REPUBLIC OF TURKEY

AKDENIZ UNIVERSITY

PUSOS: AN OPERATING SYSTEM APPROACH IN SUPERVISOR MODE FOR FOG COMPUTING

Muhammed Numan INCE

INSTITUTE OF NATURAL AND APPLIED SCIENCES

DEPARTMENT OF COMPUTER ENGINEERING

DOCTORAL THESIS

MAY 2024

ANTALYA

REPUBLIC OF TURKEY

AKDENIZ UNIVERSITY

PUSOS: AN OPERATING SYSTEM APPROACH IN SUPERVISOR MODE FOR FOG COMPUTING

Muhammed Numan INCE

INSTITUTE OF NATURAL AND APPLIED SCIENCES

DEPARTMENT OF COMPUTER ENGINEERING

DOCTORAL THESIS

MAY 2024

ANTALYA

REPUBLIC OF TURKEY

AKDENIZ UNIVERSITY

INSTITUTE OF NATURAL AND APPLIED SCIENCES

PUSOS: AN OPERATING SYSTEM APPROACH IN SUPERVISOR MODE FOR FOG COMPUTING

Muhammed Numan INCE

DEPARTMENT OF COMPUTER ENGINEERING

DOCTORAL THESIS

This thesis unanimously accepted by the jury on 28/05/2024.

Prof. Dr. Melih GÜNAY (Supervisor)

Assoc. Prof. Dr. Taner DANIŞMAN

Dr. Reza BARZEGARAN

Asst. Prof. Dr. Gökhan SEÇİNTİ

Asst. Prof. Dr. Özge ÖZTİMUR KARADAĞ

ÖZET

PUSOS: SİS BİLİŞİM İÇİN SÜPERVİZÖR MODUNDA BİR İŞLETİM SİSTEMİ YAKLAŞIMI

Muhammed Numan İNCE

Doktora Tezi, Bilgisayar Mühendisiliği Anabilim Dalı
Danışman: Prof. Dr. Melih GÜNAY

Mayıs 2024; 77 sayfa

Endüstrinin dijitalleşme devriminin ardından gelişen Nesnelerin İnterneti konsepti; verinin toplanması, depolanması ve işlenme süreçlerini doğrudan etkilemiştir. Şimdiye kadar akıllı küçük cihazların ürettiği verilerin ele alınması bulut bilişim, büyük veri ve Yüksek Performanslı Hesaplama kapsamında ele alınmaktaydı. Fakat son on yılda artan IoT cihaz sayısı ve hızlı yanıt alma beklentileri, yeni hesaplama yaklaşımlarının doğmasına neden olmuştur. Bu doğrultuda oluşan yeni trend ise sis bilişim olarak karşımıza çıkmaktadır. Sis bilişim, hizmet kalitesini artırmak için bilgi işlem, depolama ve ağ oluşturmayı, son kullanıcılara ve cihazlara yaklaştırarak gecikmeyi azaltmayı hedeflemektedir. Her bilişim sistemi gibi sis bilişim de ihitiyaçları karşılamak için bir hesaplama ortamını kullanmaktadır. Şimdiye kadar yapılan sis bilişim çalışmalarında, yaygın olarak bulut sunucu sistemlerinin domine ettiği Linux işletim sistemi sis bilişim altyapı ve hizmet uygulamalarının sunulmasında kullanılmıştır. Bu durum her ne kadar mevcut teknoloji ortamında eldeki hızlı bir çözüm gibi görülse de, sistem seviyesindeki geliştirme süreçleri ve yaygınlaştırma zorlukları bakımından uzun vadede sürdürülebilir olamayacaktır.

Bunun için, bu doktora çalışmamızda, Sis bilişimin hedeflediği gecikme zamanı azaltmasını ve ara katman hedeflerini sistemsel düzeyde ele alarak farklı bir yaklaşımla yeni bir sis bilişim işletim sistemi konsepti öneriyoruz. Bu işletim sistemi, sis bilişimin asgari niteliklerini sağlamak koşuluyla, sadece çekirdek alanında supervisor modda ve donanımsal düzeyde açık kaynak komut seti olan RISC-V mimarisi üzerinde çalışmaktadır. PusOS adını verdiğimiz bu sistem, kullanıcı alanına sahip olmadan sadece kernel alanında, veri toplama, depolama ve işleme gibi sis bilişim gereklerini yerine getirirken RISC-V çalışma

modları yardımıyla da sağlam bir sanallaştırma izolasyonu sunmaktadır. PusOS, sis bilişim uygulama katmanını, özelleştirilebilir bir nano çekirdek kullanarak gerçeklemektedir. Nano çekirdek yapısı içine modüler şekilde tanımlanan sadece etki alanı içeren uygulama katmanı sayesinde daha kolay dağıtabilir ve kompakt bir sistem elde edilir. Yaptığımız deneyler sonucu, veri toplama ve depolama yönlerinde Linux bir sistemden nasıl hızlı olduğunu, sanallaştırma testleriyle de hata tolerans ve göç etme yeteneklerinin başarısını gösteriyoruz. Bunun yanısıra PusOS açık kaynak ve özgür lisans ile geliştirilen kod tabanı sayesinde farklı sis bilişim projelerinde test yatağı olarak ta kullanılabilecektir.

ANAHTAR KELİMELER: İşletim Sistemi, Nesnelerin İnterneti, RISC-V, Sanallaştırma, Sis Bilişim

JÜRİ:

Prof. Dr. Melih GÜNAY

Doç. Dr. Taner DANIŞMAN

Dr. Öğr. Üyesi Gökhan SEÇİNTİ

Dr. Öğr. Üyesi Özge ÖZTİMUR KARADAĞ

Dr. Reza BARZEGARAN

ABSTRACT

PUSOS: AN OPERATING SYSTEM APPROACH IN SUPERVISOR MODE FOR FOG COMPUTING

Muhammed Numan İNCE

Doctoral Thesis in Computer Engineering
Supervisor: Prof. Dr. Melih GÜNAY
May 2024; 77 pages

The rise of connected devices in the Internet of Things era has accelerated industry digitization, prompting significant innovations in data collection, storage, and processing strategies. Last decade, we witnessed the handling of data produced by smart small devices has been handled within the scope of cloud computing, big data and high performance computing (HPC). However, the surging number of IoT devices and real-time needs necessitated new computing paradigms. And the new trend that has become more famous in this direction is Fog computing. Fog computing aims to reduce latency by bringing computing, storage, and networking closer to devices and end users to improve quality of service. Like every computing system, Fog computing also need to use a computing environment to meet requirements. The Linux operating system, which is widely used in cloud server systems, is used to present Fog computing infrastructure and service applications to date. Although this trend may seem like a practical solution in the current technology stack, it will not be sustainable in the long term in terms of system-level development processes and distribution deployment aspects. The fact that Fog computing has a middleware architecture necessitates the use of fewer resources and a compact operating system structure.

Therefore, this doctoral research proposes PusOS, a novel Fog computing operating system designed to minimize latency and meet middleware requirements. Built on the open-source RISC-V architecture, PusOS leverages hardware-level virtualization by running solely in supervisor mode with a nanokernel handling data collection, storage, and processing – all within the kernel space, eliminating the need for a user space. This streamlined design, featuring modular domain-specific application stacks within the nano-

kernel, translates to a compact and easily deployable system. Experiments show PusOS significantly outperforms Linux in data collection and storage tasks, while virtualization tests confirm its effectiveness in fault tolerance and application migration. Furthermore, PusOS's open-source nature makes it a valuable test platform for further Fog computing research and development.

KEYWORDS: Fog Computing, Internet of Things, Operating System, RISC-V, Virtualization

COMMITTEE:

Prof. Dr. Melih GÜNAY

Assoc. Prof. Dr. Taner DANIŞMAN

Asst. Prof. Dr. Gökhan SEÇİNTİ

Asst. Prof. Dr. Özge ÖZTİMUR KARADAĞ

Dr. Reza BARZEGARAN

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor Prof. Dr. Melih GÜNAY, who supported me at every stage in the preparation of this study and provided different perspectives with his criticisms.

Also I would like to thank all my teachers who trained me to serve for our country.

Finally special thanks to my family for their big love and support.

LIST OF CONTENTS

ÖZET i
ABSTRACT iii
ACKNOWLEDGEMENTS v
TEXT OF OATH viii
SYMBOLS AND ABBREVIATIONS ix
LIST OF FIGURES x
LIST OF TABLES xi
1. INTRODUCTION 1
1.1. Why Not Linux as Main Operating System 2
1.2. Why RISC-V as Instruction Set Architecture 4
2. LITERATURE REVIEW 6
2.1. Operating Systems 6
2.1.1. Kernel Types 7
2.1.2. User Space vs Kernel Space 13
2.2. Fog Computing 17
2.2.1. Fog Computing and Edge Computing 18
2.2.2. Fog Computing and Cloud Computing 21
2.2.3. Reference Architectures 21
2.3. RISC-V 24
2.3.1. RISC-V and Fog Computing 26
2.4. Virtualization and Fog Computing 28
2.5. Related Works 31
3. MATERIAL AND METHOD 34
3.1. Kernel Structure 34
3.1.1. LittleKernel 35
3.2. Implementation Platform 37
3.2.1. QEMU 37
3.3. Network Stack 38
3.3.1. Data Structure 38
3.3.2. Sensor Communication 42
3.4. Storage 44
3.5. Computation Stack 47

3.6. Virtualization 49
3.6.1. QMP 50
3.6.2. Fault Tolerance Method 51
3.6.3. Migration Method 52
3.7. Management and Monitoring 53
4. RESULTS AND DISCUSSION 56
4.1. Pus Controller Tool 56
4.2. Experiments 59
4.2.1. Data Collection 60
4.2.2. Data Storage 62
4.2.3. Fault Tolerance 64
4.2.4. Application Migration 66
5. CONCLUSION 69
6. REFERENCES 71
CURRICULUM VITAE

TEXT OF OATH

I declare that this study “PusOS: An Operating System Approach In Supervisor Mode For Fog Computing ”, which I present as doctoral thesis, is in accordance with the academic rules and ethical conduct. I also declare that I cited and referenced all material and results that are not original to this work.

28 / 05 / 2024

Muhammed Numan INCE

SYMBOLS VE ABBREVIATIONS

Abbreviations:

ARM	: Advanced RISC Machines
BIOS	: Basic Input/Output System
CoAP	: Constrained Application Protocol
eBPF	: Extended Berkeley Packet Filter
HAL	: Hardware Abstraction Layer
HPC	: High Performance Computing
IoT	: Internet of Things
KVM	: Kernel-based Virtual Machine
LK	: LittleKernel
LRU	: Least Recently Used
MAPT	: Message Average Processing Time
MQTT	: Message Queuing Telemetry Transport
OS	: Operating System
OSI	: Open Systems Interconnection
RISC-V	: Reduced Instruction Set Processor five
RTOS	: Real-Time Operating System
QEMU	: Quick Emulator
QMP	: QEMU Machine Protocol
QGA	: QEMU Guest Agent
QoS	: Quality of Service
SLA	: Service Level Agreement
VM	: Virtual Machine

LIST OF FIGURES

Figure 2.1 Monolithic Kernel vs. Microkernel Structure (Wikipedia) 8
Figure 2.2 Hybrid Kernel Structure (Wikipedia) 9
Figure 2.3 Nanokernel Arhitecture Structure (Bagchi 2008) 11
Figure 2.4 Exokernel Structure (Wikipedia) 12
Figure 2.5 Kernel vs. User Space Relation 14
Figure 2.6 The Fog as a Layer Between Smart Devices/Sensor and the Cloud 18
Figure 2.7 Hierarchical View - Fog and Edge Computing (Ashraf et al. 2022) 20
Figure 2.8 OpenFog Software Architecture View (OpenFog consortium 2018) 23
Figure 2.9 Layers in RISC-V Architecture 27
Figure 2.10 Virtualization Technologies on Fog Computing Architecture . . . 30
Figure 2.11 BasatNet High Level Design (Ince et al. 2022) 32
Figure 3.12 Block Diagram of the PusOS Fog Computing Environment . . . 34
Figure 3.13 PusOS Overlay on top of LK 36
Figure 3.14 HTTP and CoAP in OSI Layer (Al-Haidari and Alqahtani 2020) 43
Figure 3.15 QMP Protocol - Monitor Objects (Capitulino 2010) 51
Figure 3.16 Cloud-to-Things Continuum with PusOS Internals 55
Figure 4.17 PusCtl Web User Interface - Cluster 57
Figure 4.18 PusCtl Web User Interface - New Node 58
Figure 4.19 PusCtl Web User Interface - Node Migration 59
Figure 4.20 PusOS Crash Recovery Demonstration 65
Figure 4.21 PusOS Fault Tolerance Mechanism 66
Figure 4.22 PusOS Application Migration Mechanism 67

LIST OF TABLES

Table 2.1 Comparison of kernel types 13
Table 2.2 Comparing with fog computing and cloud computing 22
Table 3.3 Remote procedure commands 41
Table 3.4 Comparing filesystems for PusOS storage choice 45
Table 4.5 Comparison of test systems 60
Table 4.6 10ms interval sensor communication processing speed 61
Table 4.7 50ms interval sensor communication processing speed 62
Table 4.8 100ms interval sensor communication processing speed 62
Table 4.9 10ms interval I/O operation processing speed 63
Table 4.10 50ms interval I/O operation processing speed 64
Table 4.11 100ms interval I/O operation processing speed 64
Table 4.12 Fault recovery test 66
Table 4.13 Application migration test 68

1. INTRODUCTION

The proliferation of data-generating devices, fueled by technological advancements, has propelled cloud computing and big data to the forefront, impacting not only the "Internet of Things" but also diverse fields. At the same time, cloud systems are revolutionizing data storage and processing with their on-demand server availability, simplified configuration, and reduced costs. This shift unlocks agility and efficiency, allowing organizations to seamlessly handle the workloads and data volumes. While a cloud-connected continuation seems reasonable when there is a limited number of IoT devices, the growing number of devices connected to the Internet has prompted us to question direct access to the cloud. First of all, the cost of a cloud-only solution is too high to run a large-scale IoT system with more than thousands of devices distributed geographically, especially when some devices require fast response times under 10 ms (Yang 2017). Secondly, transmitting all kinds of data to the cloud takes up bandwidth space unnecessarily. Therefore, there is a need for a promising solution that brings the resources and services in closer to the end-users (Haouari et al. 2018).

To address these needs, Fog Computing introduced by Cisco in 2012 promises a solution (Bonomi et al. 2012) for geographically dispersed, low-latency, and privacy-sensitive data processing. It acts as an intermediary tier, bringing cloud-like services (network, storage, computation, management, and virtualization) closer to the edge of the network, where data is generated by connected devices. This distributed approach fosters low-latency, decentralized applications by processing data closer to its source. This focus on proximity to data and devices aligns with the growing demand for improved Quality of Service (QoS) as outlined in the Fog computing paradigm (Yousefpour et al. 2019).

There has been various Fog computing platforms that are implemented since the introduction of the concept. The common approach is the use the Linux operating system as the software layer (Li et al. 2018). These implementations may be grouped as follows:

- Serverless deployment within partial-cloud (Cheng et al. 2019),
- Proprietary systems (Forti et al. 2019),
- Fog computing simulation platforms (Coutinho et al. 2018)

According to aforementioned implementations; serverless semi-cloud provides some services of Fog computing over the cloud. Therefore, they are internet dependent. Proprietary platforms are mostly commercial and are not open source hence limited in flexibility. On the other hand, platforms developed for simulations are primarily designed for academic research, therefore not quite ready for production deployment. Because Fog computing requirements are implemented at user space, in application layer on Linux systems, they often include unnecessary modules and has extra overhead due to additional inter-process communication. Furthermore, due to complex Linux codebase, it is not easy to strip it down to micro kernel which is desired at the Fog Computing System level.

Because of the shortfalls and disadvantages of the current paradigm, we propose to design and address the Fog computing requirements at the OS level. In this study, only the principal functions such as communication, storage, computation, and virtualization of the Fog computing notion, outlined by the OpenFog consortium (2018), is considered. For performance reasons, user space modules and extra kernel modules that do not relate to fog requirements are not included. Because, the proposed Fog computing system runs in the kernel, it uses a single address space for system tasks.

In the following subsections, i) Why a Linux distribution is not considered as the Fog OS and ii) Why we choose RISC-V as the platform architecture is discussed.

1.1. Why Not Linux as Main Operating System

Fog computing lies between the Cloud and the IoT which bridges the gap. It not only shuttles data but also handles local computations, reducing the burden on the Cloud. Because Fog computing operates in diverse environments, from resource-constrained IoT devices to powerful cloud servers, it necessitates a lightweight and flexible operating system that can effortlessly run specialized applications. IoT devices at the Edge are often low cost and limited in computation power. They typically read sensors and control devices. Nonetheless once analysis and/or decision making are expected, the current paradigm is to refer to public/private Cloud where higher computation performance is achieved, though expensive. Fog computing, situated in the intermediate layer, requires cost-effectiveness alongside respectable performance to manage decision-making at fog nodes with restricted data. Consequently, there is a preference for an operating systems, unlike Linux, that

is lightweight in hardware requirements yet proficient in computation, unlike those found on Edge devices (Alwakeel 2021).

Reasons for not selecting Linux as the operating system for Fog Computing may be outlined as follows:

- Incorporating Fog computing services into the Linux kernel space presents challenges due to Linux's monolithic structure. A more viable option could be the microkernel architecture, which allows for easier integration (Fetzer 2016).
- The system should be resilient to security vulnerabilities and capable of swift repairs. Despite Linux's robust security features, there remains potential for security vulnerabilities and risks within Linux systems. Detecting and analyzing code in such a large codebase is a challenging task (Bayer 2022).
- For practical system deployment, the operating system must seamlessly run on the target hardware. Deploying a Linux system at the kernel level in this manner is not straightforward; it requires additional user-space libraries.
- The system may require rapid compilation and preparation when needed. However, compiling and assembling a Linux system is not feasible in such situations.
- The system should ensure optimized resource utilization. However, the Linux system inherently consumes a certain amount of resources, and each new process adds to this consumption. This poses a challenge for programmers operating within constrained resource environments.

Operating at the kernel level when high performance and small footprint is needed is not be possible within the current implementations that are at Linux user space. Hence, this research delves into exploring the feasibility of integrating the Fog computing services outlined by Bonomi et al. (2012) and formalized by the OpenFog consortium (2018) directly into the operating system's kernel. The proposed computing system, now augmented with Fog services, has been streamlined by removing extraneous user-space components and keeping only the essential kernel modules. As a result, the proposed computing system operates within a singular address space.

1.2. Why RISC-V as Instruction Set Architecture

Fog computing spans a diverse range of devices, many powered by ARM processors for their efficiency. However, these devices come in various configurations specific to different ARM models. This heterogeneity creates a challenge for manufacturers to agree on a single, standardized hardware design. ARM's architecture is an instruction set licensed by ARM Semiconductor. As Fog computing expands, potential limitations might emerge for mission-critical applications depending on ARM licensing terms. To address this, the next section will explore the potential of RISC-V as an alternative architecture for Fog computing.

A broad alliance of chipmakers, universities, and programmers (the RISC-V community) is working together to open-source and refine the RISC-V instruction set. This collaborative approach is fueling the rise of RISC-V as a popular choice for processor design. Because it's open-source, RISC-V gives manufacturers the flexibility to create processors for specific needs. This makes it perfect for Fog computing, where custom instruction sets can significantly enhance performance (Rosario et al. 2018). Also RISC-V's transparency enables easy tracking of architectural changes. In the realm of Fog computing and embedded systems, this transparency could disrupt the dominance of major chip manufacturers, empowering smaller companies to create their own specialized devices. The RISC-V architecture provides essential flexibility for both virtualization and bare-metal operation through its machine and supervisor operating modes (Waterman et al. 2014).

Drawing from the preceding explanations, this study is motivated by an open-source ISA architecture and a straightforward, comprehensible operating system. Given that Fog computing serves as an intermediary layer, it necessitates a computing layer that naturally simplifies from the upper echelon (Cloud) to the lower tier (IoT). Consequently, we can outline the primary contributions of this thesis as follows:

- A Fog computing operating system design abstracted from unnecessary functions except fog requirements.
- Kernel space runtime environment that handles operations without any user space, with near-zero syscall cost.
- A kernel structure that proves the supervisor advantages of the RISC-V with a

battle-tested virtualization environment.

- An open source implementation that can easily be used as a testbed for different projects.

The study begins with a comprehensive literature review (Section 2) before delving into the details of the research methodology, including the materials and method (Section 3). Results and Discussion (Section 4), showcases the findings derived from both simulations and measurements, accompanied by an in-depth discussion. Following the conclusion of the study (Section 5), a comprehensive list of cited sources is provided (Section 6).

2. LITERATURE REVIEW

Through this section, literature review about critical design issues for operating systems designed for Fog computing are discussed along with the importance of RISC-V as the instruction set architecture and the necessity of virtualization for Fog computing. In addition, important criticisms will be discussed, supported by relevant researches, as they form a background for our design and implementation choices.

2.1. Operating Systems

An operating system acts as the interface between human and computer hardware. It manages all the resources of the computer, including memory, storage, and processing power. It also provides the platform for running applications and interacting with the user. The kernel, positioned at the core of an operating system, facilitates low-level hardware communication and resource management, essential for overall system functionality. All requests from the user are transmitted to the kernel through intermediary programs, and the kernel interacts with the hardware and returns user requests. In this respect, The heart of an operating system, the kernel, plays a crucial role in performance and efficiency. Choosing the right kernel architecture is essential before designing any operating system, as it acts as the foundation for all other components and directly influences their interaction. Therefore, the choice of kernel for an operating system is important as it directly affects the following issues;

- Performance: Different kernel architectures (like monolithic or microkernel) prioritize different aspects, affecting speed and resource utilization.
- Efficiency: The kernel manages resources like memory and CPU time. Its structure dictates how efficiently these are allocated and used.
- Modularity: Other system components rely on the kernel for essential services like memory management and device interaction. Its design defines their capabilities and limitations.

In line with the above issues, which are also important for Fog computing, the kernel types that stand out in operating systems are discussed in the following subsection.

2.1.1. Kernel Types

Operating systems have relied on kernels since the dawn of centralized computers, and these kernels have constantly evolved to meet the needs of users and ever-changing hardware. The kernel acts as the brain of the operating system, loaded at startup and ever-present in memory. It allocates resources fairly among programs, facilitates smooth communication between software and hardware, and governs everything from memory and storage to devices. Also it ensures programs access shared resources fairly and preventing conflicts. By prioritizing tasks and allocating processor usage, the kernel keeps the system running smoothly. Operating in a secure zone separate from user programs, the kernel safeguards the system's core functions. This separation creates a stable environment where programs can operate freely without crashing the entire system, keeping errors isolated. In order to better understand the functions aforementioned, the core types, each with their own advantages and disadvantages, are explained below;

- **Monolithic kernel:** This is the most common type of kernel, and it consists of a single large program that runs in kernel space. The kernel acts as a high-level abstraction layer for the hardware, providing system calls for core functionalities like process management, concurrency, and memory management as shown in the Figure 2.1. These functionalities can be implemented in separate modules within the kernel space, but due to tight integration and a shared address space, a bug in any module can crash the entire system. Modern monolithic kernels like Linux, FreeBSD, Solaris offer dynamic loading and unloading of modules at runtime, however, this modularity applies only to the binary level and not the core design architecture itself (Stankov and Spasov 2006).

  Monolithic kernels, while popular, suffer from complexity due to their massive codebase (potentially thousands of lines) leading to maintenance challenges and recompiling the entire kernel for updates. This size also hinders portability to resource-constrained embedded systems. Furthermore, the inherent complexity increases the likelihood of system crashes as a single kernel error can bring the entire system down. Microkernel architectures were designed to address these limitations (Miao 2011).

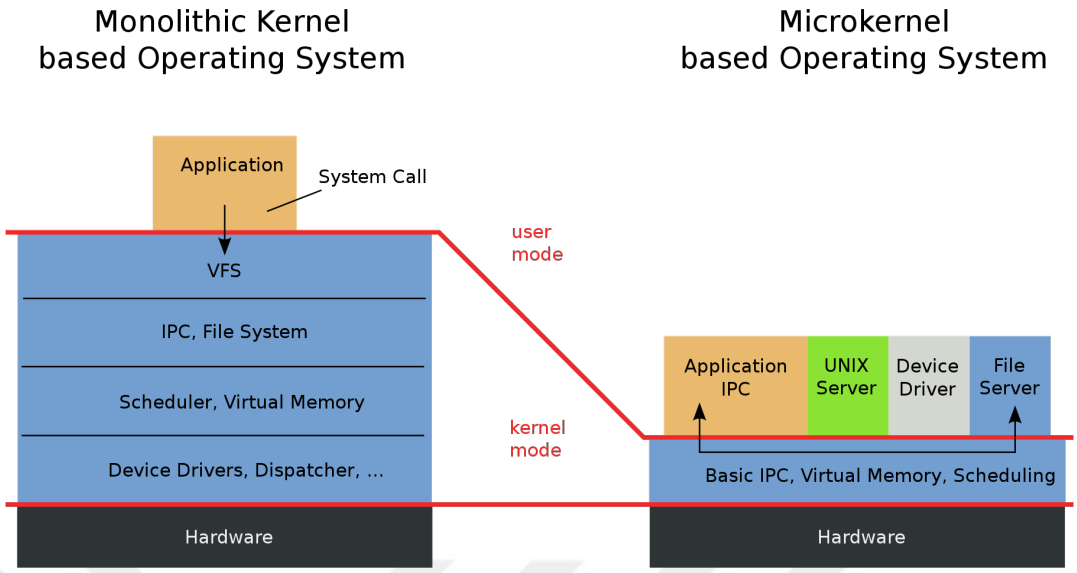

Figure 2.1. Monolithic Kernel vs. Microkernel Structure (Wikipedia)

- **Microkernel:** One approach to modern operating system design, which is becoming popular, is to build the distributed operating system as a set of independent system servers using the primitive, generic services of a microkernel. This type of kernel is much smaller than a monolithic kernel, and it only provides the most basic operating system services, such as memory management and process communication. Other operating system services, such as device drivers and file systems, are implemented as separate user-space programs that communicate with the microkernel through inter-process call (IPC) mechanism as shown in the Figure 2.1.

  Microkernel architectures address the growing complexity of modern systems by leveraging distributed computing within the virtual machine layer. This structured design allows for better system design, improved manageability of complexity, and easier comprehension. By incorporating distribution and subsystem support into the low-level kernel, microkernels introduce modularity, a key aspect missing from traditional monolithic architectures. This modularity, akin to an object-oriented approach in OS design, paves the way for the evolution of such systems in distributed environments prevalent in today's high-end computing architectures (Gien and Grob 1992). Microkernel architectures offer advantages in security and flexibility,

but haven't become the dominant design for mainstream operating systems. While some commercial products like Blackberry's QNX utilize this approach, others like Haiku and MINIX are primarily used in research applications. The reason for this lies in a trade-off, microkernels basically rely on message passing for communication between processes which can introduce performance overhead compared to monolithic kernels. This performance penalty, combined with the increased complexity of development, makes them less suitable for general-purpose use. However, microkernels remain a valuable choice for embedded systems and security-critical applications.

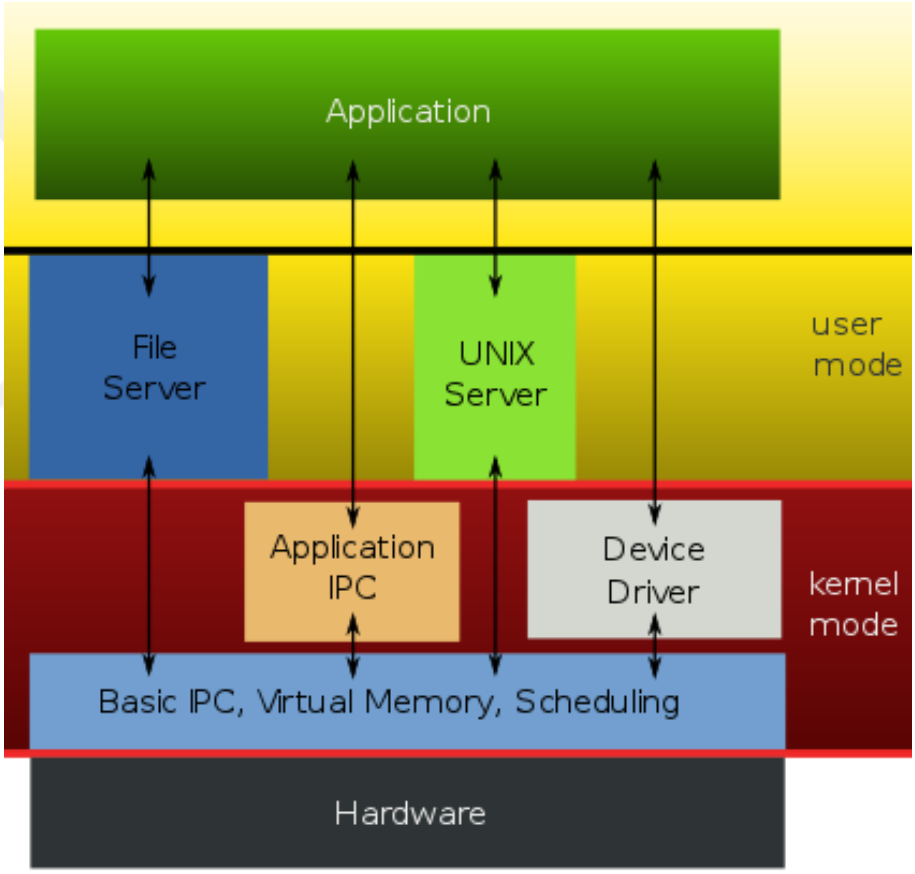

Figure 2.2. Hybrid Kernel Structure (Wikipedia)

- **Hybrid kernel:** This type of kernel combines elements of both monolithic and microkernels. It typically has a small microkernel that provides the basic operating system services, and then a number of loadable kernel modules that can be added

or removed as needed. This allows for a balance between the performance of a monolithic kernel and the flexibility of a microkernel as shown in Figure 2.2. There are examples of hybrid kernels:

  - Windows NT kernel: used in Windows NT, Windows 2000, Windows XP, and later versions of Windows.
  - Mac OS X kernel (XNU): used in macOS and iOS.
  - Mach kernel: used in early versions of macOS and in various research projects.

  It's important to note that the classification of hybrid kernels is somewhat controversial. Some experts argue that they are essentially just monolithic kernels with a different structure, while others believe they offer distinct advantages over both monolithic and microkernels (McBrewster et al. 2009).

- **Nanokernel:** This type of kernel is even smaller than a microkernel, and it only provides the bare minimum of operating system services. Nanokernel design prioritizes modularity and flexibility by separating kernel data from the functionalities that utilize it. The kernel address space is split into a minimal nanokernel core and loadable kernel devices. These devices act as dynamically attachable modules with well-defined interfaces to the kernel. They implement specific policies or algorithms and access external kernel data objects through designated import/export mechanisms which incorporates dynamic reconfiguration capability by decoupling the kernel data objects from the policies implemented in the kernel subsystems (Bagchi 2008).

  Also in projects like XtratuM (Masmano 2005), which aims to achieve temporal and spatial isolation for running multiple operating systems on a single machine, the nanokernel architecture has emerged as a powerful tool. XtratuM leverages the inherent simplicity of the nanokernel, essentially a thin software layer to provide the core functionalities of an operating system. This minimal design focuses on the essential elements: managing memory, scheduling tasks, and handling interrupts from devices. Project also establishes itself as the core system manager by controlling interrupts, offering virtual timers for simplified OS porting (adapting to new

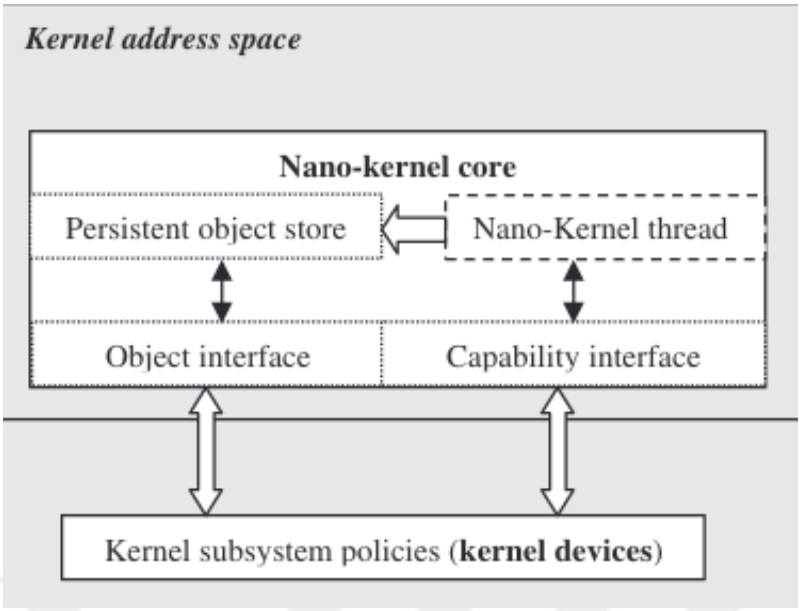

Figure 2.3. Nanokernel Arhitecture Structure (Bagchi 2008)

hardware), and providing memory isolation and management. This control over interrupts ensures stability and security, while virtual timers with high-level APIs ease development for guest OSes. The nanokernel further allocates dedicated memory regions for each OS, preventing conflicts and allowing for dynamic memory adjustments based on individual needs.

In essence, the nanokernel's interrupt control, built-in virtual timers, and advanced memory management features provide a solid foundation for building lightweight and efficient operating systems. These capabilities allow it to effectively utilize the underlying hardware while maintaining isolation and security when guest operating systems are used on it through virtualization.

- **Exokernel:** This type of kernel is a very different approach to operating system design. It does not provide any traditional operating system services, such as memory management or process scheduling. Instead, it provides a set of low-level primitives that applications can use to directly access hardware resources. Figure 2.4 shows the exokernel provides a thin layer of abstraction between applications and the hardware. Applications can make requests to the exokernel for resources, such as memory and network access. The exokernel then grants or denies these requests

based on the security policies in place.

In exokernels, resource access control happens at runtime, eliminating privilege checks during execution for speed. This trust-based approach assumes applications manage resources responsibly. However, granting untrusted libraries high-level control like scheduling can be insecure. While initial verification by the exokernel exists, malicious libraries could exploit this trust to monopolize resources, harming other processes. This highlights the trade-off between speed and security in exokernel design (Restivo-2020).

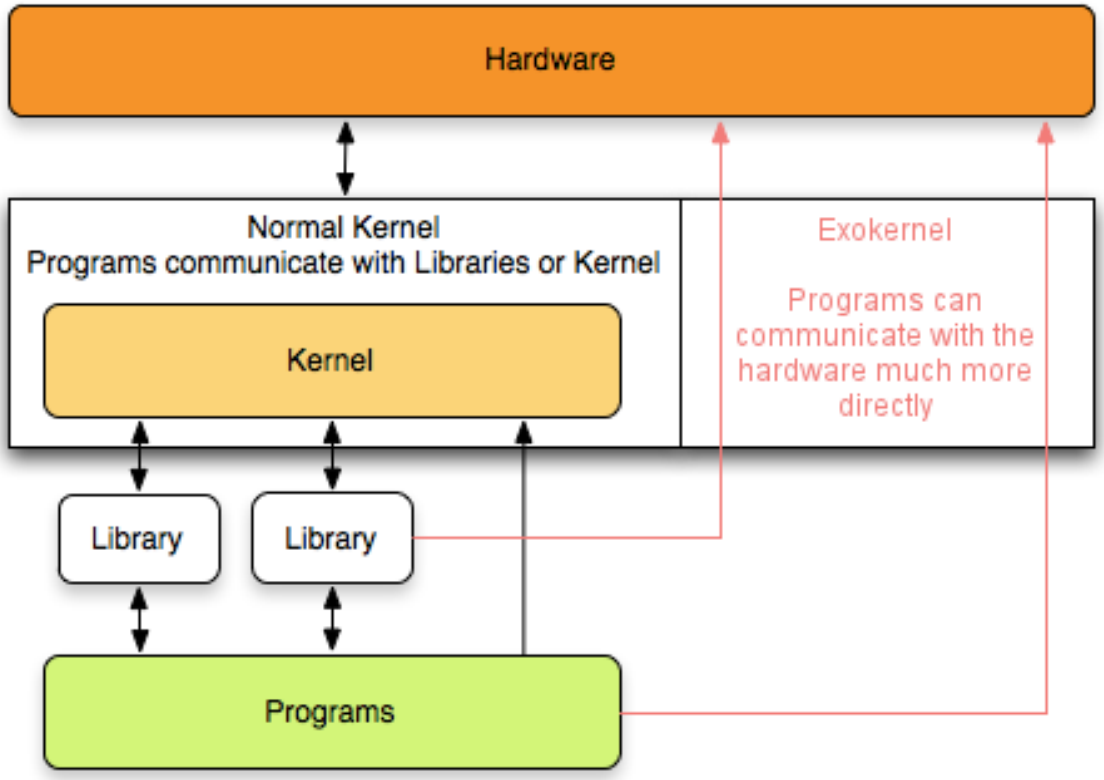

Figure 2.4. Exokernel Structure (Wikipedia)

We can summarize the following conclusions in line with the comparison made in the Table 2.1; monolithic kernels are the most common type of kernel because they offer good performance and are relatively easy to develop. However, they can be less secure and flexible than other types of kernels. Microkernels are more secure and flexible than monolithic kernels, but they can also be slower. They are often used in embedded sys-

Table 2.1. Comparison of kernel types

Type	Size	Performance	Security	Flexibility	Development
Monolithic	Large	High	Medium	Low	Easy
Microkernel	Small	Medium	High	High	Difficult
Hybrid	Medium	High	High	Medium	Medium
Nanokernel	Very small	Very high	Very high	Low	Very difficult
Exokernel	Very small	Very high	Very high	Very high	Very difficult

tems and other applications where security is a major concern. Hybrid kernels offer a balance between the performance of monolithic kernels and the flexibility of microkernels. They are a good choice for general-purpose operating systems that need to be both fast and secure. Nanokernels are very small and efficient, but they are not suitable for most general-purpose operating systems. They are typically used in embedded systems where size and efficiency are critical. Exokernels are a very different approach to operating system design, and they are not yet widely used. However, they have the potential to offer very high performance and security. In addition, current operating systems (Windows/Unix and modern RTOS) struggle to handle the diverse needs of IoT applications (Gaur and Tahiliani 2015). This has led to the development of specialized IoT systems. Since Fog computing sits above the IoT layer, it's likely to be impacted by the kernel architectures of these operating systems. This suggests that kernel selection is crucial when designing an operating system specifically for Fog computing.

2.1.2. User Space vs Kernel Space

In an operating system, the computer's memory is seperated as kernel space and user space (userland) as a general approach. User space places all user applications such as web browser, word processor, games, and everything user interact to computer. Each application gets its own private space within user space, preventing them from interfering with each other. User space plays by the rules defined by the OS, but ultimately users have more control over what happens here. Kernel space, on the other hand, is like the restricted, high-security zone controlling the memory section. Here resides the kernel, which

manages all hardware resources like memory, storage, and devices. Only the kernel has full access to this space, granting limited, controlled access to applications when needed. This separation ensures stability and security, preventing applications from accidentally or maliciously messing with critical system functions as shown in Figure 2.5.

However, in this thesis, we addressed this situation in a different way within the scope of Fog computing. We will discuss how this field separation, whose advantages and disadvantages we include below, affects our operating system approach and design.

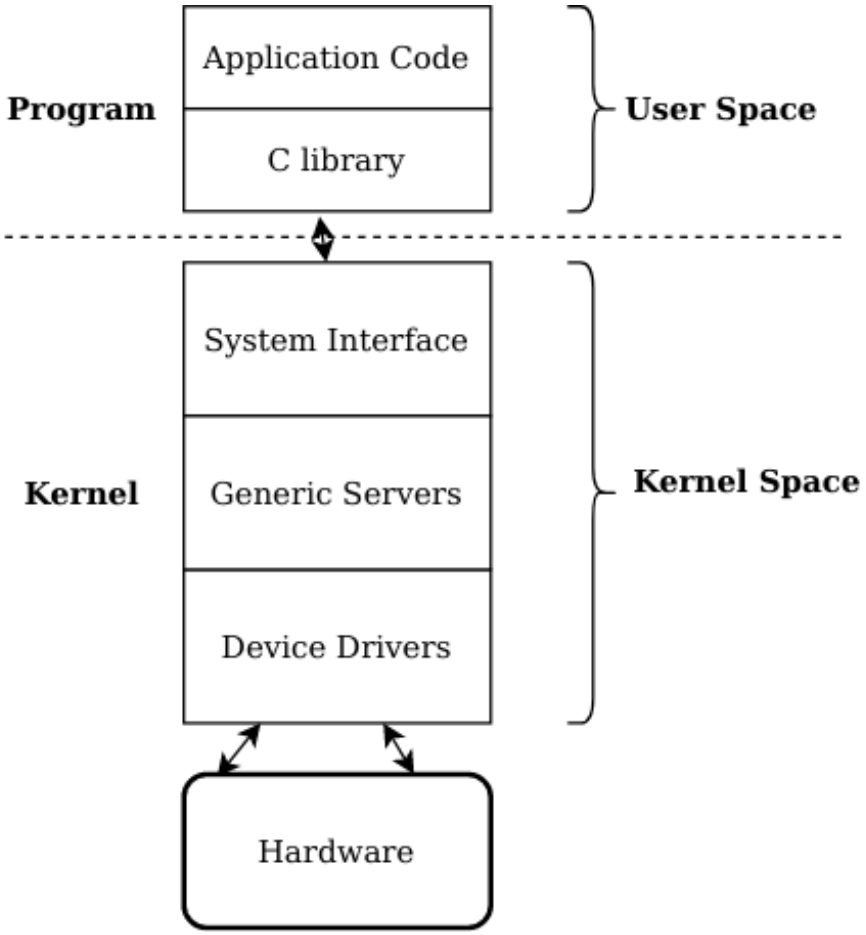

Figure 2.5. Kernel vs. User Space Relation

Using only kernel space offer some unique advantages compared to traditional operating systems with separate address spaces for each process. These advantages can be particularly beneficial for Fog computing applications, where resource efficiency, security, and real-time performance are critical. Without going into details, terminologically, the term "thread" will be used instead of the concept of "process" only in the kernel space;

- Resource efficiency: Kernel threads within a process share a unified memory space. This allows them to access the same data and code without creating copies, which is especially beneficial for memory-limited edge devices. Additionally, the lack of separate address spaces simplifies memory management, leading to potentially lower CPU overhead and improved performance, particularly for in-kernel implementations (Minghao et al. 2007).

- Simplified pointer-based communication: Any valid pointer can be used by any thread, regardless of its origin. This simplifies data sharing and communication between threads, making it easier to develop distributed applications for Fog computing environments. Since pointers are globally valid, there's no need to translate or adjust them during context switches, potentially reducing context switching overhead and improving performance.

- Enhanced security: Using only kernel space can implement more granular access control mechanisms based on capabilities or tags associated with memory regions. This allows for greater control over data access and can potentially improve security compared to traditional permission-based access control. The absence of separate address spaces and associated translation mechanisms can create a smaller attack surface, potentially making the system less vulnerable to certain security exploits.

- Persistence and real-time capabilities: The only kernel space persists across system reboots, making it easier to store and access persistent data in a unified manner. This can be beneficial for real-time applications requiring continuous access to shared data. With data residing in only kernel area, less data needs to be copied or moved between processes, potentially improving real-time performance (Heiser et al. 1998).

Overall, the advantages of using kernel space for Fog computing are significant, particularly in terms of resource efficiency, simplified communication, and potential security benefits. On the other hand, using user-space redundantly presents several disadvantages when if we want to apply to Fog computing, especially compared to other options like RTOS. Here are some key drawbacks we discussed:

- Performance overhead: User space generally run on top of a kernel, adding an extra layer of abstraction that can introduce performance overhead. This is particularly problematic for resource-constrained edge devices in fog networks, where latency and real-time responsiveness are crucial.

- Resource consumption: In monolithic systems, user processes share memory, making them resource-intensive. Switching between these processes often uses round-robin scheduling, which might benefit slightly from an approximation of least recently used (LRU) preemption algorithms when memory pressure is moderate. However, with excessive memory swapping (thrashing), such systems could see performance degradation (Wiggins and Heiser 2000).

- Limited real-time capabilities: User space are not optimized for real-time tasks, where predictable and deterministic behavior is critical. This can be an issue for applications requiring precise timing guarantees, such as industrial automation or autonomous vehicles in Fog computing scenarios.

- Security concerns: User space often have larger attack surfaces due to their complex nature and additional software layers. This can increase the risk of security vulnerabilities and exploits, particularly in edge environments where security needs are often heightened.

As a result, Fog computing leverages resource-constrained devices at the network edge. However, its reliance on a kernel-only architecture, while offering advantages mentioned previously, presents security challenges. Traditionally, user-space applications benefit from isolation provided by the operating system. This isolation can potentially be achieved within the kernel through controlled configuration, especially in a heterogeneous fog environment with diverse devices. By executing user-space applications securely within the kernel, this approach eliminates the need for a separate user space, resulting in a more compact and potentially more secure environment. Additionally, a well-defined domain within the kernel specifically designed for applications can further enhance security and reduce overall footprint.

2.2. Fog Computing

Although the term *Fog computing* was first coined by Cisco in 2012 (Bonomi et al. 2012), the concept and underlying technologies have been evolving for decades. Historically, it actually has a background dating back to the 2000s. From the fertile ground of early computing sprouted the concept of distributed computing, where data processing tasks were no longer confined to a single machine, but shared across a network of interconnected devices. The increasing popularity of wireless sensor networks for data collection and monitoring in various fields created a demand for edge computing principles. The large amount of data generated by these networks necessitated real-time processing and analysis closer to the source, rather than relying solely on centralized cloud computing. As seen in Figure 2.6, empowered by mobile and efficient computing, data processing is shifting to the edge. Soon after, in 2012, Cisco officially adopted the term and took a leading role in its commercialization through the as a computing platform that promotes the benefits of decentralized intelligence at the network edge. The growing interest in IoT and the need for real-time data processing has led to the further development and adoption of Fog computing concepts by various vendors and research groups. In 2015, the OpenFog consortium was formed by industry leaders to promote open standards and interoperability in Fog computing (Industry IoT Consortium 2015).

In the following years, ongoing research, development and standardization efforts by various organizations and industry players progressed. Fog computing has come to this day by gaining more application areas with developments in various fields such as smart cities, industrial automation, healthcare and more. Fog computing's future is intertwined with key trends; a merging with edge computing for broader decentralized processing, seamless integration with big data on Cloud computing for comprehensive data management, a heightened focus on security and privacy at the network edge, and the adoption of open standards and open-source projects to ensure wider compatibility and foster a thriving ecosystem. These trends paint a picture of Fog computing evolving as a collaborative and complementary approach for computing area. In the following parts, the clarification of the boundaries of Fog computing and its relationships with the technologies it is in close contact with will be discussed.

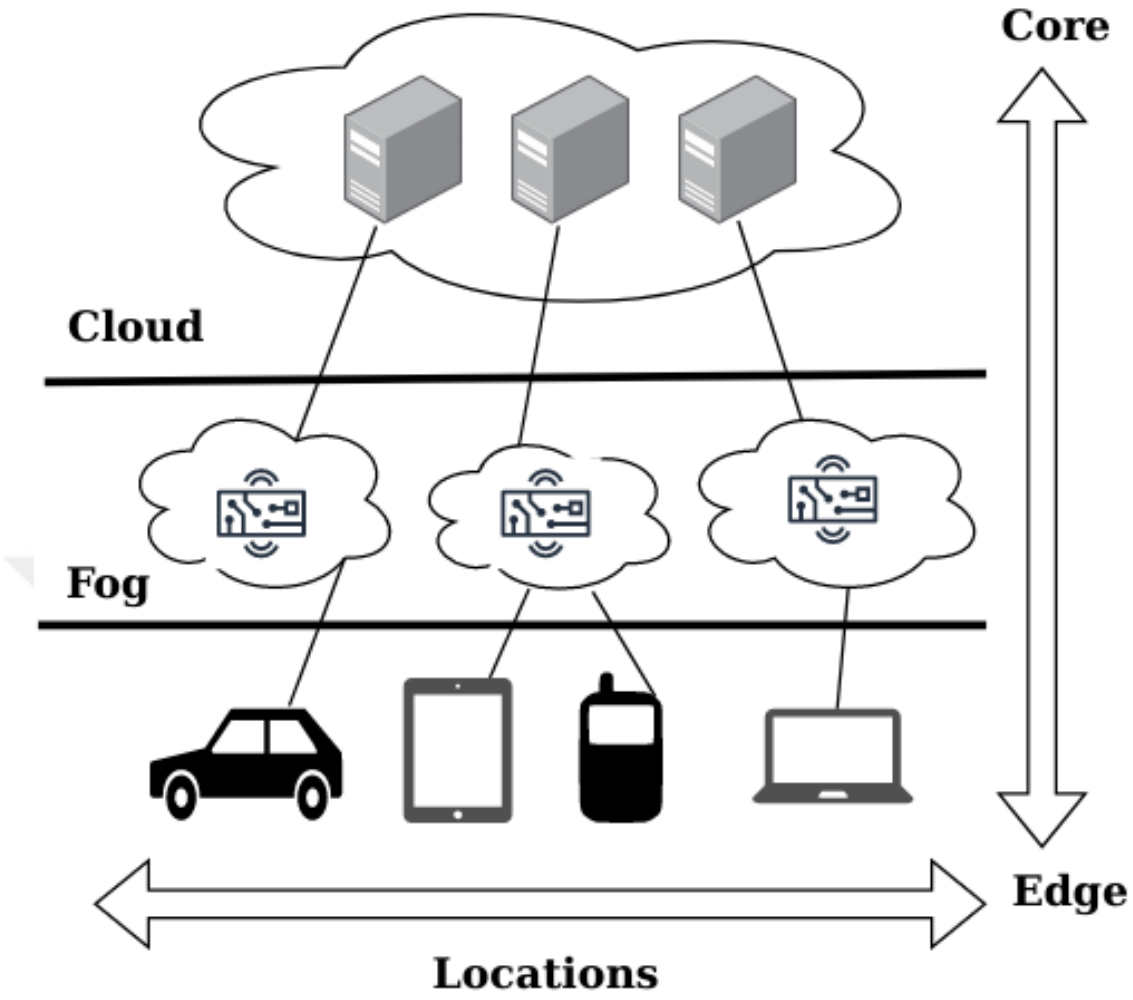

Figure 2.6. The Fog as a Layer Between Smart Devices/Sensor and the Cloud

2.2.1. Fog Computing and Edge Computing

Edge computing is a distributed computing paradigm that processes data near the source where it is generated, instead of relying on centralized data centers. This means that devices like sensors, smartphones, and other IoT devices can analyze and act on data locally, without having to send it all the way to the cloud for processing. Although Fog computing is included as an edge implementation (Dolui and Datta 2017), Edge computing and Fog computing share common concepts in focusing on processing data closer to its source, as shown in Figure 2.7, but they have the following differences (Singh et al. 2019):

- Location: Edge computing processes data locally where it's generated, while Fog computing does so on nearby gateways, acting as a middle ground between Edge and Cloud processing.

- Latency and processing power: Edge prioritizes ultra-fast processing at the source with constrained resources, while fog offers lower latency and more complex analysis closer to the edge with increased resources.

- Data filtering and aggregation: Fog computing, unlike Edge computing limited to basic analysis, offers advanced data manipulation like filtering, aggregation, and pre-processing before transferring it further.

- Connectivity and communication: Edge computing, limited in connectivity, relies on direct communication with the Cloud or other edge devices, while Fog computing acts as a hub connecting multiple edge devices and exchanging data with the Cloud.

At this point, it is necessary to mention a concept that emerged as a result of the convergence between Fog computing and Edge computing. This concept is Cloudlet, is essentially a small-scale cloud computing environment deployed at the network edge, closer to the data sources. It acts as an intermediary layer between the devices and the central cloud. It behaves like a mini-cloud that provides computing, storage, and networking resources closer to the action (Verbelen et al. 2012). But the relation between Cloudlets and Fog computing can be described in a few different ways, depending on the perspective:

- Cloudlets as part of Fog computing; as mini data centers at the network edge, are a type of fog node used for localized processing and filtering in Fog computing, reducing latency and bandwidth by handling initial data tasks before sending it to the central cloud.

- Cloudlets versus Fog computing; different scales and locations while both aim to bring computing closer to users, Fog encompasses a wider range of technologies and can be deployed at various points between devices and the cloud. Cloudlets typically reside at the network edge, closer to end-users, like base stations or WiFi

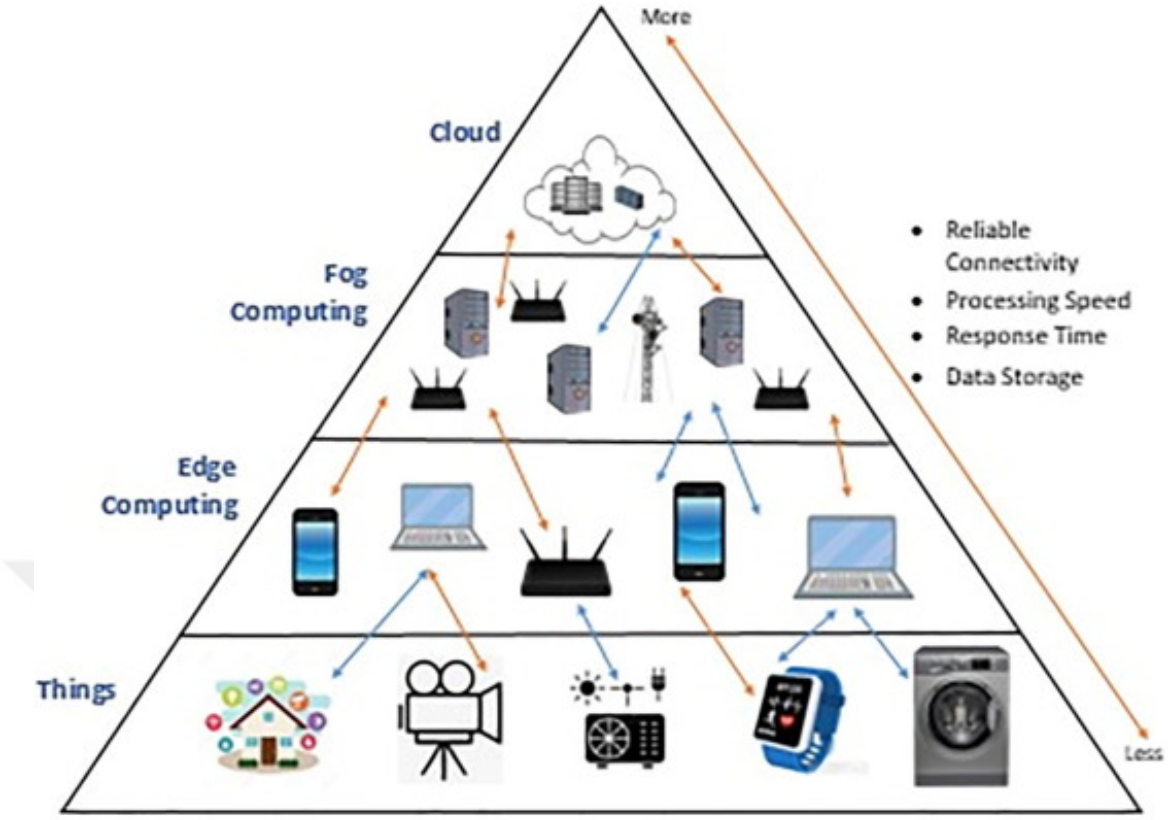

Figure 2.7. Hierarchical View - Fog and Edge Computing (Ashraf et al. 2022)

access points. According to resource limitations; Cloudlets offer limited resources compared to the vast capabilities of the central cloud. However, this trade-off allows for faster processing and lower latency due to their proximity to devices.

Overall, Cloudlets and Fog computing are interrelated concepts. Cloudlets act as building blocks or specific implementations within the broader Fog computing architecture. Both share the goal of decentralizing computing to improve responsiveness and efficiency for applications and devices at the network edge (Dolui and Datta 2017). It was included in the literature review because it was a concept term used before Fog computing. In general, we can accept Cloudlet as a primitive terminological representation of Fog computing.

2.2.2. Fog Computing and Cloud Computing

The massive amount of data generated today, known as big data, is difficult to store and analyze using traditional methods. Cloud computing, with its on-demand storage and processing power, provides the perfect platform for big data. With its distributed data processing approaches allows businesses to analyze massive datasets efficiently and cost-effectively, unlocking valuable insights that can transform their operations. Therefore, big data focuses on collecting, storing, and analyzing this data to extract insights and make informed decisions. Big data analytics typically happen in powerful, centralized processing centers where placed as Cloud computing (Singh et al. 2019). Fog computing, on the other hand, deals with processing data at the edge of the network, closer to where it's generated by devices like sensors or IoT gadgets. This allows for real-time analysis, bandwidth conservation, privacy and security. Therefore, the importance of Fog computing compared to big data or Cloud computing is mostly complementary, not competitive in action. Fog computing processes data at the edge, while big data analyzes it in centralized pools (Rayes and Salam 2017). Both are crucial for different stages of data processing. For detailed understanding, we describe the Table 2.2 which shown the differences between Cloud computing and Fog computing.

2.2.3. Reference Architectures

Fog Computing introduces the concept of Cloud-to-Thing continuum which aims to bridge the gap between traditional cloud-based systems and resource-constrained devices at the network edge. It facilitates distributed data processing closer to the source, minimizing latency and bandwidth limitations inherent in centralized cloud architectures. This approach offers significant advantages in terms of privacy, reliability, and responsiveness compared to purely cloud-centric solutions. However, it necessitates an increase in local computational resources at the edge. From an architectural standpoint, Fog computing seeks to establish itself as an intermediary layer between cloud data centers and edge devices. Notably, there's a lack of a universally accepted reference architecture for Fog computing. Ongoing research investigates the optimal number of layers required for a robust Fog computing infrastructure. Existing literature proposes architectures with var-

Table 2.2. Comparing with fog computing and cloud computing

Feature	**Cloud**	**Fog**
Location	Remote data centers	Distributed edge devices
Latency	High	Low
Scalability	Highly scalable due to vast resources	Less scalable than cloud, but can be scaled within local network
Client - server distance	Multiple hops	One hop
Security	Robust security measures, but potential vulnerabilities due to centralized storage	Requires additional security considerations for distributed environment
Cost	Pay-as-you-go model, can be cost-effective for resource-intensive tasks	Initial investment in edge devices, ongoing maintenance costs
Suitable Applications	Large-scale data processing, web applications, email	Real-time applications, IoT, smart cities, autonomous vehicles

ying levels of complexity, ranging from three to eight layers, depending on the specific application requirements (Antonini et al. 2019). Noteworthy and standardized reference architecture studies are discussed below in terms of determining the interaction environment of the operating system;

1. The OpenFog reference architecture created by the OpenFog consortium, is a widely accepted standard that provides a layered structure for Fog computing systems. It covers various functionalities like compute, storage, networking, and security, and ensures compatibility between different fog components. This flexibility allows it to adapt to diverse use cases and deployment scenarios, encompassing all aspects of Fog computing, not just the computational layer (OpenFog consortium 2018). The system is a three-layer architecture. The Fog layer, made up of Fog nodes, is further divided into four sub-layers: Platform, Node Management & Software Backplane,

Application Support, and Application Services as shown in Figure 2.8. The bottom layer is the physical hardware, or Platform Hardware. The Node Management & Software Backplane layer handles overall node management and communication between various points like remote cloud systems, edge devices, and other Fog nodes. The Application Support layer provides general-purpose micro-services like databases, storage, networking, security, and more. Finally, the Application Services layer offers various services.(Antonini et al. 2019).

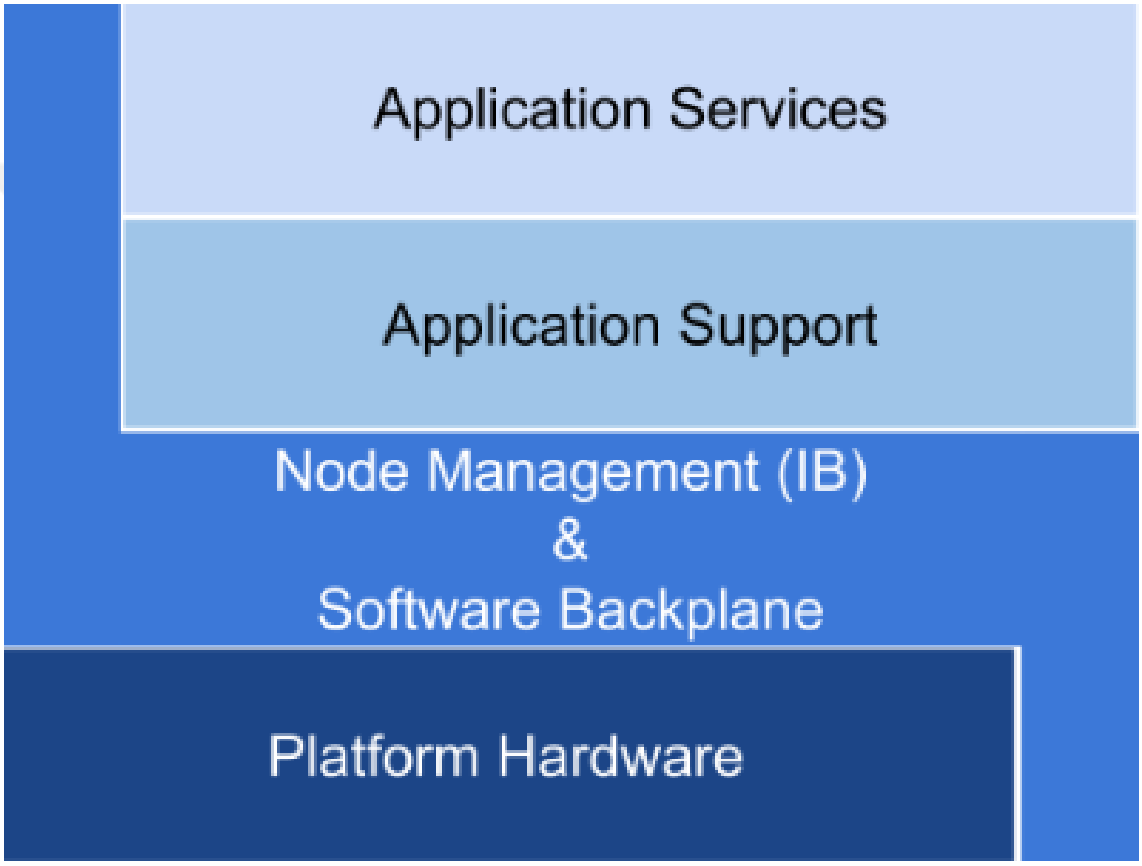

Figure 2.8. OpenFog Software Architecture View (OpenFog consortium 2018)

2. The Edge computing consortium (ECC) architecture prioritizes seamless integration between the edge layer and the cloud, emphasizing real-time processing, security, and resource management. It caters to applications requiring ultra-low latency and minimal cloud dependence, addressing specific security concerns at the edge and providing additional edge-specific details, complementing the OpenFog reference architecture (Edge computing consortium 2018).

3. Cheng et al. (2018) introduced FogFlow is a cloud-edge orchestration framework built for FIWARE that manages applications for the IoT at the Fog computing layer which allows developers to manage and automate data processing tasks at the edge of the network, where data is collected from IoT devices. FogFlow also leverages FIWARE's infrastructure to orchestrate data processing tasks across at fog nodes.

   Among these architectures, the OpenFog reference architecture serves as an important road map in building Fog computing systems. The basic building blocks, functions and connections required to create and manage fog nodes and networks are clearly highlighted in this reference architecture. Adaption of this architecture brings more fog operating systems support a more harmonious fog ecosystem by ensuring compatibility between various hardware and software components. It will also support seamless interaction between Fog and Cloud systems.

2.3. RISC-V

Emerging in 2010 as a research endeavor at the University of California, Berkeley, the RISC-V instruction set architecture (ISA) originated with the goal of addressing limitations inherent in existing proprietary ISAs. This open-source approach aimed to establish a foundation for the future of processor design. To foster widespread adoption and standardization, the RISC-V Foundation was established in 2015. This entity convened industry leaders, academic institutions, and individual contributors to collaboratively propel the development and dissemination of RISC-V technology. Since its inception, the RISC-V Foundation currently boasting hundreds member organizations. The RISC-V ISA's reach has extended to a vast array of applications, with companies adopting it for everything from microcontrollers and embedded systems to high-performance computing and data center processors.

RISC-V is gaining significant traction in various computing domains, from tiny embedded devices to high-performance servers. RISC-V also boasts a sleek core with a focus on efficient instructions for both performance and code size. Its modularity play role through optional extensions for specialized tasks like floating-point calculations, encryption, and virtualization, allowing customization for everything from tiny microcontrollers to powerful systems. And with its open-source nature fuels rapid development and a thri-

ving community, fostering constant innovation in the architecture (Asanovic and Patterson 2014). RISC-V holds significant importance due to its unique characteristics and potential impact across various computing domains. Also developers are empowered to experiment and optimize, potentially disrupting the traditional landscape by democratizing processor design and innovation for all. Also the detailed key benefits are discussed below:

- **Open-Source and Customizable**: Unlike proprietary architectures, RISC-V is openly available and royalty-free. This allows developers to tailor the instruction set to specific needs of their edge devices also prioritizes efficiency across all fronts: minimized power consumption for battery-powered and thermally sensitive devices, advanced security through custom hardware encryption extensions, and deterministic real-time performance enabled by dedicated extensions - optimizing functionality for diverse applications from mobile to industrial automation.

- **Scalability and Modularity**: RISC-V's strength lies in its adaptability, fitting devices from microcontrollers to processors. This allows fog applications to leverage a single architecture across diverse needs. This brings two key benefits: flexibility, where the same base architecture simplifies development and management across various devices within the network, and modularity, where specific extensions can be chosen to add needed features without unnecessary complexity, optimizing resource utilization and performance (Cui 2023). This makes it particularly attractive for building secure systems for the IoT where resource-constrained devices are common.

- **Security**: Open-source nature allows for wider scrutiny and potential faster patching of vulnerabilities compared to closed-source architectures. Also it has standard extensions for cryptography and trusted execution environments. These extensions provide hardware-level support for encryption algorithms and secure enclaves for protecting sensitive data and applications.(Bove 2022)

- **Lower Power Consumption**: By prioritizing simplicity and efficiency, RISC-V's minimalist design holds the potential for significantly lower power consumption compared to its proprietary counterparts. This makes it a game-changer for battery-powered devices like those in the IoT, where extending operation time and reducing

recharge frequency are crucial (Núñez-Prieto et al. 2023). Additionally, in resource-constrained environments with limited cooling capacity, RISC-V's efficiency helps manage heat dissipation, enabling powerful computing even in challenging conditions (Serrano et al. 2021).

- **Growing Ecosystem**: Fueled by a flourishing developer, manufacturer, and toolchain provider community, the RISC-V ecosystem is accelerating development cycles, broadening adoption, and bolstering tooling. This translates to quicker innovation and adoption thanks to collaborative contributions, easier implementation due to wider hardware and software compatibility, and access to constantly evolving development tools and resources.

Additionally, while the base RISC-V architecture supports classic virtualization methods, the optional hypervisor extension mode takes things a step further. This powerful expansion adds dedicated modes and registers specifically designed for hypervisors, empowering them to seamlessly manage guest operating systems with enhanced efficiency and control, extending classic capabilities and opening doors for innovative virtualization implementations (Sá et al. 2021).

RISC-V virtualization shines with its ability to consolidate resources, allowing to run multiple isolated guest operating systems on a single hardware platform. This maximizes utilization, cuts costs, and empowers enhanced security and reliability by offering strong isolation between each virtual machine. Additionally, the flexible nature of RISC-V virtualization allows independent operating systems and configurations for each virtual machine with its own virtualization modes (U, S, H, M), shown in RISC-V layers Figure 2.9, making it adaptable to diverse workloads and scenarios. In short, it's a powerful tool for optimizing resources, boosting security and unlocking customization across RISC-V hardware landscape.

2.3.1. RISC-V and Fog Computing

Fog computing leverages RISC-V's strengths to address the challenges of resource-constrained edge devices, diverse device capabilities, and cost-effectiveness in Fog deployments. RISC-V's simpler design enables lower power consumption, making it ideal

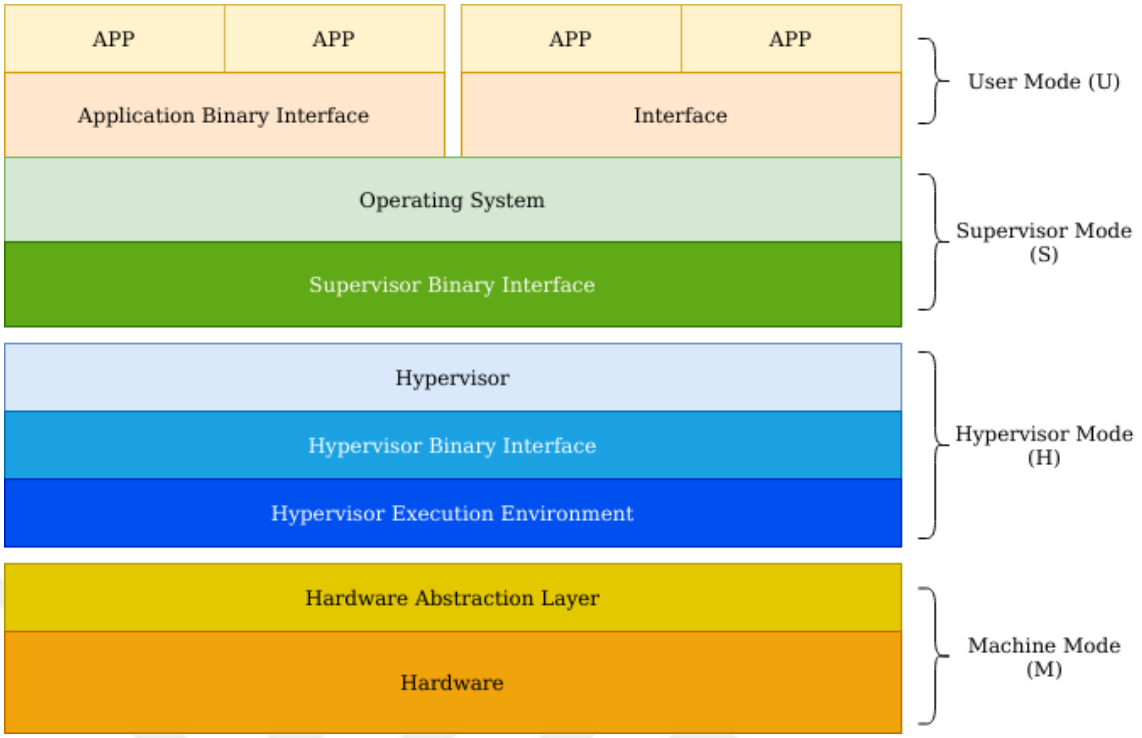

Figure 2.9. Layers in RISC-V Architecture

for edge devices with limited battery life. Its scalability allows for a unified architecture across various fog applications, regardless of the device capabilities. Also the open-source nature of RISC-V permits customization of instruction sets for specific edge-processing needs, such as security or real-time performance and simpler design potentially reduce costs for large-scale Fog deployments (Lump et al. 2023).

Fog computing acts as a springboard for RISC-V's growth by creating a thriving ecosystem of optimized tools and hardware, offering real-world testing grounds to refine its capabilities, and ultimately expanding its market presence through successful fog applications. This symbiotic relationship fuels advancements in both areas, with RISC-V becoming increasingly adept at Fog computing tasks like real-time data processing in smart factories, autonomous vehicles, and wearable health devices, ultimately accelerating innovation in edge computing.

In addition, the rise of automation and smart systems in industries and the increase in the number of smart devices cause concerns about data security. Communication and security mechanisms for Fog computing, which are architecturally located on IoT sys-

tems, are integrated into production and daily life. One approach involves optimizing resource-intensive algorithms to improve performance and security on constrained devices. However, simplified algorithms such as cryptography may have limitations due to their non-standard nature. Instead, it will be inevitable to use processors designed for a specific set of applications (Pan et al. 2021). At this point, RISC-V comes into play and will complete the security side of Fog computing at the hardware level.

2.4. Virtualization and Fog Computing

According to OpenFog reference arhitecture (2018), Fog computing extends the traditional cloud model by distributing computational resources and data storage across various layers within a network's hierarchy. This layered approach aims to maintain the core benefits of Cloud computing, such as containerization, virtualization, orchestration, manageability, and efficiency, while bringing these functionalities closer to the edge of the network. Virtualization plays a pivotal role in Fog computing by enabling resource isolation, improved security through guest OS separation, and dynamic provisioning for efficient resource utilization. We can discuss the key contributions of virtualization to the Fog computing paradigm as follows:

Resource Power: Consider deploying multiple virtual machines on a single Fog device. This can be achieved through hardware virtualization extensions and a hypervisor, which enables efficient resource sharing of the underlying physical CPU, memory, and storage. Virtualization is especially advantageous in resource-constrained environments where optimizing power consumption is essential. By leveraging hardware-assisted virtualization and proper VM isolation techniques, a hypervisor can guarantee isolation and security for each VM. This isolation prevents conflicts between VMs and strengthens the overall system's security posture.

Flexibility: Legacy hardware limitations are a thing of the past. Virtualization technology enables the creation of isolated environments on a single physical device. These VMs can run disparate operating systems and have independent resource allocation. This empowers the deployment of a broader range of solutions for Edge computing applications. Development and testing cycles are significantly accelerated as virtualization facilitates rapid provisioning, testing, and updates of new applications and services. Lump et al.

(2023) established the essential building blocks necessary to enable core functionalities within containerization and orchestration platforms for edge computing environments. These building blocks encompass network plugins, container runtimes, and the computational and networking resource prerequisites. Also centralized management tools streamline the administration of these diverse VMs across numerous fog devices, reducing complexity and simplifying overall operations.

Performance and Availability: Virtualization ensures that even when a single virtual machine experiences a problem, other VMs remain operational, minimizing service disruptions and guaranteeing high availability. Li et al. (2018) conducted experiments to confirm that virtualization effectively minimizes delays and jitter within the system by implementing a virtual environment. Also VM deployment and updates occur instantaneously, further accelerating system response times. To address unforeseen spikes in demand, workloads can be efficiently distributed across multiple VMs, resulting in smoother performance and optimized load balancing.

Portability: Virtualization enables the deployment of standardized virtual machines across heterogeneous fog nodes. This eliminates the need for device-specific tools and management techniques, streamlining operations and simplifying maintenance. Applications packaged as VMs become agnostic to the underlying hardware. This allows for effortless migration between different fog environments, enhancing deployment flexibility and adaptation to dynamic resource demands.

In terms of virtualization approaches, hypervisor and container are two well-known virtualization technologies in use. Hypervisor operates at the hardware abstraction layer (HAL), virtualizing physical resources like CPU, memory, and storage. They enable the execution of multiple virtual machines concurrently. VMs leverage hypervisor-managed resource allocation to improve overall utilization (Tseng et al. 2018). Notably, hardware-assisted virtualization extensions are prevalent in modern processors, making them suitable for Fog computing deployments. Hardware virtualization for I/O and processing allows multiple entities to share a single physical system. Even the fog node's operating system can run in isolation using VMs managed by the hypervisor. This isolation simplifies resource management. Unlike hypervisors, containers focus on operating system-level virtualization. They package an application with its dependencies, allowing it to run

consistently across environments with a compatible container engine. This approach is lightweight and offers faster startup times compared to VMs (Benomar et al. 2019). The relationships of both virtualization technologies with Fog computing servers are shown in Figure 2.10 comparatively.

In conclusion, virtualization serves as a cornerstone technology for Fog computing. By enabling resource optimization, standardization, and application portability, virtualization empowers fog to achieve its full potential. This paves the way for a new era of distributed computing innovation characterized by efficiency, scalability, and agility.

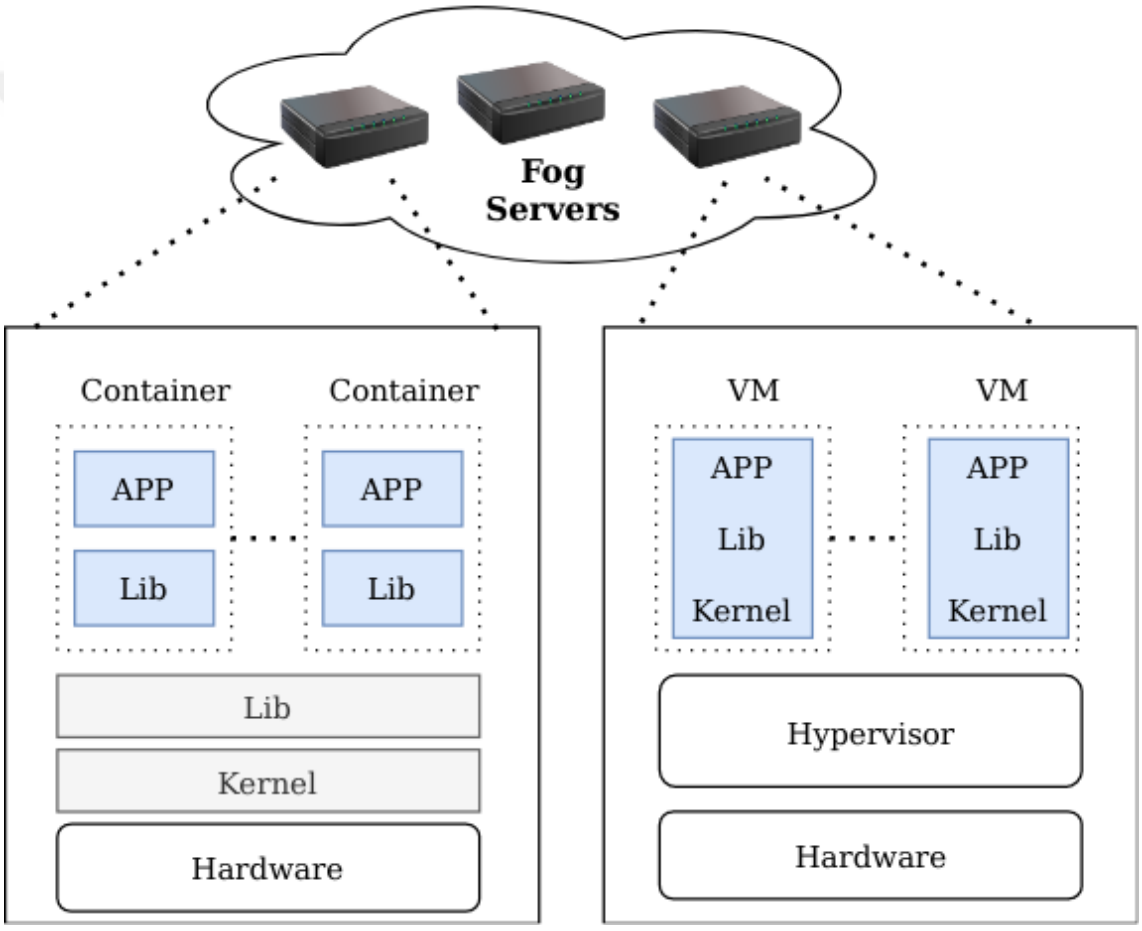

Figure 2.10. Virtualization Technologies on Fog Computing Architecture

2.5. Related Works

In the literature, as we mentioned in the introduction section, Fog computing is generally designed with a framework structure on top of Linux operating system. Studies closer to our study at the operating system level were tried to be examined. Among these, BasatNet, FogOS, OpenStack, and FORA were investigated for their relevance to our research focus.

BasatNet is a distributed computing environment which we proposed as capable of scaling up to devices with heterogeneous architecture (Ince et al. 2022) in our earlier study, allowing devices to establish clusters with resource-limited nodes and subsequently run on top of it. The solution can be thought of as a minimalist hybrid approach between HPC and big data which defines the Fog computing around. Similarly, our previous work actually depicts Fog computing as shown in the Figure 2.11, data nodes correspond to the storage layer in our work, compute nodes correspond to the computation layer (executor unit / dynamic runtime), and the coordination part corresponds to the Controller part in our study. Additionally, the study tried to ensure minimum resource consumption by using Linux containers. The deployment and management challenges encountered in that study brought forth the idea of using a customized nanokernel instead of Linux in this study and the development of a different approach for the Fog computing layer.

FogOS is an operational system design presented by Choi et al. (2017), distinct from the structural operating system concept. They proposed a operating system for IoT services. Their system pretends to view the entire IoT ecosystem as a computer and acts as an operating system for it. In their design, a reference architecture including resource management, application management, edge resource: registration, ID/addressing, and control interface sections are presented. No details were given regarding the core components of the operating system. They tried to manage all Cloud, Fog, and Edge resources, such that fog application developers can use an API to request certain resources. In this way, it is predicted that the development and distribution of IoT applications will be easier. It seems to be designed as an abstract distributed operating system instead of an operating system.

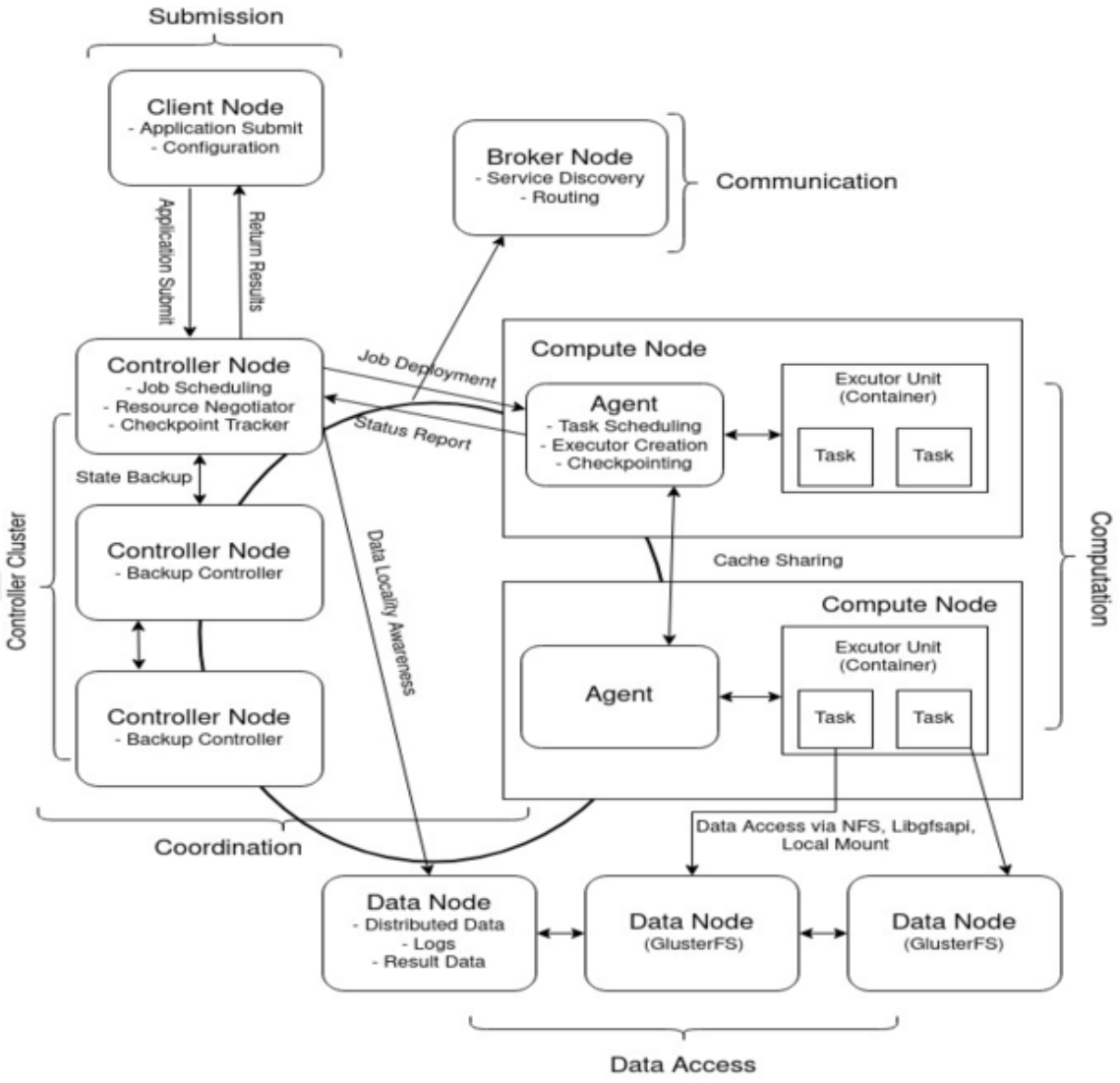

Figure 2.11. BasatNet High Level Design (Ince et al. 2022)

OpenStack is an attempt to set up a Fog computing environment with container management applications using the Stack4Things IoT framework of the OpenStack cloud infrastructure (Benomar et al. 2019). It has been shown that the volume of data communication with the cloud is reduced by the local fog network by establishing an integrated working environment in the cloud infrastructure. This system, which uses Docker as a container application, depends on the OpenStack cloud infrastructure. The system meets the storage and computation processes from OpenStack sub-components as a service. With this structure, it is more of a service integration component and the deployment processes are tightly connected to the OpenStack

ecosystem.

FORA is another study where Fog computing was discussed as a platform, considering its functions within the scope of Industry 4.0 (Pop et al. 2021). The platform is referred to as the Fog Computing Platform (FCP), which runs mixed-criticality and control applications. In FCP, multiple heterogeneous nodes are interconnected. Control applications are run as virtualized tasks within isolated partitions, each with its own operating system where RTOS were utilized in the partitions. Regarding computation, control applications run on operating systems with flexible scheduling policies, such as PikeOS (Kaiser and Wagner 2015). The hypervisor capability embedded within PikeOS creates a secure and isolated environment for executing multiple applications concurrently on edge devices. This segregation is crucial for safeguarding the integrity of the system. Using PikeOS for Fog computing, presents a similar situation to our study by harnessing its microkernel structure. While microkernel architecture typically provides hypervisor services to applications, in our case, applications are embedded within the kernel and directly access services as functions.

Regarding related studies, FogOS is more of a distributed Fog computing application platform than the emphasis on "operating system" in its name. OpenStack is a Cloud platform built on Linux base. FORA is an industry-centric Fog computing framework focused on control engineering. Our inference is, unlike these studies that we discussed in the context of comparison, the operating system mentality is presented and implemented in PusOS in the most accurate way.

3. MATERIAL AND METHOD

This section dives into the low-level implementation details of the Pus operating system, focusing on the components that constitute the kernel application layer. Below is a diagram outlines the blocks of the PusOS computing environment in Figure 3.12.

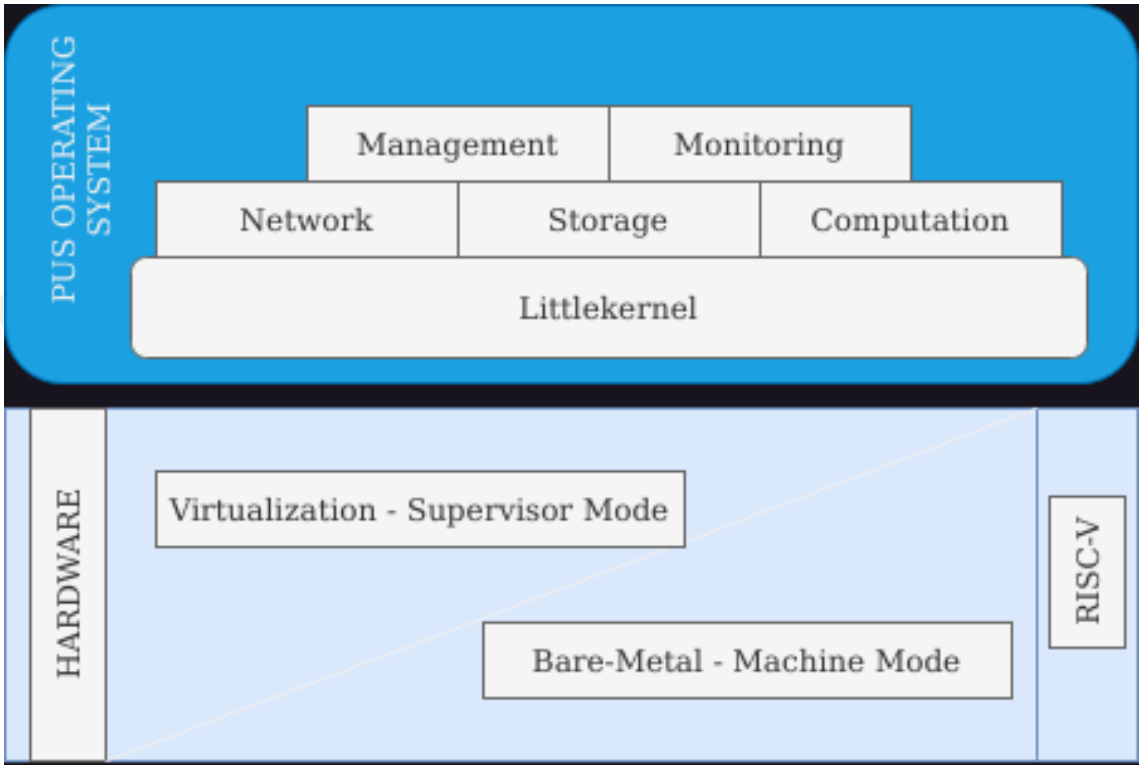

Figure 3.12. Block Diagram of the PusOS Fog Computing Environment

3.1. Kernel Structure

In our literature review, we investigated various types of kernels in operating systems. In particular, it was emphasized that nanokernels offer advantages that position them well for applications at the intersection of Cloud and Internet of Things technologies. A nanokernel-based system deployed in the Fog computing layer holds promise for significant performance and modularity gains. Motivated by this potential, we sought a suitable kernel to serve as the core for the intermediate system in our research. We leveraged the concept of being intermediate layer to analyze device functionality. In this context, bootloaders emerged as the most fitting component. Bootloaders are programs that perform

essential pre-boot checks and facilitate the transition to a higher-level operating system. Inspired by the intermediary role of bootloaders, our research, we identified the Little-Kernel (Geiselbrecht 2008) project as a compelling option. This project underpins the bootloader software for many Android devices (Android 2024) and is also employed by NVIDIA as trusted execution environments (2024). The following section delves deeper into the LittleKernel's structure and the rationale behind our selection as kernel of Pus operating system.

3.1.1. LittleKernel

LittleKernel (LK) is an open-source, highly modular, and lightweight operating system kernel specifically designed for embedded systems and IoT devices. LK stands out for its compact size and efficient operation, allowing for fast boot times and increased storage space. Its modular design with optional networking, storage, and sensor functionalities offers customization, while multi-architecture support ensures adaptability to diverse devices. Furthermore, secure boot, memory protection, and fast boot times prioritize system security and speed, and hardware abstraction simplifies interaction with various platforms with extendable debugging feature (Saha et al. 2021). It plays a critical role in the foundation of embedded systems and IoT devices by providing a reliable and secure boot process, hardware initialization, and flexible customization options to meet specific device needs and security requirements (Pundkar et al. 2022). Its robust kernel design and focus on early error detection minimizes the risk of system failures, ensuring reliable operation for these crucial applications.

In addition to the other considerations, we should also factor in whether the system is lightweight, as this might limit its suitability for applications requiring complex operating systems, and the level of customization required, as it may necessitate adaptation and additional modules for specific uses. Besides this, LK has a very understandable project structure which it can be easily built and additional software layers can be added. As seen in the Figure 3.13, the LK project consists of a number of components. These components are completely based on assembly abstraction. At the end, they are compiled and linked into one kernel image. According to this structure;

- *project* defines the naming of the association of modules collected for a purpose.

- *arch* includes implementations related to instruction architectures such as RISC-V, x86 and MIPS.
- *target* includes boards for different instruction sets such as kernel-based virtual machine (KVM), PC-x86 and Raspberry PI.
- *platform* is same as board contents but includes CPU level groupings such as Mediatek and JH7110.
- *dev* includes common hardware libraries for all architectures such as USB, timer and GPIO.
- *lib* includes basic libraries for the kernel such as thread, memory, libC.

PusOS performs layering using LK's overlay mechanism, so that a kernel-adjacent software layer can be easily created within the context of Fog computing requirements. The software layer contains; the console and server applications of Pus with *app*, the network and time libraries with *lib*, the targeted RISC-V architecture with *arch*, and the definitions and configuration related to the Quick Emulator (QEMU) board specification with *target*.

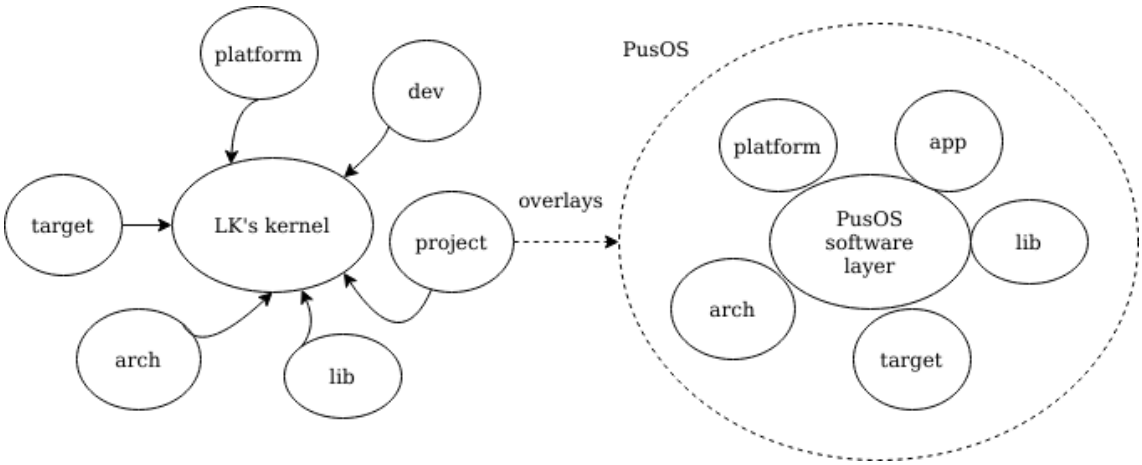

Figure 3.13. PusOS Overlay on top of LK

3.2. Implementation Platform

Choosing hardware and underlying architecture represents a fundamental step at operating system design. As we explained in the Introduction section, we chose the RISC-V architecture due to its advantages. Choosing the right hardware is also crucial decision for operating system development. We used QEMU as a implementation board to show proof of work for this study. Below is a subsection that evaluates the choice of QEMU and also touches on the challenges of using physical board.

3.2.1. QEMU

QEMU is a powerful open-source emulator, stands out for its versatility and ease of use. These qualities make it a popular choice, especially for operating system development. As our go-to development platform, it offers several advantages over physical boards, particularly during the crucial early stages of OS development and testing. We discussed its benefits at below:

- *Versatility*: QEMU's versatility shines in its ability to emulate diverse hardware architectures like x86, ARM, and RISC-V, eliminating the need for dedicated equipment for each platform. This proves invaluable for cross-platform compatibility testing. Furthermore, we can empower the QEMU to craft emulated environments, fine-tuning CPU, memory, peripherals, and even network conditions, allowing to tailor test scenarios and meticulously evaluate specific aspects.

- *Efficiency and convenience*: We need affordability, efficiency, and collaborative features when we need to use a board. QEMU is open-source and free eliminates the cost of physical boards for various architectures. Additionally, setting up and testing emulated environments is much faster than traditional methods, accelerating development cycles. Finally, sharing QEMU configurations and environments fosters collaboration and ensures reproducible test results. Therefore we could tested our configuration on different host machines.

- *Virtualization capabilities*: The combination of QEMU with KVM provides a powerful virtualization solution. While KVM leverages the system's hardware virtu-

alization extensions for near-native performance, QEMU's emulation capabilities enable guest operating systems to be run on different hardware architectures, providing a virtualization environment experience for operating system design.

- *Development and debugging*: QEMU's emulated environment provides isolation which prevents any harm to physical device on top of it. Also it is packed with powerful debugging tools like breakpoints and network analysis, then issues could be tracked efficiently.

In line with the above advantages, the board implementation of the PusOS was made using QEMU. QEMU has 32 and 64 options for RISC-V architecture. We used the RISC-V 64 (qemu-system-riscv64) option which also provides BIOS firmware (RISC-V Supervisor Binary Interface) for RISC-V user modes. As shown in the Figure 3.12, machine and supervisor modes are supported by LK at the hardware level.

3.3. Network Stack

In this section, the software components used in the network stack will be discussed. Fog computing utilizes a software stack positioned between IoT devices and the Cloud which incorporates operating systems for fog nodes, containerization for applications, middleware for data management and communication and processing. In this context, these elements facilitate efficient data flow and processing between resource-constrained devices and the Cloud. To enable communication across this network layer, a standardized effective communication protocol is essential. Fog nodes, equipped with operating systems, exchange sensor data with IoT devices, transmit data to the Cloud, and interact with other fog nodes. This necessitates a common data transmission protocol to ensure seamless communication within the entire Fog computing architecture. The following section provides information about the data transmission format choice.

3.3.1. Data Structure

In data communication, we need to serialize structured, record-like, written data in a language-independent, platform-independent, extensible way in the transmission of data for a certain purpose. This is more important if we consider the architectural structure of

Fog computing where it is close to data sources, so we chose Protobuf for this purpose in this study. Protobuf, short for Protocol Buffers, is a language-neutral and platform-independent mechanism for data serialization (Protocol Buffers 2024). It defines how structured data is encoded and decoded efficiently, making it popular for communication and data exchange in various applications, including the IoT. Protobuf is used in the PusOS to provide the following functions;

- Heartbeat message for monitoring
- Remote code execution (RPC)
- Inter-node communication
- IoT/Cloud data exchange

The first of these is to monitor the liveness of Pus nodes, a heartbeat message system built on the Raft consensus protocol (Ongaro and Ousterhout 2014) structure was implemented for fog nodes to communicate among themselves and with the Controller software. The Protobuf's code is shown as below:

```
syntax = "proto3";
message AppendEntriesRequest {
    // The uuid of the Raft node sending the request.(leader)
    string id = 1;
    // The term from the requester's perspective.
    int32 term = 2;
    // Index of entry immediately preceding new ones.
    int32 prevLogIndex = 3;
    // The term of the prevLogIndex entry.
    int32 prevLogTerm = 4;
    // New ops to append.
    repeated string entries = 5;
    // The leader's commitIndex. - received by the majority
    int32 leaderCommit = 6;
}
```

```
message AppendEntriesReply {
    // The uuid of the Raft node sending the reply.
    string id = 1;
    // The term from the respondent's perspective.
    int32 term = 2;
    // True if consensus
    bool success = 3;
    // if declined, specifying the conflicting index
    int32 conflictIndex = 4;
    // if declined, specifying the conflicting term
    int32 conflictTerm = 5;
}
```

The Raft consensus algorithm employs to trigger leader election. All servers start as followers and remain followers as long as they receive valid RPCs (heartbeat) from a leader or candidate. A leader sends periodic heartbeats to all his followers to maintain his authority (Howard and Mortier 2020). In this way, we periodically collect heartbeat requests from all fog nodes. This requests are tracked through a server software called as Controller. The Controller makes the necessary alive time calculations according to the heartbeat requests coming from the nodes. In response to the requests received, index-increased response messages are returned. If it does not receive a request below the specified heartbeat rate, it performs a recovery process. This mechanism should be transmitted between nodes quickly and effectively. Therefore Raft provides a suitable protocol structure for this usage also by using the Raft protocol definition, we ensure forward compatibility for advanced implementations.

Another use of Protobuf is for RPC communication. In Pus operating system, RPC communication is designed to play role between the fog nodes, the client, and Controller. In general, a client uses this method to call remote functions on fog nodes. In our remote code execution implementation, the commands sent by the client consist of two fields: function and data. The data field can be thought of as a parameter for the function that is demanded to be run. Each sent request has a request ID field. After the sent command are received by the Pus node, the command is executed directly in the serial console of

the operating system and the received output is returned according to the same request number. The commands executed are mostly commands with short responses which are places in the Table 3.3. Also the Protobuf structure used in this interactivity is shown at below:

```
syntax = "proto3";

message CmdRequest {
    int32 id    = 1;  // request id
    string fn   = 2;  // function name
    string data = 3;
}

message CmdResponse {
    int32 id    = 1;
    string data = 2;
}
```

Table 3.3. Remote procedure commands

Command	Description
reboot	restart the node
systime	get the system time
coapcli	fetches the sensor data
datastore	bring the data from storage
fibon	start the fibonacci test
test	run the system tests

In addition, the Protobuf can be used for communication between fog nodes and data transfer to the Cloud. For these cases, extensible data protocols should also be prepared with the Protobuf protocol structure.

3.3.2. Sensor Communication

In Fog computing, sensor communication is designed to address limitations of Cloud-based IoT. Fog nodes behaves as powerful gateways, sit closer to sensors and handle initial data processing. Sensors can use standard protocols like WiFi or Bluetooth to talk to the fog node, reducing latency and bandwidth use compared to sending everything directly to the Cloud. This allows real-time actions based on sensor data, for example, a fog node in a factory analyzing machine sensor data to predict maintenance needs.

Fog computing relies on various sensor communication protocols like MQTT and CoAP besides of traditional HTTP base protocols for efficient data exchange. These protocols are lightweight and suited for resource-constrained sensor devices, enabling them to publish data to a central broker in the fog layer which can then perform local processing, filter data, or forward it to the Cloud for further analysis. We primarily used the CoAP protocol to run the tests we designed in this study. CoAP is a specialized web transfer protocol designed for devices with limited resources and unreliable networks. Unlike HTTP, which can be complex and power-hungry, CoAP prioritizes small message sizes to minimize data fragmentation (Ali 2018). Both CoAP and HTTP operate at the application layer as shown in Figure 3.14, but CoAP's streamlined approach makes it a more suitable choice for resource-constrained environments. While CoAP interacts with lower layers for network transmission, its core function of defining communication methods and data exchange falls firmly within the application layer's responsibilities. This is where CoAP establishes resource identifiers, defines request/response patterns, and handles data manipulation.

CoAP protocol was used to communicate with the sensor device in the established environment. In our setup, the CoAP uses simple messages and leverages UDP for efficient communication. The sensor sends requests (like GET or PUT) to a server to access or update data, mirroring a web interaction with minimal overhead. Below is a step by step explanation of how this process works on both sides in detail;

- Sensor side:

    1. Sensor collects data:
        - The sensor periodically or based on events measures its environment and

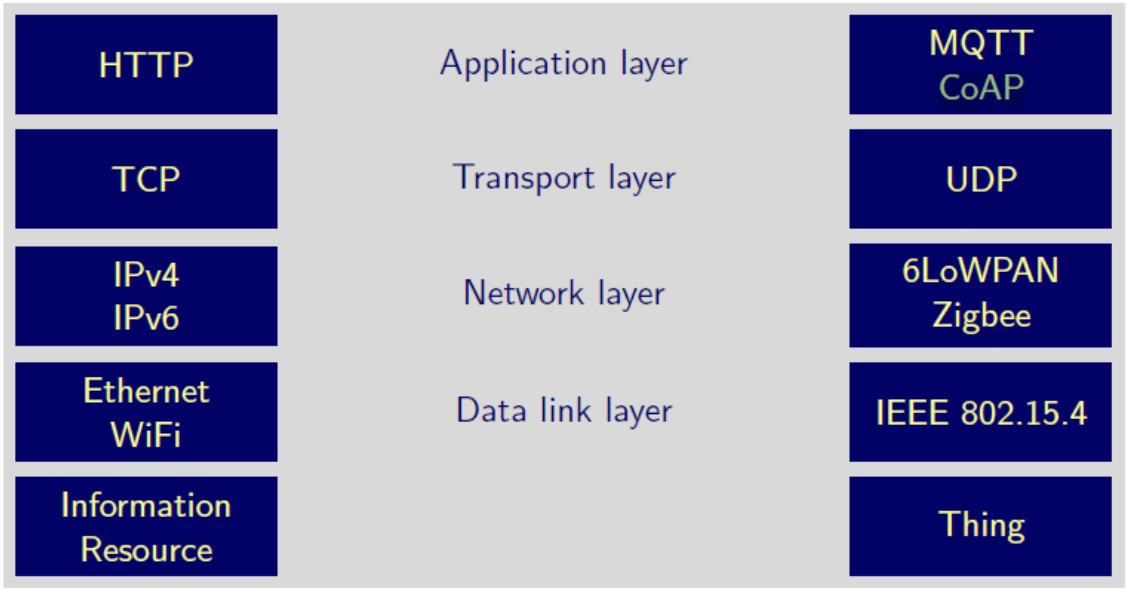

Figure 3.14. HTTP and CoAP in OSI Layer (Al-Haidari and Alqahtani 2020)

stores the data.

2. Prepare CoAP message:
   - Set Request method: POST for transmitting data.
   - Set URI: Specify the resource on the server to which the data will be sent.
   - Set Payload: Include the sensor data in a suitable format such as JSON.
   - Set Content-Type: Specify the format of the payload.
   - Optionally, include additional information like sensor ID, timestamp, etc.
3. Send CoAP message:
   - Send the prepared message to the server using the selected CoAP endpoint.
4. Handle response:
   - If successful (Code 20x), acknowledge receipt and continue.
   - If error (Code 4xx or 5xx), re-transmit the message after a backoff delay.
   - If timeout occurs, resend the message.
5. Repeat: Go back to step 1.

Server side:

1. Listen for CoAP messages:

- The server listens on the CoAP port for incoming messages.

2. Parse CoAP message:

   - Extract the request method, URI, payload, and other relevant information from the received message.

3. Validate message:

   - Check if the request is valid.

4. Process data:

   - Extract the sensor data from the payload.

   - Store the data in a file or forward it to another application.

5. Send response:

   - Send a CoAP response message with an appropriate code (204 Created for successful reception).

   - Optionally, include additional information in the response payload.

3.4. Storage

In this section, how the data is stored, which file system the necessary applications are made with, examples and methods are explained. As mentioned in Introduction section, storage feature is crucial for Fog computing as it allows fog nodes to process and analyze data locally instead of sending everything to the Cloud. This reduces latency, bandwidth usage, and improves responsiveness for real-time applications, especially beneficial for the massive amounts of data generated by Iot devices. A comparison of LittleFS, which we chose as the file system, and other candidate file systems is included in Table 3.4.

According to comparison, we can summarize while FAT32 and Ext2 offer compatibility and journaling respectively, they lack wear leveling and are designed for hard drives, making them less ideal for resource-constrained embedded systems with flash memory. Also Ext2 lags behind in terms of complexity and Fat32 at error correction. LittleFS strikes a good balance between performance and efficiency, with wear leveling for flash longevity. For even tighter resource constraints, SPIFFS prioritizes simplicity and excels with static data or infrequently updated files but it has no directory and wear-leveling support. On the other hand, LittleFS boasts a compact code size, keeping the application

Table 3.4. Comparing filesystems for PusOS storage choice

Feature	**LittleFS**	**SPIFFS**	**FAT32**	**Ext2**
Resource Usage	Low	Very low	Higher	Higher
Wear Leveling	Yes	No	No	No
Fragmentation	Less prone	More prone	fragmented	fragmented
Error Correction	Limited	No	No	Yes
File System Size	Limited	Limited	Larger	Larger
Power Cons.	Low	Low	Moderate	Moderate
Read Performance	Good	Good	Good	Good
Write Performance	Good	Good (small)	Slower (random)	Slower (random)
Complexity	less	less	less	more

code and data spacious, while also requiring minimal RAM for operation. It delivers good read/write speeds, even on devices with limited resources. It has robust resilience against common storage challenges: wear leveling evenly distributes writes, extending lifespan; power failure safety minimizes data corruption risk; and error resilience utilizes features like checksums and metadata redundancy to ensure data integrity. Also LittleFS stands out with its feature-rich design, offering directory support for organized file management, a wide range of features like hard and symbolic links, file attributes, and more, and remarkable cross-platform compatibility supporting various architectures and microcontroller boards.

In line with the above advantages, we ported LittleFS, which can be implemented with a few header files, under the "lib" directory. As will be discussed in the "Results and Discussion" section, the essential function is that the sensor data could be stored and read in a "file" format. Listed below are the basic steps of writing to a file that we use to store data in sensor communication:

1. Open the file: Use the open() function with the "append" mode ("a") to open the existing file. If the file doesn't exist, it will be created. Check the return value (File pointer) to ensure successful opening.

2. Seek to the end of the file (optional): While technically not required for appending, seeking to the end ensures data is added at the right location, especially if using write(). Use the seek() function with SeekEnd mode to move the file pointer to the end.

3. Write the data: Use either write() function or print() function, write() takes the data buffer, data size, and returns the number of bytes written. print() takes the data to be printed and appends it as strings with line breaks by default.

4. Close the file: Use the close() function to close the file and flush any pending data. This is crucial to ensure data integrity and avoid file corruption.

Additionally, LittleFS has a flexible configuration structure. In this way, the file system can be used more effectively with custom fine-tuning settings according to different storage behaviors. In the code block below, the settings used in our system and their explanations are included as comments.

```
const struct lfs_config lfs_cfg = {
// block device operations
.read  = &lfs_bio_read,
.prog  = &lfs_bio_prog,
.erase = &lfs_bio_erase,
.sync  = &lfs_bio_sync,
// tuning
.read_size = 32,
.prog_size = 32,
.cache_size = 4096,
// Size of the logical block size used
//by the filesystem in bytes.
.block_size = 4096, //SECTOR_SIZE,
// Number of blocks in the filesystem.
.block_count = 512,
// a multiple of the read and program sizes,
```

```
    //and a factor of the block size.
    .lookahead_size = 4096,
    .block_cycles = 1024,
    // Maximum size of file names in bytes.
    .name_max = 32,
};
```

3.5. Computation Stack

The computation stack, the pillar of Fog computing, acts as a processing bridge between smart devices and the Cloud. It can produce local analytics for real-time decision-making while performing basic tasks such as data pre-processing and filtering to reduce bandwidth usage and latency. In addition, due to the proximity of the computation layer to the data source, advanced security and privacy requirements are met.

Our computing environment is implemented in a two-pronged approach within the operating system. The first of these are libraries that provide essential functions compiled directly into the kernel as static modules. These libraries provide tight integration and efficient access between each other in the execution of functions. However, a completely static build mechanism limits the flexibility. To address this, we introduce a dynamic runtime environment residing in system memory. This allows for expansion beyond the static kernel, enabling customization. Essential libraries leverage static compilation for optimal performance, while the runtime environment utilizes dynamic loading to handle data manipulation at runtime. Bridging this gap is a lightweight script language (Haster 2017) ported as a runtime library. This strategic choice provides users with the ability to create custom scripts and dynamically interact and perform operations with the computational environment, although core functionality remains tightly embedded in the kernel statically. Below we justify our choice in terms of Fog computing by discussing the advantages of using a script language;

1. Script languages are generally known for their conciseness and ease of use, allowing developers to quickly write and test code compared to compiled languages. This is

crucial in Fog computing, where rapid prototyping and iteration are essential due to the dynamic nature of edge environments.

2. Script languages typically have smaller footprints than compiled languages, requiring less memory and processing power to run. This is advantageous for fog nodes, which often have limited resources compared to cloud servers.

3. Script languages are mostly interpreted, meaning the code is translated and executed line-by-line during runtime. This allows for dynamic adjustments and easier debugging compared to compiled languages, which can be beneficial in the unpredictable and dynamic edge environment.

4. Pre-built function libraries can be imported at runtime, thus supporting cumulative development, significantly reducing development time and effort.

5. Script languages are generally platform-independent, meaning the same code can often run on different fog node hardware with minimal modifications. This flexibility simplifies deployment and maintenance across diverse edge environments.

To show the syntax of the language used, how a quick sort algorithm is coded in the shell environment and its working outputs are shown at below:

```
> # A quicksort implementation
> fn qsort(data)
    let [x, ..data] = tbl(data)
    if (len(data) == 0)
        return [x]

    let small = filter(fn(y) -> y <  x, data)
    let large = filter(fn(y) -> y >= x, data)
    return qsort(small) ++ [x] ++ qsort(large)

> print( qsort ([3,4,7,9,0,1,6,5,2,8]) )
> [0,1,2,3,4,5,6,7,8,9]
```

The choice between implementing the algorithm in as kernel application module (C/C++) or a scripting language offers distinct advantages. Kernel modules provides potentially faster execution and finer control over system behavior. However, developing kernel modules is complex and requires the understanding of the kernel internals. Scripting languages often provide built-in libraries for common tasks, making them suitable for prototyping and applications with less stringent performance requirements. While scripting languages offer a convenient way to interact with the algorithm's output, using a kernel module allows for deeper intervention in its logic. C/C++ provides full control over the decision-making process within the algorithm. This is particularly valuable for applications based on simple decision trees, where the logic can be directly translated into code structures like if-else statements and switch cases. This fine-grained control over logic execution can be crucial for optimizing performance and tailoring the algorithm's behavior to specific domain needs.

3.6. Virtualization

Beyond using QEMU as a implementation platform, it plays a crucial role in operating system virtualization due to its unique capabilities with KVM feature. Unlike traditional hypervisors that rely on hardware virtualization extensions, QEMU employs software emulation to simulate various hardware components like processors, memory, and devices. This allows it to run guest operating systems on a host system even if the underlying hardware architectures differ significantly. KVM unlocks near-native performance for virtual machines on compatible Linux hosts. By working with the KVM module, the system shifts from emulation to leverage the processor's built-in virtualization features. This allows virtual machines with the same architecture as the host to run significantly faster.

This software-based approach offers several advantages. Firstly, QEMU boasts exceptional platform independence, functioning seamlessly across diverse operating systems like Windows, Linux, and macOS. Secondly, its extensive hardware emulation capabilities enable it to run a wide range of guest operating systems, including older or less common ones, fostering broader compatibility. These qualities make QEMU a valuable tool for tasks like testing software compatibility across different OS environments, developing and debugging embedded systems, and exploring legacy operating systems. There-

fore leveraging QEMU's hardware virtualization capabilities through its dynamic binary translation and device emulation, we employed it as the virtualization platform for our research platform. This choice ensured compatibility with various guest operating systems and hardware architectures while enabling hypervisor support for efficient virtual machine management.

3.6.1. QMP

We use the QEMU Machine Protocol (QMP) to communicate with QEMU to demonstrate virtualization capabilities of the proposed operating system. The QMP is a JSON-based communication protocol that allows applications to control and query QEMU instances at the machine level. It acts as a control panel for QEMU instances, offering programmatic ways to manage their entire lifecycle: starting, stopping, pausing, and configuring them. It also functions as an information hub and allows to query QEMU about the guest operating system, its current state, and the details of its hardware components as shown in Figure 3.15.

Additionally, QMP acts as a bridge for sending commands directly to the guest OS using the QEMU guest agent (QGA). It boasts several benefits are; its JSON format offers efficient, lightweight communication, minimizing data transfer while remaining human and machine-readable for easy parsing. It also supports receiving notifications from QEMU about events within the virtual machine, and being cross-platform ensures compatibility across different operating systems where QEMU runs. QMP offers various use cases, including writing automation scripts for managing QEMU instances in testing, integrating with other tools for cloud deployment and development workflows, remotely monitoring and controlling virtual machines, and even debugging guest operating systems through commands sent via QGA.

QMP offers a client-server architecture where applications connect to the QMP server running within QEMU. Communication happens through JSON-formatted commands and responses, with clients and servers agreeing on capabilities beforehand. QMP offers a client-server architecture where applications connect to the QMP server running within QEMU. Communication happens through JSON-formatted commands and responses, with clients and servers agreeing on capabilities beforehand. Additionally, to be able

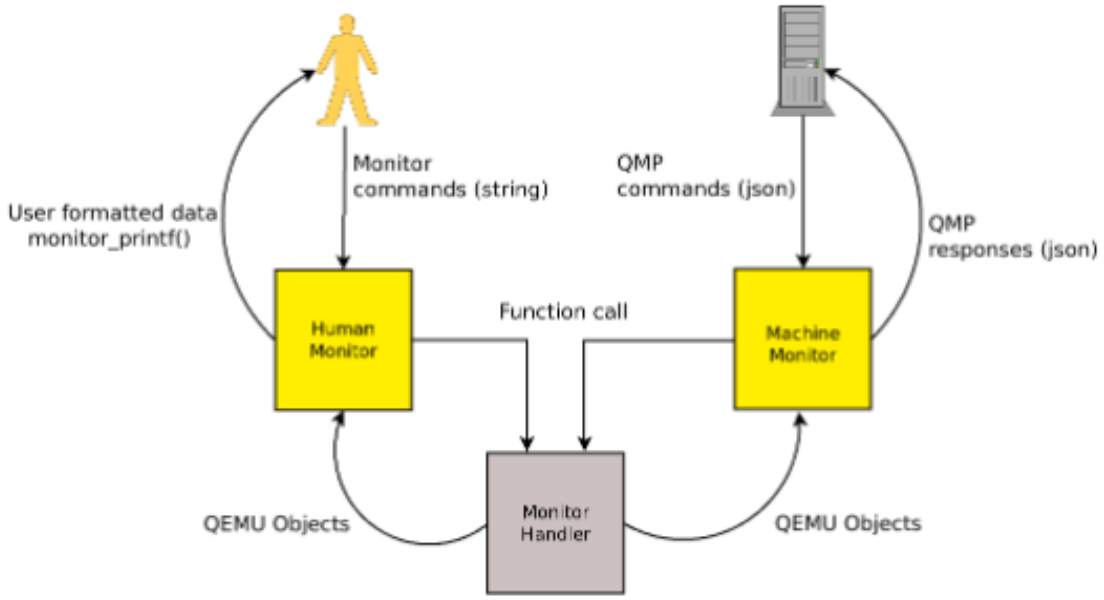

Figure 3.15. QMP Protocol - Monitor Objects (Capitulino 2010)

to implement more advanced guest management, the QEMU Guest Agent API requires the setup of an agent application within the guest operating system for communication. However, since QMP is more advantageous for the PusOS, there is no need to use QGA within the system. The following subsections describe the virtualization capabilities demonstrated using QMP.

3.6.2. Fault Tolerance Method

To demonstrate the potential of system virtualization to achieve fault tolerance capability, we use the external tool we developed together with the QMP for monitoring and control. Although QEMU lacks native fault tolerance capabilities, with the approach we developed, service continuity is ensured by recovering the workload in case of failure. Specific details of the applied methodology are given below:

1. Setup: To enable fault tolerance in QEMU instances, we configure them for communication by enabling QMP access with specific QEMU start parameters, Pus node is prepared for checkpointing or replication according to the controller tool and defined communication channels between the QEMU instances.

2. Monitoring and heartbeats: To enable communication between QEMU instances and a controller tool, each QEMU instance will periodically send heartbeat messages through QMP to the controller. The controller monitors these messages and if a heartbeat is not received within a set time frame, it identifies the corresponding QEMU instance as having failed.

3. Failure detection and recovery: Upon detecting a failing primary instance, the controller tool leverages QMP to initiate recovery actions: if replication is used, the replicated VM state is migrated to a healthy backup, and if checkpointing is used, the latest checkpoint is restored on the backup using QMP commands. Subsequently, the backup instance boots up, assuming the workload of the failed primary instance.

4. Availability and Integrity: In order to ensure high availability and data integrity, consistency testing is performed within the application itself within the operating system to isolate faulty instances. Additionally, the most appropriate heartbeat configuration is determined by regular fail-over tests.

3.6.3. Migration Method

Another step in demonstrating virtualization capabilities is the migration process. Since we use QEMU as a virtualization backend, we benefit from the QMP and QEMU's VM migration capabilities. Below we detail the methodology applied for the migration process and show how it is done:

1. Setup:

   (a) In preparation for migration, we enabled QMP access on both the source and destination VMs, and ensured they are connected through the network for data transfer, and configured shared storage for VM images if we are moving them to a different host.

   (b) To initiate migration, we sent a "migrate" command via a QMP message to the source virtual machine, specifying the destination virtual machine's ID or URI within the command itself.

2. Pre-migration on source VM: In preparation for migration, the source virtual machine pauses the guest operating system, then concurrently begins backing up its memory and disk data while establishing a connection to the destination VM via TCP.

3. Data transfer and state synchronization: The source VM transmits memory pages and disk image chunks to the destination VM through the established TCP stream and then progress and status updates are communicated via QMP messages.

4. Post-migration on destination VM: Upon receiving the data stream, the destination VM performs a three-step process: first, it saves the transferred memory and disk image to its own storage, then it loads the recovered state of the VM, and finally, it resumes the execution of the guest operating system using QMP commands.

5. Finalization: Following a successful migration, we ensured proper resource management by cleaning up temporary resources like migration snapshots on the source VM. Additionally, powering off the VM or preparing it for a subsequent migration are in optional state.

3.7. Management and Monitoring

Remote manageability is crucial for Fog computing systems, whether they run directly on the hardware or via virtualization. The specific methods for remote access will vary depending on the deployment scenario. A Pus node can be managed using both RPC and serial access. RPC methods enable system functions like reboot, backup, and migration. These methods are implemented using Protocol Buffers, a machine communication protocol, for fast command transmission. Additionally, fog nodes can utilize RPC for collaboration. The virtualized environment presents different management and access libraries contingent upon the underlying virtualization technology deployed. In PusOS, this management is provided via Linux KVM. For this purpose, we explained QMP API usage as mentioned in section 3.6. to communicate over the network.

Designed to support a distributed environment consisting of multiple interconnected nodes, our operating system presented a challenge in terms of manageability and monitoring. To address this, we proactively developed the Pus controller tool (PusCtl). PusCtl

acts as the coordinator for our cluster, simplifying the administration and oversight of a network of Pus nodes. Beyond core management functionalities, PusCtl offers a web-based interface that empowers clients to conduct crucial tests. These tests include simulating fault tolerance scenarios and application migration processes. By combining these capabilities, PusCtl ensures the smooth operation and flexibility of our complex distributed system, while also performing the task of monitoring Pus nodes.

Due to its distributed nature and potential blind spots, Fog computing has some challenges for monitoring compared to Cloud systems. However, effective monitoring is crucial for proper regulation and management. Therefore, it is essential to collect and orchestrate up-to-date information on the status and running tasks of fog nodes and communication links. Actions such as offloading services to more powerful nodes and optimizing placement based on historical data guarantee service level agreements (SLAs), ensure smooth operation (Costa et al. 2020). Besides the heterogeneity of monitored nodes, other situations related to frequency, topology, and communication patterns can also be observed. It may be desirable to monitor the health status of the system running on the fog node and its performance of the assigned tasks. To provide decision-making with actionable data, PusOS offers real-time system monitoring through multiple channels. Nodes, supporting TCP/IP protocols, allow access via HTTP for live system status and process observation, as well as through various APIs for detailed system information. These data channels can also be seamlessly integrate with log servers for comprehensive record keeping.

Lastly, the Figure 3.16 also shows the position of PusOS in the continuum from the Cloud to the IoT. Accordingly, while smart devices communicate with the network layer of the operating system through IoT communication protocols such as MQTT, CoAP and HTTP, the storage unit of the system stores the incoming data (LittleFS) and after performing the necessary operations in the computation unit (Dynamic Runtime), saves them as analysis outputs and transmits them to the Cloud (HTTP API). In addition, these fog nodes which places the PusOS can communicate among themselves via customized data serialized protocol (Protobuf). In case of any fault or application migration, virtualization capabilities come into play by using QMP.

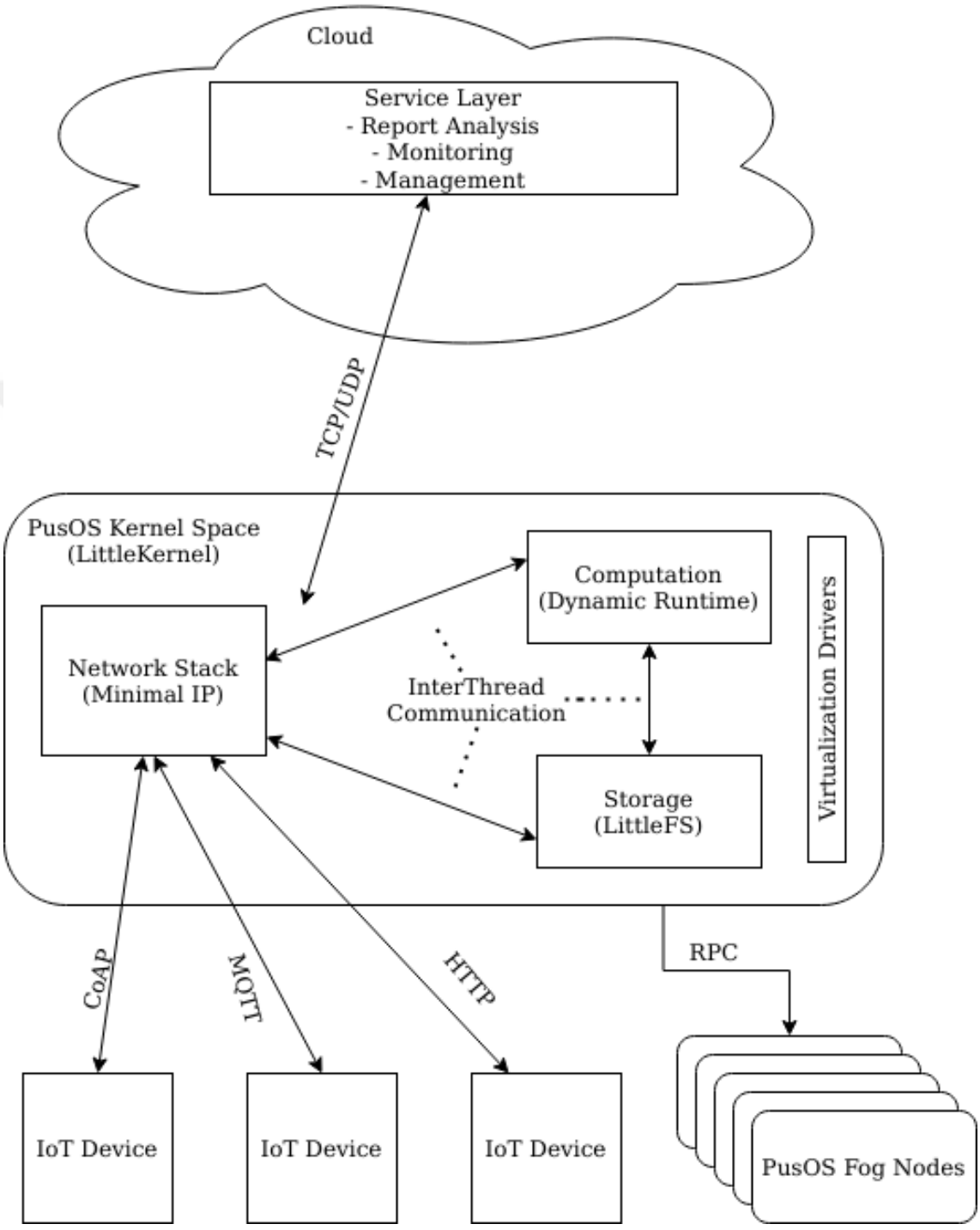

Figure 3.16. Cloud-to-Things Continuum with PusOS Internals

4. RESULTS AND DISCUSSION

In this section, experiments and obtained results showing the supervisor and virtualization capabilities of PusOS in terms of Fog computing are presented. Before detailed explanations of the experiments, the PusCtl management/monitoring tool is explained which we mentioned in Section 3.7. and used in performing the experiments. Detailed explanations are given using examples and figures on how to manage a fog node cluster established by using this tool.

4.1. Pus Controller Tool

Previously, we discussed the challenges of managing and monitoring Pus cluster, especially when the operating system creates links across multiple nodes. While basic management is possible through serial way by using terminal applications but difficulties arise as the number of nodes increases. This is where PusCtl comes in which we developed using the Go programming language and it communicates with nodes via UDP and provides a user-friendly web user interface through HTTP.

The Figure 4.17 showcases a screenshot of the PusCtl web user interface after a cluster is established. The area highlighted in green serves as the monitoring zone. It presents information about each node in a tabular format, including respectively:

1. Pus Node ID (ID)
2. Virtual IPv4 address of a node (Local IP)
3. QEMU communication port (Port)
4. Host systems's virtual IPv4 address (Host IP)
5. Time of the last heartbeat message sent by a node (Heartbeat)
6. Status of the machine running the node (Machine Status)

The red rectangular area below the management zone offers management functions. It includes respectively:

1. A *IP list* listbox to display IP information of nodes

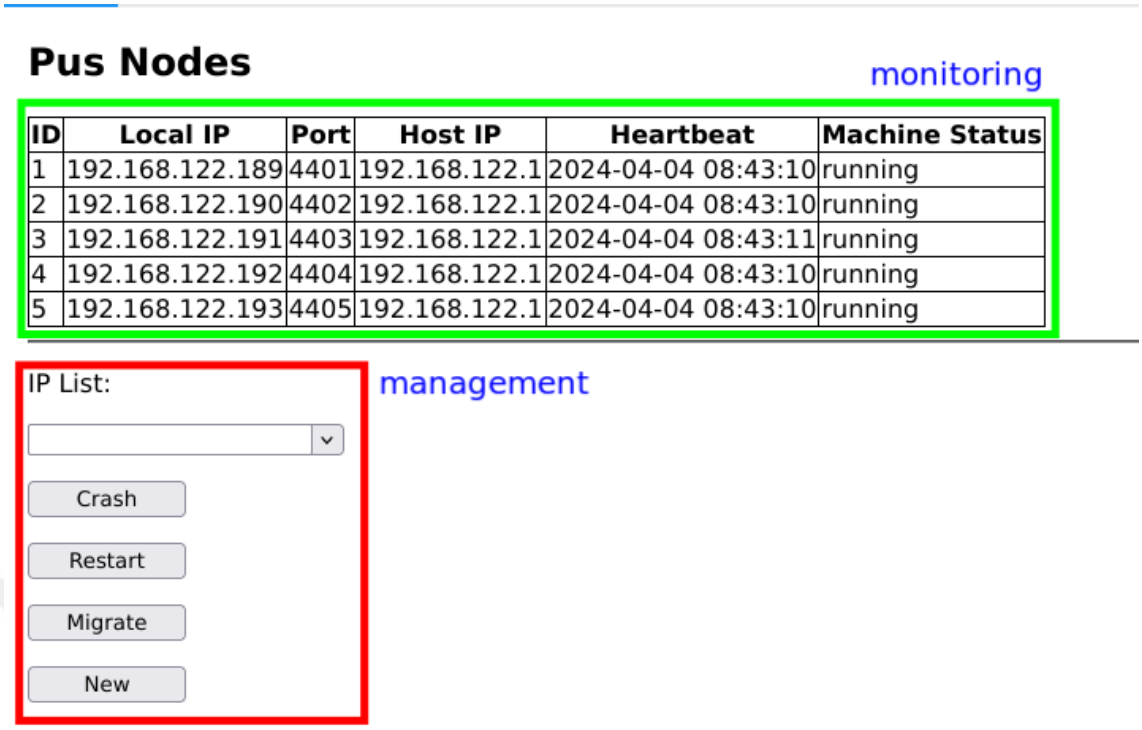

Figure 4.17. PusCtl Web User Interface - Cluster

2. A *Crash* button to send a crash signal to selected node
3. A *Restart* button to reboot a node
4. A *Migrate* button to start a migration process between nodes
5. A *New* button to add a new node to the cluster

Adding a new node to the cluster via web user interface is depicted in Figure 4.18. The steps are as follows:

1. IP Selection: Choose an IP address for the new node from the provided IP list. The list might display consecutive IPs based on the current cluster configuration.
2. Adding the Node: Click the *New* button to initiate the node addition process.
3. Node Status Change: After adding the new node, observe its status in the monitoring area. The status should change to "inmigrate" which indicates that the QEMU

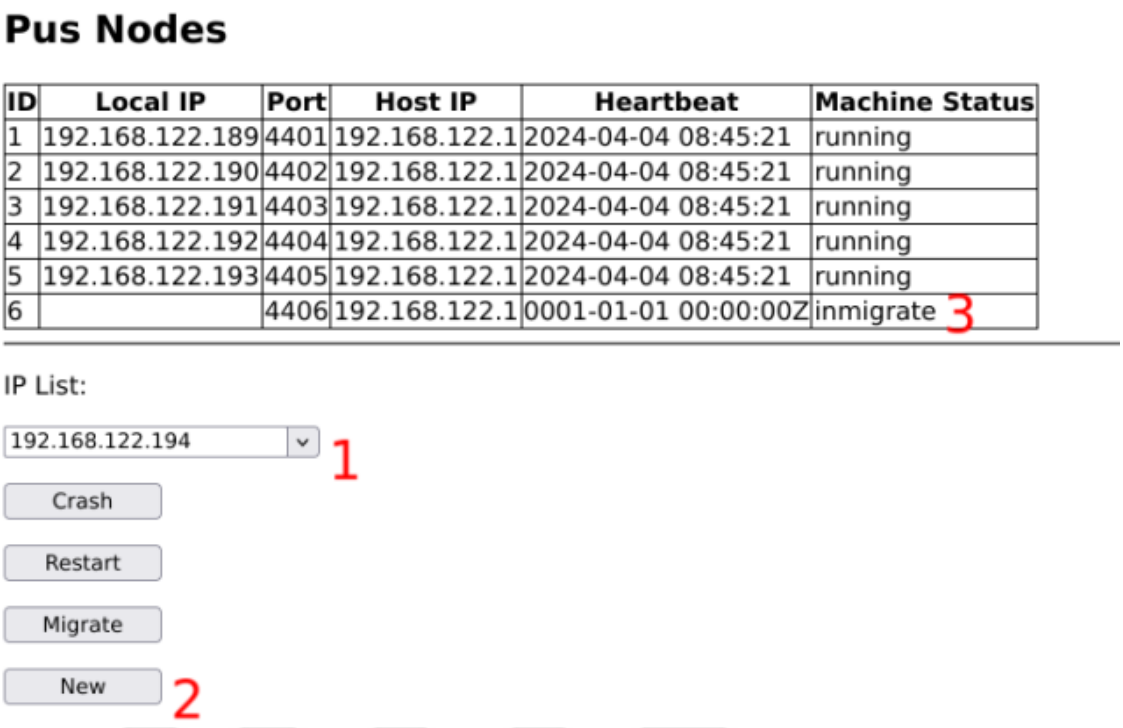

Figure 4.18. PusCtl Web User Interface - New Node

instance can get a migration from other instance, is a specific term used in the context of the QEMU virtualization.

After adding a new node to cluster then we can **migrate of a node** to newly added node , is depicted in Figure 4.19. The steps are as follows:

1. Node Selection: Select the node to migrate from the provided IP list.
2. Migration Initiation: Click the *Migrate* button to start the migration process.
3. Data Transfer: Observe that information from the selected node (ID 1) is transferred to the newly added node. The migrated node's status will also change to "running".
4. Post-Migration State: Once the migration is complete, the migrated node will transition to a "postmigrate" state, indicating successful migration. The local IP value is also copied to the new machine during this process then will empty at the old one's field.

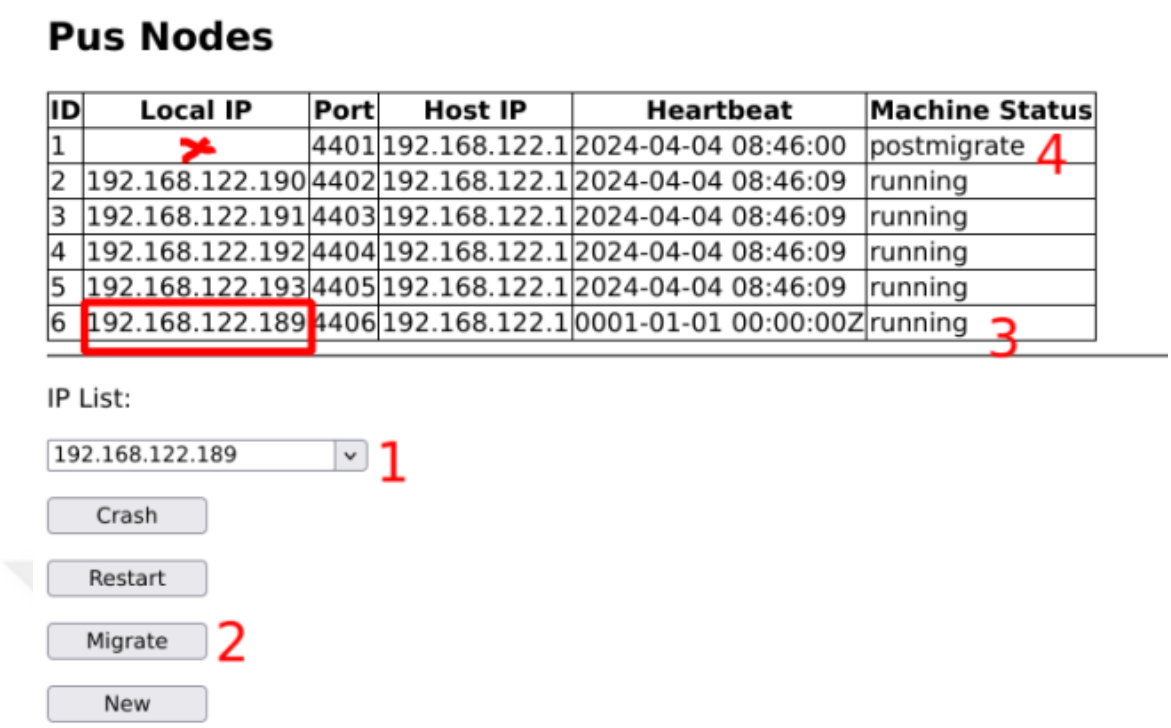

Figure 4.19. PusCtl Web User Interface - Node Migration

4.2. Experiments

Within the scope of PusOS's experimental studies, the following test scenarios were carried out and explained in detail in the relevant subsections;

- Data Collection
- Storage Management
- Fault Tolerance
- Application Migration

The following subsections (Data Collection and Storage Management) include tests to evaluate the other components of PusOS, especially their ability to run only in the kernel space. All tests were performed by using QEMU virtual machine. Fedora Linux distribution[1], which contains many development tools, was chosen as the competitor system

[1] version: Fedora-Developer-38-20230519.n.0.SiFive.Unmatched.and.QEMU

for comparison. The distribution's QEMU image was obtained and tested using the same QEMU machine parameters as the PusOS. Since the PusOS consists of only one ELF file, it is given to the QEMU virtual machine as a kernel parameter. Both systems were made ready with the same network and disk configuration as using the test system. The table 4.5 below lists the features of the PusOS and Linux system selected to be used in the tests.

Table 4.5. Comparison of test systems

Feature	**PusOS**	**Fedora Linux**
Kernel	LittleKernel	Linux
Kernel Type	nano	monolithic
Process Structure	kernel thread	userspace process
Filesystem	LittleFS	Ext4
Minimum RAM amount(MB)	40	150
Deployment Method	only kernel file (ELF)	disk image file (IMG)

4.2.1. Data Collection

In this evaluation, we designed a mechanism that generates 4-byte float type virtual temperature sensor data. A client application where places in the test systems, pulls the temperature values by using the CoAP protocol. With the application we developed in the PusOS, the sensor data was captured. The application pulls the data in a loop with the count parameter it receives which shown in Algorithm 1.

A client application has also been implemented on the Linux system side for comparison. A Linux application is a command-line application that reads sensor data with as many cycles as the number of inputs it receives. Linux and PusOS client applications are implemented by using same algorithm. Technically, in both applications, a thread has been created to accomplish each sensor data reading and serial printing tasks. Threads are run with the create, resume, and join function process. Therefore, the sensor data were taken sequentially. It was done with 10ms, 50ms and 100ms interval times after each read/print tasks as operation. In the tables (Table 4.6, Table 4.7, Table 4.8) below, the number of operation, how long it takes for applications to complete this operation and the

Algorithm 1 Sensor Communication Test

Require: $count > 0$ {message count number as count}

Require: $interval > 0$ {delay time for the next operation as interval}

for $i = 0$ to $count$ **do**

{task-1 read temperature value from sensor}

$tempVal = readTemperature()$

{task-2 print temperature value to serial output}

$printSerial(tempVal)$

{delay for the next operation }

$delay(interval)$

end for

message average processing time (MAPT) is given. The tests were repeated 10 times and the average values of the operation times were taken.

Table 4.6. 10ms interval sensor communication processing speed

	PusOS		Linux	
Message Count	**Total(s)**	**MAPT(ms)**	**Total(s)**	**MAPT(ms)**
1000	10.8	0.8	13.52	3.52
2000	21.74	0.87	28,71	3.69
3000	32.6	0.86	41.05	3.68

According to results (Table 4.6, Table 4.7, Table 4.8), the client application running on PusOS was completed in a shorter time than the counterpart application running on Linux, since it runs only in the kernel space. As the number of sensor values pulled increased, the difference between processing the tasks times became more evident. While PusOS captures and prints a message at different delay times between 0.7-0.9 ms, this time is recorded as 3.4-4 ms in the Linux application. Since there are no system calls in the PusOS, changing delay times did not make a difference for the application in PusOS. However, as the latency decreases on the Linux side, the message processing time also

Table 4.7. 50ms interval sensor communication processing speed

	PusOS		Linux	
Message Count	**Total(s)**	**MAPT(ms)**	**Total(s)**	**MAPT(ms)**
1000	50.7	0.7	53.40	3.4
2000	101.37	0.69	106.82	3.41
3000	152.06	0.68	160.26	3.42

Table 4.8. 100ms interval sensor communication processing speed

	PusOS		Linux	
Message Count	**Total(s)**	**MAPT(ms)**	**Total(s)**	**MAPT(ms)**
1000	100.7	0.7	103.95	3.95
2000	201.44	0.72	207.76	3.88
3000	302.2	0.73	311.7	3.9

decreases because of Linux kernel tracks pages have been recently accessed are kept on the "active" list, with just-accessed pages put at the head of the list. But eventually, in terms of average value between 10 and 100 ms delay times, PusOS outperforms operates up to 4.86 times faster than Linux.

4.2.2. Data Storage

This section covers the measurement of I/O work done using the file system. Storing to a file has been added as an additional job to the values received by sensor reading and written to the serial. While PusOS stores the sensor data by using its LittleFS module, Linux has saved it to a file by using Ext4 file system as a comparison. Both systems used a Qcow2 formatted QEMU block disk at first.

While each sensor reading is made, each record has been tested to perform a file open/close function. For each read value, the file was opened (fileOpen) and the value was written (fileAppend) and lastly the file was closed (fileClose) functions were applied under *Task-3*, as shown in Algorithm 2.

In this test, these processes were performed sequentially with a interval of 10, 50 and

Algorithm 2 I/O Operation Test

Require: $count > 0$ {message count number as count}

Require: $interval > 0$ {delay time for the next operation as interval}

for $i = 0$ to $count$ **do**

{task-1 read temperature value from sensor}

$tempVal = readTemperature()$

{task-2 print temperature value to serial output}

$printSerial(tempVal)$

{task-3 store temperature value to disk}

$f = fileOpen("records.txt")$

$fileAppend(f, tempVal)$

$fileClose(f)$

{delay for the next operation }

$delay(interval)$

end for

100 ms. In this way, it is aimed to compare the I/O operation speeds of PusOS with its Linux competitor. Comparative values are given in the tables (4.9, 4.10, 4.11) below.

Table 4.9. 10ms interval I/O operation processing speed

	PusOS		Linux	
Message Count	**Total(s)**	**MAPT(ms)**	**Total(s)**	**MAPT(ms)**
1000	25.38	15.38	14.05	4.05
2000	48.03	14.01	28.24	4.12
3000	75.82	15.27	42,54	4.18

In the tables (4.10, 4.11), very fast time values were obtained with LittleFS by using 50ms and 100ms interval times. Except 10ms, PusOS accomplishes the data storage tasks just 0.1-0.5 ms more than data collection tests per message. This means that the latency time spent on opening, writing and closing files is almost zero.

Table 4.10. 50ms interval I/O operation processing speed

	PusOS		Linux	
Message Count	**Total(s)**	**MAPT(ms)**	**Total(s)**	**MAPT(ms)**
1000	50.74	0.74	54.07	4.07
2000	101.42	0.71	107.84	3.92
3000	152.19	0.73	161.97	3.99

Table 4.11. 100ms interval I/O operation processing speed

	PusOS		Linux	
Message Count	**Total(s)**	**MAPT(ms)**	**Total(s)**	**MAPT(ms)**
1000	100.73	0.73	105,2	5.2
2000	201.47	0.74	210,7	5.35
3000	302.32	0.77	315,24	5.08

With 10ms, it is seen that Linux outperforms operates up to 3.6 times faster than PusOS. This is because latency values nearly 10ms are reported as problematic thresholds for LittleFS design. Despite this, as a result of our experiments, above 10ms values, PusOS still exceeds the system call-free performance. In terms of average value between 50 and 100 ms delay times, PusOS operates up to 5.5 times faster than Linux competitor.

4.2.3. Fault Tolerance

The experiment performed in this section is based on the scenario that if the PusOS encountered a crash while it is running, the system will recover itself and continue where it left off as shown in Figure 4.20. For this purpose, a simple application that calculates the Fibonacci series after the system boots has been launched. While the test application is printing the numbers of the series to the serial interface, the system is intentionally crashed via the remote management API. After the system crashes, it has been shown that it can restore itself from the last successful operating point and continue its operation from where it left off. The application that prints the Fibonacci series numbers to the serial at 1-second intervals. We run 10 virtual machines which each machine as fog node separately

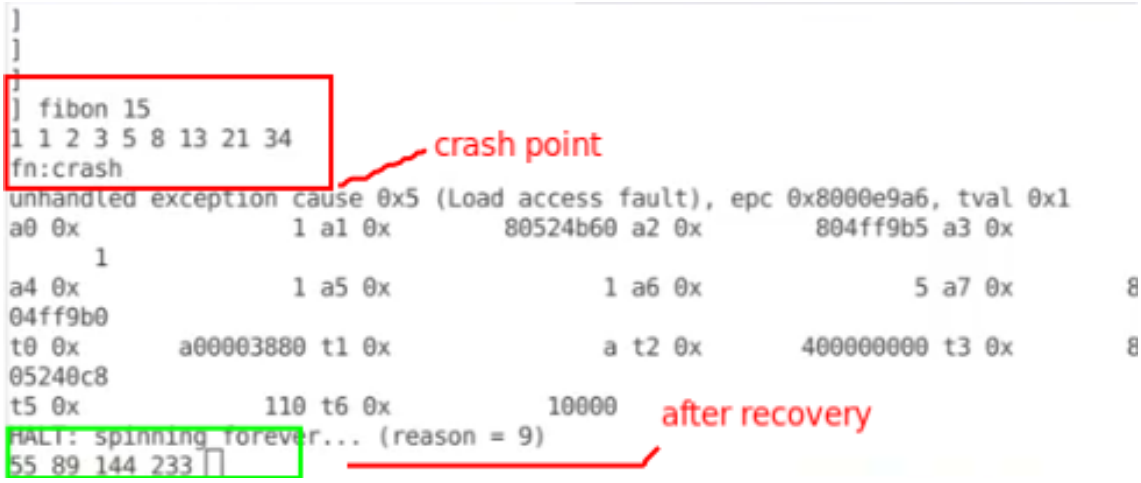

Figure 4.20. PusOS Crash Recovery Demonstration

runs PusOS with Fibonacci series application inside it. We have four test scenarios:

1. Failing 1 VM randomly,
2. Failing 3 VMs randomly.
3. Failing 5 VMs randomly,
4. Failing 10 (all) VMs

Each scenario repeats as five times to agreed on the average time values of the test. In the setup recovery environment, the system sent a heartbeat message every 1 second. Also after each Fibonacci series number is written, a snapshot of the system is taken with a 1-second delay and saved to the block disk on which the system is running. The recovery of process is also carried out via this snapshot. Saving and restoring the current state of the system is implemented by using QEMU's QMP API. The functioning of the fault-tolerance mechanism shown in detail in Figure 4.21

According to the results shown in table 4.12, the system continued the application from where it left off by restoring itself from the last successful execution point within a maximum 2sec loading delay. In addition, the 1-second delay while printing the numbers is also included in the recovery time. We are evaluating this time as jitter recovery time because when we send load signal to host machine of a crashed PusOS, the virtual machine loads the last successful snapshot and then system continues to run and sends the

Table 4.12. Fault recovery test

Fail Scenario	Avg Recovery Time(s)	Jitter Recovery(s)
1 VM	43	< 2
3 VM	34.2	< 2
5 VM	47.6	< 2
10 (All) VM	26.5	< 2

heartbeat message to heartbeat server (controller). Hereby jitter recovery time equals the difference of load time and first heartbeat message after loading.

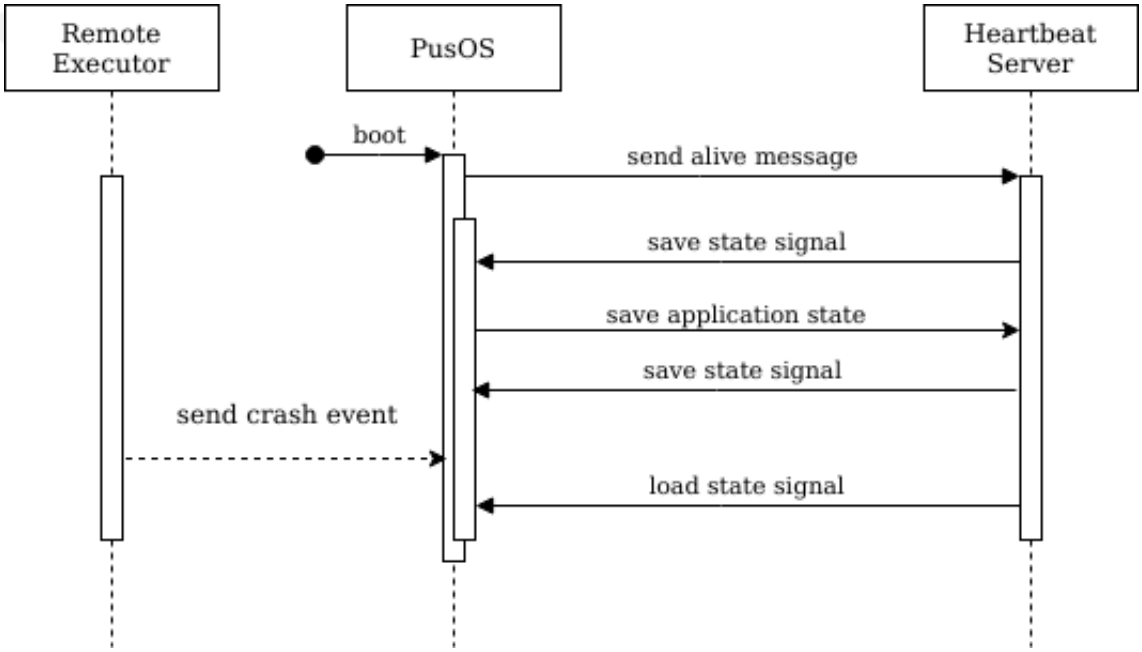

Figure 4.21. PusOS Fault Tolerance Mechanism

4.2.4. Application Migration

This experiment demonstrates the application migration capabilities of PusOS, based on the assumption that multiple Pus nodes are running on different host systems. As shown in Figure 4.22, a discovery service is defined so that nodes can get information about each other. After each Pus node boots, it registers itself to the *Discovery Server* and receives a current registration table response, including the status of other nodes. Using

this table, it plans application and system crash migrations. Then the remote executor intentionally sends a migrate signal to start a migration via remote management API. According to the Figure 4.22, Node-1 queries the current registration table from the *Discovery Server* and depending on the response, it migrates the application directly to the target host system where the Node-2 is running by using the QEMU API.

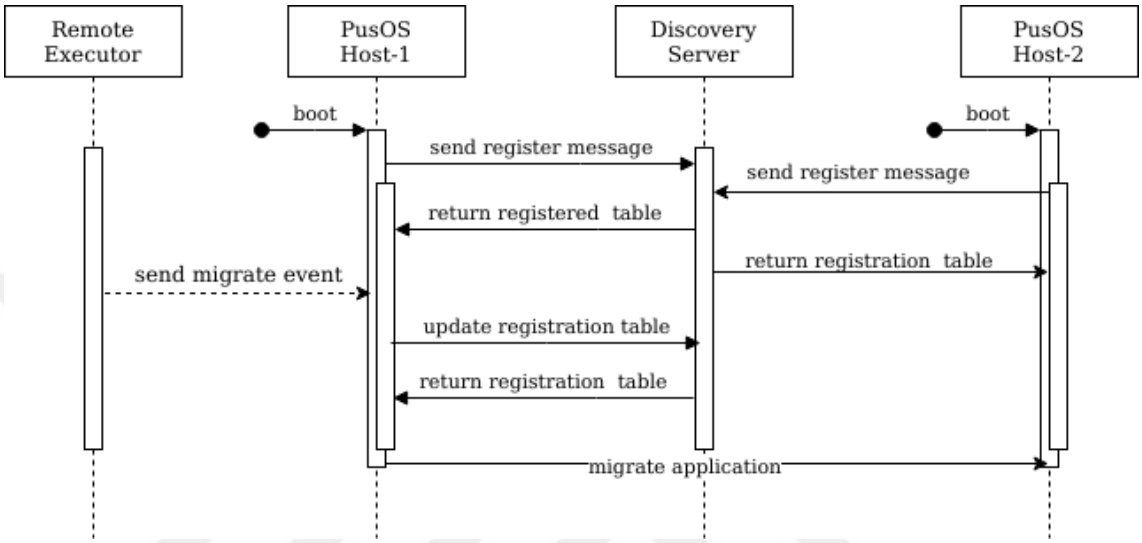

Figure 4.22. PusOS Application Migration Mechanism

In the test, a simple Fibonacci series application was used, similar to the fault-tolerance experiment. While this application was printing the numbers of the series to serial interface, the application which was running intentionally, was migrated to another host, allowing the application to continue where it left off on the new host. The application that prints the Fibonacci series numbers to the serial at 1-second intervals. The Table 4.13 shows the different migration delays at first column that means be migrated after delay, and second and third columns shows that non-migrated and migrated printing times respectively. Last column is difference from the t1 time of non-migrated and migrated t2 time of printing the number which is considered as jitter time of migration.

According to test, ten numbers of Fibonacci series take 9 second for printing in non-migrated state, with migration it takes 11 second at maximum, then we conclude that the system migrated successfully and continued to run where it left off under 2 seconds with 0.25 seconds average jitter time.

Table 4.13. Application migration test

Delay Time(s)	**Printing Time(s)**	**Printing+Migration(s)**	**Jitter Time(s)**
4	4.006	4.139	0.133
5	5.006	5.125	0.119
6	6.009	6.293	0.284
7	7.008	7.148	0.140

5. CONCLUSION

This thesis study introduces a novel operating system approach specifically tailored for Fog computing on RISC-V architecture. The proposed system operates entirely in kernel space, granting it direct access to hardware resources and enabling significant performance optimizations. The results obtained regarding the operating system aspect are as follows;

- **Kernel-Centric Design**: Because the operating system executes only within the kernel space, the context switch time between the user space and kernel space is eliminated. This elimination results in latency reduction and improves responsiveness. Such performance gain is critical in Fog computing applications that often necessitate real-time or near real-time processing, ideally below 10 milliseconds as expected from Fog computing platforms. The effectiveness of this approach has been demonstrated through data collection tests, achieving performance $4-5$ times faster than its Linux counterpart.

- **Effective File System**: We evaluated a file system usage performance designed specifically for embedded systems to meet Fog computing storage requirements. The file system underwent custom configuration updates and achieved file operations with near zero overhead during data storage tests. Based on to the results obtained, a delay of $0.1 - 0.5$ ms was recorded in addition to data collection time. This also confirms that working in the kernel space only improves file I/O significantly.

In line with the results we obtained in the experimental stage, the fault tolerance and application migration virtualization functions required for both clustering and Fog computing have been successfully verified. The observations are as follows;

- **Fault Tolerance**: We evaluated fog node/s resilience in a simulated cluster failure triggered by hardware faults or network disruptions. The results confirmed successful recovery of affected nodes by the cluster coordinator. All nodes resumed operation from the last checkpoint, ensuring minimal service disruption during repeated tests.

- **Application Migration**: The experiment focused on the ability to migrate applications running on PusOS to a different host within the cluster. This capability is crucial for load balancing and handling situations requiring sudden workload shifts. The relevant test successfully did migration between nodes, demonstrating seamless application migration without impacting functionality.

In conclusion, experiments have demonstrated PusOS's design and implementation effectiveness, proving its suitability as a Fog computing operating system. Developing the operating system on the RISC-V architecture offered a foundational background for our ongoing studies at the system level, enabling us to explore various user modes of RISC-V at the virtualization level. Moreover, the open-source nature of PusOS, coupled with its lack of licensing fees, facilitates accessibility for other researchers and developers, empowering them to experiment with, utilize, and contribute to the community.

For future work, the operating system, which is designed and implemented on the virtual machine in this study, will will have a demonstration on a realistic platform in future studies which is important in terms of using the operating system in real-life scenarios. In addition, the deployment process will be modernized to make the system more compact, and it will be possible to run domain-targeted applications more easily.

6. REFERENCES

Al-Haidari, F. and Alqahtani, E. 2020. Securing Communication between Fog Computing and IoT Using Constrained Application Protocol (CoAP): A Survey. Journal of Communications, (15): 14-30.

Ali, A. Constrained Application Protocol (CoAP) for the IoT. 2018. IoT Seminar, High Integrity System, Frankfurt University of Applied Science.

Alwakeel, A. 2021. An Overview of Fog Computing and Edge Computing Security and Privacy Issues. *Sensors*, (21): 8226.

Android Open Source Project. 2024. https://source.android.com/docs/security/features/trusty [Last access time: 01.04.2024].

Antonini, M. and Vecchio, M. and Antonelli, F. 2019. Fog Computing Architectures: A Reference for Practitioners. *IEEE Internet of Things Magazine*, (2): 19-25.

Asanović, K. and Patterson, D. A. 2014. Instruction sets should be free: The case for RISC-V. EECS Department. University of California, Berkeley, Tech. Rep. UCB/EECS, 146.

Ashraf, M. and Shiraz, M. and Abbasi, A. and Albahli, S. 2022. Distributed application execution in fog computing: A taxonomy, challenges and future directions. Journal of King Saud University - Computer and Information Sciences, (34): 3887-3909.

Bagchi, S. 2008. Nano-kernel: a dynamically reconfigurable kernel for WSN. *In 1st International ICST Conference on Mobile Wireless Middleware, Operating Systems and Applications*. 10.

Bayer, M. 2022. Securing and Hardening Embedded Linux Devices - case study based on NXP i.MX6 Platform. *International Conference on Future Internet of Things and Cloud (FiCloud)*, 181-189.

Benomar, Z. and Longo, F. and Merlino, G. and Puliafito, A. 2019. Enabling Container-Based Fog Computing with OpenStack. International Conference on Internet of

Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 1049-1056.

Bonomi, F. and Milito, R. and Zhu, J. and Addepalli, S. 2012. Fog Computing and Its Role in the Internet of Things. *Association for Computing Machinery*, 13-16.

Bove, D. 2022. Secure Services for Standard RISC-V Architectures. Proceedings of the 17th International Conference on Availability, Reliability and Security, ACM, (23): 7.

Capitulino, L. 2010. https://www.linux-kvm.org/images/1/17/2010-forum-qmp-status-talk.pp.pdf [Last access time: 01.04.2024].

Cheng, B. and Solmaz, G. and Cirillo, F. and Kovacs, E. and Terasawa, K. and Kitazawa, A. 2018. FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities. *IEEE Internet of Things Magazine*, (5): 696-707.

Cheng, B. and Fuerst, J. and Solmaz, G. and Sanada, T. 2019. Fog Function: Serverless Fog Computing for Data Intensive IoT Services. *IEEE International Conference on Services Computing (SCC)*, 28-35.

Choi, N. and Kim, D. and Lee, S. and Yi, Y. 2017. A Fog Operating System for User-Oriented IoT Services: Challenges and Research Directions. IEEE Communications Magazine, (55): 44-51.

Costa, B. and Bachiega, B. and Rebouças, R. and Rosa, M. and Araujo, A. 2020. Monitoring fog computing: A review, taxonomy and open challenges. Computer Networks 1389-1286.

Coutinho, A. and Greve, F. and Prazeres, C. and Cardoso, J. 2018. Fogbed: A Rapid-Prototyping Emulation Environment for Fog Computing. *IEEE International Conference on Communications (ICC)*, 1-7.

Cui, E. and Li, T. and Wei, Q. 2023. RISC-V Instruction Set Architecture Extensions: A Survey. *IEEE Access*, (11): 24696-24711.

Dolui, K. and Datta, S. 2017. Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing. *Global Internet of Things Summit (GIoTS)*, 1-6.

Edge Computing Consortium. 2018. Edge Computing Reference Architecture 2.0. http://en.ecconsortium.net/Uploads/file/20180328/1522232376480704.pdf [Last access time: 20.11.2018].

Fetzer, C. 2016. Building Critical Applications Using Microservices. *IEEE Security & Privacy*, 6 (14): 86-89.

Forti, S. and Pagiaro, A. and Brogi, A. 2019. Simulating FogDirector Application Management. *Simulation Modelling Practice and Theory*, 102021.

Gaur, P. and Tahiliani, M.P. 2015. Operating Systems for IoT Devices: A Critical Survey. *IEEE Region 10 Symposium*, 33-36.

Geiselbrecht T. 2010. Little Kernel. https://github.com/littlekernel/lk [Last access time: 01.04.2024].

Gien, M. and Grob, L. 1992. Micro-kernel based operating systems: Moving UNIX onto modern system architectures. *Proceedings of the UniForum*, 19 (1): 43-55.

Haster C. 2017. http://mu-script.org [Last access time: 01.04.2024].

Haouari, F. and Faraj, R. and AlJa'am, J. 2018. Fog Computing Potentials, Applications, and Challenges. *International Conference on Computer and Applications (ICCA)*, (8): 399-406.

Heiser, G. and Elphinstone, K. and Vochteloo, J. and Russell, S. and Liedtke, J. 1998. The mungi single-address-space operating system. *Softw: Pract. Exper.*, (28): 901-928.

Heng, S. and Chen, N. and Ralph, D. 2015. Combining Mobile and Fog Computing: Using CoAP to Link Mobile Device Clouds with Fog Computing. *IEEE International Conference on Data Science and Data Intensive Systems*, 564-571.

Howard, H. and Mortier, R. 2020. Paxos vs Raft: have we reached consensus on distributed consensus?. Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, ACM, (8): 9.

Rayes, IEEE 1934. 2018. IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing. *IEEE Std 1934*, 1-176.

Industry IoT Consortium. 2017. OpenFog Reference Architecture. https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf [Last access time: 01.04.2024].

Ince M.N. and Günay, M. Ledet, J. 2022. Lightweight distributed computing framework for orchestrating high performance computing and big data. Turkish Journal of Electrical Engineering and Computer Sciences, 1571-1585.

Kaiser R. and Wagner S. 2015. Evolution of the PikeOS Microkernel. IEEE Access.

Li, C. and Xue, Y. and Wang, J. and Zhang, W. and Li, T. 2018. Edge-Oriented Computing Paradigms: A Survey on Architecture Design and System Management. *Association for Computing Machinery Computing Survey*, (51): 39.

Li, J. and Jin, J. and Yuan, D. and Zhang, H. 2018. Virtual Fog: A Virtualization Enabled Fog Computing Framework for Internet of Things. IEEE Internet of Things Journal, (5): 121-131.

Lumpp, F. and Barchi, F. and Acquaviva, A. and Bombieri, N. 2023. On the Containerization and Orchestration of RISC-V architectures for Edge-Cloud computing. Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum, ACM, 21-28.

Masmano, M. and Ripoll, I. and Crespo, Alfons. 2005. An overview of the XtratuM nanokernel. Alpha Press. 01.

McBrewster, J. and Miller, F.P and Vandome, A.F. 2009. Kernel - computing: Monolithic kernel, Microkernel, Hybrid kernel, Exokernel, History of operating systems, Time-sharing, Unix, History of Mac OS, AmigaOS, History of Microsoft Windows. Alpha Press. 120.

Miao, H. 2011. Analysis of practicality and performance evaluation for monolithic kernel and micro-kernel operating systems. *International Journal of Engineering (IJE* 5(4): 277.

Minghao, K. and Chyang, K.Y. and Karuppiah, E.K. 2007. Performance Analysis and Optimization of User Space versus Kernel Space Network Application. *5th Student Conference on Research and Development*, 1-6, Selangor, Malaysia.

Núñez-Prieto, R. and Castells-Rufas, D. and Terés-Terés, L. 2023. RisCO2: Implementation and Performance Evaluation of RISC-V Processors for Low-Power CO2 Concentration Sensing. Micromachines, 14.

NVIDIA. 2024. https://trustedfirmware-a.readthedocs.io/en/latest/components/spd/tlk-dispatcher.html [Last access time: 01.04.2024].

Ongaro, D. and Ousterhout, J. In search of an understandable consensus algorithm. 2014. USENIX ATC'14, USENIX Association, 305–320.

OpenFog consortium. 2018. IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing. *IEEE Std*, 1-176.

Pan, L. and Tu, G. and Liu, S. and Cai, Z. and Xiong, X. 2021. A Lightweight AES Coprocessor Based on RISC-V Custom Instructions. *Security and Communication Networks*, 13.

Pundkar, S. R. and Karmakar, S. P. and Mishra, S. K. and Singh, S. and Vrind, T. 2022. Energy Aware Dynamic Load Balancer for Embedded Multi-core Systems. *35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID)*, 56-61.

Pop, P. and Zarrin, B. and Barzegaran, M. and Schulte, S. and Punnekkat, S. and Ruh, J. and Steiner, W. 2021. The FORA Fog Computing Platform for Industrial IoT. Information Systems, (98): 101727.

Protocol Buffers. 2024. https://protobuf.dev [Last access time: 01.04.2024].

QMP Reference Manual. 2024. https://qemu-project.gitlab.io/qemu/interop/qemu-qmp-ref.html [Last access time: 01.04.2024].

Rayes, A. and Salam, S. 2017. Cloudlets: Fog computing defining. In: Internet of Things from hype to reality. *Springer International Publishing, Cham*, 139–164.

Restivo, J. 2020. A zero kernel operating system: Rethinking microkernel design by leveraging tagged architectures and memory-safe languages. Doctoral dissertation, Massachusetts Institute of Technology, 22.

Rosario, V. and Pisani, F. and Gomes, A. and Borin, E. 2018. Fog-Assisted Translation: Towards Efficient Software Emulation on Heterogeneous IoT Devices. *International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 1268-1277.

Sá, B. and Martins, J. and Pinto, S. 2021. A First Look at RISC-V Virtualization From an Embedded Systems Perspective. IEEE Transactions on Computers, 71, 2177-2190.

Saha, A. and Udava, R. and Bidari, M. and Prasad, M. and Raju, V. and Vrind, T. 2021. TraFic — A Systematic Low Overhead Code Coverage Tool for Embedded Systems. *IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 1-6.

Serrano, R. and Sarmiento, M. and Duran, C. and Nguyen, K. and Hoang, T. and Ishibashi, K. and Pham, C. 2021. A Low-Power Low-Area SoC based in RISC-V Processor for IoT Applications. 18th International SoC Design Conference (ISOCC), 375-376.

Singh, S.P. and Nayyar, A. and Kumar R. and Sharma A. 2019. Fog computing: from architecture to edge computing and big data processing. *Journal of Supercomputer*, (75): 2070-2105.

Stankov, I. and Spasov, G. 2006. Discussion of microkernel and monolithic kernel approaches. *In International Scientific Conference Computer Science*.

Tseng, F. and Tsai, M. and Tseng, C. and Yang, Y. and Liu, C. and Chou, L. 2018. A Lightweight Autoscaling Mechanism for Fog Computing in Industrial Applications. IEEE Transactions on Industrial Informatics, (14): 4529-4537.

Verbelen, T. and Simoens, P. and De Turck, F. and Dhoedt B. 2012. Cloudlets: Bringing the cloud to the mobile user. *Proceedings of the third ACM workshop on Mobile cloud computing and services*, 29-36.

Waterman, A. and Lee, Y. and Patterson, D. and Asanovi K. 2014. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. https://api.semanticscholar.org/CorpusID:60142097 [Last access date: 01.04.2024].

Wiggins, A. and Heiser, G. 2000. Fast address-space switching on the StrongARM SA-1100 processor. *Computer Architecture Conference, 2000. ACAC 2000. 5th Australasian*, 97-104.

Yang, S. 2017. IoT Stream Processing and Analytics in the Fog. *IEEE Communications Magazine*, 55 (8): 21-27.

Yousefpour, A. and Fung, C. and Nguyen, T. and Kadiyala, K. and Jalali, F. and Niakanlahiji, A. and Kong, J. and Jue, JP. 2019. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, (98): 289-330.

CURRICULUM VITAE

MUHAMMED NUMAN İNCE

EDUCATION

Doctor of Philosophy	Akdeniz University
2020-2024	Institute of Natural and Applied Sciences,
	Department of Computer Engineering, Antalya
Master of Science	Akdeniz University
2018-2020	Institute of Natural and Applied Sciences,
	Department of Computer Engineering, Antalya

WORK EXPERIENCE

Lecturer	Alanya Alaaddin Keykubat University
2021-continues	Distance Education Application and Research Center, Antalya

PUBLICATIONS

Articles Published in International Refereed Journals

1- Michail, S., Ledet, J. W., Alkan, T. Y., İnce, M. N., Günay, M. (2023). A journal recommender for article submission using transformers. Scientometrics, 128(2), 1321-1336. Doi: 10.1007/s11192-022-04609-x

2- Ince, M. N., Günay, M., Ledet, J. (2022). Lightweight distributed computing framework for orchestrating high performance computing and big data. Turkish Journal of Electrical Engineering and Computer Sciences, 30(4), 1571-1585. Doi: 10.55730/1300-0632.3866

Papers Delivered in International Conferences and Printed as a Proceedings

1- Günay, M., İnce, M. N. (2021). Building an Open Source Big Data Platform Based on Milis Linux. ICAIAME 2020, (pp. 33-41). Springer International Publishing.

2- İnce M. N., Ak M., Günay M. (2020). Blockchain based distributed package management architecture. 2020 5th International Conference on Computer Science and Engineering (UBMK), 238-242. Doi: 10.1109/UBMK50275.2020.9219374.

3- İnce M. N., Gunay M., Ledet J. (2019). Building An Open Source Linux Computing System On RISC-V. UBMYK 2019, 30(4), 1013-1020. Doi: 10.2298/FIL1604013C.

4- Günay, M., İnce, M. N., Cetinkaya A. (2019). Apache Hive Performance Improvement Techniques for Relational Data. 2019 International Artificial Intelligence and Data Processing Symposium (IDAP), (pp. 1-6). Doi: 10.1109/IDAP.2019.8875898.