we will explore the fault tolerance mechanisms employed by RODS, specifically focusing on leader failure detection and the process of electing a new leader.

Algorithm 5. Leader Failure Detection and Election

Input: All candidate nodes in the RODS cluster

1 **Upon detecting the absence of the leader node by controller component:**;
2   Nodes initiate the election process;
3   Fetch the latest information about all other candidate nodes in the cluster;
4 **Election Initialization:**;
5   Nodes introduce themselves as candidates for leadership;
6   Add a small random delay before sending election ballots to avoid ties;
7 **First Round of the Election:**;
8   Nodes count the votes received from other candidates;
9   Communicate the election results to every other node [a];
10 **Election Result Evaluation:**;
11   If a candidate did not receive the expected message or received conflicting results:;
12   Realize that it did not succeed in the election;
13   If a candidate receives the expected election result:;
14   Declare itself as the new leader;
15   Update the database and storage across all candidate nodes and replicas;
16   **Election Resolution:**;
17   If a clear winner has emerged, the new leader takes control, and the system proceeds;
18   If no clear winner is determined:;
19   Initiate a second round of the election to determine the leader;
20   **Repeat the election process until a new leader is elected or a clear winner emerges**

Output: New leader elected and database/storage updated accordingly

---

[a] The reason behind this is explained in Discussion chapter section 5.2.

As Algorithm 5 enumerates, when a leader node becomes disconnected from the cluster for any reason, the first nodes to notice its absence are those directly connected to it. These nodes initiate an election process by gathering the latest information about all candidates in the cluster. A small random delay is introduced before sending the election ballots to introduce themselves as candidates for leadership to prevent potential ties or situations where only two candidate nodes remain. After the initial round of the election, each participating node counts the votes and communicates the results to every other node. Suppose a candidate node did not receive the expected message or received conflicting results. In that case, it realizes that it did not succeed in the election. On the other hand, if a node receives the expected election result, it declares itself as the new leader, and the database and storage are updated across all candidate nodes and replicas. In the event that none of the above scenarios occur,

indicating no clear winner, the nodes can proceed to a second round of the election to determine the leader. This ensures that a new leader is elected and the system can continue its operation without interruption. Appendix 8 is reserved for the code that implements the modified version of the RAFT algorithm that is described in this subsection.

The fault tolerance mechanism implemented in RODS provides a robust solution for handling leader failures. By enabling candidate nodes to initiate an election process and using a multi-round approach, the system ensures the selection of a new leader to maintain the continuity of operations. Through this fault tolerance mechanism, RODS enhances its resilience and guarantees the stability and availability of the distributed system.

4.4. Summary

Through the acquiring the methodology, in this chapter, the second research question of this thesis is addressed, the development of a specialized orchestration platform for edge computing, by demonstrating how the RODS platform, with its distributed task distribution mechanism and fault-tolerant architecture, can effectively manage edge nodes and optimize task allocation. The implementation process involved

1. the careful selection of components,
2. the utilization of communication interfaces and storage mechanisms, and
3. the integration of consensus mechanisms to ensure the system's reliability and fault tolerance.

The implementation also allowed us to assess the scalability and usability of the RODS platform through extensive testing and evaluation. We gained insights into the framework's efficiency, memory management, and concurrent processing capabilities by conducting experiments and analyzing the collected data. Moreover, these findings contribute to the overall understanding of the Rust programming language and its applicability in developing resilient distributed systems for Edge computing scenarios.

5. DISCUSSION

This chapter delves into a comprehensive analysis of the RODS platform, highlighting its comparison with other works, practical challenges encountered during implementation, generalizability in cloud environments, and overall impact on edge computing orchestration to fulfill the third research question of this thesis. This chapter aims to provide insights into the comparison between RODS and other existing platforms in terms of memory and CPU consumption, present practical lessons learned from the implementation process, propose the applicability of container-based technologies for enhanced isolation in cloud environments, and summarize the contributions and advancements brought by the RODS platform. Furthermore, the chapter discusses the limitations encountered during the implementation process. It presents potential areas for future enhancements, such as consensus mechanisms, security enhancements, advanced scheduling policies, and integration of a new federated learning framework. By exploring these essential aspects, we gain a deeper understanding of the capabilities, limitations, and potential future directions of the RODS platform in the context of edge computing.

5.1. Brief Comparison with State-Of-The-Art

In this section, a brief comparison between the RODS platform and state-of-the-art orchestration systems is provided, including Oakestra, Kubernetes, K3s, and MicroK8s. While it is important to note that this thesis does not aim to provide an extensive performance analysis and benchmarking of these platforms, this section offers low-key insights into their key features and highlights how RODS addresses certain limitations in the context of edge computing orchestration. Two different infrastructure was considered to set up the experiment to compare the results. The first setup is conducted on ARM-based devices, a combination of NVIDIA Jetson Nano [56] and Raspberry 4 [57] devices, with 4GB of memory and four cores each of CPU ARM A57 and ARM A72, respectively. The other setup consisted of provisioned virtual machines on CSC[5] servers with ten vCPUs and 16GB of memory. RODS resource usage for memory and CPU utilization was roughly the same in those environments. To draw the figures, the highest numbers are picked for RODS in this section.

The results of the Figure 7 are adapted from [18] since it was impossible to find any specific instructions to reproduce the result of Oakestra's framework. As Figure 7 illustrates the difference between CPU usage in the compared platforms, Kubernetes, Oakestra, and RODS -in idle mode and different numbers of nodes- have a constant value for CPU usage, which demonstrates that these platforms can scale out up to large number of nodes without overusing the resources. Moreover, RODS is slightly using fewer CPU times than Oakestra, although both outperform Kubernetes in this comparison. This comparison is not fair from different points of view. For example, In Kubernetes, there are other services functioning in the system to ensure the stability of the deployment in Kubelet, ranging from networking services to constant health

[5]IT for Science provider at `https://www.csc.fi`

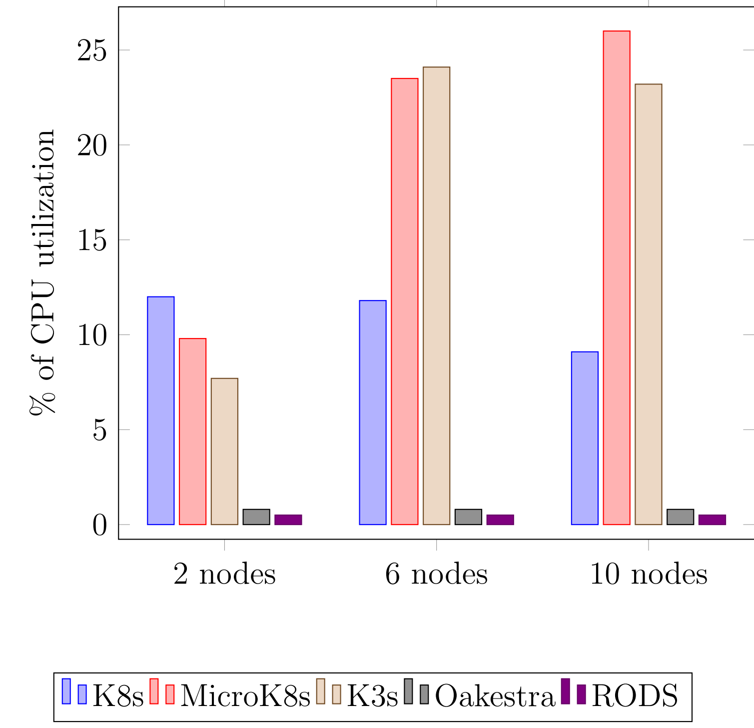


Figure 7. Percentage of CPU utilization in different platforms.

checking of side-services. In contrast, RODS and Oakestra do not have such services inside.

The Rust language provides better memory management as introduced in section 3.2. The advantage of this feature is evident in Figure 8. RODS outperforms every other platform in memory consumption significantly. However, as explained before, these comparisons can be misleading or unfair. For example, Oakestra is written in Python, a high-level scripting language, and clearly, it will use more resources due to its interpreting nature, likewise, the group of authors in [58] mentioned benchmarking flaws in their article.

5.2. Address to Practical Challenges

During the implementation of the RODS platform, several practical challenges were encountered, and valuable lessons were learned. This section reflects on these challenges, discusses any identified limitations or trade-offs, and shares insights gained from overcoming them. Additionally, it highlights the practical implications of these challenges and lessons for the future development and deployment of the RODS platform.

One of the key challenges addressed during implementation was ensuring consistency within the RODS platform. To achieve this, a mechanism was devised where updates made by the leader node were broadcasted to all directly connected nodes, which, in turn, propagated the updates to their locally connected nodes. This process continued in a cascading manner throughout the network. To prevent duplicate updates in cases where there are looped connections, the leader node sends both the

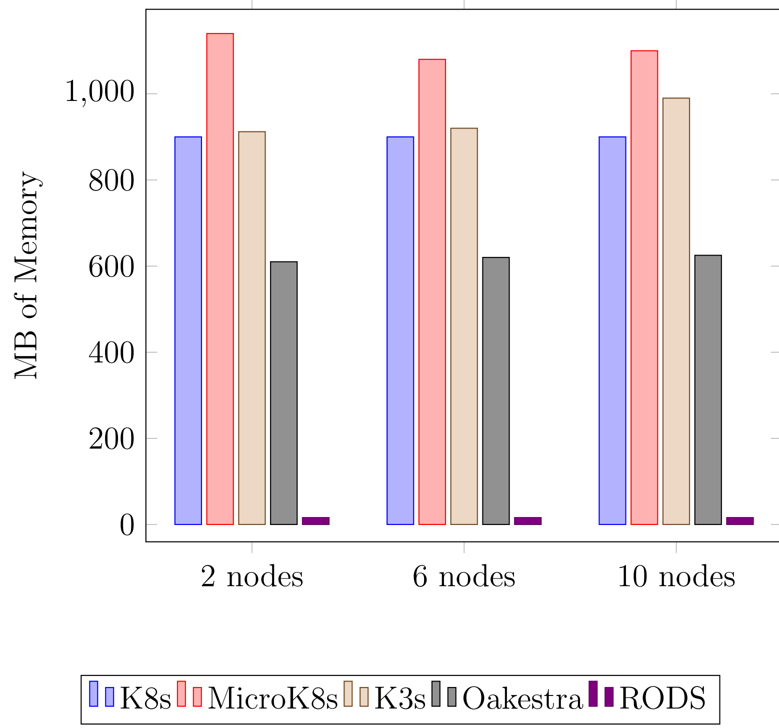


Figure 8. Comparison of memory usage in different platforms.

update and a unique log ID to the connected nodes. This ensured that the same update was not pushed multiple times as it is possible in the model 3 in Figure 6. As depicted in Figure 9, a consistency diagram illustrates this process, highlighting the flow of updates and log IDs within the RODS platform. Another way of a write operation is the scenario where a candidate node wants to update the data storage. In this scenario, the candidate node will send the write operation to the leader, and then the leader will repeat the above-mentioned update process in all the nodes.

Another valuable lesson learned during implementation was related to holding elections and selecting a new leader. The failure of a leader could potentially result in partitioning the devices within the cluster. To address this, the RODS platform incorporates a feedback mechanism by seeking confirmation from all other candidate nodes in the system regarding the election results. This ensures that all candidate nodes acknowledge and recognize the elected leader, thereby mitigating the possibility of partitioning. Figure 10 visually demonstrates this scenario and emphasizes the importance of confirming the election results with all candidate nodes.

In Figure 10, the cluster topology is depicted, showcasing the convergence of two partitions formed by nodes joining from different points, namely the leader and the candidate (Figure 10.(a)). In Figure 10.(b), a scenario unfolds where the leader becomes disconnected from the cluster, leading two directly connected nodes to detect this failure ahead of others. Consequently, these nodes initiate an election process based on the consensus algorithm described in 4.3. Figure 10.(c) displays the occurrences of Election 1 and Election 2, along with the respective results indicated above each candidate. It becomes evident that both candidates perceive themselves as the winner, judging by the votes and the number of connected nodes supporting them. However, a crucial node positioned between them possesses the capability to ascertain

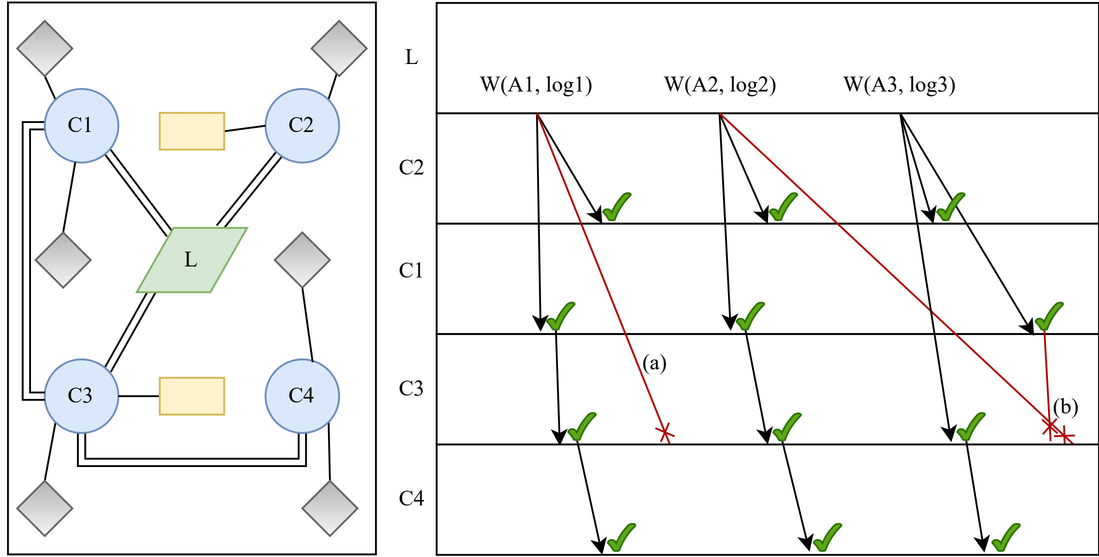


Figure 9. Illustration of an example in consistency model. The left figure is the topology, and the right figure shows the flow. As marked with (a) in the figure, the leader sent the `write` on data A with a value of 1 to every connected node. Candidate 3 receives this operation from two sources, i.e., the leader and Candidate 2. However, it only accepts one of those operations because the log IDs are the same. Since Candidate 2's message is earlier than the leader's, the other one will be dropped. The scenario in the marked (b) in the figure shows that a `write` operation on data A with a value of 2 was received after the `write` operation on data A with a value of 3. Since the log ID of the latter is greater than the former, the former operation will be dropped.

the true winner by validating the received results (Figure 10.(d)). As demonstrated in Figure 10.(e), the node that held Election 1 emerges as the rightful winner, ultimately assuming the role of the new cluster leader (Figure 10.(f)).

Valuable lessons were learned, and insightful approaches were developed while overcoming the challenges faced during implementing the RODS platform. One notable lesson was the importance of adopting a communication model that prioritizes message passing over shared memory. A famous slogan in Go Lang [59] documentations states that *"Do not communicate by sharing memory; instead, share memory by communicating"*. This principle aligns with the design philosophies of programming languages like Rust and Go, emphasizing communicating between concurrent processes rather than relying on shared memory. By adhering to this principle, the RODS platform achieved several benefits. First, it promoted a clear separation of concerns and encapsulation, as each component communicated with others through well-defined message interfaces. This reduced the complexity associated with managing shared memory and minimized the chances of data corruption or race conditions. Second, the message-passing model facilitated fault isolation and improved the system's resilience. In the event of a failure or crash in one component, it had minimal impact on others since they relied on message passing instead of directly sharing memory. This approach enabled better fault tolerance and enhanced the overall robustness of the RODS platform.

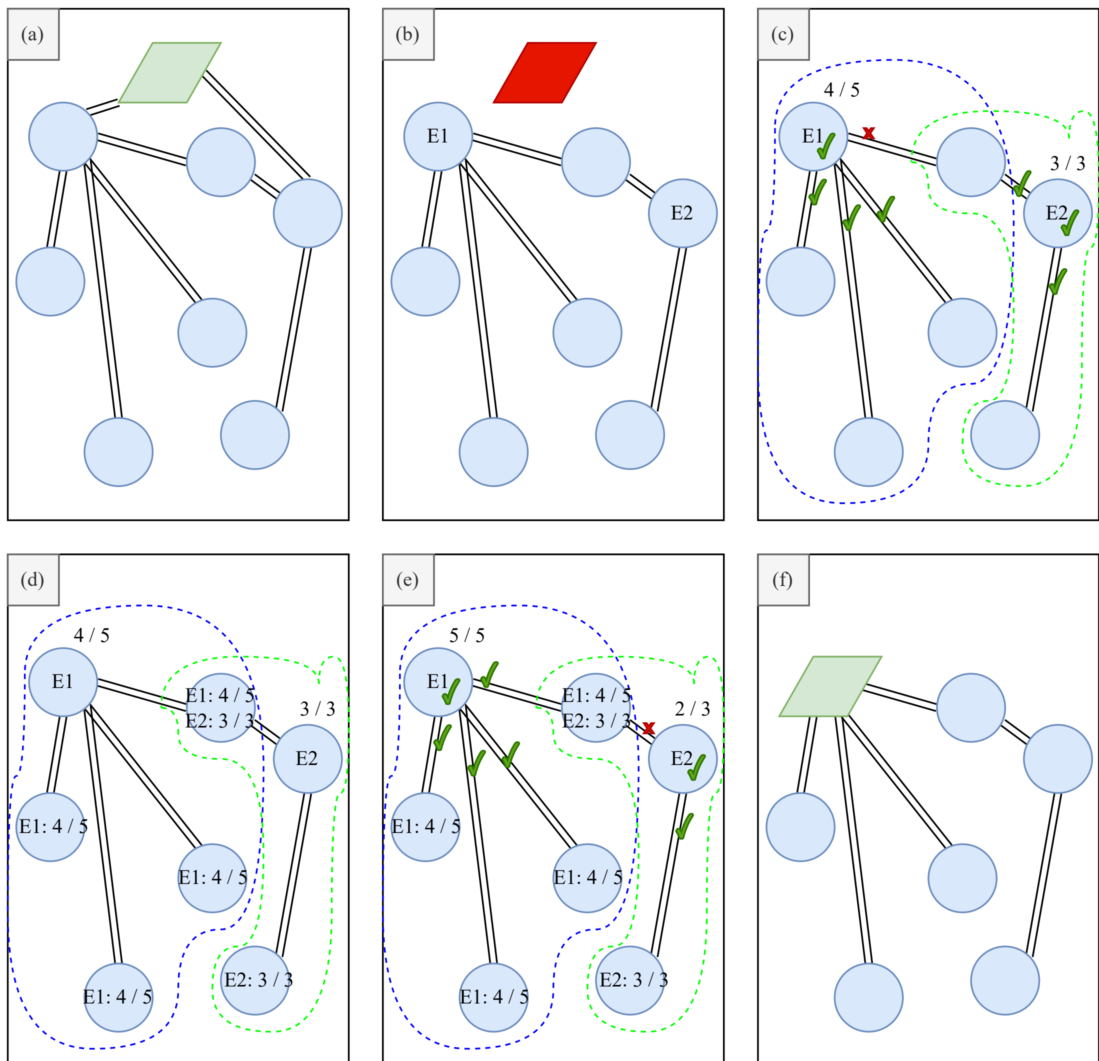


Figure 10. Demonstration of an election to choose new leader, ordering from (a) to (f).

5.3. Limitations and Future Enhancements

The following section discusses the limitations of the RODS platform in its current implementation. It presents potential solutions and future enhancements to address these challenges. By acknowledging these limitations and proposing ways to overcome them, the RODS platform can continue to evolve and meet the evolving needs of edge computing environments more effectively.

5.3.1. Consensus

One of the limitations of the RODS platform lies in the limited set of implemented consensus algorithms, primarily based on the Raft consensus protocol. While Raft provides a robust and fault-tolerant approach for achieving consensus in distributed systems, it does have certain limitations. For instance, relying on a single leader introduces a single point of failure. In the event of leader failure, the system incurs overhead in selecting a new leader, impacting overall system performance. A creative

approach involving AI and agent-based systems can be considered to overcome these limitations and explore new avenues for achieving consensus in the RODS platform.

By leveraging the capabilities of AI and allowing nodes to communicate freely with each other, an agent-based consensus mechanism can be designed. In this approach, each node can act as an intelligent agent that employs learning algorithms to make consensus decisions based on the information it receives from other nodes. The agent-based consensus system would distribute decision-making authority across multiple nodes, eliminating the reliance on a single leader. By allowing nodes to learn from historical data and adapt their decision-making processes, the system can dynamically adjust to changing conditions and ensure efficient consensus even in the presence of failures or dynamic network topologies. This innovative approach introduces several benefits. Firstly, it eliminates the single point of failure, enhancing the fault tolerance and robustness of the consensus mechanism. Secondly, by leveraging AI and learning algorithms, the system can adapt and optimize its decision-making processes based on the specific characteristics and requirements of the edge computing environment. This can lead to improved scalability, performance, and responsiveness. However, it is important to note that implementing such an AI-based consensus mechanism would introduce its own challenges. Designing effective learning algorithms, ensuring proper communication and coordination among nodes, and addressing potential security and privacy concerns would require further research and development. Nevertheless, exploring the potential of AI and agent-based systems for consensus in the RODS platform opens up exciting opportunities for enhancing the reliability and efficiency of edge computing orchestration.

5.3.2. Scheduling Policies

The task scheduling algorithm implemented in the RODS platform is currently limited to a basic FIFO approach. While FIFO scheduling provides simplicity and fairness in task execution, there are instances where it may not achieve optimal performance. Two such instances are task prioritization and dynamic workload scenarios. In task prioritization, certain tasks may have higher urgency or criticality compared to others. With FIFO scheduling, tasks are executed strictly in the order they arrive, regardless of their priority. As a result, high-priority tasks may experience delays if a large number of lower-priority tasks are queued ahead of them. This can lead to decreased responsiveness and potential violations of performance requirements. Similarly, in dynamic workload scenarios where tasks have varying resource requirements or time constraints, FIFO scheduling may not efficiently utilize available resources. For example, suppose a resource-intensive task arrives early in the queue and consumes a significant portion of available resources. In that case, subsequent tasks with lower resource requirements may experience resource starvation or increased waiting times, impacting overall system efficiency.

To address these limitations, a more sophisticated task scheduling algorithm, such as Shortest Job First (SJF) or Priority Scheduling, can be employed. SJF scheduling prioritizes tasks based on their expected execution time, allowing shorter tasks to be executed first and potentially reducing overall turnaround time. Priority Scheduling assigns priority levels to tasks based on their urgency or importance, ensuring that

high-priority tasks are executed ahead of lower-priority tasks. Compared to FIFO scheduling, these advanced algorithms provide better performance in scenarios where task prioritization or dynamic workload management is crucial. They optimize resource utilization, reduce response times, and enhance overall system efficiency. However, it is important to mention that implementing these algorithms introduces additional complexity in terms of task prioritization criteria and dynamic priority adjustments, which require careful consideration and fine-tuning.

By implementing more advanced task scheduling algorithms, the RODS platform can provide enhanced performance and responsiveness, particularly in scenarios where task prioritization and dynamic workload management are critical. The choice of scheduling algorithm should align with the specific requirements and characteristics of the edge computing environment, striking a balance between fairness, responsiveness, and efficient resource utilization.

5.3.3. Security Enhancement

The RODS platform incorporates several security enhancements to ensure data and communications' confidentiality, integrity, and authenticity. One of the primary security measures implemented is using a private/public key mechanism in TLS protocols. This cryptographic mechanism establishes secure connections between nodes, preventing unauthorized access and eavesdropping. By employing TLS with private/public key encryption, the RODS platform establishes a secure channel for communication, safeguarding sensitive information exchanged between nodes. Additionally, integrating a trusted Root CA further enhances security by verifying the authenticity of certificates presented by nodes during the handshake process. This helps prevent the possibility of joining threats or unauthorized entities gaining access to the system. System-wide implementation of a certificate renewal component is essential to be considered. Currently, in the RODS platform, the process of signing certificates for new nodes joining the system relies on the leader or the cluster. While this mechanism ensures secure certificate issuance, it presents limitations in terms of scalability and efficiency. A more robust and scalable certificate renewal component should be implemented to address this limitation. This would involve automating the process of certificate generation and signing, allowing new nodes to obtain valid certificates without relying solely on the leader or cluster. By distributing the responsibility of certificate signing, the system can accommodate a more significant number of nodes joining the network and reduce the overhead on the leader or cluster. In addition to secure certificate management, the RODS platform also emphasizes the importance of securely storing security keys. Security keys, including private keys for TLS encryption, should be stored safely and protected to prevent unauthorized access or misuse. Utilizing secure key management practices, such as encryption and access control mechanisms, ensures the confidentiality and integrity of the keys, thereby minimizing the risk of key compromise and potential security breaches. The RODS platform can strengthen its overall security posture by addressing these security considerations and implementing the necessary enhancements. Incorporating a scalable certificate renewal component and robust key management practices will

bolster the framework's ability to securely operate in edge computing environments, protecting sensitive data and ensuring the integrity of communication channels.

5.3.4. A New Framework for Worker Nodes

One of the limitations of the current RODS platform is its support for Python runtime in worker nodes. While Python offers excellent flexibility and a wide range of libraries for general-purpose programming, it may not be the optimal choice for specific specialized tasks, such as machine learning algorithms used in federated learning applications.

A new framework, similar to the FLOWER framework [60], can be developed to address this limitation and provide a more comprehensive environment for federated learning. This framework would cater specifically to machine learning applications, offering a dedicated environment that supports the execution of machine learning algorithms across the distributed edge network. The new framework would provide developers with a standardized interface and tools to deploy and manage federated learning applications within the RODS infrastructure. It would encompass features such as data partitioning, model aggregation, and communication protocols tailored to the unique requirements of federated learning scenarios. By leveraging this framework, developers can harness the power of distributed machine learning and train models collaboratively across the network of edge devices. This specialized framework for machine learning in the RODS ecosystem would open up opportunities for advanced analytics and decision-making at the edge. It would allow developers to harness the computational capabilities of edge devices and enable them to perform complex machine-learning tasks closer to the data source, minimizing the need for centralized processing and reducing latency. The framework can provide additional functionalities such as model versioning, model lifecycle management, and security mechanisms to protect the confidentiality and privacy of sensitive data. By incorporating these features, the RODS platform can empower developers to build and deploy sophisticated federated learning applications securely and efficiently.

5.4. Generalizability and Applicability

The RODS platform can be significantly enhanced in terms of generalizability and applicability by integrating container-based technologies like Docker [61] and LXC [62]. These technologies offer numerous advantages that contribute to improved resource isolation, scalability, and overall efficiency within cloud environments. One significant benefit of containerization is the ability to encapsulate applications and their dependencies into portable and lightweight containers. Packaging the RODS platform components and their dependencies as containers makes it easier to deploy and manage the platform across different cloud environments. Containers provide consistent execution environments, ensuring seamless operation of the RODS platform across diverse infrastructure setups. Moreover, containerization facilitates efficient resource isolation as each container operates within its own isolated environment. This isolation ensures that the RODS platform components can effectively utilize resources without interference or conflicts with other services running on the same

infrastructure. The dynamic allocation and management of resources for individual containers enhance scalability and enable the RODS platform to adapt to varying workload demands.

However, incorporating container-based technologies into the RODS platform introduces various challenges and considerations. One such challenge is the design and orchestration of deploying multiple containers that constitute the platform. Proper management of container lifecycles [63], networking, and storage interactions is crucial to ensure the smooth functioning of the RODS platform. Additionally, evaluating the impact of containerization on performance, security, and manageability is essential. Although containers offer lightweight virtualization, there may be a slight performance overhead due to the additional layer of abstraction. Security considerations, such as ensuring isolation between containers and securing container images, must be addressed to protect sensitive data and maintain the integrity of the RODS platform. Effectively managing many containers and their configurations requires robust container orchestration tools and efficient monitoring mechanisms. By addressing these challenges and considerations, the RODS platform can benefit from adopting container-based technologies in cloud environments. The enhanced resource isolation, scalability, and manageability provided by containers contribute to the platform's generalizability, allowing it to be effectively deployed and utilized across various cloud infrastructures.

In addition to the considerations mentioned above, network-related issues play a crucial role in the generalizability and applicability of the RODS platform when integrating container-based technologies in cloud environments. Containerization introduces a network layer that facilitates communication between different containers and external services. However, careful management of network configurations is necessary to ensure seamless connectivity and efficient data transfer within the RODS platform. One challenge lies in establishing network connections and enabling container communication, mainly when the platform is distributed across multiple cloud instances or nodes. Proper network design, including configuring network interfaces, routing, and firewall rules, becomes critical to maintain secure and reliable container communication. Furthermore, deploying the RODS platform in cloud environments requires considerations for network bandwidth and latency. As the platform interacts with various components such as databases, storage systems, and external services, network performance directly affects overall system responsiveness and data transfer speeds. Optimizing network configurations, implementing caching mechanisms, or utilizing content delivery networks (CDNs) can help mitigate latency issues and improve the performance of the RODS platform. Security, from networking point of view, is another aspect related to network considerations [64]. When containerizing the RODS platform, ensuring proper network segmentation and isolation is vital to protect sensitive data and prevent unauthorized access. Implementing network security measures, such as encryption, secure communication protocols, and strict access control policies, helps safeguard the platform and its data from potential network-related vulnerabilities. Lastly, monitoring and troubleshooting network-related issues become crucial in a containerized environment. Efficient network monitoring tools and mechanisms allow administrators to promptly identify and address network bottlenecks, connectivity problems, or performance issues. Proactive monitoring helps maintain the stability and reliability of the

RODS platform's network infrastructure. By addressing these network-related issues, the RODS platform can leverage the benefits of container-based technologies in cloud environments while ensuring seamless and secure container communication, optimizing network performance, and providing reliable network infrastructure for the platform's operations.

5.5. Summary

In the discussion chapter, various essential aspects, including a brief comparison with state-of-the-art approaches and addressing practical challenges, were encompassed. The comparison highlights the unique features and advancements offered by RODS, showcasing its potential to cater to evolving edge computing needs. Practical challenges such as consensus, scheduling policies, security enhancement, and a new framework for worker nodes are then discussed. An innovative distributed mechanism using AI and agent-based systems is proposed to add an innovative method for consensus. Advanced scheduling algorithms like SJF or Priority Scheduling are suggested to overcome the limitations of the basic FIFO policy. Security enhancement involves scalable certificate renewal and secure key management practices. Additionally, a specialized framework dedicated to machine learning is proposed to enhance the Python runtime in worker nodes. The chapter emphasizes the generalizability and applicability of RODS by integrating container-based technologies like Docker, considering network configurations and security. By addressing these challenges and incorporating enhancements, the RODS platform can become a standard tool for hybrid and flexible environments.

6. CONCLUSION

The concluding chapter of this thesis provides an overview of the impact, contributions, and a summary of the research conducted on the design and implementation of the RODS platform for edge computing orchestration. This chapter reflects upon the findings and outcomes of the study, highlighting the significance of the research in addressing the limitations of existing orchestration platforms, advancing the field of edge computing, and providing practical insights for future development and deployment.

6.1. Impact and Contribution

The impact and contribution of this study in the field of edge computing orchestration are significant and far-reaching. The RODS platform has made substantial advancements in the field by addressing existing orchestration platforms' limitations and explicitly targeting the requirements of edge computing. This research has shed light on the gaps that exist between the unique demands of edge computing and the capabilities of platforms like Kubernetes. The RODS platform serves as a specialized orchestration solution designed explicitly for edge environments. Its design, algorithms, and architectural innovations have filled gaps in knowledge and understanding within the rapidly evolving field of edge computing orchestration. By optimizing resource allocation, handling intermittent network connectivity, and meeting stringent latency requirements, the RODS platform has made notable contributions to the field. An important contribution of this thesis lies in the practical implications and lessons learned from implementing the RODS platform. The challenges encountered during implementation have provided valuable insights for future development and deployment efforts. By sharing these experiences, this research enables researchers and developers to grasp a thorough understanding of the practical implications and considerations involved in implementing an orchestration platform in edge environments. Furthermore, the impact of the RODS platform extends beyond edge environments to its potential applications in cloud environments. The platform gains enhanced generalizability and applicability by proposing the integration of container-based technologies like Docker and LXC. This expansion opens up new possibilities for resource isolation, scalability, and network management, providing a robust and flexible orchestration solution for edge computing within cloud environments. In summary, the contributions of this thesis, in addition to the answers to introduced research questions, can be summarized as follows:

- The design and implementation of the RODS platform addresses the limitations of existing orchestration platforms in edge computing scenarios.
- Filling the gaps in knowledge and understanding within the field of edge computing orchestration.
- Providing practical insights, lessons learned, and valuable experiences from the implementation process.

- Proposing adapting container-based technologies for enhanced generalizability and applicability in cloud environments.

The findings and outcomes of this study have the potential to significantly influence and shape the future of edge computing research and practice. The RODS platform opens up avenues for further exploration and development in the field of edge computing orchestration, paving the way for improved resource allocation, efficient network management, and seamless integration of edge and cloud environments. The potential implications and future directions resulting from this research will contribute to advancing the capabilities, performance, and reliability of edge computing systems, benefiting various industries and applications that rely on edge computing technologies.

6.2. Summary

In the rapidly evolving field of edge computing, there is a growing need for specialized orchestration platforms that can effectively address edge environments' unique challenges and requirements. This thesis aims to contribute to the advancement of edge computing orchestration by developing a specialized orchestration platform called RODS (Reliable Orchestration for Distributed Systems). The motivations behind this research stem from the limitations of existing orchestration platforms, particularly Kubernetes, when applied to edge computing scenarios. The need for a specialized platform tailored to the demands of edge environments and the gaps in knowledge in this field prompted the investigation and development of the RODS platform.

This thesis answered the following research questions:

1. 1. What are the limitations of Kubernetes and other alternatives when applied to edge computing scenarios?
    - The limitations of Kubernetes in edge computing scenarios have been thoroughly examined in the related work chapter, highlighting the specific challenges that arise when deploying Kubernetes in edge environments.
2. How can a specialized orchestration platform be developed to address the identified limitations of Kubernetes and provide an alternative solution better suited to the challenges posed by edge environments?
    - The methodology and implementation chapters present the development of the RODS platform, which addresses the limitations identified in Kubernetes and other state-of-the-art platforms. Innovative approaches, algorithms, and architectural designs have been explored to optimize resource allocation, handle intermittent network connectivity, and meet the stringent latency requirements of edge computing.
3. What are the practical challenges in the implementation and development of the proposed orchestration platform?

- The discussion chapter reflects on the practical challenges encountered during the implementation of the RODS platform. Valuable lessons have been learned, particularly in terms of ensuring consistency, avoiding shared memory communication, and handling network-related issues. These insights have practical implications for the future development and deployment of the RODS platform.

The consensus algorithm employed in RODS ensures reliable data replication and consistency across the distributed system. By broadcasting updates and using unique log IDs, the platform prevents duplicate updates and handles potential network loops. Additionally, the leader election process safeguards against leader failures and minimizes the partitioning of devices in the cluster. A brief comparison has been made between RODS and other state-of-the-art orchestration platforms, including Oakestra, Kubernetes, K3s, and MicroK8s. While this thesis does not aim to provide in-depth performance comparisons, low-key insights have been gained from the comparison, highlighting the unique features and advantages of RODS in addressing edge computing challenges. The discussion chapter has provided valuable insights into the practical challenges encountered during the implementation of the RODS platform. Discourses have been learned concerning the matter of avoiding shared memory communication and utilizing Rust. Furthermore, the potential of integrating container-based technologies like Docker and LXC to enhance the generalizability and applicability of RODS in cloud environments has been explored. Considerations regarding resource isolation, scalability, and network-related challenges in this context have been discussed. The overall impact and contribution of this study in the field of edge computing orchestration are significant. The RODS platform effectively addresses the limitations of existing orchestration platforms in the context of edge computing, providing a specialized solution tailored to the unique demands of edge environments. By filling the gaps in knowledge and advancing edge computing research and practice, this thesis offers valuable insights and practical implications for developing and deploying orchestration platforms in real-world edge computing scenarios.

Furthermore, the findings from the research questions and the lessons learned from the implementation and development of RODS highlight the potential for further advancements and future directions in the field of edge computing orchestration. The integration of container-based technologies demonstrates the adaptability and generalizability of RODS in cloud environments while also highlighting the challenges and considerations that need to be addressed. This thesis contributes to the understanding and practical implementation of edge computing orchestration, offering a specialized platform and valuable insights for researchers and practitioners in the field. The RODS platform stands as a significant step forward in addressing the challenges of edge computing. It paves the way for future advancements and innovations in this exciting and evolving domain.

7. REFERENCES

[1] Garcia Lopez P., Montresor A., Epema D., Datta A., Higashino T., Iamnitchi A., Barcellos M., Felber P. & Riviere E. (2015), Edge-centric computing: Vision and challenges.

[2] Cardellini V., Lo Presti F., Nardelli M. & Rossi F. (2020) Self-adaptive container deployment in the fog: A survey. In: Algorithmic Aspects of Cloud Computing: 5th International Symposium, ALGOCLOUD 2019, Munich, Germany, September 10, 2019, Revised Selected Papers 5, Springer, pp. 77–102.

[3] Atieh A.T. (2021) The next generation cloud technologies: a review on distributed cloud, fog and edge computing and their opportunities and challenges. ResearchBerg Review of Science and Technology 1, pp. 1–15.

[4] Yaqoob A., Bi T. & Muntean G.M. (2020) A survey on adaptive 360 video streaming: Solutions, challenges and opportunities. IEEE Communications Surveys & Tutorials 22, pp. 2801–2838.

[5] Shafique K., Khawaja B.A., Sabir F., Qazi S. & Mustaqim M. (2020) Internet of things (iot) for next-generation smart systems: A review of current challenges, future trends and prospects for emerging 5g-iot scenarios. Ieee Access 8, pp. 23022–23040.

[6] Zhang K., Leng S., He Y., Maharjan S. & Zhang Y. (2018) Mobile edge computing and networking for green and low-latency internet of things. IEEE Communications Magazine 56, pp. 39–45.

[7] Simon S. (2000) Brewer's cap theorem. CS341 Distributed Information Systems, University of Basel (HS2012) .

[8] Kubernetes: Production-grade container orchestration. URL: `https://kubernetes.io/`. Accessed 25.5.2023.

[9] Cloud native computing foundation. URL: `https://www.cncf.io`. Accessed 6.6.2023.

[10] Curry D., Gitlab moves operations from microsoft azure to google cloud. URL: `https://www.rtinsights.com/gitlab-azure-google/`. Accessed 14.6.2023.

[11] k0s: Kubernetes for edge/iot. URL: `https://k0sproject.io`. Accessed 6.6.2023.

[12] K3s: A lightweight kubernetes. URL: `https://k3s.io`. Accessed 6.6.2023.

[13] Microk8s. URL: `https://microk8s.io`. Accessed 6.6.2023.

[14] Docker swarm documentation. URL: `https://docs.docker.com/engine/swarm/`. Accessed 6.6.2023.

[15] Apache mesos. URL: `https://mesos.apache.org`. Accessed 6.6.2023.

[16] Nomad: Hashicorp's orchestration tool. URL: https://www.nomadproject.io. Accessed 6.6.2023.

[17] Aws greengrass service documentation. URL: https://aws.amazon.com/greengrass/. Accessed 6.6.2023.

[18] Bartolomeo G., Yosofie M., Bäurle S., Haluszczynski O., Mohan N. & Ott J. (2022) Oakestra white paper: An orchestrator for edge computing. arXiv preprint arXiv:2207.01577 .

[19] Arcaini P., Riccobene E. & Scandurra P. (2015) Modeling and analyzing mape-k feedback loops for self-adaptation. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE, pp. 13–23.

[20] Kuenzer S., Bădoiu V.A., Lefeuvre H., Santhanam S., Jung A., Gain G., Soldani C., Lupu C., Teodorescu Ş., Răducanu C. et al. (2021) Unikraft: fast, specialized unikernels the easy way. In: Proceedings of the Sixteenth European Conference on Computer Systems, pp. 376–394.

[21] Nanovms. URL: https://nanovms.com. Accessed 6.6.2023.

[22] Kuo H.C., Williams D., Koller R. & Mohan S. (2020) A linux in unikernel clothing. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1–15.

[23] Satyanarayanan M., Lewis G., Morris E., Simanta S., Boleng J. & Ha K. (2013) The role of cloudlets in hostile environments. IEEE Pervasive Computing 12, pp. 40–49.

[24] Zhang L., Yang W., Hao B., Yang Z. & Zhao Q. (2023) Edge computing resource allocation method for mining 5g communication system. IEEE Access .

[25] Cao K., Liu Y., Meng G. & Sun Q. (2020) An overview on edge computing research. IEEE access 8, pp. 85714–85728.

[26] Rahimi M.R., Ren J., Liu C.H., Vasilakos A.V. & Venkatasubramanian N. (2014) Mobile cloud computing: A survey, state of art and future directions. Mobile Networks and Applications 19, pp. 133–143.

[27] Pan J. & McElhannon J. (2017) Future edge cloud and edge computing for internet of things applications. IEEE Internet of Things Journal 5, pp. 439–449.

[28] Wu W., Zhang Q. & Wang H.J. (2019) Edge computing security protection from the perspective of classified protection of cybersecurity. In: 2019 6th International Conference on Information Science and Control Engineering (ICISCE), IEEE, pp. 278–281.

[29] Chen C., Gu H., Lian S., Zhao Y. & Xiao B. (2022) Investigation of edge computing in computer vision-based construction resource detection. Buildings 12, p. 2167.

[30] García-Valls M., Cucinotta T. & Lu C. (2014) Challenges in real-time virtualization and predictable cloud computing. Journal of Systems Architecture 60, pp. 726–740.

[31] Kokkonen H., Lovén L., Motlagh N.H., Partala J., González-Gil A., Sola E., Angulo I., Liyanage M., Leppänen T., Nguyen T. et al. (2022) Autonomy and intelligence in the computing continuum: Challenges, enablers, and future directions for orchestration. arXiv preprint arXiv:2205.01423 .

[32] Salah T., Zemerly M.J., Yeun C.Y., Al-Qutayri M. & Al-Hammadi Y. (2016) The evolution of distributed systems towards microservices architecture. In: 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST), IEEE, pp. 318–325.

[33] De Vries W.A. & Fleck R.A. (1997) Client/server infrastructure: a case study in planning and conversion. Industrial Management & Data Systems 97, pp. 222–232.

[34] Kulesza K., Kotulski Z. & Kulesza K. (2006) On mobile agents resistance to traffic analysis. Electronic Notes in Theoretical Computer Science 142, pp. 181–193.

[35] Kulesza K. & Kotulski Z. (2003) Decision systems in distributed environments: Mobile agents and their role in modern e-commerce. Information in 21st Century Society, University of Warmia and Mazury Edition, Olsztyn , pp. 271–282.

[36] Newman S. (2021) Building microservices. " O'Reilly Media, Inc.".

[37] Vianden M., Lichter H. & Steffens A. (2014) Experience on a microservice-based reference architecture for measurement systems. In: 2014 21st Asia-Pacific Software Engineering Conference, vol. 1, IEEE, vol. 1, pp. 183–190.

[38] Taleb T., Samdanis K., Mada B., Flinck H., Dutta S. & Sabella D. (2017) On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. IEEE Communications Surveys & Tutorials 19, pp. 1657–1681.

[39] De Brito M.S., Hoque S., Magedanz T., Steinke R., Willner A., Nehls D., Keils O. & Schreiner F. (2017) A service orchestration architecture for fog-enabled infrastructures. In: 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC), IEEE, pp. 127–132.

[40] Zhang C. (2020) Design and application of fog computing and internet of things service platform for smart city. Future Generation Computer Systems 112, pp. 630–640.

[41] Svorobej S., Bendechache M., Griesinger F. & Domaschka J. (2020) Orchestration from the cloud to the edge. The Cloud-to-Thing Continuum: Opportunities and Challenges in Cloud, Fog and Edge Computing , pp. 61–77.

[42] Barika M., Garg S., Zomaya A.Y., Wang L., Moorsel A.V. & Ranjan R. (2019) Orchestrating big data analysis workflows in the cloud: research challenges, survey, and future directions. ACM Computing Surveys (CSUR) 52, pp. 1–41.

[43] Jiang Y., Huang Z. & Tsang D.H. (2017) Challenges and solutions in fog computing orchestration. IEEE Network 32, pp. 122–129.

[44] Bartolomeo G., Bäurle S., Mohan N. & Ott J. (2022) Oakestra: An orchestration framework for edge computing. In: Proceedings of the SIGCOMM '22 Poster and Demo Sessions, SIGCOMM '22, Association for Computing Machinery, New York, NY, USA, p. 34–36. URL: https://doi.org/10.1145/3546037.3546056.

[45] Jimenez L.L. & Schelen O. (2020) Hydra: Decentralized location-aware orchestration of containerized applications. IEEE Transactions on Cloud Computing 10, pp. 2664–2678.

[46] Maymounkov P. & Mazieres D. (2002) Kademlia: A peer-to-peer information system based on the xor metric. In: Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7–8, 2002 Revised Papers, Springer, pp. 53–65.

[47] Cozzolino V., Ott J., Ding A.Y. & Mortier R. (2020) Ecco: Edge-cloud chaining and orchestration framework for road context assessment. In: 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI), IEEE, pp. 223–230.

[48] Mehar S., Senouci S.M., Kies A. & Zoulikha M.M. (2015) An optimized roadside units (rsu) placement for delay-sensitive applications in vehicular networks. In: 2015 12th Annual IEEE consumer communications and networking conference (CCNC), IEEE, pp. 121–127.

[49] Vo N.S., Duong T.Q., Tuan H.D. & Kortun A. (2017) Optimal video streaming in dense 5g networks with d2d communications. IEEE Access 6, pp. 209–223.

[50] Castellano G., Esposito F. & Risso F. (2019) A service-defined approach for orchestration of heterogeneous applications in cloud/edge platforms. IEEE Transactions on Network and Service Management 16, pp. 1404–1418.

[51] Castellano G., Esposito F. & Risso F. (2019) A distributed orchestration algorithm for edge computing resources with guarantees. In: IEEE INFOCOM 2019-IEEE Conference on Computer Communications, IEEE, pp. 2548–2556.

[52] Jung R. (2020) Understanding and evolving the rust programming language. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647 .

[53] Pereira R., Couto M., Ribeiro F., Rua R., Cunha J., Fernandes J.P. & Saraiva J. (2017) Energy efficiency across programming languages: how do energy, time, and memory relate? In: Proceedings of the 10th ACM SIGPLAN international conference on software language engineering, pp. 256–267.

[54] Raju P., Kadekodi R., Chidambaram V. & Abraham I. (2017) Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 497–514.

[55] Sears R. & Ramakrishnan R. (2012) blsm: a general purpose log structured merge tree. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 217–228.

[56] Nvidia jetson nano developer kit. URL: `https://developer.nvidia.com/embedded/jetson-nano-developer-kit`. Accessed 25.5.2023.

[57] Raspberry 4 specifications. URL: `https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/`. Accessed 25.5.2023.

[58] van der Kouwe E., Heiser G., Andriesse D., Bos H. & Giuffrida C. (2019) Sok: Benchmarking flaws in systems security. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P), IEEE, pp. 310–325.

[59] Go language documentation: Concurrency. URL: `https://go.dev/doc/effective_go#concurrency`. Accessed 25.5.2023.

[60] Beutel D.J., Topal T., Mathur A., Qiu X., Fernandez-Marques J., Gao Y., Sani L., Li K.H., Parcollet T., de Gusmão P.P.B. et al. (2020) Flower: A friendly federated learning research framework. arXiv preprint arXiv:2007.14390 .

[61] Docker. URL: `https://www.docker.com`. Accessed 7.6.2023.

[62] Lxc: Linux containers. URL: `https://linuxcontainers.org`. Accessed 7.6.2023.

[63] Wu S., Tao Z., Fan H., Huang Z., Zhang X., Jin H., Yu C. & Cao C. (2022) Container lifecycle-aware scheduling for serverless computing. Software: Practice and Experience 52, pp. 337–352.

[64] Bui T. (2015) Analysis of docker security. arXiv preprint arXiv:1501.02967 .

8. APPENDICES

Appendix 1 Implemented RAFT consensus algorithm in Rust for RODS

Consensus in RODS

Algorithm 6. RAFT Consensus

```
use std::{collections::HashMap, net::ToSocketAddrs};

use rand::Rng;
use tokio::{select, sync::mpsc};

use crate::{
    control::db_interface::{get_all_candidates, set_this_node_leader},
    database::db::DB,
    proto::{Consensus, CtlMsg, Header},
};

pub(crate) async fn election(
    ctl_tx: mpsc::Sender<CtlMsg>,
    consensus_tx: mpsc::Sender<Consensus>,
    mut consensus_rx: mpsc::Receiver<Consensus>,
    mut db: DB,
) {
    let mut my_id = 0;
    let mut get_all_nodes = false;
    let mut broadcast_election = false;
    let mut election_timeout = false;
    let mut my_vote = 0;
    let mut votes = 0;
    let mut voters = 0;
    let mut yes = 0;
    let mut count_winner = 0;
    let mut is_ongoing = false;
    // id: voters, number of yes
    let mut elections: HashMap<u32, (u32, u32)> = HashMap::new();
    loop {
        select! {
            msg = consensus_rx.recv() => {
                if let Some(msg) = msg {
                    match msg {
                        Consensus::ElectionStart(id) => {
                            if is_ongoing { continue }
                            my_id = id;
                            votes = 0;
                            my_vote = 0;
                            voters = 0;
                            yes = 0;
                            count_winner = 0;
                            let candidates = get_all_candidates(&mut db);
                            for n in candidates {
                                ctl_tx.send(CtlMsg::ConnectToNode(

↪   (n.address.to_socket_addrs().unwrap().next().unwrap(),
                                    n.id, n.node_type))
                                ).await.unwrap();
                            }
                            get_all_nodes = true;
                            election_timeout = true;
                            is_ongoing = true;
                            elections = HashMap::new();
                            elections.insert(my_id, (0, 0));

                        },
```

```
                        Consensus::ElectionBallot(candidate_id) => {
                            elections.insert(candidate_id, (0, 0));
                            if my_vote == 0 {

↪   ctl_tx.send(CtlMsg::MsgToNode((candidate_id,
↪   Header::ElectionVoteCast((my_id, true))))).await.unwrap();
                                my_vote = candidate_id;
                            }else {

↪   ctl_tx.send(CtlMsg::MsgToNode((candidate_id,
↪   Header::ElectionVoteCast((my_id, false))))).await.unwrap();
                            }
                        },
                        Consensus::Vote((_nid, yesno)) => {
                            votes += 1;
                            if yesno {
                                yes += 1;
                                elections.get_mut(&my_id).unwrap().1 =
↪   yes;
                            }
                            if votes == voters {
                                // Send election results to others
                                ctl_tx.send(CtlMsg::Broadcast((2,
↪   Header::ElectionResult((my_id, votes, yes))))).await.unwrap();
                            }
                        },
                        Consensus::OwnVote(id) => {
                            votes += 1;
                            if my_vote == 0 {
                                my_vote = id;
                                yes += 1;
                                elections.get_mut(&my_id).unwrap().1 =
↪   yes;
                                count_winner += 1;
                            }
                            if votes == voters {
                                // Send election results to others
                                ctl_tx.send(CtlMsg::Broadcast((2,
↪   Header::ElectionResult((my_id, votes, yes))))).await.unwrap();
                            }
                        },
```

```
                    Consensus::ElectionResult((candidate_id, votes,
    ↪  number_of_yes)) => {
                        elections.get_mut(&candidate_id).unwrap().0 =
    ↪  votes;
                        elections.get_mut(&candidate_id).unwrap().1 =
    ↪  number_of_yes;
                        if elections.iter().map(|(_e, (v, _ny))|
    ↪  v).all(|&v| v > 0) {
                            let mut max_ny = 0;
                            let mut max_v = 0;
                            let mut max_id = 0;
                            let mut tie_flag = false;
                            for (id, (v, ny)) in &elections {


                                if ny > &max_ny {
                                    max_ny = ny.to_owned();
                                    max_v = v.to_owned();
                                    max_id = id.to_owned();
                                    tie_flag = false;
                                }
                                else if ny == &max_ny {
                                    if v > &max_v {
                                        max_id = id.to_owned();
                                        max_v = v.to_owned();
                                        tie_flag = false;
                                    }
                                    else if v == &max_v {
                                        tie_flag = true;

                                    }

                                }
                            }
                            if tie_flag {
                                ctl_tx.send(CtlMsg::Broadcast((2,
    ↪  Header::ElectionTie))).await.unwrap();
                            }
                            else {
                                ctl_tx.send(CtlMsg::Broadcast((2,
    ↪  Header::ElectionWinner(max_id)))).await.unwrap();
                            }
                        }
                    },
                    Consensus::ElectionWinner(winner_id) => {
                        if winner_id == my_id {
                            count_winner += 1;
                            if votes == count_winner {
                                // anounce the leader
                                ctl_tx.send(CtlMsg::Broadcast((3,
    ↪  Header::AnounceLeader(my_id)))).await.unwrap();
                                election_timeout = false;
                                db.set_leader(true);
                                set_this_node_leader(&mut db,
    ↪  my_id).await;
                            }
                        }
                    },
```

```
                        Consensus::ElectionTie => {
                            // re election
                            log::debug!("Re election because of tie! resp!")
                        },
                        Consensus::CandidatesLen(len) => {
                            log::debug!("candidate len: {:?}", len);
                            voters = len;
                            elections.get_mut(&my_id).unwrap().0 = voters;

                        },
                        Consensus::LeaderAnnounced(_leader_id) => {
                            election_timeout = false;
                            is_ongoing = false;
                        }
                    }
                    }
                    else {
                        break;
                    }
                }
                _ = tokio::time::sleep(std::time::Duration::from_millis(100)),
↪   if get_all_nodes => {
                    let sleep_duration = {
                        let mut rng = rand::thread_rng();
                        rng.gen_range(10..40)
                    };

↪   tokio::time::sleep(std::time::Duration::from_millis(sleep_duration *
↪   100)).await;

↪   ctl_tx.send(CtlMsg::GetAllConnectedCandidates).await.unwrap();
                    get_all_nodes = false;
                    broadcast_election = true;

↪   consensus_tx.send(Consensus::OwnVote(my_id)).await.unwrap();

                }

                _ = tokio::time::sleep(std::time::Duration::from_millis(1)),
↪   if broadcast_election => {
                    broadcast_election = false;
                    ctl_tx.send(CtlMsg::Broadcast((2,
↪   Header::ElectionBallot(my_id)))).await.unwrap();

                }

                _ =
↪   tokio::time::sleep(std::time::Duration::from_millis(10000)), if
↪   election_timeout => {
                    // single node
                    if elections.get_mut(&my_id).unwrap().0 == 1 {
                        election_timeout = false;
                        db.set_leader(true);
                        set_this_node_leader(&mut db, my_id).await;

↪   ctl_tx.send(CtlMsg::SetThisLeader(my_id)).await.unwrap();
                    }
                    else {
                        // re-election
                        log::debug!("Re election because of tie!");
                        log::debug!("elecetions {:?}", elections);
                    }
                }


            }
        }
}
```