DEMOCRATIZING THE INTERNET OF THINGS THROUGH PLATFORM VIRTUALIZATION

by

Galal Hassan

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
September 2022

Abstract

Towards realizing the vision for Industry 4.0 (I4.0) and (the recently introduced) Industry 5.0 (I5.0) rapid research and development in their core enablers is essential. Along with the recent advances in cloud computing, most research efforts focus on the upper layers of the Industrial Internet of Things (IIoT). However, a significant number of challenges remain unresolved in the lower layers. We believe that the key to catalyze the development of IIoT is to focus on accelerating the development of Internet of Things (IoT) platforms. However, software development in the IoT industry is significantly challenged when compared to other industries (e.g. financial services). This disadvantage is primarily due to the diverse number of IoT platforms available, additionally, the lack of cross-platform software compatibility further hinders the ability to apply modern software development methodologies in the IoT sector. In addition, the IoT industry involves a wide diversity of peripherals, each using a different type of interfacing technology. Such diversity creates a wide range of challenges to large-scale IoT development and deployment. Despite the numerous research efforts in current literature, there remains a massive lack of critical features that enable the broad adoption for large-scale applications.

Our theory is to entwine IoT platform development with popular software and project development concepts, specifically, Continuous Integration (CI), Continuous

Deployment (CD), Separation of Concerns (SoCo), and post-deployment operations. The core objective of our thesis is to accelerate and democratize IoT platform development. Towards this end, we propose Platform Independent Library of Things (PILoT), a platform-agnostic library of architectures and solutions that enable virtualized IoT platforms, peripherals, and an End-to-End (E2E) CI/CD capable IoT development architecture. Furthermore, we present our designs for two of the key enablers of PILoT and assess their viability and performance. HIVE enables virtualization of platforms and peripherals through an IoT containerization engine. While WhiteBus, enables True Plug-and-Play (TPnP) in IoT through a concept we introduced dubbed peripheral phantoms. Before concluding our thesis, we present our journey using PILoT to design and implement an IIoT solution from scratch in less than six calendar months.

Co-Authorship

1. Galal Hassan, Abdulmonem M. Rashwan, and Hossam S. Hassanein. "Reimagining IoT Development Through Software-Defined Development Kits" Journal Article, In Preparation

2. Galal Hassan, Abdulmonem M. Rashwan, and Hossam S. Hassanein. "HIVE: A Journey Towards Democratizing Low-Level IoT Platform Research and Development." IEEE Internet of Things Journal, Under Review

3. Galal Hassan, Abdulmonem M. Rashwan, and Hossam S. Hassanein. "SandBoxer: A Self-Contained Sensor Architecture for Sandboxing the Industrial Internet of Things." In 2019 IEEE International Conference on Communications Workshops (ICC Workshops), pp. 1-6. IEEE, 2019.

4. Galal Hassan, Abdulmonem M. Rashwan, and Hossam S. Hassanein. "WhiteBus: A Platform Independent Plug-and-Play Interface for IoT Infrastructures." In ICC 2019-2019 IEEE International Conference on Communications (ICC), pp. 1-6. IEEE, 2019.

5. Galal Hassan, and Hossam S. Hassanein. "MoT: A deterministic latency MAC protocol for mission-critical IoT applications." 2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC). IEEE, 2018.

Acknowledgments

Today, as I sit and type, I conclude a fantastic journey. A quest filled with challenges and adventures worthy of a J. R. R. Tolkien novel. As I arrive at the destination, I recall times when I failed before I succeeded, gave up before I persevered, and struggled before I triumphed. And as I prepare to conclude this chapter of my life, I realize I would have never arrived at the destination without being surrounded by all my colleagues, friends, and family.

I'm incredibly grateful to Prof. Hossam Hassanein. Before this journey started, you believed in me and gave me a chance; very few were willing to give. You were more than a research supervisor; you helped me grow as a researcher and person. You gave me freedom of speech and thought, and I will forever value your honest, constructive and fierce feedback, which has always helped me regain my focus on the bigger picture. I would not be who I am today without your nurturing and support. Thank you.

I am also grateful to Dr. Abdelmoneim Rashwan. You've had a profound belief in my work and abilities from the first week at TRL, often more than I had in myself. Along this journey, you have become a friend, a co-author, and a mentor. Your guidance and insightful suggestions have always unblocked me when I've reached dead ends.

Special thanks to Dr. Mohamed Adel. I will always value our conversations and debates during our time at the lab and at the airport. We've spent considerable time together on this journey, and I will never forget the challenges we've conquered together. At the end of this journey, I will forever be grateful for the trips and projects when we always "Stole The Show" together.

I would also like to extend my deepest gratitude to the examination committee. I truly enjoyed our discussions and value your feedback and improvement suggestions. Thank you Prof. Aboelmagd Noureldin, Prof. Cheng Li, Prof. Khalid Elgazzar, and the examination chair Prof. Selim Akl.

I would like to extend my sincere thanks to Dr. Sharief Oteafy. On my first day on campus, you helped me to craft the path that would become the seed of my journey. I will always be thankful for your invaluable insights and advice. Thanks also to Dr. Khalid Elgazzar. You were my first co-author at Queen's University, and although our other research visions did not align, I will always remember our discussions and debates.

I also had the great pleasure of working with Dr. Abdullah Maaitah. Your hardware (and crafting) expertise were instrumental in the projects we worked on together. I will always value your support; thank you.

I cannot leave Queen's University without mentioning Dr. Deena Salem. You were the first person to welcome me on my first night back in Canada in 2014. You were my mentor in my Undergrad but my friend for life. You have never wavered in your support, and I will always be grateful for your guidance and friendship.

Basia Palmer was particularly helpful to me (and everyone at TRL) during this journey. You are the backbone of the lab. Thank you for all your help figuring out

the logistics of everything at the lab. I'd also like to recognize the assistance of all my colleagues at the lab. Thank you, Wenjie Li, Hesham Farahat, Abdalla Abdelrahman, Amir Mohamed, Ashraf Alkhresheh, Mohannad Alharthi, Mohamed El-Maradny, Faria Khandaker, Sara Elsayed and Mary Zarif Riad.

I will always be indebted to my parents, Mounir and Somaya. Your love and support have been and will always be the beacon of light at the end of the tunnel. Thank you to my sister, Ayat, for always being there for me and telling me that I am awesome even when I didn't feel that way. I'll always be grateful to my brother, Omar, for teaching me to "Never Give Up". I am deeply thankful to my bro, Mohamed Saleh. I remained sane on this journey because of your support, jokes, and companionship.

Above all, I would like to thank my wife, Sara K. Your unconditional love is why this journey can finally conclude. I look forward to all our future journeys together. I will always be grateful for when you constantly listened to me rant and talk things out, and for the sacrifices you made for me to pursue a Ph.D.. I love you.

Although this journey has come to an end, I am looking forward to the next quest.

"I love to travel, but I hate to arrive." $\sim$ Albert Einstein

Statement of Originality

I hereby certify that all the work presented in this PhD thesis is original, and all notions and/or techniques attributed to others have been fully acknowledged in accordance with the proper referencing practices.

Contents

Glossary

$_m$**S^{3}4G** S^3 4G Module.

$_m$**S^{3}E** S^3 Ethernet Module.

$_m$**S^{3}G** S^3 Guardian Module.

$_m$**S^{3}L** S^3 Long Range (LoRa®) Module.

$_m$**S^{3}N** S^3 Near Field Communications (NFC) Module.

$_m$**S^{3}S** S^3 Sensing Module.

$_m$**S^{3}Sat** S^3 Sattelite Module.

$_m$**S^{3}T** S^3 Tamper Module.

$_m$**S^{3}W** S^3 Wireless Fidelity (WiFi) Module.

$_m$**S^{3}Z** S^3 Zoner Module.

4G Fourth Generation Wireless.

AIO Analogue Input/Output.

API Application Programming Interface.

ARM® Advanced Reduced Instruction Set Computer (RISC) Machine.

BEAR BEE Execution Auditing Rover.

BEE Basic Execution Entity.

BEEs Basic Execution Entities.

BLE Bluetooth Low-Energy.

BS Base Station.

BW BandWidth.

CD Continuous Deployment.

CD-CD Container Drop Continuous Delivery.

CF Continuous Feedback.

CI Continuous Integration.

CO Carbon Monoxide.

CoAP Constrained Application Protocol.

COVID-19 Coronavirus Disease 2019.

CPS Cyber Physical Systems.

CPU Central Processing Unit.

CR Coding Rate.

CRC Cyclic Redundancy Checking.

CS Chip Select.

CSV Comma Separated Values.

DB Database.

DC Duty Cycle.

DE low Data-rate optimization Enable.

DevOps Development & Operations.

DIO Digital Input/Output.

E2E End-to-End.

EC Edge Computing.

EEPROM Electrically Erasable Programmable Read-Only Memory.

FPGA Field-Programmable Gate Array.

FSK Frequency Shift Keying.

GND Common Ground.

GPIO General Purpose Input/Output.

GPS Global Positioning System.

GSM Global System for Mobile communications.

GUI Graphical User Interface.

H explicit Header.

HAL Hardware Abstraction Layer.

HDB Hardware Definition Blueprint.

HEAL Hive Engine Abstraction Layer.

HIVE Hardware Independent Virtual Engine.

HLA High Level Architecture.

HV Hypervisor Virtualization.

I^2C Inter-integrated Circuit.

I4.0 Industry 4.0.

I5.0 Industry 5.0.

IaC Infrastructure as Code.

IC Integrated Circuit.

ID Identification.

IDE Integrated Development Environment.

IEEE Institute of Electrical and Electronics Engineers.

IIoT Industrial Internet of Things.

IO Input/Output.

IoT Internet of Things.

IS Instruction Set.

ISA Instruction Set Architecture.

JVM Java Virtual Machine.

LED Light Emitting Diode.

LoRa® Long Range.

LoRaWAN® Long Range Wide Area Network.

MAC Media Access Control.

Mb Mega-bit.

MCU Microcontroller Unit.

MoT MAC on Time.

MQTT Message Queuing Telemetry Transport.

NCAP Network Capable Application Processor.

NFC Near Field Communications.

OOP Object Oriented Programming.

OS Operating System.

OTA Over-The-Air.

PC Personal Computer.

PDB Peripheral Definition Blueprint.

PHY Physical Layer.

PILoT Platform Independent Library of Things.

PL PayLoad bytes.

PMU Power Management Unit.

PnP Plug-and-Play.

PoC Proof of Concept.

QA Quality Assurance.

QUEEN Quick Universal Execution Entity Nexus.

RAM Random Access Memory.

RISC Reduced Instruction Set Computer.

ROM Read Only Memory.

RSSI Received Signal Strength Indicator.

RTOS Real-Time Operating System.

RX Receive.

S^3.IO S^3.IO Cloud.

S^3AS S^3 All-Sense.

S^3N S^3 Node.

S^3P S^3 Proximity.

S^{3}R S^3 Router.

S^{3}Z S^3 Zoner.

S3 Smart Sensing Solutions.

SCL Serial CLock.

SCS Sensor Controller Studio.

SD Secure Digital.

SDA Serial DAta.

SDAA Sensor Data Acquisition Algorithm.

SDDK Software-Defined Development Kit.

SDI Serial Data In.

SDLC Software Development Life Cycle.

SDO Serial Data Out.

SDR Software-Defined Radio.

SE Sensing Element.

SeN Sensor Node.

SF Spreading Factor.

SiN Sink Node.

SoC System on Chip.

SoCo Separation of Concerns.

SPI Serial Peripheral Interface.

TCP/IP Transfer Control Protocol/Internet Protocol.

TDMA Time-Division Multiple Access.

TEDS Transducer Electronic Data Sheet.

TI Texas Instruments.

TOA Time On Air.

TPnP True Plug and Play.

TX Transmit.

UART Universal Asynchronous Receiver-Transmitter.

UDP User Datagram Protocol.

UI User Interface.

UPnP Universal Plug and Play.

USB Universal Serial Bus.

$\mathbf{V}_c c$ Common Collector Voltage.

VCPU Virtual Central Processing Unit.

VDK Virtual Development Kit.

VDKaaS Virtual Development Kit as a Service.

VID Virtual IoT Device.

VM Virtual Machine.

VMM Virtual Machine Monitor.

VSoC Virtual System on Chip.

WASM WebAssembly.

WBL WhiteBus Line.

WBM WhiteBus Module.

WBMSI WhiteBus Master Serial Interface.

WBUSI WhiteBus Universal Serial Interface.

WiFi Wireless Fidelity.

WSN Wireless Sensor Network.

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Over the past three centuries, civilization has gone through three industrial revolutions: the first based on mechanization, water and steam power; the second based on mass production, assembly lines, and electricity; the third and current relies on computers and automation [110]. In November 2011, the German government planted a seed for the fourth industrial revolution for 2020, and the term Industry 4.0 (I4.0) was coined [61]. I4.0 consists of four design principles, *Interoperability*, *Information Transparency*, *Technical Assistance*, and *Decentralized Decisions* [56]. Nine different technologies are involved in satisfying each design principle, each introducing a multitude of challenges in the context of I4.0 [110]. However, I4.0 is based mainly on the integration of both Cyber Physical Systems (CPS) and Industrial Internet of Things (IIoT) technologies [145]. It is prevalent that realizing I4.0 will not be possible without rapid advancement in its technological pillars [25, 106]. Furthermore, customization and personalization are at the core of Industry 5.0 (I5.0) [31]. Therefore, for our research, we focus on the challenges brought forward by these concepts and technologies.

In 2020, the Coronavirus Disease 2019 (COVID-19) pandemic caused a global supply-chain chaos affecting a myriad of business sectors [39]. The effects of these

disruptions remain a reality even two years after the start of the pandemic, with multiple workarounds being employed by multiple businesses [121]. According to [130], Inmarsat created "digital twins"[1] of their supply chain to help mitigate the supply-chain issues. Furthermore, 77% of their survey respondents deployed an Internet of Things (IoT) project in 2020, compared to 21% in 2018. Such a rise in demand has not left the IoT industry immune from the unfortunate supply-chain issues. For instance, silicon-chip production lead times skyrocketed from an average of six to nine weeks to a staggering twenty-six weeks [63]. Thus, causing many of the IoT development kits, peripherals, and components to be delayed. Due to delays in electronic component shipping, several IoT developers are blocked and unable to proceed with developing new systems. These developers are searching for alternatives to the traditional development methodologies in the IoT sector. As a result, the progression of I4.0 has been hindered and was not fully realized in 2020.

Furthermore, such global events have ignited the interest in a more accelerated, secure, and efficient IoT platform development pipeline [30]. In order to convert the current status-quo of IoT development into a cross-platform haven, significant research efforts are required. Traditionally, the IoT platform development journey involves efforts across the different layers of the IoT architecture [124]. However, each layer of the architecture requires developers to have experience in a different area of software development, embedded, web, cloud, and mobile [124]. Despite the extensive research efforts in accelerating the development across the higher layers of the IoT architecture, to the best of our knowledge, the development cadence of the lower layers is still severely challenged; specifically, the sensing elements, sensing nodes, and IoT

[1] "*A digital twin is a virtual representation of an object or system that spans its lifecycle, is updated from real-time data, and uses simulation, machine learning and reasoning to help decision-making.*" - IBM

routers/gateways. Furthermore, IoT applications are usually tightly coupled to their underlying platforms, making it extremely challenging for general-purpose and generic platforms to be used by diverse IoT applications [37].

Additionally, developing sensor units or peripherals for IoT applications is currently the combined responsibility of the sensor/peripheral manufacturers and IoT platform developers. The manufacturers are responsible for creating the sensing elements and providing the proper documentation and examples. At the same time, the developers are responsible for implementing and using the sensing elements in IoT sensor nodes for different applications. Even though manufacturers sometimes develop software drivers for their manufactured sensors, IoT platform developers still need to port the driver to the different hardware platforms and operating systems. The challenges for inexperienced IoT platform developers rarely stop here. It is common that once a development kit is delivered, some developers decide that it is unsuitable for their use case after using it in a quick prototype. Such uncertainty delays the development of the entire IoT platform and increases costs significantly.

Towards a world in which embedded peripherals are interchangeable, platforms are modular and reusable, and post-deployment software maintenance is feasible, we propose a platform-agnostic framework optimized for IoT development. The proposed research aims to enable a seamless adoption of modern Software Development Life Cycle (SDLC) approaches within any IoT platform development project, thus catalyzing the research and development of IoT, IIoT and respectively I4.0 and I5.0. Moreover, our research adopts the concept of software containers and sandboxes to introduce the concept of hardware containers and development-sandboxing, thus, enabling the Separation of Concerns (SoCo) paradigm in IoT platform design and development.

1.1 Motivation

Despite the diverse and widespread platforms targeted for the IoT space, the technical innovation and development cadence remains hindered by the number of challenges that still exist [137]. Furthermore, cross-compatibility is often overlooked due to the never-ending competition between IoT device manufacturers. As a result, the IoT market became fragmented with numerous proprietary technologies that are not cross-platform compatible [137]. Hence, a sharper focus on the challenges involved in such technologies is needed. While the higher layers of IIoT and CPS have been the primary focus of many researchers, the infrastructure of these technologies is often overlooked [106]. Despite having distinctive higher layer implementation philosophies, both technologies share a similar if not common infrastructure. Unfortunately, implementing such infrastructures is not unified, with each system fundamentally different. Such diversity in design escalates the difficulties when interfacing vendor-manufactured IoT devices and peripherals. With each peripheral using one type of interfacing bus, the innovation of peripheral manufacturers is severely throttled by the need to satisfy multiple interfacing technologies. Such challenges across the industry hinder the progression towards the I4.0 vision. Hence, a solution to bridge the different interfacing technologies is crucial.

Software development methodologies have been evolving rapidly over the past decade, and with increasing business requirements driving the pace of development, methodologies such as Agile have replaced Waterfall [76]. However, the need for faster software deployment and production cycles has pushed the industry towards a methodology that tightly couples the development team with the operations team, namely Development & Operations (DevOps). The applicability of the DevOps

approach in different software industry sectors has been explored in current literature. However, the feasibility of adopting DevOps within the IoT and IIoT sectors remains a subjective topic [77]. To truly embrace the vision of I4.0 and I5.0, an objective consensus on the feasibility of DevOps in the IoT and IIoT industry is required. Furthermore, the challenges facing such adoption must be addressed. One challenge is the process of deploying an application from a development environment to a production environment [8].

Furthermore, the software development process involved in the lower layers of an IIoT platform articulates using platform-specific code. Microcontroller firmware is usually programmed in C and runs at a very low level, requiring expert knowledge to use a Real-Time Operating System (RTOS). Although a traditional development process usually delivers highly optimized code, it creates a multitude of challenges in a world where flexible, portable, and agile code is the number one requirement [17]. To that end, a platform-independent software development process is vital to excelling in developing both IIoT and CPS technologies.

Contemporary methodologies to integrate IoT peripherals have rendered IoT devices non-customizable. This lack of customization results in application-specific IoT platforms that require extensive development time and expertise [141]. With research efforts dating back to 1984 [96], Plug-and-Play (PnP) is no longer a novel concept. However, within the IoT context, the use of PnP technologies remains challenging due to the resource-constrained platforms used in IoT. Since PnP is part of the IoT roadmap [48], there has been a growing interest in the adoption of the technology within the realm of IoT and IIoT.

Another trending research area in IoT is security, and a significant source of

security vulnerabilities is the lack of software updates post-deployment [112]. As is the case with Wireless Sensor Network (WSN), post-deployment software updates are a must-have feature of modern IoT platforms. However, realizing such a vision for IIoT is not a trivial task; previous efforts have yet to propose an efficient method.

Finally, almost none of the commercial IoT development platforms focus on the full development journey an entire solution employs. Such platforms either focus on dashboarding [60, 47, 29, 128, 55], cloud infrastructure [87, 7], or embedded development [11, 44, 38]. To the best of our knowledge, the only commercially available platform that provides a vision relevant to ours is Particle [100]. However, despite offering an End-to-End (E2E) solution, Particle requires the usage of the hardware it offers, making it very challenging to use other hardware. This type of vendor lock-in is one of the primary reasons behind the fragmented IoT market we live in [137]. Towards a future in which the core concepts of DevOps, namely, Continuous Integration (CI) and Continuous Deployment (CD) are integrated into the IoT development process, a shift in the current IoT development paradigm with a unified framework that fills the gaps is essential.

1.2 Contributions

This thesis proposes Platform Independent Library of Things (PILoT), a software development framework that streamlines IoT development and enables modern SDLCs in the field by introducing the following concepts. A docker-like low-footprint containerization engine (Hardware Independent Virtual Engine (HIVE)) for IoT devices that allow developers to create once and deploy on any platform. The concept of peripheral phantoms encapsulate all peripheral requirements (drivers and descriptors) on the device and a hardware interfacing bus (WhiteBus) to enable our proposed TPnP paradigm. Finally, using PILoT our proposed concepts of Software-Defined Development Kit (SDDK)s and Virtual Development Kit (VDK)s can be realized.

Furthermore, the thesis outlines our vision of a cross-platform haven for IoT solutions. It describes the core components of PILoT across the chapters to present by the end a wholesome view of what this paradigm entails and the challenges to seeing it manifest.

1.2.1 Re-envisioning a streamlined IoT platform development with the PILoT framework

The current status-quo of IoT development is hindered by the complexity of writing, maintaining, and operating different levels of software solutions that work in tandem to achieve the intended application requirements [65]. We present PILoT to redefine the traditional IoT development pipeline and bring modern SDLC capabilities to the world of IoT development. Furthermore, our framework enables the concept of complete resource-sharing for IoT peripherals and devices across the globe. In PILoT, developers use Infrastructure as Code (IaC) to software-define their hardware and

networks in the Design & Plan stage. They are followed by developing and testing their application logic without needing physical (or local) hardware access using IoT containers in the Develop stage. In the Deploy stage, they then describe and implement a previously infeasible CI/CD pipeline. Before finally operating, controlling, monetizing, and managing their IoT fleet in the Operate stage.

1.2.2 Platform Agnostic IoT Development

The development of IoT platforms involves an intertwined relationship between software and hardware. Today's IoT firmware is developed to target a specific hardware platform with a specific underlying processor architecture. Porting the firmware to a different platform involves significant development effort. Thus, a high level of code reuse is vital to accelerate the development of new IoT platforms. PILoT proposes using resource-constrained IoT-tailored software containers. The low-footprint containers allow developers to develop and test on any platform and then run their application on any other platform. The containerization engine proposed, HIVE, encapsulates drivers, hardware blueprints, and application scripts. Additionally, HIVE uses a custom byte-code tailored for resource-constrained embedded devices, resulting in a low-footprint high-performance Virtual Machine (VM).

1.2.3 Enabling IoT Peripheral Interoperability

The interoperability of the current IoT peripheral market is fractured. Diverse manufacturers create and use a manifold of standards and interfaces. Furthermore, peripheral drivers (if provided by the manufacturer) require significant effort to port correctly before use on an IoT platform. Combining hardware blueprints with dynamic

applications running on IoT containers enables the first step of TPnP. To achieve the whole vision, we propose a shift in driver development responsibility from embedded system developers to peripheral manufacturers. We introduce the concept of peripheral phantoms to encapsulate the peripheral drivers and descriptors and further enable the virtualization of the peripherals. Finally, we propose a hardware interfacing bus, WhiteBus, that abstracts the underlying hardware peripheral interfaces on a platform to provide a universal bus to be used by TPnP supported devices in addition to legacy devices.

1.2.4 Catalyst for Realizing Remote Development

Today's global supply-chain chaos and economic crisis have caused a severe drop in the supply of hardware components and development kits. Furthermore, inexperienced developers are usually uncertain of the fitness of a particular development kit until it is delivered and a Proof of Concept (PoC) built. It is then common for these developers to realize they would require a different development kit and would have to endure the in-availability, delivery delays and inflated costs of the current market. Therefore, we propose the concept of Software-Defined Development Kits (SDDKs) and Virtual Development Kits (VDKs). Using PILoT, HIVE, and WhiteBus, manufacturers and developers can virtualize and use virtual development kits locally or on the cloud without physical hardware access. Thus, reducing business risk and accelerating IoT platform development towards the full vision of I4.0.

1.3 Organization of The Thesis

The remainder of this thesis is organized as follows. We discuss background and related work in Chapter 2, propose the PILoT paradigm in Chapter 3 and discuss it's potential as a catalyst towards I4.0. We then propose HIVE in Chapter 4 as the embodiment and implementation of the IoT containerization engine proposed in PILoT. We propose WhiteBus in Chapter 5 as a universal interfacing bus towards the realization of the TPnP paradigm introduced with PILoT. We discuss how we used PILoT to design, develop, implement, and deploy an IIoT monitoring solution in less than six months in Chapter 6. Finally, we conclude our thesis and present future directions in Chapter 7.

Chapter 2

Background

The first step in our IoT space defragmentation journey was to investigate and learn about the different concepts in this space. In this chapter, we outline our findings and describe the primary principles related to our research. However, before we delve into some related research, we present and discuss commonly used terminology in an effort to align our discussions for the remainder of this thesis.

2.1 Terminology Alignment

Due to the diversity of research in the IoT field, different terminology is used to refer to different elements in the IoT realm. For our research, we use the classic IoT architecture as illustrated in Fig. 2.1. The Network and Perception layers consist of one (or more) star networks with a centralized Sink Node (SiN). At the leaves of the network (in the Perception Layer) are Sensor Nodes (SeNs) with one (or more) Sensing Elements (SEs) attached. The Application layer consists of cloud infrastructure and a user-facing application that displays and executes business logic.

2.2 The IoT Development Paradigm

The main focus of our research is to revolutionize the IoT development journey; therefore, we first start by describing the current status-quo.

The Internet of Things (IoT) paradigm spans a myriad of application and research focuses. Regardless of the application, the goals of an IoT system remains constant, collect data from physical devices, learn from the data, and act on the data [124]. The Alliance for Internet of Things Innovation released a High Level Architecture (HLA) towards efforts of IoT standardization. The proposed HLA involves a sophisticated architecture model for a standardized IoT that is based on the classic IoT architecture [119]. The classic architecture is three-tiered, consisting of several sensing devices in the perception layer, one or more gateways in the network layer, and a cloud infrastructure in the applications layer [48]. Figure 2.1 illustrates a traditional three-tiered architecture. As a consequence of the increased popularity of IoT and IIoT, several challenges spanning from a single layer to cross layers have resurfaced [139]. Additionally, each of the core IoT components in each layer has countless challenges when it comes to development [124].

Countless hours of research and development to resolve the lack of standardization between IoT platforms can be considered one of the core reasons behind the extended development-cycle new IoT platforms go through [4]. Towards the standardization of IoT, industry giants such as Microsoft [87] and Amazon [7] have successfully tackled the interoperability challenges in the cloud/backend layer; however, lower layer mismatch issues remain unaddressed. Towards addressing the standardization issues on the end-devices and gateways, we discuss software-related challenges of current IoT architectures, development, and post-deployment support from the perspective of

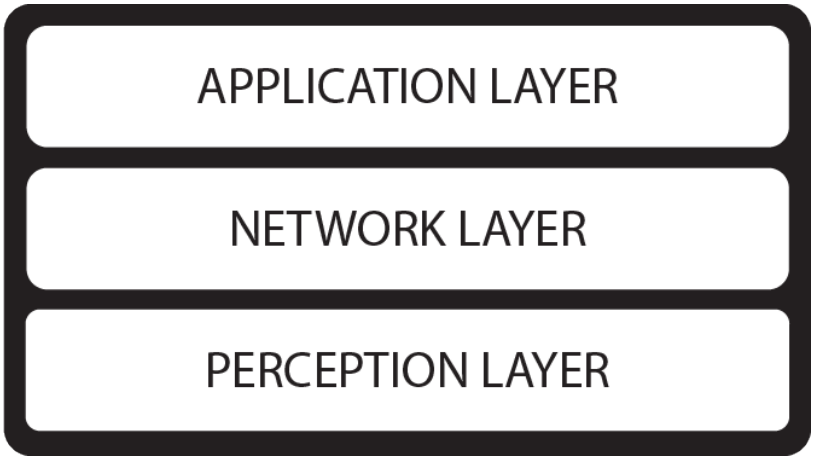


Figure 2.1: A traditional IoT/IIoT three-tiered system architecture

these lower-layer IoT elements in the following subsections.

2.2.1 IoT Prototyping

Two common methods used in prototyping IoT platforms involve either virtual platforms in the application layer or generic embedded system platforms in the perception and network layer, with the latter being the widely used method [132]. Such a process involves using embedded system platforms, such as the Raspberry Pi [43] or Arduino [11] for the sensing devices and gateways. These platforms have been popularized because of their flexibility and extensive online community, with hobbyists and companies creating expansion boards that host various sensing elements, thus, making rapid prototyping a reality. However, despite the flexible environment such platforms provide, they are not geared towards production environments. These generic platforms are bloated with unnecessary features for most IoT applications, consuming more of the already constrained resources available for IoT applications. Therefore, transitioning from a development/prototyping environment to a production

environment is often very challenging [77].

The second prototyping method used to kick-start IoT projects involves using software-defined virtual IoT platforms. However, this method remains limited in popularity due to the number of inherent challenges. A key challenge lies in the abstraction of IoT devices on the cloud towards enabling virtual sensors and participatory sensing. To resolve this, NTT Communications Science Laboratories researchers have developed a virtual machine for wireless sensor nodes to abstract IoT devices on the cloud [122]. At the same time, researchers at EURECOM proposed an Edge Computing (EC) architecture to create Virtual IoT Device (VID). These virtual devices abstract the underlying physical device capabilities into a single VID, thus providing a positive step towards enabling virtual sensing [32]. Unfortunately, these efforts fail to run multiple IoT applications on a single device, a feature we believe is critical for low-risk rapid prototyping and for virtual sensing to become a widespread reality.

2.2.2 IoT End-Node Development

The perception layer involves SeNs that communicates with SiNs to relay the information to the backend. These SeNs run an embedded firmware in contrast to traditional software applications. Embedded software architectures, can be categorized into full-stack Operating System (OS) based devices, partial stack OS based devices, and devices with no OS [117]. Partial stack OS devices have a special-purpose RTOS that sometimes includes a network stack or requires implementation. In such architecture, there are four primary levels of firmware code implementation, application, peripheral drivers, RTOS, and Hardware Abstraction Layer (HAL).

- **Real-Time Operating System (RTOS)** is usually developed and implemented

by a third party, and the developer is not usually concerned with the development of this layer. However, in some instances, the developer might need to implement or modify the network stack within the RTOS. In such cases, the network stack implementation depends on the communication hardware installed on the platform and, therefore, can be platform-specific or application-specific. However, we assume the network stack and communication hardware to be unchanged per platform for discussion purposes.

- **Hardware Abstraction Layer (HAL)** is responsible for abstracting and exposing lower-level System on Chip (SoC) hardware capabilities to the RTOS. Depending on the IoT platform being used, the RTOS could have a compatible or non-compatible HAL, thus, an IoT developer might need to port the HAL before being able to use a specific RTOS. This layer is usually implemented once per IoT/embedded-system platform.

- **Peripheral Drivers** are software explicitly implemented for each peripheral device connected to a platform, such as sensors, actuators, and external memory. Because all implemented software is compiled, then assembled into machine-code firmware and then flashed onto the device, a developer must know all the peripherals connected and the different hardware ports they are connected to. Such a process also constrains the ability to change sensors or peripherals after installing the firmware.

- **Application** this firmware level is usually the highest and most versatile, but it must still be baked into the full firmware alongside the other layers. Therefore, the implementation is usually tightly coupled with the peripherals attached to

the platform.

Software development in each software architecture layer is considered firmware development. Unlike traditional software engineering practices, firmware engineering slightly differs in design, construction, debugging, testing, verification, and maintenance techniques [33]. A traditional development process usually delivers highly optimized code due to the firmware being developed in C and running at a very low level. However, this type of development requires expert knowledge, creating a multitude of challenges, especially in a world where flexible and portable code is the number one requirement [17]. Additionally, in contrast with traditional applications, firmware is often flashed (with the use of specialized hardware equipment) on an IoT device and requires a reboot. Thus, reprogramming or changing an IoT device's logic is not usually feasible.

2.2.3 Post-Deployment Support

Once an IoT device is deployed in the field, it is usually practically impossible to gain physical access to update the firmware or diagnose a failure. Therefore, adopting popular software engineering paradigms (such as Agile or DevOps) in the realm of IoT software development is very challenging. This restricted access is especially challenging when an IoT application involves deploying hundreds or thousands of devices [136]. Thus, remote firmware updates, also known as Over-The-Air (OTA) updates, are essential for the rapid development of IoT devices. Furthermore, we believe that the ability to execute multiple applications with minimal downtime is critical for the future of IoT.

OTA reprogramming is the term for reprogramming a particular device without

physical access to it. Reprogramming deployed sensor nodes over-the-air is essential to maintaining WSNs. It is more challenging than reprogramming regular nodes due to the limited storage, power, processing, and communication capabilities [116]. The desired reprogramming features are to minimize the usage of the limited resources available while providing the node with the updated software image. One of the main advantages of OTA programming is the ability to resolve post-deployment software bugs without physically accessing the nodes.

Despite receiving little attention in recent years, the demand in this area of research is increasing. One of the recent researches [115] developed an efficient OTA programming technique that requires no reboot of the node and zero-flash requirements. This feature is achieved by altering the code on the device instead of reprogramming it. The scheme compares the old and new images. After a detailed comparison, the difference is sent to the device, minimizing the requirement of sending the entire image to be reprogrammed. Since desired code alterations are already in place, a node reboot is not required. Another work [26] developed a platform dubbed SYNAPSE++ that allows reprogramming deployed sensor nodes with maximum error resolution and without the need for onsite intervention. The main focus was to reduce the time required for reprogramming a massive network of wireless sensors and to eliminate the human physical intervention previously required for programming each node individually.

In addition to solving the zero-downtime reprogramming, another challenge associated with OTA upgrades is the wireless transfer of the firmware data. Due to the constrained resources on IoT devices, it is necessary to use a communication scheme that minimizes packet collisions, ensures timely delivery, and has a short Time

On Air (TOA). Some common protocols used for post-deployment support include Message Queuing Telemetry Transport (MQTT), Constrained Application Protocol (CoAP), Pub/Sub, ZeroMQ, and HTTP [142, 16]. However, a majority of the common protocols are tailored for SiNs and communication over an underlying Transfer Control Protocol/Internet Protocol (TCP/IP)/User Datagram Protocol (UDP) layer. Such requirement is not always available on low-end SeNs. Which are commonly not directly connected to the internet and instead use low-footprint radio (e.g. Bluetooth, ZigBee, LoRa®.) to relay the information through a SiN [142]. Additionally, a protocol such as MQTT requires a continuous connection between the devices, which would quickly drain the battery of any SeN.

2.3 Adoption of Development & Operations in IoT

One of the core accelerators of software development in the past decade was the transition to DevOps. However, despite the fact that the rapid rise in its adoption in the software industry [46], adoption rates remain extremely low in the IoT development space. Such a slow adoption rate can be attributed to the number of challenges inherent in the low-level development involved in IoT development [92]. DevOps has been the topic of interest for researchers in the field of IoT for some time [3]. However, most research focuses solely on higher layers of the IoT architecture, specifically, the cloud/application layers. In contrast, lower layers (sensor hardware/firmware) are generally ignored.

The main general requirements for a successful DevOps adoption are CI, CD, and Continuous Feedback (CF) [10]. Each requirement involves a series of steps forming a circular chain across the development and operations team. This connection bridges the gap between both teams and improves collaboration over other methodologies, such as Agile [136]. Furthermore, only a handful of recent research focuses on a full DevOps pipeline for IoT [50].

2.3.1 Continuous Integration

The first requirement for a full DevOps pipeline is CI. Multiple stages are involved in achieving this requirement: plan, code, test, and build. Despite CI being successfully implemented in the cloud layer [14], challenges still exist in the code, test, and build stages for other layers, such as the sensing layer. Such challenges include restricted access to IoT hardware and extended development iteration time. This restriction limits automated testing and thus invalidates the CI pipeline. Therefore, the ability

to virtualize (or mock) IoT devices and peripherals is required to enable a complete CI pipeline.

2.3.2 Continuous Deployment

This stage usually entails automated integration testing and deploying to a staging environment identical to the production environment. Once all tests pass in the staging environment, deployment to the production environment is the next step; however, this is highly challenging in IoT sensor devices since they are usually deployed in enormous numbers with restricted physical access. An identical sensor device is generally used as a staging device and used for testing; however, this is a manual process and is usually not flexible enough to cover different testing scenarios. Furthermore, to fully realize an ideal CD pipeline, an efficient OTA automated deployment and upgrade system is required.

2.3.3 Continuous Feedback

The final stage in the DevOps pipeline involves operating and monitoring the system. Despite this being a typical operation in the software industry, it remains challenging for resource-constrained IoT sensor devices [78].

2.4 IoT Development Sandboxing

One of the fundamental concepts of computer science is the notion of SoCo, one such implementation of the concept is the containerization of software. With the rising popularity of DevOps and cloud computing, containerized software became a core pillar of every developer's toolchain. The concept, however, is commonly confused with running virtual machines. A container, on the other hand, abstracts a layer running on top of an operating system. Thus, a container can be thought of as a virtual software environment. In the IoT realm, containers are reserved for higher layer implementations due to the vast computing resources required to run containers. Nonetheless, sandboxing needs not only to be implemented for execution isolation but also has a more substantial potential in developing an entire IIoT system. Therefore, to bring the notion of SoCo into the lower layers of the IoT paradigm, we propose the use of a scripting-based IoT software-development architecture to implement a containerized sandboxed execution environment.

The term sandboxing is widely used to describe the concept of isolating the execution of software programs in separate restricted environments. Thus, safeguarding parallel and underlying software systems from unauthorized software access [104]. Research on different sandboxing mechanisms has been on the rise for the past decade [101]; for instance, fog computing uses virtualization to enable sandboxing [105]. Apple uses sandboxing in its iOS operating system to isolate each application and limit access to the file system, network, and the underlying hardware [89]. On the other hand, Google's Android uses a sandboxing architecture based on the Linux kernel [2]. The sandboxing concept has been adopted in the WSN paradigm to create a live forensics framework [143] and enable software-based memory protection [66] for

wireless sensor nodes. However, sandboxing adoption in the field of IoT has been very limited.

2.4.1 Containerization

The principle of SoCo has been at the core of software engineering since the introduction of Object Oriented Programming (OOP), giving birth to the concept of resource virtualization to become one of the key enablers of cloud computing. Traditional virtualization techniques, such as Hypervisor Virtualization (HV) requires a Virtual Machine Monitor (VMM) that runs on top of a host OS to create a VM that is hosting its own isolated OS [27].

Software containers theorized in 2006 [23] is a lightweight alternative to virtual machines that have recently been popularized to provide isolation and resource management [27]. Containerization engines such as Docker [21] can encapsulate applications and their dependencies within a virtual container, running on top of a host OS. Thus, multiple containers share a host OS kernel. Although a shared kernel seems like a security downgrade from the security provided by VM isolation, the benefits of using a container have been investigated from a security perspective by [62]. The research found that the high level of isolation and resource limitation helps harden the host's security to a level closer to that provided by the VM alternative.

Part of the roadmap for the future of IoT is enabling the containerization concept within its realm [123], thus, enabling one of the essential core concepts of DevOps. Towards this end, research such as [92] has been focused on bringing Docker to Linux-based IoT platforms. However, Docker remains unfeasible for resource-constrained embedded systems not running a Linux OS. It is simpler to adopt DevOps in the

web development domain partly due to the infrastructure abstraction provided by virtualization [77].

We believe that a containerization engine tailored for resource-constrained embedded/IoT devices would serve as one of the main ingredients of the IoT development catalyst.

2.4.2 Embedded Scripting

Scripting describes a type of programming language at a higher level than system programming languages (often referred to as high-level languages) such as C, C++, and Java. Instead of compiling a program ahead of time to generate machine code (as is the case with system programming), an interpreter runs script commands during run-time [98]. Among the many advantages of scripting and most relevant to our research are platform independence, rapid development and run-time execution. Unfortunately, running a traditional scripting engine on an embedded system is not trivial, which motivated researchers to develop stripped-down versions of Lua and Python, eLua [40] and MicroPython [19] respectively.

Another example of optimizing scripting engines for embedded systems is TinyScript, an open-source scripting engine designed from scratch for memory-constrained microcontrollers [69]. Another is Tapper, a lightweight scripting engine designed for resource-constrained wireless sensor networks [140]. Alternatively, projects such as TinyOS [68] focused on enabling run-time system support on resource-constrained embedded systems through a low-footprint OS, aiming to support programming abstractions and configurations. However, due to the extensive generic features included with TinyOS (and similar embedded OS), only a handful of platforms are capable of

running the OS [68].

2.4.3 Run-time Virtual Machines & Platform Independence

Most run-time VMs create a software-defined Turing-complete machine to interpret dynamically typed programming (scripting) languages. The Java Virtual Machine (JVM) is one of the most popular VMs for all platforms. Despite the added benefits of execution isolation and platform independence, a VM is usually significantly less performant compared to native bare-metal implementations [97]. Due to this performance penalty, using interpreted languages in an IoT platform has been frowned upon by researchers and industry developers alike [70]. As a result, interest in a resource-constrained tailored scripting language has been sparked in the past decade, with efforts to create embedded versions of popular scripting languages. microPython, for example, is an implementation of Python 3 for embedded and resource-constrained systems [19]. On Unix, microPython requires 0.5 MBs and approximately 256 KBs for embedded platforms. Additionally, the start-up memory usage of microPython is north of 20 KBs [103]. Another example is mRuby, an embedded version of the Ruby VM [125] designed specifically for resource-constrained devices. However, the footprint of mRuby is >512KBs [70], rendering it expensive for most IoT SeNs.

Multiple efforts investigated the use of Web Assembly on IoT; however, the memory requirements are at least 100 KBs [138], with the WebAssembly (WASM)3 interpreters requiring a Read Only Memory (ROM) size of 64 KB and Random Access Memory (RAM) size of 85 KB [144]. Finally, eLua is an embedded version of Lua [40] with the primary goal of providing a fully functional development environment running on the embedded device itself [34]. In contrast to other frameworks, eLua does not run

as an interpreter on top of the OS; instead, it runs on the bare-metal; resulting in eLua having a footprint of 64 KB [70] and only limited to platforms to which eLua has been ported. Porting to new platforms is possible; however, it requires massive development effort.

Other efforts such as Everylite are built from the ground up as a lightweight scripting engine; developed specifically to enable the concept of micro-tasks in IoT applications [70]. The interpreter has a small footprint of 37 KBs and performs very similarly to that of C [70]. Contrary to Everylite, JerryScript uses an implementation of IoT.js - "a framework for the internet of things" [45]. However, JerryScript requires 196 KBs of memory [70].

Texas Instruments (TI) developed Sensor Controller Studio (SCS), a development environment tailored for TI chips with an embedded sensor controller onboard. The premise of SCS was to provide developers with a simplified development environment to design peripheral logic for execution on the sensor controller chip [134]. Once development on SCS is complete, the tool generates embedded-C files that contain the firmware image to be flashed on the Sensor Controller chip [127]. Once the source files are generated, the developer would need to integrate and use the exposed functions in their embedded-C firmware code. At the time of writing our thesis, the Sensor Controller approach had several limitations, including the small number of supported microcontrollers and the lack of inter-thread communications. The latter is a significant performance bottleneck since data would have to be offloaded to the main Central Processing Unit (CPU) before another thread can consume it.

The SandBoxer concept built upon the sensor controller approach and theorized a lightweight kernel layer that isolates the different applications running on an IoT

device from the sensor drivers within. Additionally, it generalized the benefits of the TI approach to the entire node application rather than just the sensor logic. Although SandBoxer was designed with IIoT applications in mind, the concept can be expanded to create a lightweight scripting engine optimized for resource-constrained IoT applications [54].

Different scripting frameworks exist in both IoT and non-IoT realms. However, to our knowledge, none provide driver-level platform independence. Therefore, our work introduces a different VM architecture to enable platform independence for low-level drivers. Additionally, in contrast to most scripting frameworks, our proposed work requires the program to be pre-compiled into custom bytecode, thus, reducing the footprint requirements.

2.5 The Lack of Adoption of Plug and Play for IoT

Another pillar that lacks popular support in the IoT realm is PnP. Moreover, despite this no longer being a novel concept, the adoption of PnP for peripherals used with resource-constrained devices is limited. Since the early digital days, transducers have digitized the surrounding environment. One of the earliest and most basic sensing elements was used to detect the temperature of electronic circuitry and trigger a cooling fan as feedback. Such applications brought the sensing age as we know it and planted the seed of what is now known as IoT. However, the journey has not been without its challenges, as the diversity of transducer manufacturers made diverse hardware interfacing buses [64]. Standardization to mitigate the issue was met with disinterest from the industry, mainly due to overhead costs and limitations imposed on the manufacturer's side [64]. However, with the promise of zero configuration on the user side, the industry was again intrigued with the concept of PnP. A recent study suggests that the IoT road-map involves PnP for devices beyond the year 2025 [48].

In a typical IoT architecture, the perception layer involves several SeNs. Such devices are usually resource-constrained and deployed in large numbers in most IoT applications [133]. The SeN architecture consists of an application layer, a networking stack, and an OS layer as illustrated in fig 2.2. As is the case with WSN nodes, an IoT sensor node hardware interfaces with one or more transducers. The OS layer is responsible for communicating and exposing such transducers to the upper layers of the node through software applications commonly known as drivers. Due to the heterogeneity of transducer manufacturers and IoT platforms, no single standardized interoperable methodology exists to interface the two. Thus, transducer manufacturers provide their customers with a sample driver program (commonly written in C), which

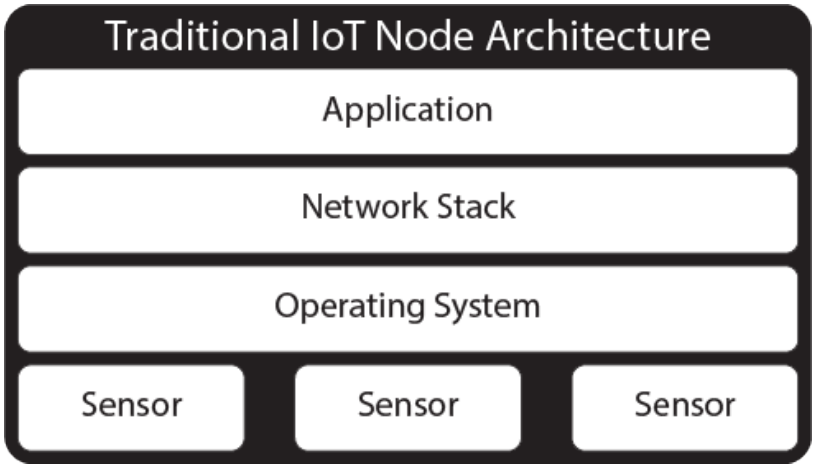


Figure 2.2: A basic IoT node system architecture

the IoT system developer must refactor to ensure compatibility with the developed platform. Such a development pipeline presents one of the key unresolved challenges in the perception layer [133].

Additionally, once an IoT system is developed, it is often very challenging to customize the designed node with different peripherals. This lack of customization is an integral aspect that is throttling the advancement of IIoT due to the need to redesign a platform to re-use it in a different application. Such redesign leads to a longer time-to-market and costly development and testing procedures [64]. When combined with the IoT paradigm, the concept of PnP enables a customizable platform that can adapt to different applications without the need for redevelopment. Despite the added simplicity and flexibility, PnP presents some substantial challenges to be considered feasible for the IoT paradigm. Towards this end, numerous solutions have been proposed; one such example is Universal Plug and Play (UPnP), a set of protocols developed by an industry consortium led by Microsoft. The protocols are used for service discovery to standardize the communication between control points [6].

Jini, on the other hand, improves on UPnP by using Java at its core, thus enabling platform-independence, an essential feature for IoT [6].

Despite the focus in the literature on the upper layers for PnP, research such as SPROUTS attempts to dig into the lower layers and provide a simple PnP functionality for sensors and IoT devices [64]. However, the sensor drivers are still required to be ported and tested by the IoT platform developers. Moreover, the security risks presented with architectures such as UPnP (although mitigable) pose a significant threat when integrated within the IIoT industry [1]. Therefore, the ultimate TPnP experience requires isolating driver logic implementation from platform-specific software. Preferably achievable through a platform-independent scripting language. Additionally, a PnP system needs to be hot-pluggable[1], offer a zero-configuration setup, have a low-energy footprint, and be compatible with the available interface ports.

2.5.1 Drivers and Interfacing

Regardless of the communication bus used, the current status quo requires the developer to write lengthy drivers for each transducer from each manufacturer. To simplify and modularize the development, system architects usually use a dedicated Microcontroller Unit (MCU) for each transducer that is specifically programmed to collect the transducer readings and communicate them to the main MCU on the central data collection device. Such a method can radically increase the development and production costs as the number of transducers increases. Furthermore, despite reducing the transducer driver development complexity, developing a new IoT system becomes

[1] A hot-pluggable peripheral can be disconnected and connected without interruption to the application or requiring a system reboot.

more complex and radically reduces the flexibility of using different transducers post-production. This complexity is due to the developer needing to reprogram the MCU every time another sensor is to be used [28]. Therefore, the ability to plug and unplug a transducer seamlessly and without a dedicated MCU would significantly reduce the development time and risk while allowing a more straightforward prototyping mechanism for researchers, accelerating research and development of IIoT.

2.5.2 Universal Serial Bus

Universal Serial Bus (USB) has been the standard for PnP functionality since its introduction in the early 1990's [83]. However, such technology did not spread into the IoT and embedded system realms due to its impracticality on such resource-limited platforms [102]. USB requires a dedicated chip/microcontroller on the sensor to be connected with the IoT system, introducing a higher level of complexity for both the manufacturer and the system designer [71]. Moreover, such a requirement increases the cost and energy consumption of using the USB interface. In order for a system designer to integrate USB interfacing, a driver must be developed for each separate sensor, given that the MCU on the IoT system supports USB natively [71].

2.5.3 IEEE 21451

The Institute of Electrical and Electronics Engineers (IEEE) registered a series of 21451 standards to add PnP capabilities to sensors and transducers. The standard proposed a unified bus for connecting analogue transducers by requiring an Electrically Erasable Programmable Read-Only Memory (EEPROM) to be attached to each transducer. Capabilities such as self-identification, self-description, self-calibration, and standard

data formats [72] were introduced. The EEPROM is to be loaded with a Transducer Electronic Data Sheet (TEDS) that is responsible for identifying the transducer as well as providing the Network Capable Application Processor (NCAP) with specific parameters to be used in calculating the sensed data. As with traditional datasheets, TEDS contains information about the manufacturer, the type of transducer, serial number, accuracy, measurement range, calibration data, and the supported data formats. Although still active, the standard was only adopted by a handful of sensor manufacturers. By design, the standard is more inclined toward analogue sensors than digital ones, hence, lacking the support of critical features such as interfacing protocol and sensor register descriptions [120]. Additionally, embedding a memory unit with the sensor was expensive at the time and impractical, given that memory units had far less capacity and were not as miniaturized as they are today.

The NCAP is to be developed by a system designer and out of the scope of the sensor manufacturer; hence, the manufacturer's signal conditioning and processing are not implemented. This standard intensifies the necessary development from the system designer's side while preventing the manufacturers from ensuring the intended implementation of their sensors [57]. To use the standard with a digital sensor, system developers are still required to write an interfacing driver program and use a dedicated MCU with the sensor [28]. Therefore, to render the standard more accessible for different types of sensors and transducers and enable TPnP functionality, it is essential to reduce the burden on system developers and move the driver program from the system to the sensor/transducer side. Additionally, eliminate the need for a dedicated MCU on the sensor side.

2.5.4 mikroBUS

As an alternative to the USB interface for embedded systems, researchers at MikroElektronika proposed a standard (mikroBUS) for add-on boards. The standard describes a pin configuration layout used by add-on board manufacturers, consisting of 2x8 female headers. MikroBUS is now a popular choice among manufacturers to support add-on functionality on embedded systems. However, to comply with the standard, sensor manufacturers must use the entire 2x8 headers in the add-on board, even if not all pins are required. Such requirement increases the cost and size of the sensors and adds a considerable overhead on the manufacturers. From the system designer's perspective, using mikroBUS limits the design options to platforms that support the interface or requires the designer to integrate the standard interface as part of the design process. Not only does this complicate the design procedure, but they also require the designer to write code for each board to be connected. Similar efforts include Arduino Shields [93], and Raspberry-Pi Hats [80].

2.5.5 Other Efforts

Efforts that aim to provide PnP functionality for IoT devices and embedded systems require an extra microcontroller to implement the communication protocol. In [99], the authors propose a modular PnP architecture technology that is similar to USB but optimized for embedded systems. In their research, they require each sensor or actuator to include a dedicated driver chip (microcontroller) to implement the communication protocol for the proposed interface bus. Their effort has the benefit of improvement over USB for embedded systems and IoT devices. However, the overhead incurred by the sensor manufacturers remains an issue.

Despite the advances in PnP technologies, current solutions fail to provide a method that has the simplicity of USB with the flexibility of bare-metal interfacing. Additionally, taking it a step further and enabling CI/CD pipelines by reducing the number of hours required to interface a new transducer in an IoT system. We believe that such a system must be able to provide a flexible interfacing bus for the manufacturers while also providing developers with a standardized and portable driver development solution.

2.6 Real-Time Operations

The ever-changing and demanding nature of the software industry has led to a significant rise in the need for post-delivery real-time support and operations. Such demand is no stranger for the IoT and IIoT industry. The consumer and business needs have evolved, requiring real-time feedback and operations on deployed IoT systems. To meet this demand, CI and CD became key pillars of every software development company [77]. Towards CI/CD support in the IoT domain, OTA reprogramming is essential.

2.6.1 Over-The-Air Reprogramming

Towards enabling a CI/CD pipeline, reprogramming IoT devices from the cloud is necessary, in fact the future of IoT relies on remote reprogramming [123]. However, once IoT sensor nodes are deployed, it is usually virtually impossible to regain physical access to perform maintenance and software updates [115]. Such challenge is the motivation behind remote reprogramming of the nodes, usually referred to as Over-The-Air (OTA) Reprogramming. Due to the limited energy capacity IoT sensor nodes have, it is crucial for OTA reprogramming to not just transmit as little data as possible but also transfer the data efficiently with minimal radio packet loss and idle wait periods. This challenge is out of scope for our work and has been addressed in other research efforts such as [5] and our previous efforts [53], and [52].

OTA reprogramming techniques are subdivided into System Level, Module Level, Differential, and VM [116]. All techniques except the VM require transferring large quantities of data over the radio and a reboot of the node. However, the VM technique demands significant CPU and memory capabilities and resources from

the node, rendering it unsuitable for IoT ventures [115]. Therefore, research efforts have naturally ignored the VM technique as a viable solution for the IoT space. An emerging new technique uses software containers as the modular element to update. However, this technique can be considered a variant of the VM technique due to the sandboxing nature of containers. Scripting Over-The-Air is one effort that utilizes such a technique using a JavaScript (JerryScript) based container [12].

The closest solution to the best of our knowledge is [13]. The researchers proposed a cross-layer solution combining cloud-based graphical programming with an RTOS running on a low-end IoT device, on top of which a containerized JavaScript environment is hosted. The architecture enabled scripts to be pushed over the internet to an edge device that forwards the scripts OTA to an end-device. However, the solution's performance remains unsuitable for resource-constrained IoT devices.

2.6.2 Networking Protocols

One key aspect often overlooked in the literature is the network layer challenges. Such as the packet delivery ratio that affects the real-time aspect of OTA programming [53]. Different Media Access Control (MAC) layer protocols are used to mitigate the issue. However, synchronous ones are prone to clock drifts [129]. Hence, most Time-Division Multiple Access (TDMA) protocols re-synchronize clocks between the different nodes and the Base Station (BS) [129]. This drift is caused by the dissimilar hardware used for the different types of nodes and the environmental factors in which this node operates. A case in point, the crystal oscillators of a node would cause a time drift of up to 0.18 seconds every hour [135].

2.7 Summary

Our primary goal is to accelerate the development of IoT platforms and solutions. In this chapter, we have identified and discussed the different key technologies and research that have the potential to act as a catalyst for development. We believe that enabling and advancing CI/CD in the IoT realm is critical to the maintainability and adoption of modern SDLCs. Furthermore, the concept of development sandboxing on resource-constrained devices has the potential to render IoT development to become platform agnostic and maximize code-reuse. In addition to containerization and sandboxing, to realize TPnP, a solution that offers the simplicity of USB with the flexibility of bare-metal is necessary. Finally, enabling virtualization of peripherals and IoT nodes enables the concepts of SDDKs and VDKs. Concepts that ultimately have the potential needed to skyrocket the development of new IoT solutions. However, current research on each technology outlined lacks the feasibility to be adapted for the IoT development paradigm. To this end, we propose PILoT to address the shortcomings of the current status quo.

Chapter 3

Platform Independent Library Of Things (PILoT) - A Platform-Agnostic Haven

In addition to supply-chain complications, the current IoT development paradigm employs methodologies inefficient for rapid prototyping and E2E modern SDLCs. If we are to witness the full I4.0 and I5.0 visions in the 2020s, a significant shift in the development process is necessary. However, to accelerate and democratize the development of IoT platforms, we believe that any proposed shift in the development paradigm must satisfy the following requirements.

1. **Virtualization of IoT hardware**: Current IoT development methodologies require access to IoT hardware very early in the process, therefore, the ability to virtualize IoT hardware can significantly accelerate the development.

2. **E2E CI/CD support**: Full E2E testing in embedded systems development remains challenging [50], therefore, solutions that enable modern SDLCs is necessary.

3. **Streamlined peripheral development**: Due to the diverse number of peripheral manufacturers, a large portion of a developer's time is wasted writing or porting peripheral drivers. In a platform-agnostic IoT world, the duty of driver development can be shifted to the manufacturer. A manufacturer develops the driver once for a specific peripheral, and all developers using the peripheral can use it with minimal development or porting time.

4. **True Plug and Play**: In addition to a more streamlined sensor development process, the developer still needs to develop an application that addresses each used peripheral separately. Developing an IoT application for a device that uses a LoRa® modem, for instance, restricts this device to this modem on the same hardware port and pins. Thus, rapid prototyping and platform reuse are severely restricted.

5. **Platform independence**: Code reuse is a core catalyst in the software development process; however, due to the diverse nature of IoT platforms, a large portion of developed applications are platform-dependent and, therefore, do not allow code reuse. Furthermore, achieving a high level of code reuse allows open-sourced and crowd-sourced applications to be utilized in the IoT paradigm.

6. **Contextual execution** Multi-application support and resource sharing have the potential to allow developers to quickly test applications on already deployed IoT infrastructure. Thus, reducing the risk and accelerating decision-making during development.

7. **Real-Time Operations**: Once an IoT device is deployed in production, it is generally very challenging to gain physical access to the device. Such constraint

poses a challenge for software updates and fault diagnoses. Additionally, the need for seamless zero-downtime and zero-code-change deployments is growing among IoT developers' wish lists.

We envision a world in which IoT hardware and software can be developed independently of each other. One in which an IoT hardware engineer can develop and test the drivers for their hardware peripheral long before their hardware design materializes. A world where software developers can use virtual IoT development kits locally or remotely in the cloud. Towards this end, we propose PILoT; a platform-agnostic library of IoT architectures and solutions that enable virtualized IoT platforms, virtualized peripherals, and an E2E CI/CD capable IoT development architecture. Furthermore, to address the growing need for hardware access, we believe that providing virtual customizable IoT development kits as a service has the potential to disrupt the industry. Our proposed work would provide developers with instant access to a development platform, thus, reducing time, cost, and risk.

3.1 Enabling Paradigms

The primary goal of PILoT is to accelerate and democratize IoT platform development while reducing business risks. Towards this end, we propose and discuss a number of enabling paradigms in the following sub-section. Towards satisfying the requirements for a democratized IoT platform development world, we propose a number of core paradigms in PILoT.

3.1.1 True Plug-and-Play (TPnP)

Towards streamlining peripheral development and satisfying requirement 3, we propose the concept of embedding the peripheral driver with its hardware; we dub this paradigm TPnP. In this paradigm, the responsibility of driver development is shifted from embedded system developers to peripheral manufacturers. This shift in development responsibility ensures that the peripherals operate as intended by the manufacturers. Additionally, the manufacturer can embed the driver in a memory unit on the peripheral, allowing a supported IoT or embedded platform to read and execute the driver seamlessly. The current status quo involves most manufacturers offering peripheral hardware for sale while providing a sample driver for free. The sample driver is most commonly developed for a single platform using C/C++. While the hardware usually consists of a sensing element or actuator embedded with an interfacing unit or control unit. Figure 3.1 illustrates an example PILoT temperature sensor. The sensing element remains embedded with an interfacing/control unit; however, we propose adding an embedded memory unit that hosts the drivers. The onboard storage enables a peripheral to be abstracted and isolated from the platform it would be connected to. Thus, a platform would not need to know about the peripheral

beforehand.

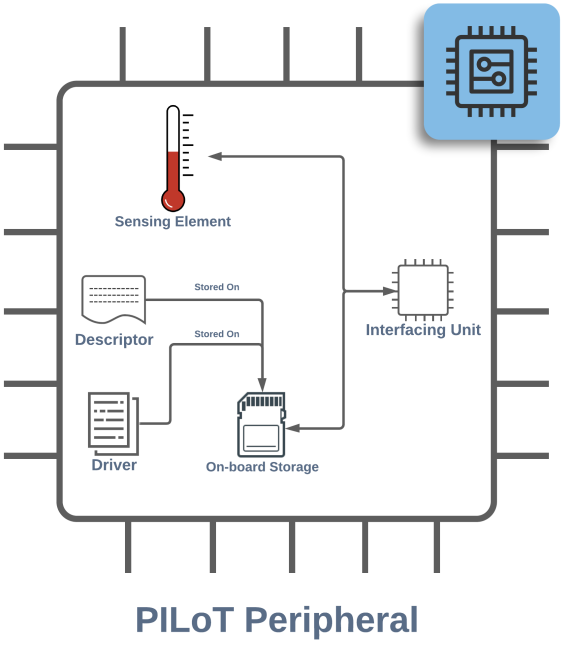


Figure 3.1: The proposed peripheral architecture in PILoT

Figure 3.2 illustrates how the the full vision of TPnP can be achieved with PILoT. A PILoT peripheral would communicate with an IoT node that integrates a universal bus. The driver and peripheral descriptor are read from the embedded storage upon connection with the node. Then the communication proxy unit creates a direct connection between the SoC and the peripheral using the interface found in the descriptor. Finally, the node can communicate directly with the sensing element on board using the appropriate communication protocols the same way it would if there was no universal bus.

Towards materializing this paradigm, we outline the following set of requirements.

1. Peripheral modules should include a memory chip that hosts the embedded driver and a logic chip to facilitate the TPnP logic.

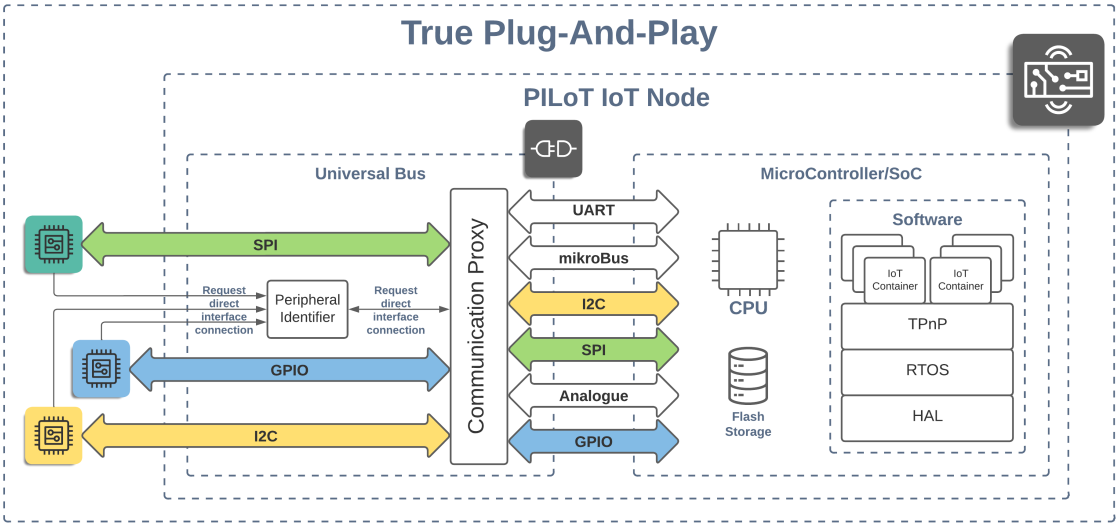


Figure 3.2: The True Plug-and-Play paradigm

2. The embedded driver should expose a set of interfacing functions. Namely, configuring, reading, writing, and controlling.

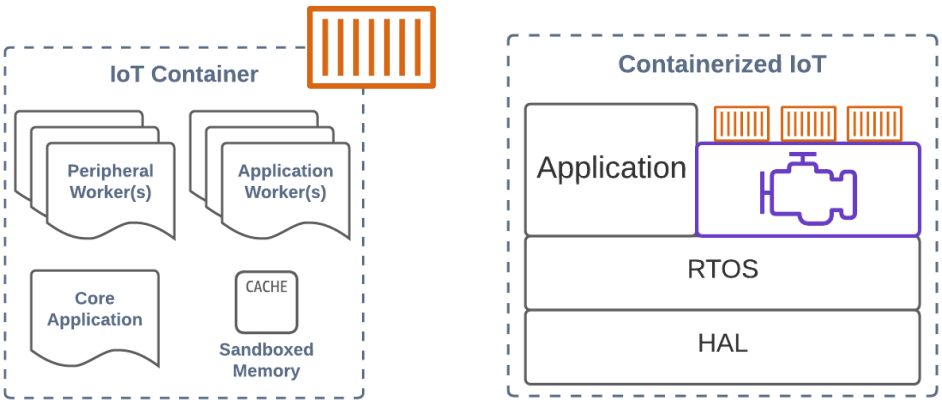
3. The descriptor should include several identifiers that can be used to correctly communicate and identify the peripheral.

4. To maximize adoption and reduce overhead, the TPnP system should remain backwards compatible with current peripherals and IoT nodes. Thus, developers and manufacturers can use the available stock of peripherals without rendering them obsolete.

5. The universal bus should be able to communicate with PILoT peripherals to read the contents of the onboard memory, then become transparent and provide a direct connection between the node and the peripheral.

3.1.2 IoT Containers

Towards enabling IoT development to a wider audience, and satisfying requirements 1 and 5, we propose the concept of IoT containers. This paradigm involves a docker-like container system tailored for resource-constrained devices. A container in this context is a collection of application software, hardware drivers, and operating system layers combined in a single low-footprint package deployed on an IoT sensor node. Just as Docker [8] accelerated innovation in the cloud computing space, we believe that bringing this paradigm to the resource-constrained IoT devices has the potential to democratize IoT development. Contrary to recent research, our proposal runs containers on IoT smart objects, not routers/coordinators. In addition to platform independence, IoT developers would be able to virtualize IoT nodes, allowing them to run an application on any machine/platform exactly as it would run on the real device.



(a) IoT Container (b) Containerized IoT Device

Figure 3.3: Visualization of the tree before and after correct deletion of the root node

Figure 3.3 illustrates how an IoT container runs as an application on top of the

OS layer of an IoT sensor node. We build on the functional programming paradigm and subdivide the IoT container into a single core application, one (or more) workers, and a sandboxed cache memory, as shown in figure 3.3(a). The core application in the proposed paradigm represents the high-level logic of the intended application and is responsible for coordinating between the workers. In contrast, a worker is an atomic unit of execution responsible for a single task. There would exist two types of workers, peripheral and custom. Peripheral workers control the peripherals connected to the device, commonly referred to as a device driver. When combined with our TPnP paradigm, the peripheral workers would get dynamically loaded when plugged in. On the other hand, custom workers are created by the IoT application developer and have limited access to the system resources.

To truly harness the power of this paradigm, we outline the following requirements.

1. The containers should be low-profile in terms of memory, storage and processing time.

2. The contained logic should use a development method that is simple to understand and use. Nevertheless, powerful to realize a wide range of applications.

3. The containerization engine should be able to run containers on any platform without the need to change the contents of the containers.

4. The containerization engine should ideally be platform agnostic.

3.1.3 Container Drop Continuous Delivery

Towards reducing the upfront risk associated with deploying an IoT system and satisfying requirements 6 and 7, we propose the Container Drop Continuous Delivery (CD-CD) paradigm. Alongside the IoT containers paradigm, this paradigm builds on the traditional OTA upgrades concept by allowing nodes to change the container/application to execute with no-reboot in real-time. Additionally, this flexibility plants the seed towards real-time customizability of applications, thus enabling supreme personalization towards I5.0.

To correctly implement this paradigm, we outline the following set of requirements.

1. The context switch must not cause a node to reboot.

2. The context switcher should fully utilize the onboard memory before storing the state on disk.

3. The containerization engine should report and record how much CPU, memory, and energy a container requires and consumes.

4. The context switcher should be able to utilize the resource usage and requirements for scheduling and switching containers. This behaviour should also be configurable by the IoT sensor node owner.

3.2 Envisioned Architecture

Combining the enabling paradigms allows PILoT to provide a catalyst for the development of IoT solution development. To better illustrate a user's journey when using PILoT, we subdivide the discussion of our architecture into four core phases (Design & Plan, Develop, Deploy, and Operate) that resemble an E2E development pipeline of an IoT solution. Each phase involves a number of key components that work in tandem to satisfy one (or more) of the requirements we outlined earlier in this chapter. Figure 3.4 illustrates a detailed outline of the different phases and capabilities involved in PILoT.

3.2.1 During Designing & Planing

In this phase, a PILoT user is crafting their future IoT solution architecture using available (purchasable/shareable) hardware or creating their blueprint for a new one. Once the hardware is planned, the user can design their IoT network architecture from the sensing layer to the cloud layer and beyond. The outcome of this phase is a fully-designed IoT application blueprint that will be used in the following phases.

The blueprint is the core element of any PILoT IoT solution. It abstracts the definitions of all the necessary components of the application. In a way, the blueprint is the IaC of a PILoT IoT project. On a high level, the blueprint consists of a list of locations where the solution is deployed. Within each location is a reference to a network definition and the Global Positioning System (GPS) coordinates of the location. The remainder of the blueprint includes a list of network definitions, a list of hardware definitions, and a list of policy definitions. Algorithm 3.1 illustrates a snippet of a high-level blueprint.

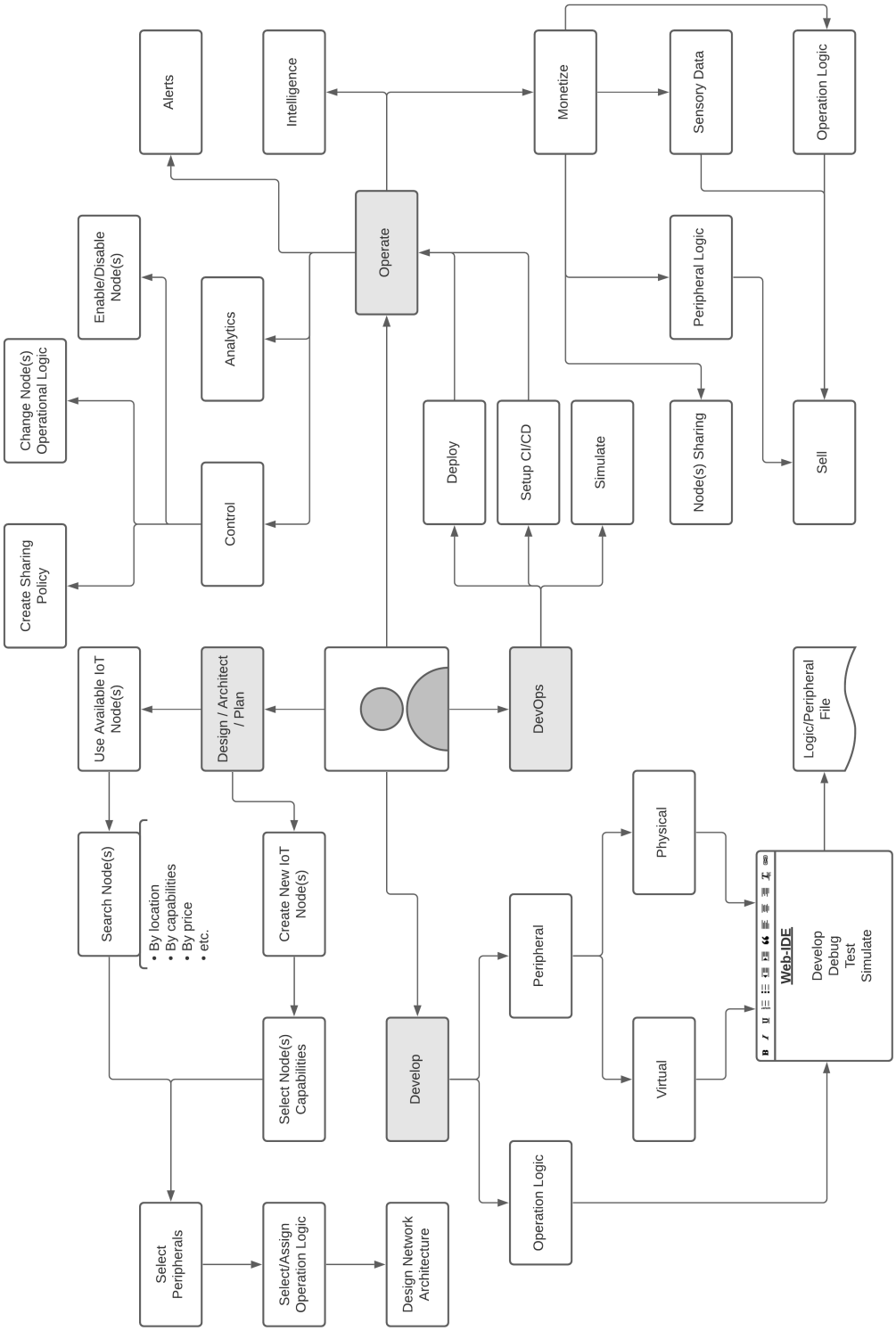


Figure 3.4: The PILoT staged architecture

Algorithm 3.1 High-level example of a PILoT blueprint

```
company: Company A
project: Super Secret Project
deployment_locations: ...
network_templates: ...
hardware_definitions: ...
policies: ...
```

Deployment Abstraction

The blueprint describes an IoT application and is used in PILoT to provision the necessary resources. During the design phase, the blueprint can reference virtual/physical elements already created or ones still to be created in the develop phase. The blueprint consists of a multilevel definition, with the highest level being *deployment_locations*. This field contains a list of geographical locations representing one or more network(s), either already deployed, to be deployed, or both. Algorithm 3.2 outlines an example *deployment_locations* section.

Such a level of abstraction enables a developer to plan (and re-design) the IoT system long before the hardware materializes. Furthermore, since a blueprint can dynamically change anytime throughout the lifetime of the IoT system, the business risk associated with ever-changing requirements is significantly reduced.

Algorithm 3.2 Example of a *deployment_locations* section in a PILoT blueprint

```
deployment_locations:
  - id: '242'
    name: Mississauga Factory
    coordinates: '43.68174702298701, -79.61782363586379'
    network_definition: standard
  - id: '331'
    name: Kingston Factory
    coordinates: '44.219338045350895, -76.55455735155752'
    network_definition: standard
  - id: '344'
    name: Ottawa Factory
    coordinates: 'Virtual'
    network_definition: advanced
```

Network Architecture Abstraction

The second highest level definition in the blueprint lists all the network configurations available to the application. This list does not specify actual deployments; instead a construct of different network deployment templates. This list of *network_templates* is what the *deployment_locations* field references in each geographical location. A single network template contains two lists, routers, and nodes. Each list contains a reference to a hardware template and a number representing how many of such templates are in the network. Abstracting networks in this manner enables flexibility in design and lower risk upfront.

Algorithm 3.3 displays an example *network_templates* section with two types of networks *standard* and *advanced*. The number of nodes in the *advanced* section is set to -1 to represent an unknown/unlimited number of nodes of type *sensor-node-unknown-features* that can connect to this network. However, the *standard* network template has a maximum number of 15 nodes of type *sensor-node-1*.

Algorithm 3.3 Example of a *network_templates* section in a PILoT blueprint

```
1  network_templates:
2    - name: downtown-building-network
3      routers:
4        - router: sink-1
5          count: 1
6      nodes:
7        - node: sensor-node-1
8          count: 15
9    - name: uptown-building-network
10     routers:
11       - router: sink-1
12         count: 1
13     nodes:
14       - node: sensor-node-unknown-features
15         count: -1
```

Algorithm 3.4 Example of a *hardware_templates* section in a PILoT blueprint

```yaml
hardware_definitions:
  - name: sink-1
    type: router
    features:
      - name: lora_trx
        type: radio_controller
        driver: lora_gw_trx
        parameter_overrides: sink1_lora_overrides
      - name: gsm_trx
        type: radio_reporter
        driver: huawei_gsm_32x
        parameter_overrides: sink1_gsm_overrides
    operational_logic: sink1_logic
  - name: sensor-node-1
    type: node
    features:
      - name: temperature
        type: sensor
        driver: temperature_tsys01
        parameter_overrides: node1_temp_overrides
      - name: heater
        type: actuator
        driver: heater_custom
      - name: lora_trx
        type: radio_reporter
        driver: lora_node_trx
        parameter_overrides: node1_lora_overrides
    operational_logic: node1_logic
  - name: sensor-node-unknown-features
    type: node
    operational_logic: node1_logic
```

Hardware Abstraction

The blueprint's lowest and most powerful level definition is the *hardware_templates* section. Despite offering the lowest level of abstraction, this section is considered the core of any PILoT blueprint. The *hardware_templates* field contains a list of virtual or physical hardware devices that represent IoT sensor nodes or routers. Each entry in the list defines a template for a hardware device containing an optional list of features this device supports. In addition, each node or router template includes a reference to the operational logic software it will execute.

As illustrated in Algorithm 3.4, a feature entry contains the name, type, driver, and parameter overrides of a specific feature. A feature in the context of a hardware

template represents a peripheral connected to the hardware, such as sensors, actuators, or communication peripherals. The type field in the feature entry outlines how this peripheral will be used in the application; for instance, a radio peripheral can be used to report data to the next layer in the IoT architecture or to receive commands from the upper layers. The example algorithm defines *sink-1*, a router that communicates with the sensor nodes using a LoRa® radio and the cloud back-end using a 4G/5G radio. The types for these radios are defined as *radio_controller* and *radio_reporter* respectively. The type field can contain one of the following pre-defined values.

- radio_controller
- radio_reporter
- sensor
- actuator
- screen

Algorithm 3.5 Example of a *network_policy* section in a PILoT blueprint

```
policies:
  - name: flexible_network_policy
    type: network_policy
    allow_node_registration: true

  - name: fixed_network_policy
    type: network_policy
    allow_node_registration: false
    known_nodes:
      - D0E45C4A-56FA-11EC-8683-FE4AFB1BFCCB
      - D0E45D4E-56FA-11EC-8683-FE4AFB1BFCCB
      - D0E45D76-56FA-11EC-8683-FE4AFB1BFCCB
      - D0E45DA8-56FA-11EC-8683-FE4AFB1BFCCB
      - D0E45DB2-56FA-11EC-8683-FE4AFB1BFCCB
```

Rules & Policies

Although this section is not required during the design & plan phase, it abstracts the policies to be defined in future phases of PILoT. There are two types of policies supported, *network_policy* or *sharing_policy*. An example of a network policy is outlined in Algorithm 3.5. In the example, a network policy allows any node to join the network, while the other policy defines a set of known nodes allowed to connect.

3.2.2 In Development

In this phase, PILoT allows users to actively develop and test their application logic using (crowd-sourced) pre-developed application logic, develop their own, or a combination of both. Additionally, the user can develop the drivers for the hardware peripherals using virtualized hardware. The two core types of development in this phase are application and driver development. Traditionally, this development requires physical access to the hardware devices, which significantly limits the cadence of platform developers. However, PILoT's develop phase allows the execution of code on virtual hardware devices allowing developers to iterate quicker than the traditional methodology. Figure 3.5 illustrates how the develop phase works.

Towards enabling hardware virtualization, we adopt the concept of IoT containers. A single container would encapsulate both application and driver logic. The blueprints created in the design & plan phase are used to map the virtual hardware to the underlying physical hardware; each container is allowed access to this mapping.

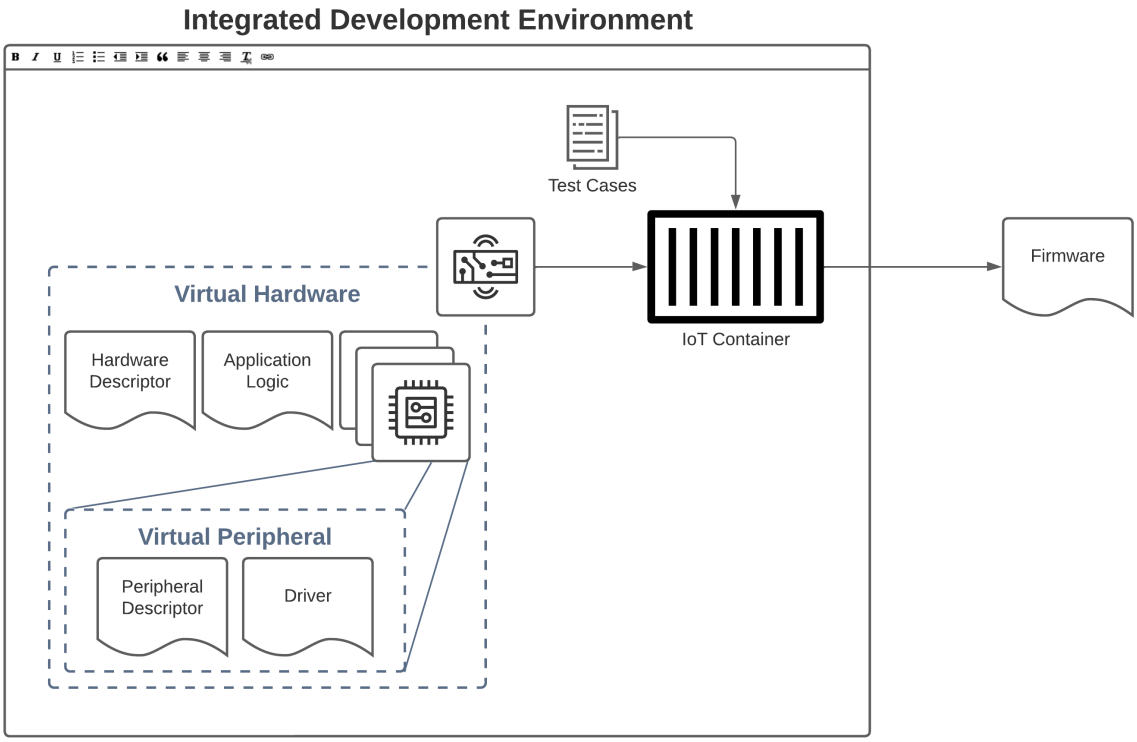


Figure 3.5: PILoT's develop stage example

Container Execution Model

As discussed earlier, an IoT container is tailored to run on the devices in the lowest layer of an IoT architecture, namely, the IoT sensor nodes. The execution model of a container starts with the core application controlling several workers. The core application would run on the main thread and spawns a separate sub-thread for each worker. Each execution loop processes one instruction on the main thread, followed by the complete execution of all sub-threads. Therefore, it is essential that sub-threads are short running or periodically yield.

The containerization engine should be able to isolate containers while allowing multiple containers to execute simultaneously with minimal context-switching overhead.

The engine would include an interface that is platform specific. This interface would be the only component required to be ported to new platforms, with everything else being platform agnostic. Traditionally, developers port all drivers and application logic in addition to the HAL for new platforms; however, using PILoT, they would be required to only port the interface in addition to the HAL. In addition to standard containerization engine components, Fig. 3.6 illustrates the core components envisioned for our engine.

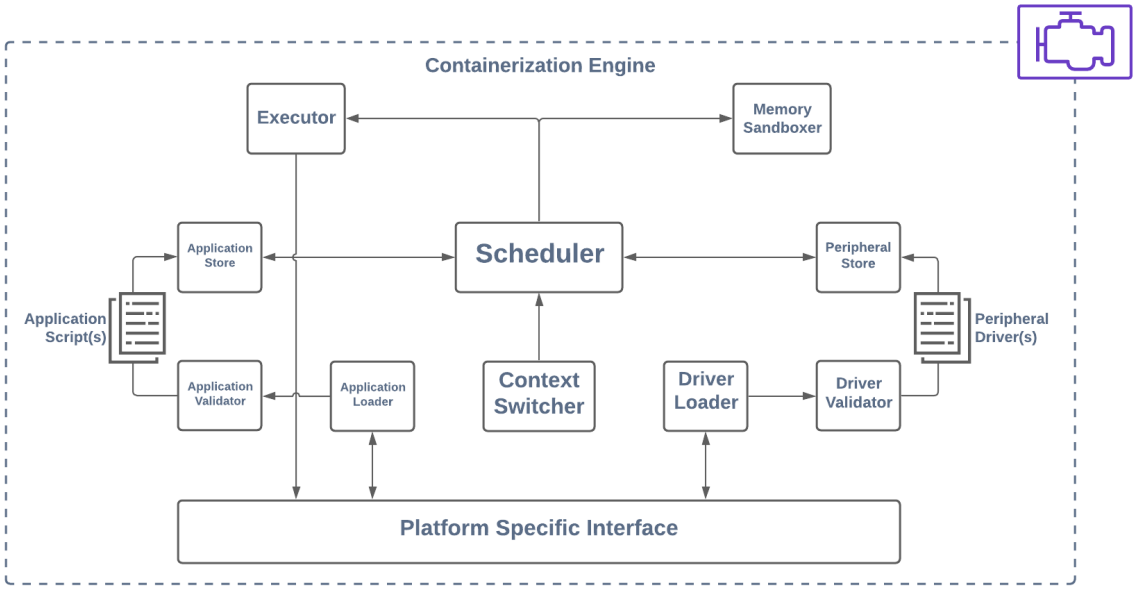


Figure 3.6: The containerization engine

The Core Application

This is the brain of an IoT container; it can access the resources for all the workers within the container and communicate outside of the container. The application itself is very high-level and is unaware of the connected peripherals. Instead, it is provided with a list of peripheral workers to control. A developer can then completely develop

the application logic without knowing the exact peripherals connected beforehand. An example algorithm is illustrated in 3.6. In this example, the application will initialize all workers, proceed to process all workers infinitely, then sleep for two seconds. The *ProcessAllWorkers* step would execute the *Process* worker on each peripheral. At the end of each worker, the core would determine whether or not to reschedule the worker again for the next execution cycle.

Algorithm 3.6 A simple generic PILoT core application

```
core:
  - name: GENERIC_IoT_EXAMPLE
    worker_ttl: 2000
    code: |
      InitializeAllWorkers();
      RunForever{
        ProcessAllWorkers();
        SleepForSeconds(2);
      }
```

Peripheral Workers

Peripheral workers are a special type of workers that would be dynamically loaded from the peripherals connected to the device. Alongside the peripheral descriptor, we expect manufacturers/developers to provision at least the following workers for each peripheral. *Initialization*, *Processing*, *Read* and/or *Write*. Optional workers include *Sleep*, *DeepSleep*, and *WakeUp*. Algorithm 3.7 outlines an example driver for an Light Emitting Diode (LED) strip containing three colours, red, blue, and yellow. The *Initialization* worker simply prepares the pins and sets some variables. While the *Write* worker uses the pre-defined variable values to change the output on the LEDs. The more complex *Processing* worker runs forever; however, for each iteration, the worker would flip the status of each LED before sleeping for one second.

Algorithm 3.7 An example PILoT driver for an LED strip

```yaml
peripheral:
  - name: LED_STRIP
    class: DIO
    product_id: 223432
    vendor_id: 553421
    certificate: 352F5B0BCBFA6E1ABEE263927DB42D40
    pins:
      - name: red_led
        type: GPIO_PIN
      - name: blue_led
        type: GPIO_PIN
      - name: yellow_led
        type: GPIO_PIN
workers:
  - type: "Initialization"
    code: |
      SetOutput(red_led)
      SetOutput(blue_led)
      SetOutput(yellow_led)
  - type: "Processing"
    parameters_in:
      - redStatus: OFF
        blueStatus: OFF
        yellowStatus: OFF
        sleepDuration: 1
    code: |
      RunForever{
        redStatus = Inverse(redStatus)
        blueStatus = Inverse(blueStatus)
        yellowStatus = Inverse(yellowStatus)
        PutGPIO(red_led, redStatus)
        PutGPIO(blue_led, blueStatus)
        PutGPIO(yellow_led, yellowStatus)
        SleepForSeconds(sleepDuration)
      }
  - type: "Write"
    parameters_in:
      - redStatus: OFF
        blueStatus: OFF
        yellowStatus: OFF
    code: |
      PutGPIO(red_led, redStatus)
      PutGPIO(blue_led, blueStatus)
      PutGPIO(yellow_led, yellowStatus)
```

Custom Workers

In contrast to *peripheral workers*, custom workers are written and installed by the developer in the same way as the core application. However, contrary to the core application, each custom worker has sole responsibility and limited access to the application memory. We expect the everyday use of custom workers to be for pre-processing sensory data before relaying it to the network router. An example of an application with a custom worker is illustrated in algorithm 3.8. The application represents a typical IoT sensor node algorithm, reading a sensor, preparing the data, then reporting it back to the router. This application assumes at least one sensor peripheral and one radio peripheral being connected. The custom worker is responsible for preparing the data to be sent/reported back to the IoT router; it first loops through all connected peripherals and then adds the sensor name and data to the data to report array.

Application Development Language

The develop phase's core is writing/implementing the application, peripheral, and custom workers. We believe using byte-code-based VM programming languages to execute the IoT containers would help achieve such requirements. However, the primary challenge with such languages is the high memory and processing footprint associated with running the VM. In the resource-constrained world of embedded systems, such virtualization is not always feasible and requires high-end devices. Popular VM-based languages (e.g. Java) are usually feature-bloated for use-cases in the embedded realm; in turn, embedded versions of some languages were created to overcome this limitation (e.g. eLua, uPython). Using one such embedded language (or the creation of a tailored

Algorithm 3.8 A complete IoT PILoT application

```
core:
  - name: COMPLETE_IoT_EXAMPLE
    worker_ttl: 2000
    code: |
      Initialize_All_Workers()
      RunForever{
        Read_All_Workers
        Process_All_Workers()
        PREPARE_DATA_REPORT
        Wirte_All_Workers()
        SleepForSeconds(2)
      }
workers:
  - name: PREPARE_DATA_REPORT
    parameters_out:
      - DataToSend: []
    code: |
      DataToSend = []
      ForEach(peripheral in Connected_Peripherals)
      {
        If (peripheral.class is DIO or AIO)
        {
          DataToSend.Insert(peripheral.name)
          DataToSend.Insert(peripheral.sensor_data)
        }
      }
```

language) would satisfy the platform-independence requirement and lower the effect of the challenges brought by traditional VM-based languages. Despite the language used for the VM, a compiler can be created to translate any high-level language to the underlying byte-code; thus, simplifying the development of applications and drivers for PILoT.

Furthermore, peripheral vendors can develop drivers in this language and embed them in their peripherals' memory, hence, facilitating the use of PnP. During development, a developer can test their code on any platform due to the platform-independent property inherited from using a VM-based language. Finally, applications can be exported as files and transferred to physical devices and are expected to operate with minimal-to-no alterations.

3.2.3 For Deployment

Traditionally in this phase, the IoT solution is deployed in the field. Depending on the application, the IoT nodes would be manually installed on-site, pre-installed in equipment, or dropped off an airplane in massive numbers. Such installation methodology insinuates that the software is ready and will not be updated after installation due to the difficulty of physically reaccessing the sensor nodes. However, using PILoT, once the IoT solution is designed and the application logic developed and tested, the deploy phase allows the user to define CI/CD logic that automates deployment and testing on virtual or physical devices. In addition to being able to run scenario simulations on a virtual test-bed of IoT devices. It is necessary to support efficient OTA upgrades towards achieving such automation. With sandboxed IoT containers supported on resource-constrained devices, it is possible to realize a different perspective of OTA upgrades (CD-CD).

Using containerized applications, the core underlying OS executes the containerization engine. Thus, upgrades to the containers would not require a node reboot. Furthermore, with the containers executing the application logic as VMs, the application code is represented as low-footprint byte-code scripts that can be efficiently transferred over a wireless link. Another requirement for CD-CD was the ability to upgrade with zero downtime. With the execution model described earlier, the containerization engine can switch contexts between containers with minimal overhead; an upgrade can be pre-loaded in memory and swapped with the current running version only when ready.

Multi-context Node Installation

In applications with different environments or execution contexts, creating a different IoT node hardware for each one is common. Such diversity of hardware is due to the different sensors used in each context. For instance, use acoustic communication modules for nodes submerged under water and LoRa® radio modules for nodes floating on the surface. Another example is using a Carbon Monoxide (CO) sensor in one shipping container and a humidity sensor in another. Such a challenge is trivial in some applications but can be an ordeal in others.

An application such as shipping containers involves a massive number of different sensors to be used; therefore, reusing the same hardware with different sensors is necessary. Traditionally, reusing the same hardware involves a bloated unified firmware to be installed that supports all possible future sensors to be used. However, with PILoT's TPnP, the sensors would encapsulate the driver logic and can be dynamically switched without the need for a bloated firmware on the node. Furthermore, such an application can require different sensors depending on the container's stage in the shipping journey. For example, a tamper-detection module might be required when the container is on board a freight ship. However, once the container is delivered, the tamper module would no longer be needed; instead, a personnel-access module would be installed. This switch of modules will be performed with zero downtime and no reboot if they support PILoT's TPnP.

CI/CD Pipeline

During development and after deployment, developers can automate firmware upgrades. Such an automated pipeline is traditionally very challenging due to the hardware

access requirement in the current status-quo. Using PILoT, CD-CD upgrades enables developers to automate firmware upgrades. Additionally, virtual IoT hardware allows a CI pipeline to run the automated system, scenario, and integration tests. An example of how testing works on virtual hardware was illustrated in Fig. 3.5; to automate this process, a developer would add the standard source-control hooks into their repository, and the tests would trigger automatically. Additionally, the output firmware can be added to the pipeline artifacts, offloaded to a cloud storage bucket, or installed directly on a physical device. Figure 3.7 illustrates an example pipeline using PILoT.

3.2.4 In Long-term Operate

Once an IoT solution is deployed, this phase is where system owners and operators spend most of their time. The operate phase allows the user to monetize their deployed IoT hardware, run business intelligence on the collected data, set up alerts, view analytics and control their deployed fleet of IoT devices. The two core requirements of any IoT solution is to collect & view sensor readings and run business intelligence applications on the data collected. On top of the traditional IoT solution capabilities, PILoT expands on them to enable resource sharing and system control. The majority of the operate phase is deployed and executed on cloud infrastructure. Figure 3.8 illustrates an overview of the envisioned cloud architecture. We discuss the capabilities and their architecture in the following subsections.

Monitor

The ability to monitor the IoT system involves storing and displaying the collected data. The micro-services deployed include a sensor data access service, a dashboarding

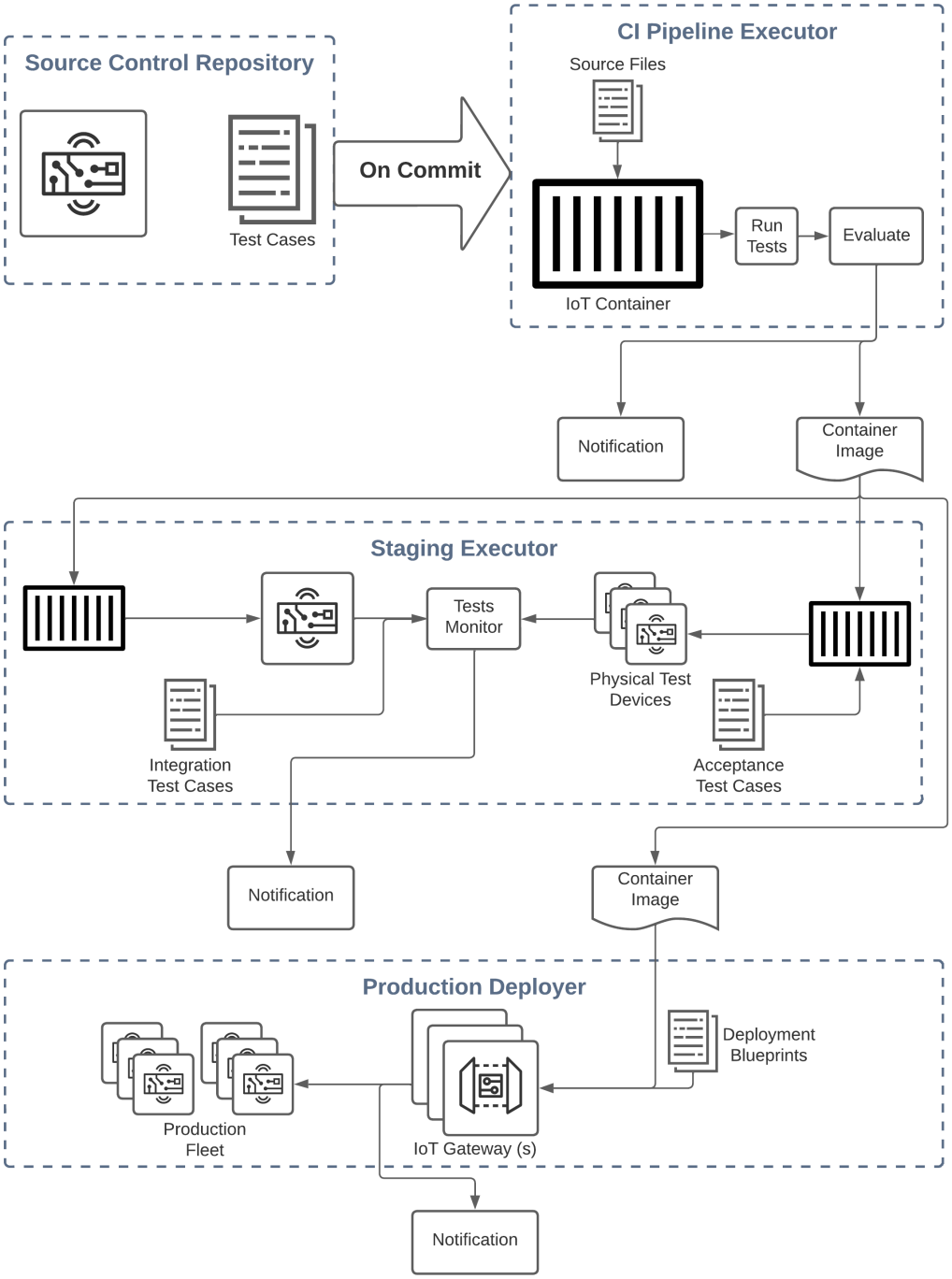


Figure 3.7: An example PILoT CI/CD pipeline

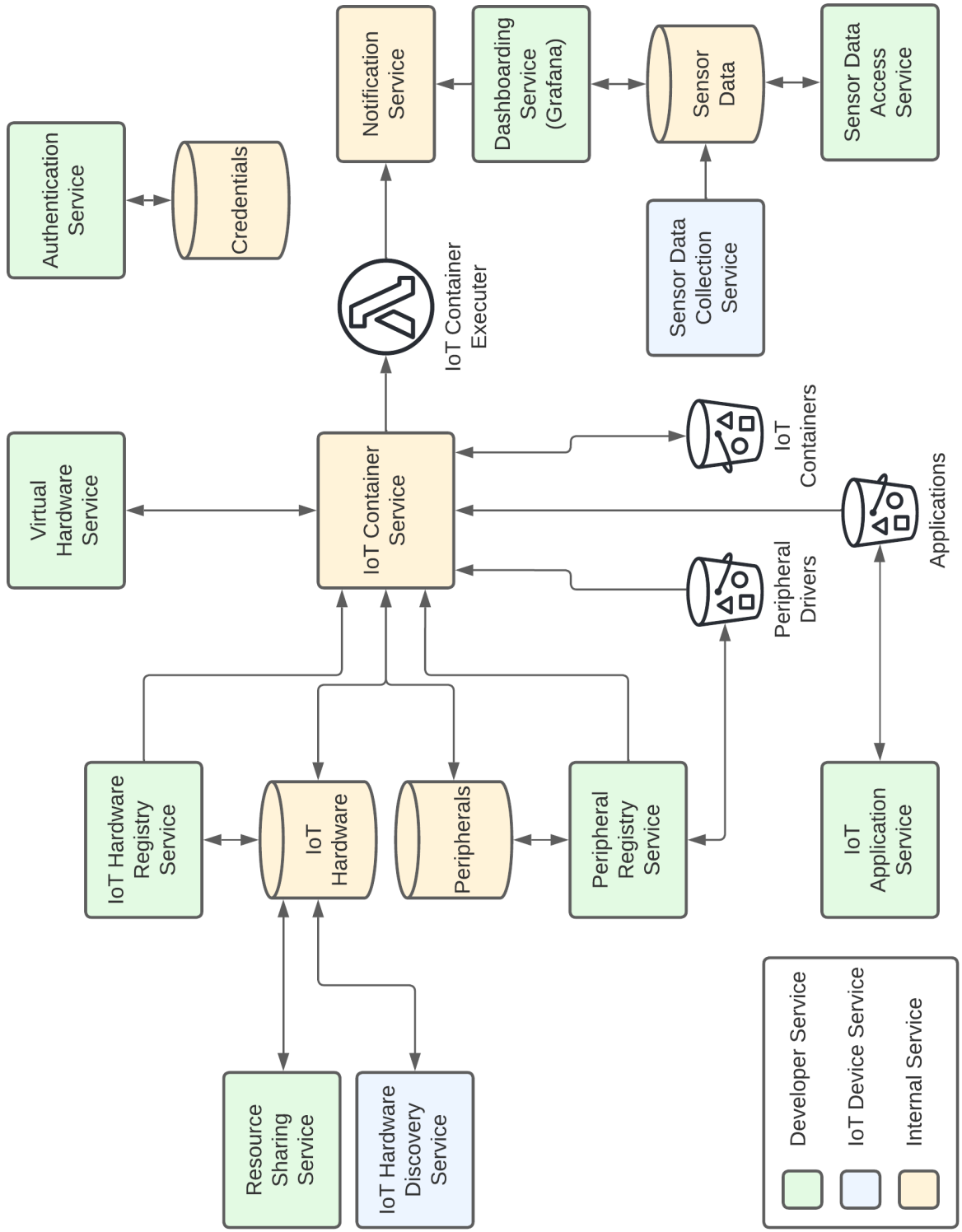


Figure 3.8: An overview of PILoT's cloud service architecture

service, and a notification service. The sensor data service enables the developer to expose Application Programming Interfaces (APIs) from PILoT for consumption by a mobile/web application. In contrast, the dashboarding service allows developers and users to create and view IoT data analytics dashboards, in addition to setting up access control and notifications/alerts based on different data rules and thresholds.

Learn

In addition to viewing a dashboard with the collected sensory data, developers can run machine learning algorithms against the sensor data and hook them into changing a specific node's operation. Furthermore, the marketplace provides data engineers access to a crowd-sourced library of IoT data models.

Monetize

Traditionally, once a system reaches the operation phase, monetizing and changing the operation of the IoT fleet is often very challenging. However, PILoT allows system owners to share their deployed IoT hardware, their developed logic/drivers, or their collected sensor data on a public marketplace. This capability provides IoT system owners with an additional avenue of income, thus reducing operational, development, and maintenance costs.

Control

Finally, developers and operators can remotely control the deployed nodes and re-program them in real-time OTA. Control involves the ability to put nodes to sleep, disable/enable features, switch active containers, and reboot a specific node.

3.3 A World With PILoT

PILoT aims to become a primary catalyst for IoT solutions development, providing benefits such as backwards compatibility with previous paradigms, lower-risk and shorter time to market, TPnP, and platform independence. Different industries and applications can harness these benefits; however, the true potential can be demonstrated with real-world use-cases. Towards this end, we outline and discuss different applications spanning multiple industries and different levels of expertise in the field.

In the following subsection, we discuss an application that can benefit from integrating PILoT. We outline the requirements, the options available, and how the traditional approach would be used before we discuss how PILoT would be integrated.

3.3.1 Peripheral Manufacturing

An embedded peripherals manufacturer intends to design a new version of their radio module line to use the latest modulation chip (LoRa®). This manufacturer has previously been a leading provider of Bluetooth and Zigbee radio modules. Manufacturing a LoRa® based module would be a foreign yet eerily familiar venture. The plan is to create a module that simplifies the usage of the Semtech SX1262 Integrated Circuit (IC) [114]. Towards this end, the IC would be installed alongside the required components to drive the application of the radio. In addition, once the module hardware is finalized, the manufacturer plans to create a sample application (or driver) to facilitate the integration and usage of their new module. Finally, the manufacturer employs several tests to validate the proper operation of the designed module.

Traditionally, the initiative would follow an agile project planning methodology,

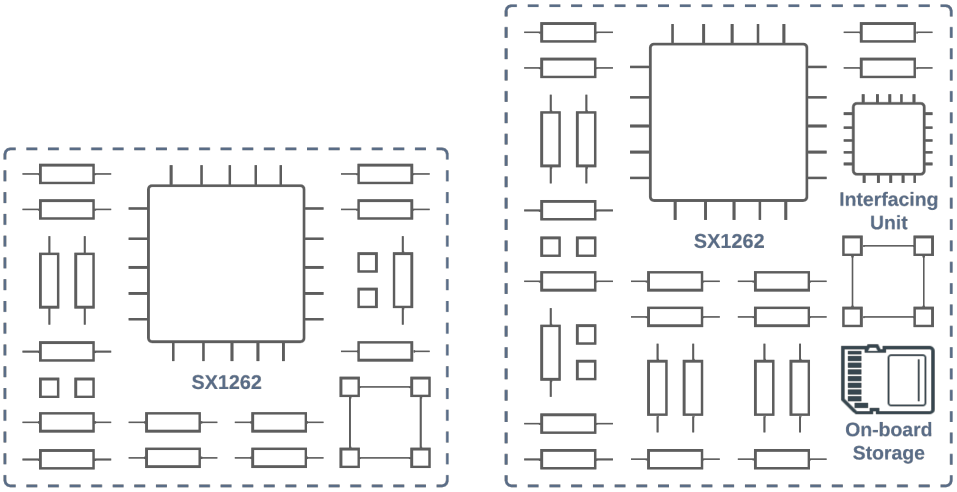
with multiple iterations of the hardware and software shipped to customers. Furthermore, the customer/developer using the module would need to acquire and refactor the software to support the underlying platform and application use case. This refactor introduces a potential risk of misimplementation of the original software and causing the module to malfunction; in turn, it directly affects the manufacturer's reputation and the new module while risking the user's system's failure.

On the journey towards finding a solution to the software misimplementation risk, the lead engineer on the project came across research introducing the PILoT paradigm. Intrigued by the benefits of reaching a wider audience and simplifying user integration, the engineer decides to study the implications of supporting PILoT. After further investigation, the engineer identified the following key findings on the project's application of PILoT.

- Zero additional hardware effort is required to support a minimal version of PILoT.
- An extra number of hardware elements are required to support PILoT's TPnP functionality.
- A PILoT driver can either be generated from the traditional C-based driver or using the PILoT cloud-based User Interface (UI).

Considering the previous findings, the engineer presented the following proposals to their manager. Figure 3.9 illustrates an abstract example of the hardware differences between the two proposals.

1. Follow the traditional approach to create the module and create a C-based software/driver suitable for conversion to a PILoT-based driver. The pros and



(a) Traditional Approach Module (PILoT compatible) (b) PILoT's TPnP Compatible Module

Figure 3.9: The possible hardware design options for the module

cons of this proposal are outlined as follows.

- ✓ The team uses the traditional approach in designing both the hardware and the software.
- ✓ Lower risk in project management.
- ✓ The shipped product supports both traditional and PILoT-based systems.
- ✓ Users/developers can use the module with PILoT supported systems for a low-risk and shorter time to market.
- ✓ The manufacturer is able ensure correct operation of the module by users/developers using the PILoT driver.
- × Additional work is required to ensure the C-based software/driver is suitable for conversion.
- × Additional software is introduced and must be maintained by the team

(PILoT driver).

2. Design a PILoT-TPnP compatible module that supports both traditional and PILoT users/developers. The pros and cons of this proposal are outlined as follows.

   - ✓ The shipped product supports both traditional and PILoT-based systems.
   - ✓ Users/developers are able to use the module with any PILoT based system with near-zero effort.
   - ✓ The manufacturer is able ensure correct operation of the module by users/developers using the PILoT driver.
   - × Additional work is required to create a TPnP-compatible hardware module.
   - × The additional hardware components might affect the size and energy footprint, as well as the cost of the final module.
   - × Additional software is introduced and must be maintained by the team (PILoT driver).

Hardware Design

Given that PILoT is backwards compatible with all peripherals, zero hardware considerations are required to support PILoT in the first proposal. Once the hardware is designed and finalized, the team can create the PILoT hardware descriptor as described in algorithm 3.9. However, since the hardware is not designed to support TPnP, the descriptor would be provided to the user alongside the software drivers.

Alternatively, the second proposal involves a duo of core components to be included in the hardware design: a storage unit and an interfacing unit. The storage unit is

Algorithm 3.9 The hardware descriptor of the first proposal

```yaml
peripheral:
  - name: LORA_MODULE
    class: RADIO
    product_id: LORAMS1262
    vendor_id: 422311
    certificate: 352F5B0BCBFA6E1ABEE263927DB42D40
    pins:
      - name: dio_0
        type: GPIO_PIN
      - name: dio_1
        type: GPIO_PIN
      - name: dio_2
        type: GPIO_PIN
      - name: radio_reset
        type: GPIO_PIN
      - name: miso
        type: SPI_MISO_PIN
      - name: mosi
        type: SPI_MOSI_PIN
      - name: clk
        type: SPI_CLK_PIN
      - name: cs
        type: SPI_CS_PIN
```

essential to host the module blueprints, including the hardware descriptor and module drivers. The interfacing unit implements the PILoT-TPnP protocol and multiplexes between the storage unit and the SX1262. The same hardware descriptor outlined in algorithm 3.9 would be used in this solution; in addition to being provided to the user online alongside the software drivers, it would also be hosted on the storage unit.

Software Design

Traditionally, manufacturers create software samples in C and provide them online through their website or a source-control platform such as Github. The samples provided are usually developed and tested on a single platform and must be refactored/ported to other platforms by the user/developer.

Both proposed solutions involve creating the samples/drivers in C and creating the PILoT version. When designing the driver, manufacturers are usually presented

with the following two design philosophies:

1. Bare bones driver, also known as a low-level driver.

2. Application ready driver, also known as a high-level driver.

The first design philosophy entails providing a driver that features pure interaction with the peripheral, such as manipulating the registers on the peripheral directly. This option requires a high level of understanding of how the peripheral works; users/developers would need to understand and refer to the datasheet before using or porting the driver. However, this style of driver design empowers the user to control the full functionality of the peripheral. The manufacturer would provide a driver that exposes functionalities such as Reset, Write Register, and Read Register. One example of such design philosophy is the Semtech SX1262 driver provided for the IC [75]. In this case, the PILoT driver implementation is straightforward; however, it would not realize the full potential of PILoT's intended design. Algorithm 3.10 outlines how this driver could be implemented.

The second design philosophy advocates simplicity over flexibility. The driver abstracts the low-level interactions with the peripheral through high-level interfaces that expose the desired functionality of the peripheral. This option requires minimal investment in understanding the internal workings of the peripheral and more on how the peripheral aids the application. A well-designed driver using this style can provide the user with very high flexibility (similar to the first option). In this case, the manufacturer would expose the functionalities through the HAL interface, Initialize, Transmit, Receive Blocking, Receive In Background, and Set/Get Register. Unfortunately, this style of driver design requires a more considerable upfront investment on the manufacturer's side, in addition to a more complex driver software to maintain.

Algorithm 3.10 The PILoT bare-bones style driver

```
workers:
  - type: "Initialization"
    code: |
      SPI_Settings(125000, MSB_FIRST, SPI_MODE1)
      SetOutput(radio_reset)
  - type: "Read"
    parameters_in:
      - register_addr
    parameters_out:
      - register_data
    code: |
      # read 16 bits
      register_data = READ_SPI_REG16(register_addr)
  - type: "Write"
    parameters_in:
      - register_addr
      - register_data
    code: |
      # write 16 bits
      WRITE_SPI_REG16(register_addr, register_data)
  - type: "Custom"
    id: "Reset"
    code: |
      # Turn off the radio
      PutGPIO(radio_reset, OFF)
      # Sleep for 20 milliseconds
      SleepForMS(20)
      # Turn on the radio
      PutGPIO(radio_reset, ON)
      # Sleep for 10 milliseconds
      SleepForMS(20)
```

Manufacturers usually take it a step further and implement a minimal sample that uses this driver style; one example is the LoRaMac-node driver provided by Semtech [74]. This driver design requires a more complex PILoT driver implementation; however, it utilized its full potential to accelerate and simplify the user's development.

Discussion

Regardless of which approach the manufacturer decides to use (full/partial PILoT support), the benefits reaped in the process greatly outweigh the extra effort required. With the partial support approach, the manufacturer would follow the traditional development methodology they always used; the only difference is how their software

driver would be implemented. For the C-based driver to be suitable for conversion to a PILoT driver, the implementation should expose an API layer. Although this approach might be foreign to some manufacturers, it is a standard approach toward reusable device drivers [20].

3.3.2 Industrial Internet of Things

A global shipping firm plans to employ an IIoT solution to keep track of shipping containers. The firm contracts an IoT solutions company to design the customized system. We discuss the case study in detail in 6.

3.3.3 Remote Virtual Development Kits

Development kits are the de facto standard for embedded system development. Almost all sensors, peripherals, modules, and platforms have a development kit available by the manufacturer. Such kits allow developers to prototype and test a specific item's work before integrating it within the hardware platform. Even though development kits are usually more expensive than buying the item directly, a kit allows the software developer to start using the module before the hardware engineers integrate it into the platform. Not only does this save time, but it also reduces risk; if a developer uses a kit and finds that the module is not suitable for their application, they would look for alternatives without integrating it into their hardware.

Although development kits come with multiple benefits, it is usually time-consuming and costly to order a development kit for each module in the system. The current delays with the supply chains significantly increase this time exponentially. PILoT enables the concept of Virtual Development Kit as a Service (VDKaaS) through

SDDK. We envision such a service to build on top of PILoT's cloud architecture and provide a playground for a peripheral or platform in development.

Peripheral Development Kit

For instance, building on the application discussed in Section 3.3.1, we discuss how the manufacturer can build a SDDK for their peripheral as follows.

1. Upload the hardware descriptor of the peripheral to PILoT.
2. Upload the peripheral driver to PILoT.
3. Create and upload a hardware descriptor for the development kit to PILoT.
4. Create and upload a sample application for the development kit to PILoT.

A potential physical module purchaser can now test and virtually use the peripheral on PILoT in minutes. If the developer is happy with the peripheral, they can integrate the virtual peripheral into their PILoT system while the physical peripheral arrives in the mail. During the shipping time, the developer can develop, test, and virtually use the peripheral within their application with zero-physical access to any hardware.

3.3.4 Introduction of New Business Opportunities

Since PILoT redefines the IoT platform development process, it introduces many business opportunities. For instance, one opportunity is building a business responsible for creating a PILoT-TPnP silicon chip that facilitates full implementation of PILoT by manufacturers. Instead of a manufacturer adding a storage and interfacing unit, in addition to programming them, they would instead buy, configure, and install the

TPnP chip. Porting opportunities include porting currently available peripherals to fully compatible PILoT peripherals and porting PILoT's container engine to different platforms. In comparison, development opportunities include the development of drivers and applications and then selling them on PILoT's marketplace.

3.4 Summary

This chapter introduced our vision to a platform-agnostic IoT haven. It is crucial to note that PILoT redefines the traditional IoT development pipeline. The current status-quo uses an outdated waterfall development methodology with minimal parallelism and restricts designs to ones highly dependent on the underlying hardware platforms. However, PILoT enables developers to utilize modern SDLCs such as DevOps through three enabling paradigms; TPnP, IoT containers, and CD-CD. We presented an outline of PILoT's architecture and discussed the potential challenges. Finally, we discussed how PILoT has the potential to disrupt the industry through different use-cases and potential business opportunities. In the next chapter, we discuss our proposed implementation of an IoT containerization engine that is core to PILoT's architecture.

Chapter 4

Things HIVE - For a Streamlined IoT Development and Operations

Software containers are the natural evolution to VMs; however, current containerization engines are resource hungry and not tailored for the IoT domain [27]. Furthermore, efforts to adopt containers on IoT devices focused on devices running Linux through Docker. Resource-constrained IoT sensor nodes that execute embedded firmware (sometimes without an OS) received minimal to zero attention.

Towards materializing core elements in PILoT's vision, we present the Things Hardware Independent Virtual Engine (HIVE); a platform-independent containerization engine tailored for resource-constrained embedded devices. The HIVE approach adopts PILoT's IoT development architecture to satisfy requirements 1 to 3, 5 and 7.

4.1 The HIVE Approach

Towards democratizing IoT platform development, we propose Hardware Independent Virtual Engine (HIVE), a containerization engine and development framework designed specifically for IoT platform development. The HIVE approach adopts PILoT's paradigm of separating development responsibilities between platform developers and peripheral manufacturers to achieve a highly optimized development pipeline. Additionally, HIVE enables a higher level of code-reuse compared to bare-metal development implementations.

The engine includes a light-weight VM for resource-constrained IoT devices; the Byte-code is designed to enable rapid development and deployment of IoT platforms. Due to its platform-independent nature, a developer using HIVE can develop and test on their local machine without access to an IoT hardware device. Thus, enabling the virtualization of IoT devices and satisfying PILoT's requirement 1. Furthermore, once deployed to the real hardware device, the developed application on the local machine will still operate as expected. Such a feature enables parallelized development of an IoT platform, where the software development can start before the system design is complete. As illustrated in Figure 4.1, the contemporary development journey is reminiscent of the waterfall development methodology. However, using PILoT and HIVE enables and promotes improved development parallelism more in line with modern DevOps methodologies.

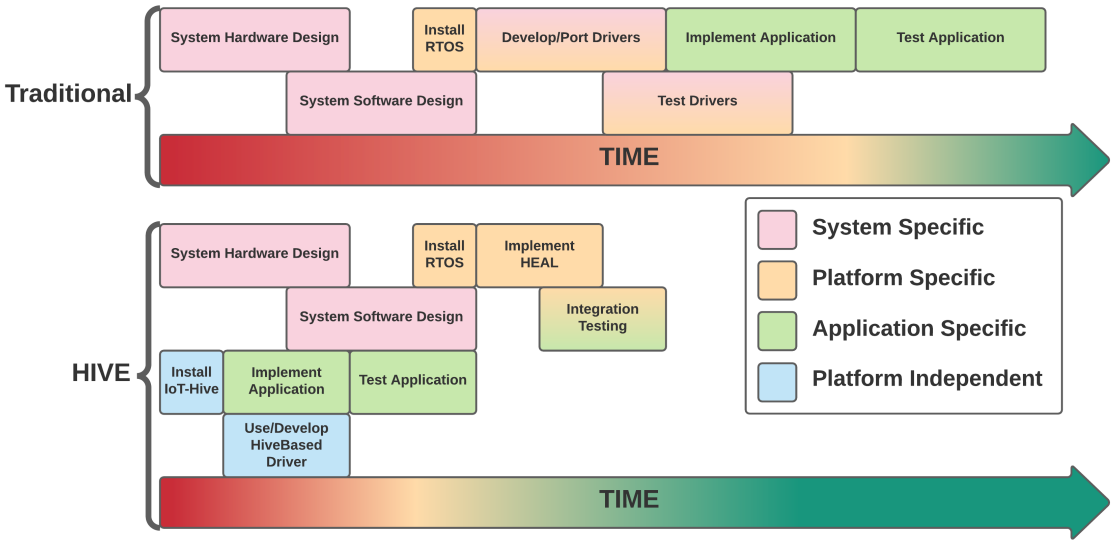


Figure 4.1: Development journey comparison between the traditional IoT development methodology and the HIVE approach

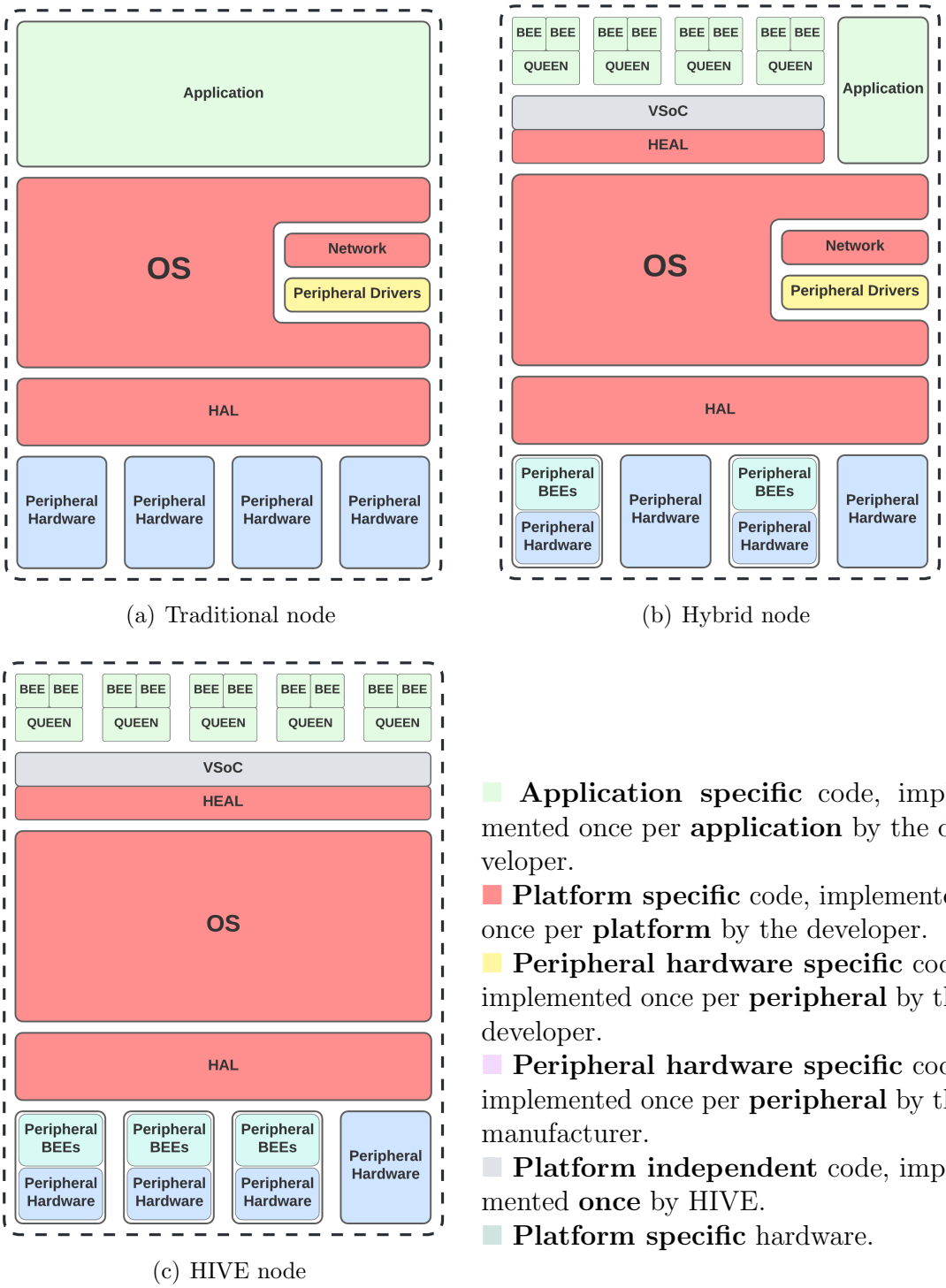
4.2 The HIVE Architecture

HIVE attempts to solve the constraints inherent with the traditional IoT platform architectures by using a (lightweight, platform-agnostic) containerization engine baked as part of the firmware. Additionally, the engine can dynamically load the application scripts from storage into memory; therefore, the application code can be decoupled from the other layers. The HIVE architecture enables a myriad of applications (including TPnP and CD-CD updates); however, it can also have security implications which we discuss in Section 4.5.

HIVE's software architecture replaces the application layer with two HIVE specific layers, namely, the HIVE-Core and the HIVE applications. The HIVE-Core consists of two sub-layers, Hive Engine Abstraction Layer (HEAL) and the Virtual System on Chip (VSoC). The only part of HIVE that is not platform agnostic is HEAL and must be ported to a new platform before using HIVE. In contrast to porting every peripheral driver on top of the underlying platform's HAL in the current status-quo, porting HEAL to a new platform is reminiscent of implementing a HAL layer and is independent of any peripheral specificity. Figure 4.2 outlines a comparison between a traditional, hybrid and a HIVE IoT node. We discuss each part of the HIVE software architecture in the following sub-sections.

4.2.1 HIVE Applications

The top layer in the architecture is the HIVE application layer. Adopting the concepts and requirements described in PILoT's container execution model, a basic HIVE application consists of a single Quick Universal Execution Entity Nexus (QUEEN) and one or more Basic Execution Entities (BEEs). A Basic Execution Entity (BEE)



(a) Traditional node

(b) Hybrid node

(c) HIVE node

Figure 4.2: Traditional vs. HIVE IoT nodes

is the highest level application implementation in a HIVE stack. One BEE script can be considered an atomic unit of execution and is written in HIVE byte-code. A compiler can convert any high-level language into HIVE byte-code; however, this is out of scope for our discussion and is part of our future work. There are three types of BEEs, a peripheral BEE, a custom BEE, and a QUEEN BEE.

Each BEE has its stack-based VM responsible for executing its byte-code; however, the execution path for some byte-code instructions differs based on the BEE type. All BEEs encapsulated together, represents a single PILoT IoT container.

Quick Universal Execution Entity Nexus (QUEEN)

The core application in a container (as described in PILoT) is the QUEEN. There can only be a single QUEEN in a HIVE application. A developer may choose to implement all of the application code inside the QUEEN program. However, such a model does not fully utilize the full capabilities of HIVE; using the QUEEN to specify the order in which the other BEEs are executed allows for far more complex applications. The QUEEN application is usually unaware of which peripherals are connected to the device; however, the QUEEN describes how the application would behave if a particular type of peripheral is connected.

Peripheral BEEs

PILoT outlines the use of peripheral workers, HIVE implements this vision as peripheral BEEs. Each connected peripheral can have one or more associated peripheral BEEs. The peripheral BEEs are further subdivided into stages, **INIT**, **ACT**, **READ** and **WRITE**. Furthermore, peripheral BEEs are divided into classes based on the

type of peripheral; the fundamental classes supported in our current design are **Digital Input/Output (DIO)**, **Analogue Input/Output (AIO)**, **SCREEN**, and **RADIO**. All BEEs associated with a peripheral share a virtual port that the programs target. This port is logically mapped to physical platform pins.

Custom BEEs

Finally, PILoT's custom workers are implemented as custom BEEs in HIVE and are a more constrained type of QUEEN. Custom BEEs can be considered co-routines. One of the fundamental differences between a QUEEN and a custom BEE is the execution scheduling. A custom BEE yields to other BEEs either on completion, through a yield byte-code, or on thread expiry. However, a QUEEN yields after executing each byte-code.

4.2.2 HIVE-Core

The second layer in the HIVE architecture is the HIVE-Core. This layer is the brain of the containerization engine, responsible for encapsulating containers in the application layer and communicating with the underlying OS or HAL layers. The HIVE-Core consists of the low-level interface (HEAL) and a software implementation of an embedded SoC similar to microchips such as MSP-432 [126] and ESP-32 [42]. We call this software implementation a VSoC. Although our discussion relies on HIVE being implemented on top of an OS/RTOS, HEAL can be implemented to communicate directly with the HAL and bypass the OS requirement, thus allowing HIVE to work on both OS-based and bare-metal architectures.

As with any SoC, the VSoC consists of a CPU (a Virtual Central Processing Unit

(VCPU) in our case) accompanied by multiple units, systems, and modules. One of the core benefits of this VSoC is providing developers with a unified extendable virtual platform that they can learn once, develop multiple times, and deploy on any IoT device that supports the VSoC; all with ZERO porting effort.

The HIVE VSoC consists of an extendable number of components. The current proposed version includes three core components, the VCPU, the Honey Pot[1], and the Bee Map, as well as several extendable engines, namely, a TPnP engine, an Encryption engine, a Timer engine, the Power engine, a Clock engine, a Flash engine, and some peripheral interfaces in the Peripherals engine. We further discuss some of these engines in Section 4.2.5. Figure 4.3 illustrates the different core components of the HIVE-Core.

4.2.3 HEAL

HEAL is a middleware abstraction layer to be implemented by platform developers. This layer consists of interface functions called by certain byte-code instructions being executed on the VSoC. Each of the engines in the VSoC is interfaced by the HEAL layer and exposes the underlying features of the IoT device platform to the VSoC. HEAL is the only platform-dependent element of HIVE due to its inherent access to platform-specific resources. However, HEAL is only dependent on the RTOS and microprocessor/microcontroller the IoT device is based on. Thus, once HEAL has been ported to a specific RTOS and microprocessor/microcontroller, HIVE becomes compatible with all IoT platforms that use them. Additionally, developers can open-source their HEAL implementations for other developers to use.

[1]The term honey pot used in HIVE is not related to the one commonly used in cybersecurity. We use the term honey pot to refer to the in-memory storage available to the applications.

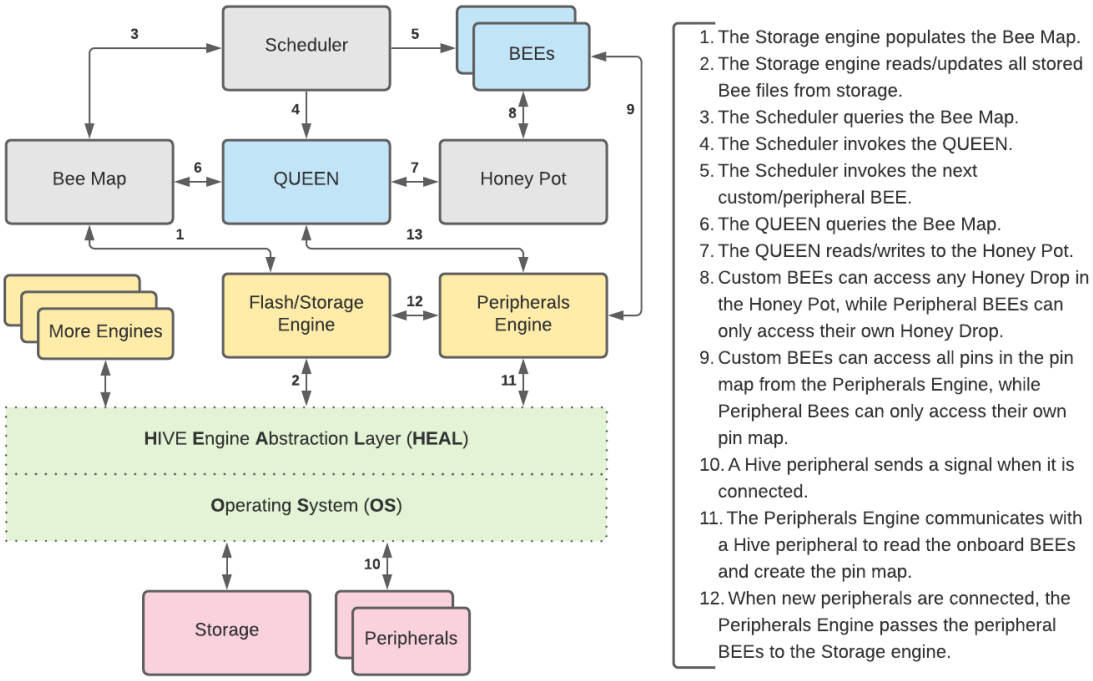


Figure 4.3: Details of the HIVE-Core

Developers implementing HEAL on a new platform first need to define the pin mapping, then implement the interface functions.

Pin Mapping

Internally, the HIVE VSoC consists of an extendable 100-pin virtual chip. The developer first maps each virtual pin to the underlying hardware platform pins. Future work could include a Graphical User Interface (GUI) tool that allows developers to map the pins with near-ZERO effort. Developers could also export their maps for others to use as open-source projects. A HIVE application does not need to know the underlying hardware platform pins; instead, the virtual chip maps the application logic to different virtual hardware ports. Each of the virtual ports can then be mapped

to physical hardware interfaces directly by the developer using the HEAL layer or is automatically mapped to HIVE-compatible peripherals through the TPnP engine.

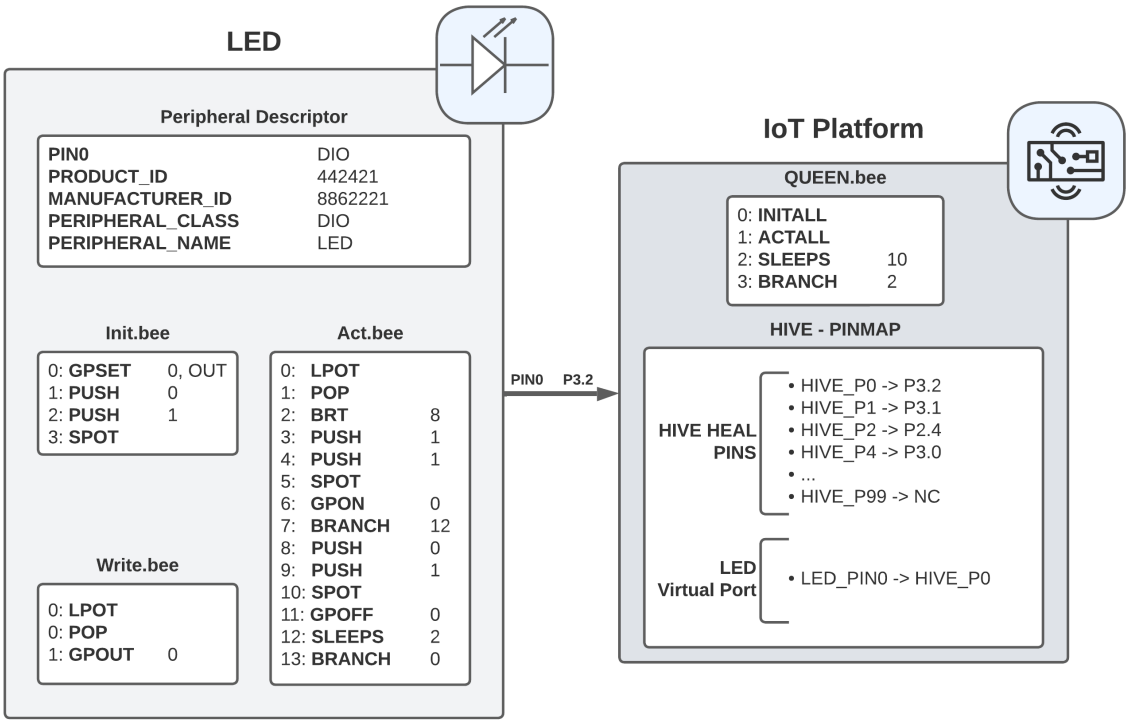


Figure 4.4: Example LED peripheral pin-mapped to a HIVE IoT platform

Fig. 4.4 illustrates an example LED peripheral implemented in HIVE. This peripheral contains a descriptor in addition to multiple peripheral BEEs. This combination of files represents a digital twin of the physical peripheral we dub a "Peripheral Phantom". The peripheral descriptor of the LED module indicates a single DIO pin labelled "PIN0", which once physically (or virtually) connected to a pin on the IoT platform would get mapped to an *LED Virtual Port*. This virtual HIVE pin/port is mapped by the developer to the underlying physical hardware pin as illustrated in the *HIVE - PINMAP* in Fig. 4.4. The mapping between the peripheral phantom pins and the HIVE virtual port pins is the responsibility of the TPnP engine. However, the

developer can also hard-code this into HEAL if the TPnP feature is not desired.

Interfacing Functions

The second and final effort required by developers implementing HEAL is the interface functions. These functions communicate directly between the HIVE containerization engine and the underlying software architecture, whether through an OS layer or directly through the HAL. Developers can choose to implement some or all of the interfaces; however, implementing some interfaces is necessary for the operation of HIVE. We outline some of the interface functions as follows.

1. **Timing** functions include *GetTimestampInMs* and *SleepMs*. The former returns the current system time in milliseconds, while the latter puts the system to sleep for the specified number of milliseconds. The *SleepMs* function can be implemented to put the underlying hardware to sleep, set the OS thread to sleep, and perform a blocking delay[2]. All timing functions are required and used by HIVE to schedule the execution of different BEEs.

2. **General Purpose Input/Output (GPIO)** functions include writing and reading from a pin, setting the pin direction, and setting a pull-up or pull-down resistor on a pin. Due to the versatility of HIVE, some platforms may never require the use of GPIO pins, and the developer could ignore the implementation of these functions.

3. **Inter-integrated Circuit (I^2C)** interfacing functions include initializing and configuring an I^2C port, in addition to reading and writing a fixed number or

[2]A blocking delay is performed by forcing the microprocessor to spend a certain number of instruction cycles on no operations. Thus consuming the microprocessor time and preventing other instructions from being executed for the set period.

a dynamic number of bytes. These functions are optional, and the developer can ignore the implementation. However, a developer may choose to create a software implementation of an I^2C port for platforms that do not have a dedicated hardware I^2C bus.

4. **Serial Peripheral Interface (SPI)** and other communication (e.g. Universal Asynchronous Receiver-Transmitter (UART)) interfacing functions are similar to the I^2C functions and can also be ignored by the developer depending on the platform.

5. **Memory** access interfacing functions include pure reading and writing to on-board memory. Additionally, include abstracted functions to read BEE files from storage. These types of functions are required since this is what HIVE uses to load the QUEEN and BEE scripts.

4.2.4 Virtual System on Chip (VSoC)

The HIVE VSoC communicates with HEAL through modular engines. The architecture of the VSoC uses a stack-based VM as the CPU, the Honey Pot as the RAM and the BEE Map as the ROM. (Recall that the term honey pot used in HIVE is not related to the one commonly used in cybersecurity.)

Virtual Central Processing Unit (VCPU)

We refer to the scheduler in the VSoC as a VCPU. It handles the execution scheduling of the different BEEs in a single IoT container. The HIVE VCPU follows a process VM similar to the DALVIK VM used in Google's Android OS [67]. Each BEE has its 32-Bit stack-based VM that executes on one of the virtual cores of the VCPU. Hence,

each BEE has its unique and isolated stack and registers. The platform developer can define the number of virtual cores of the VCPU to adapt to the underlying hardware capabilities. Since each virtual core is responsible for the execution of a single VM, we use the terms VMs and VCPU cores interchangeably.

In HIVE's execution model, a BEE file describes an atomic unit of execution; it can be thought of as a single module or function. In contrast, a QUEEN file specifies the order in which each of the BEEs is executed. The VCPU sandboxes the execution and prevents access to external OS threads. One fundamental assumption is that the underlying physical CPU has a single core, and the RTOS used runs a single task at a time. However, HIVE is designed to support and utilize multi-core platforms in the future.

The Instruction Set Architecture (ISA) implemented by the VCPU cores is a 32-Bit stack machine with a minimal footprint on memory, explicitly designed for constrained embedded systems and IoT devices. The VCPU provides two layers of sandboxes, a global sandbox and a bee-level sandbox. The global sandbox represents a single IoT container and isolates the entire HIVE from the RTOS, while the bee-level sandboxes isolate the code running in each BEE to itself. The data flow in HIVE is coordinated through a virtual shared RAM, the Honey Pot; for instance, a QUEEN can deposit data (Honey Drop) in a peripheral's section in the pot for future processing.

In contrast to programming languages such as Python and Lua, the byte-code in HIVE's VCPU cores is tailored to embedded system functionality. For instance, the Instruction Set (IS) includes instructions that natively interact directly with the underlying hardware. Thus, reducing the number of virtual instruction cycles required to perform the same operation. Furthermore, since the VCPU runs multiple VMs, the

IS includes instructions that can access the shared memory (Honey Pot) in addition to local registers and RAM.

BEE Scheduling

The threading system in HIVE is inspired by the threading module in TI-RTOS, and FreeRTOS [35], [18]. Although each BEE executes on a separate VCPU-core, HIVE still executes on a single underlying RTOS thread. Therefore, we use a modified round-robin scheduling algorithm in HIVE; algorithm 4.1 illustrates how the VCPU schedules the BEEs.

The QUEEN is considered the coordinator in an application and yields after every instruction cycle. In contrast, BEEs yield under three conditions, explicitly by the developer, at the completion of the program, or on a thread timeout. A BEE Execution Auditing Rover (BEAR) is a monitoring application running on a separate underlying RTOS thread, responsible for timing out a BEE. It is also responsible for ejecting a BEE from being executed if multiple timeouts for the same BEE are detected. This behaviour is essential to protect the HIVE from thread-starvation.

We considered two different approaches when it came to scheduling. With the first approach, the QUEEN controls when a BEE instruction is to be executed, which involves the BEE data structure being loaded onto the stack and released between each instruction. This overhead proved to be very under-performing when compared to the other approaches. In the second approach, the BEE [3] runs indefinitely and yields to other BEEs or the QUEEN. This approach reduces context switching and thus is more performant than the first approach. Figure 4.5 illustrates an example of a simple IoT application and how it gets scheduled on the VSoC. The QUEEN

[3]Except when explicitly defined to halt by the developer.

is responsible for adding BEEs to the scheduling queue. The VCPU then schedules BEE execution from the queue according to algorithm algorithm 4.1. Once a BEE terminates, the next BEE in the queue is scheduled. However, if the BEE yields, it gets re-queued for future execution. When a yielding BEE is re-queued, it preserves its current state to resume execution in the future.

Algorithm 4.1 HIVE scheduling

```
while QUEEN is not TERMINATED do
    Execute QUEEN  if BEEQueue is not EMPTY then
        for i ← 0 to numOfBEES in BEEQueue do
            while BEE is not YIELDING do
                Execute BEE[i]
            end
            if BEE[i] is not TERMINATED then
                Add BEE[i] to BEEQueue
            end
        end
    end
end
```

Memory & Storage

The honey pot is the only shared memory between BEEs and can be considered the RAM of the VCPU. Each BEE has its isolated dynamic area in the pot called a *Honey Drop*. However, the QUEEN and custom BEEs are allowed access to any honey drop in the pot; peripheral BEEs are only allowed access to their drop. This access policy adds a layer of security to prevent malicious code from affecting the device. Each

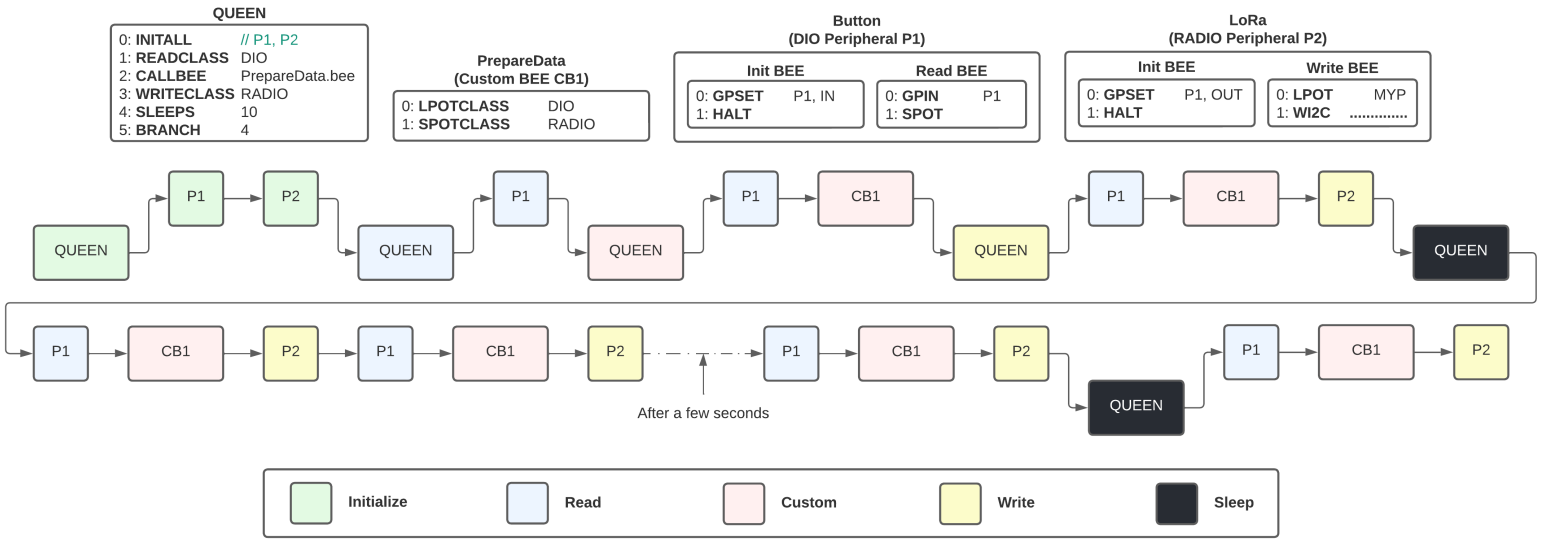


Figure 4.5: An example IoT application and how it gets scheduled on the HIVE-VSoC

The QUEEN starts by scheduling an initialization on all connected peripherals. Since the INIT scripts of all modules HALT at the end, the INIT scripts get removed from the execution upon completion.

In contrast, the READ, WRITE, and the CUSTOM BEEs do not include a HALT or YIELD instructions. Therefore, the BEEs get re-added to the queue for future execution.

We note that when the QUEEN executes the SLEEPMS instruction, it no longer gets scheduled in the execution queue until the sleep/yield time expires.

After which the QUEEN re-executes the SLEEPMS instruction again, forcing the QUEEN to clear the execution queue for the BEEs.

honey drop can be of one of the following types.

- **cfg**: Used to perform runtime configuration of the task before the task is started
- **input**: Used to pass input data to the task (for example, dynamic parameters for an external sensor)
- **output**: Used to pass output data to the System CPU application (for example, accelerometer data)
- **state**: Internal variables used to store the state of the task between iterations

In contrast, the Bee Map resembles the program memory; where all the BEEs to be executed are cached; this resides on the heap of HIVE's memory. However, all BEEs are optionally stored in flash storage for future caching if needed.

4.2.5 Extendable Engines

The HIVE-VSoC uses engines to communicate between the VCPU and HEAL. An engine is a middleware that encapsulates and abstracts more complex logic on top of HEAL. Since an engine uses interfaces exposed by HEAL, they are platform agnostic and do not require porting by developers. We discuss a select number of engine examples as follows.

Flash Engine

The storage engine interacts directly with HEAL interfaces to read HIVE BEEs from flash storage.

Peripherals Engine

This engine is responsible for abstracting the process of loading peripheral BEEs through the TPnP engine or the flash engine.

TPnP Engine

The TPnP engine is responsible for detecting when a new peripheral is connected, mapping the physical pins to HIVE ports, and reading the peripheral phantom from the storage on board the peripheral to the platform disk.

Multicontainer Engine

This engine allows an IoT platform to switch between different containers by loading a new container image from disk through the flash engine and invoking the context switch on the VSoC.

4.3 HIVE Container Execution Model

Each IoT HIVE container represents a single application, which encapsulates a single QUEEN and multiple BEEs (custom or peripheral). The QUEEN serves as the high-level application loop, while each BEE represents a single task running on the IoT node. HIVE exposes underlying system resources to a container through HEAL. The container can then utilize such resources in the QUEEN or any of the BEEs.

When a BEE is instantiated, a minimum of 180 bytes of memory are allocated on the container heap. The number of bytes can be modified based on the needs of the BEE and represents the VM's stack and data. A section in the honey pot is also reserved for this VM for cross-BEE communication. A BEE can specify a number of variables or constants stored in the honey pot. Peripheral BEEs would only have access to this data. However, custom BEEs and the QUEEN can access any drop in the pot.

Peripherals are classified under one of the following classes, *DIO*, *AIO*, *SCREEN*, or *RADIO*. Furthermore, a peripheral BEE can be in one of the following stages, *INIT*, *READ*, *WRITE*, *ACT*, or *CUSTOM*. The custom stage is similar to a custom BEE but defined within the peripheral phantom. In contrast, a custom BEE does not have a classification or belong to a certain stage.

In addition to the VM IS, the QUEEN has access to an additional set of instructions responsible for controlling the flow of the application execution. As discussed in Section 4.2.4, the scheduling process runs BEEs indefinitely unless they explicitly yield or terminate with a HALT instruction. The QUEEN can schedule or terminate BEEs using one of the following instructions. By design, only a single BEE can be active at any point in time; this is to prevent a BEE trying to access a particular resource (e.g.

SPI) from interrupting a BEE currently using this resource. However, different levels of software interrupt remain a feature to be implemented in our future work.

- **CALL_BEE** schedules the execution of the BEE specified by the operand of the instruction.
- **INIT_ALL** schedules the execution of all BEEs associated with the *INIT* stage and belonging to any class.
- **INIT_CLASS** schedules the execution of all BEEs associated with the *INIT* stage and belonging to the class specified by the operand.
- **ACT_ALL** schedules the execution of all BEEs associated with the *ACT* stage and belonging to any class.
- **ACT_CLASS** schedules the execution of all BEEs associated with the *ACT* stage and belonging to the class specified by the operand.
- **READ_ALL** schedules the execution of all BEEs associated with the *READ* stage and belonging to any class.
- **READ_CLASS** schedules the execution of all BEEs associated with the *READ* stage and belonging to the class specified by the operand.
- **WRITE_ALL** schedules the execution of all BEEs associated with the *WRITE* stage and belonging to any class.
- **WRITE_CLASS** schedules the execution of all BEEs associated with the *WRITE* stage and belonging to the class specified by the operand.

- **HALT_BEE** terminates the execution of the BEE specified by the operand of the instruction.
- **HALT_ALL** terminates the execution of all BEEs regardless of stage and class.
- **HALT_ALL_INIT** terminates the execution of all BEEs in the *INIT* stage.
- **HALT_ALL_ACT** terminates the execution of all BEEs in the *ACT* stage.
- **HALT_ALL_READ** terminates the execution of all BEEs in the *READ* stage.
- **HALT_ALL_WRITE** terminates the execution of all BEEs in the *WRITE* stage.
- **HALT_CLASS** terminates the execution of all BEEs of a specific class. For instance *HALT_CLASS RADIO* would terminate all *INIT*, *READ*, and *WRITE* BEEs belonging to the *RADIO* class.

4.4 Performance Analysis

We implemented a proof-of-concept for our work and ported HEAL to some platforms. This section discusses the performance analysis we performed on HIVE compared to popular alternatives. However, first, we present the metrics used in our evaluation process.

4.4.1 Evaluation Metrics

To evaluate HIVE, we consider performance and footprint-related metrics. Execution time is among the former, while the total compiled binary and dynamic memory usage are among the latter. Furthermore, we evaluate the relative development effort through a holistic analysis of the development speed using HIVE compared to bare-metal implementations.

4.4.2 Implementation Assumptions

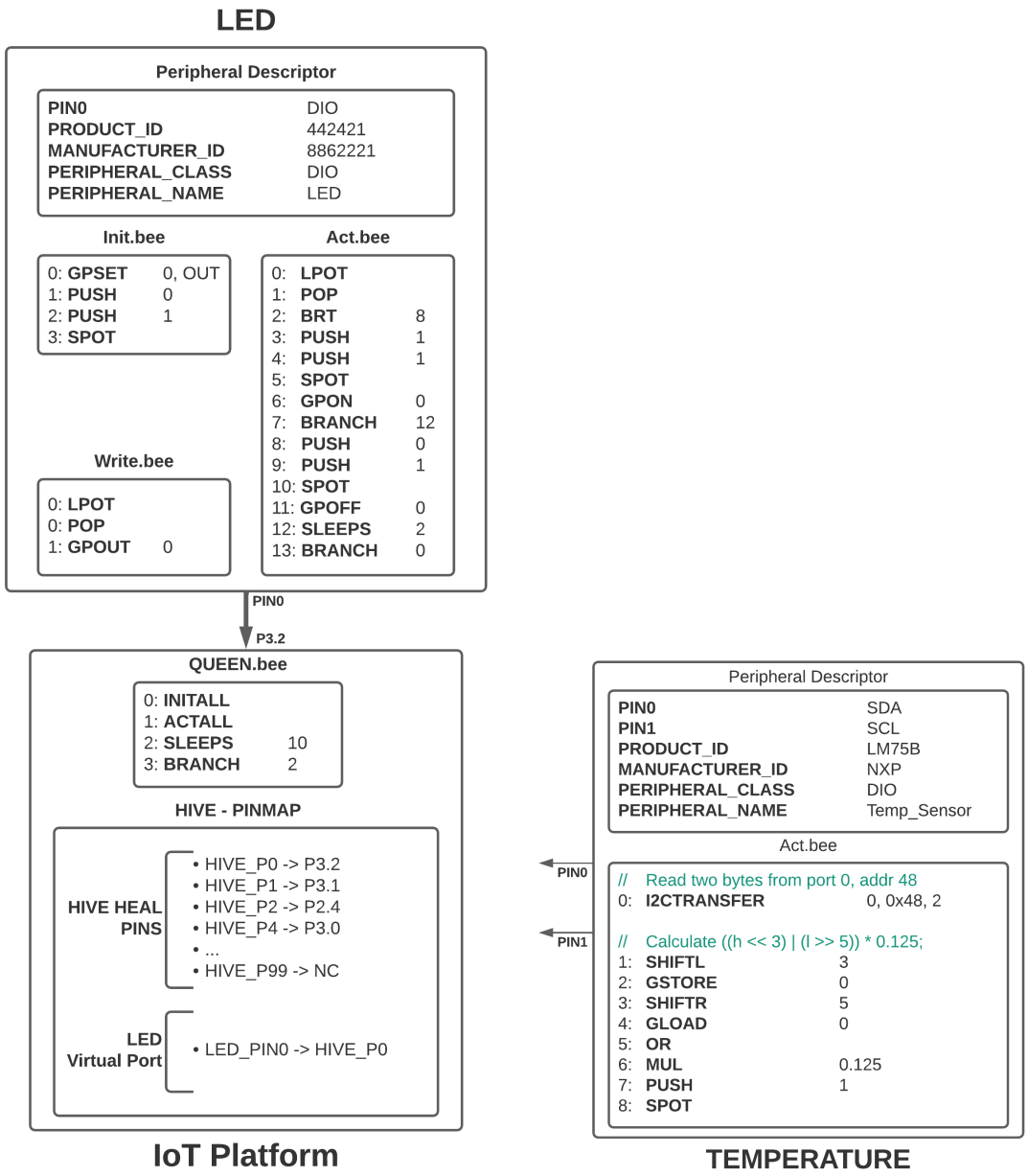
Before we outline the details of our implementation, we indicate a few assumptions and missing features that will be implemented in the future. The implemented features were carefully considered to ensure a fair comparison and accurate results. One limitation with the proof-of-concept is the lack of support for 64-bit floating-point numbers. However, support for 32-bit floating points is implemented. The current implementation of the VM follows the switch dispatch pattern; a more optimized version is part of our planned future work. We expect improved execution time in the future since the switch dispatch pattern does not fully utilize branch prediction features of modern CPUs. Another implementation decision was embedding the QUEEN and BEEs within the compiled binary. In the final implementation, the

QUEEN and custom BEE scripts would be stored on flash storage and loaded into memory at the initialization phase of the engine. Similarly, peripheral BEEs would be loaded and cached from the connected peripherals during runtime. We believe that due to our choice of embedding the programs within the compiled binary, all captured HIVE results would have a negligible amount of inaccuracy. Finally, HIVE is currently implemented without a compiler. All examples are written directly in HIVE's native byte-code. However, a compiler program will be developed as part of our future work to convert from a high-level language to HIVE's byte-code.

4.4.3 Implementation Details

We implemented a proof-of-concept of HIVE in C with the C99 standard, for which the implementation source code will be available through GitHub [51]. We have also ported HEAL to three test platforms: Advanced RISC Machine (ARM®) A based Raspberry Pi, STM-based ESP-32, and ARM® M based MSP-432. The reasons behind the choice of platforms are two-fold, evaluating the effort required to port a HIVE application to different platforms; and demonstrating a full CI/CD pipeline in IoT platform development. Additionally, we implemented HIVE-compatible drivers for several peripherals, namely, an LED, a 7-Segment Display, a temperature sensor, a microphone, and a LoRa® radio. For simplicity purposes, all peripherals (except the LED) use I^2C as the interfacing bus protocol. However, HIVE supports other popular interfacing bus protocols. Finally, we developed some applications in HIVE that use the earlier peripherals.

- **Hello World** is the first application developers are exposed to due to its simplicity. The application prints the "Hello World From The Queen!" onto the



(a) HIVE implementation of the LED flasher application

(b) HIVE implementation of the Temperature sensor peripheral

Figure 4.6: Example peripheral implementations

console.

- **LED flasher** is the *Hello World* of embedded application development. This example serves different testing purposes for developers, including testing clock/timing implementation, testing hardware access, and testing the initial compile/flash procedure. Figure 4.6(a) illustrates the setup for this application.

- **Sense & Report** is a typical IoT sensor application in which a sensor reading is taken and then reported. In our case, we use a temperature sensor and a LoRa® radio to assess this application. Figure 4.6(b) illustrates the temperature sensor implementation in HIVE. We also developed a modified version of this application (Temperature Monitor) for performance evaluation.

- **Temperature Monitor** communicates with a temperature sensor over I^2C, converts the binary number into decimal Celsius representation, prints the value to the console, and displays the temperature value on an I^2C-based 7-segment display.

- **nth Harmonic Series** calculates the $100,000,000^{th}$ harmonic number in the harmonic series [36]. This application evaluates the performance of HIVE in looping, branching, and large amounts of float data computation.

- **Real-time sensing & reporting** describes an IoT application in which continuous sensing is required, and reporting occurs only under certain conditions. For instance, an animal social study program in which IoT sensors are attached to animals. The sensors continuously monitor the audio levels and only reports once a certain dB level is maintained. We use a microphone and a LoRa® radio to assess the threading capability and performance of HIVE.

4.4.4 Peripheral Drivers

Despite the difference in implementation of the drivers, the logic remains unchanged between all bare-metal platforms. The HIVE approach for the drivers follows the same logical flow as the bare-metal implementations; however, it is divided into four main categories initialization, action, read, and write. A HIVE peripheral may implement one or more of these categories into separate BEE files. On the contrary, typical bare-metal drivers only implement functional abstractions to interact with the peripheral rather than logical operational abstractions. Hence, we omitted the action in our bare-metal implementations and focused on initialization, reading, and writing. A HIVE driver implementation also involves a descriptor section reminiscent of the IEEE 1451.2 [72] TEDS. We propose that a HIVE peripheral driver be implemented by the peripheral manufacturer and provided alongside the hardware upon purchase. Such a shift in implementation responsibility reduces the probability of error, for instance, when a peripheral involves complex calculations.

LED

is an output peripheral, hence, does not require the read BEE. Figure 4.6(a) illustrates the HIVE driver implementation as part of the application. The bare-metal implementation is almost identical, except for the Act.bee logic. The initialization involves setting the pin direction to which the LED is connected. In the HIVE version, it is also used to initialize a value in the LED honey-drop. The write logic involves writing a value provided to the write functionality on the hardware pin. The HIVE implementation reads this value from the LED's honey-drop. Finally, the action logic involves reading the honey-drop value and turning the LED either ON or OFF based

on this value; it also prepares the honey-drop for the next time this functionality is triggered before going to sleep for 2 seconds.

7-Segment Display

is a typical I^2C controlled seven-segment display with two digits. The initialization implementation involves preparing the I^2C port and the honey-pot entry. The action stage tested the peripheral by incrementing and displaying a number on the 7-segment. Finally, the write stage reads the value in the honey-pot, converts the value to a hexadecimal representation, and then prints the value on the display using the appropriate I^2C procedure.

Temperature Sensor

a typical peripheral of this type requires initialization, input and action functionalities in its driver implementation. For our experiments we used an I^2C temperature sensor (LM75B) by NXP [95]. The initialization functionality involves preparing the I^2C port parameters. Figure 4.6(b) illustrates the action functionality in the HIVE driver implementation for this peripheral. The read functionality reads two bytes from the I^2C port and stores the value in the honey-pot. The action functionality extends the reading flow by first converting the value read from the I^2C bus into a float value representing the temperature in Celsius before storing it in the honey-pot. Our implementation of the Act.bee script demonstrates how the sensor manufacturer can ensure that an accurate calculation/conversion of the values is sustained across different platforms, a capability that manufacturers are currently unable to guarantee.

Microphone

the driver implementation is almost identical to that of the temperature driver. The main differences include the I^2C address, the register initialization parameters, the conversion algorithm in the action, and the descriptor section.

LoRa® Transceiver

is the only peripheral in our tests that has significantly different logic between the bare-metal and HIVE implementations. Albeit the most complex and time-consuming, the driver implementation provides developers with a high level of flexibility. The LoRa® module requires two-three GPIO pins in addition to the SPI pins. The initialization process involves setting up the LoRa® module with a series of register values; however, it must also allow the developer to define initial values for each register on the peripheral. The read flow saves data from the radio to the honey-drop, while the write flow transmits the data available on the honey-drop. The action flow has different execution paths, and each execution path is triggered through a value in the honey drop. The different paths involve changing the LoRa® radio state, modifying the value of one of the registers, controlling the internal buffer, and other LoRa®-specific functionalities. In contrast, the bare-metal implementation adopts and ports the LoRa® driver implemented by the manufacturer [74]. We used an SX1276-based LoRa® module for our experiments. Specifically, the RFM95W by HopeRF [59].

4.4.5 Environment setup

We used multiple platforms to demonstrate HIVE's platform-independence capabilities in addition to simulating a full CI/CD pipeline. All development was performed on a Unix-based MacOS 11 machine and then deployed to one of the IoT devices. The Raspberry Pi 4 ran a Debian-based buster 10 Raspi-OS and deployed it through a remote SSH connection. The Raspberry Pi 4 was used as the main IoT platform to gather the performance and footprint results. However, all results will be relatively similar on the other platforms. All the peripherals were purchased from and were I^2C based. However, the peripheral drivers were all custom-developed except for the LoRa® driver, which was ported from the C library provided by Semtech [74].

For performance and footprint-related evaluation tests, we compared HIVE with several frameworks running several applications. We chose three applications: Hello World, Temperature Monitor, and the nth Harmonic Series. The choice of applications considered publicly available libraries and tools to minimize the effort in implementation; this represents a typical methodology a developer would follow. The framework set up and build versions used are discussed as follows.

- **C** was used to benchmark the performance and footprint of the applications.
- **Python** version 3.7.3 was used with the smBus module installed.
- **microPython** was built using the Unix port, version v1.16-38, and git commit 7ec95c276 from the main branch on the microPython Github repository [86]. Additionally, we developed an I^2C library to support the Raspberry-Pi 4 platform; this library was not included during build-time and was loaded during runtime. The choice of not baking the library within the microPython binary best simulates

how microPython is used in production.

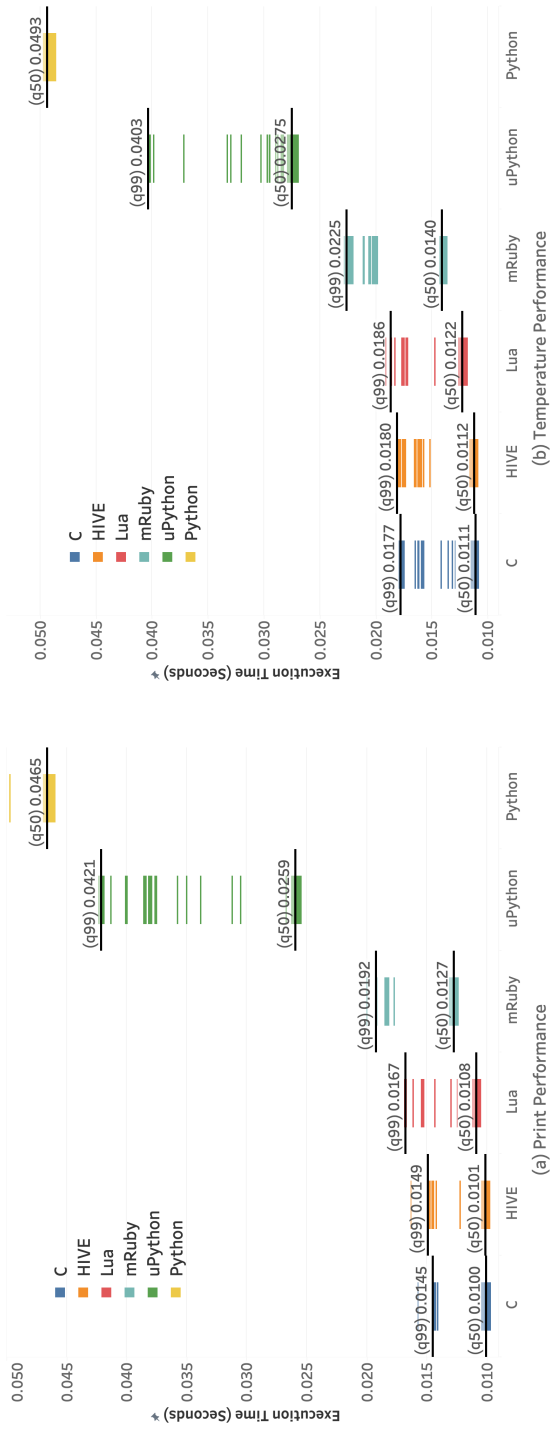
- **mRuby** was built using the default configuration, version mRuby 3.0.0 (2021-03-05), and git commit d63c0df6 from the main branch on the mRuby Github repository [81]. Additionally, mRuby-raspberry [131], and mRuby-sleep [82] modules were added to the default configuration during build time. Furthermore, due to the lack of support for the mRuby-raspberry on the Raspberry-Pi 4, we implemented a port that we contributed to the open-source library.

- **Lua** version 5.3.3 was used with the Lua-periphery module installed.

4.4.6 Performance & Footprint Evaluation

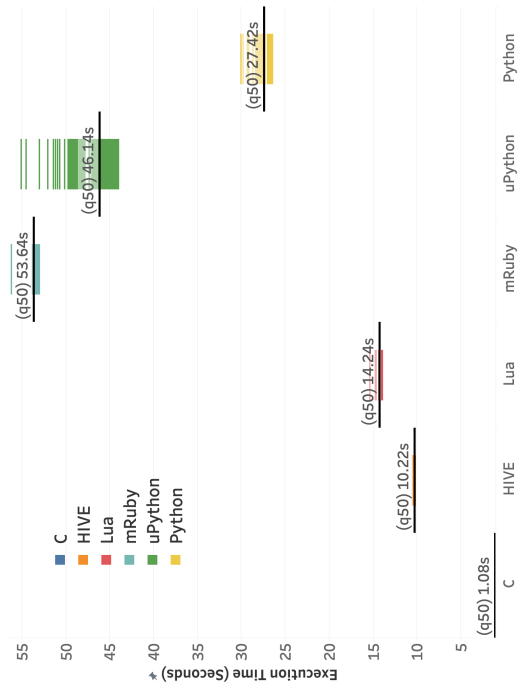
We use execution time as the primary performance indicator when evaluating HIVE. We compare HIVE's results to bare-metal, Lua, Python, uPython, and mRuby. Since eLua can only be deployed as a firmware image and does not provide a VM version that runs on top of an OS/RTOS, we do not use it in our performance evaluation and instead use Lua. Furthermore, we had to implement a Lua interface library since Lua does not have native support for low-level hardware peripheral access. Similarly, we contributed open-source ports to some mRuby and microPython libraries to support peripheral access features on the Raspberry-Pi 4. Our choice of Raspberry-Pi 4 as the platform for performance analysis was due to it being the platform of choice among popular industry IoT cloud providers (i.e. Microsoft [87]). The Raspberry Pi 4 uses a Cortex-M processor, a common choice among resource-constrained IoT devices.

The chosen applications were profiled for execution time using a custom-developed bash script. The script captures the timestamp in nanoseconds before and after executing an application using one of the frameworks and appends it to a Comma



(a) Print Performance

(b) Temperature Performance



(a) Print performance

(b) Temperature performance

(c) Harmonics Performance

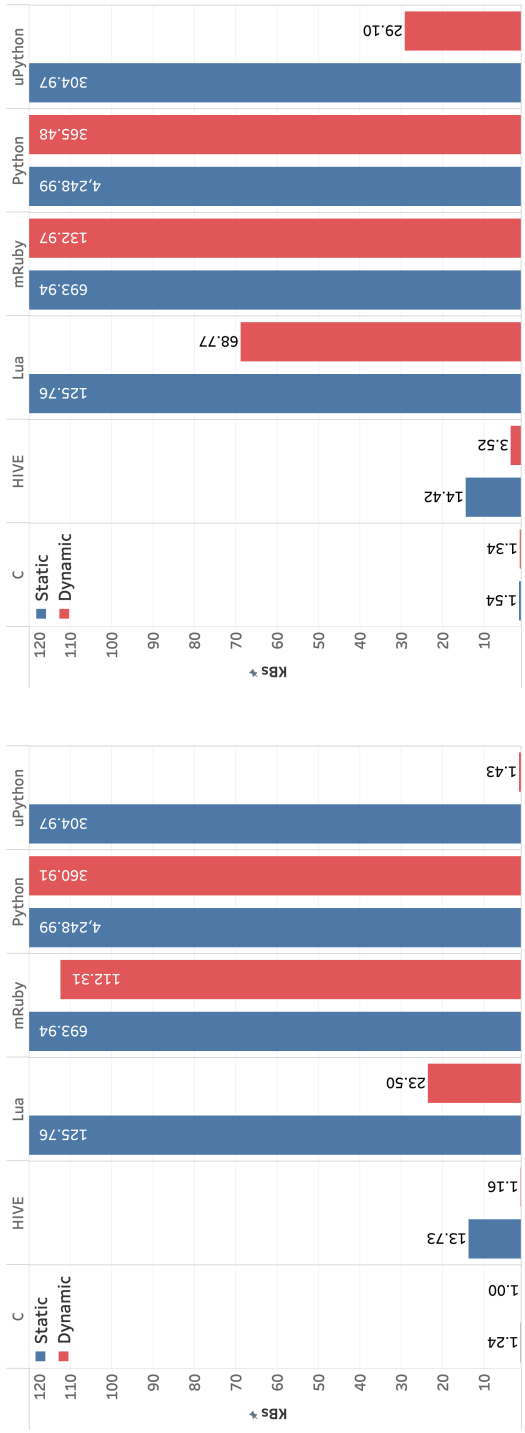
(c) Harmonics performance

Figure 4.7: HIVE performance evaluation

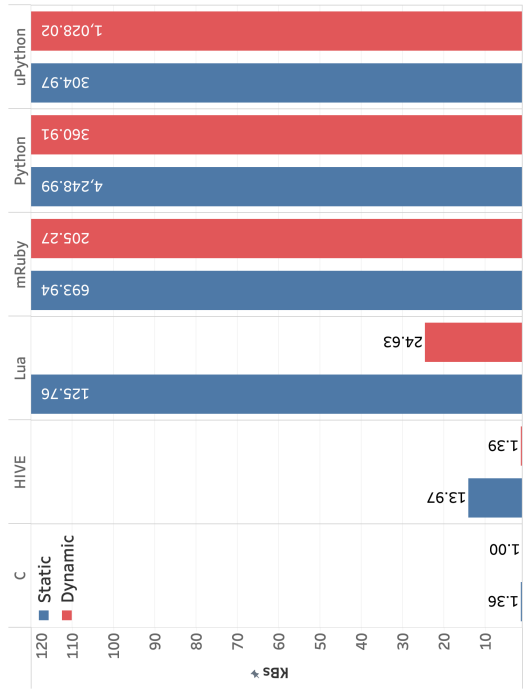
Separated Values (CSV) file; this process is repeated one hundred times. Figure 4.7 illustrates the execution time in seconds for each profiled application. From Figure 4.7-a, we observe that HIVE and Lua are almost as fast as C. However, when compared to other frameworks specifically tailored for resource-constrained embedded devices, namely, mRuby and microPython, HIVE was significantly faster, with 2,600 and 15,800 microseconds faster in the 50th percentile and 4,300 and 27,200 microseconds faster in the 99th percentile when compared to mRuby and microPython respectively. Similar results were observed with the Temperature application as illustrated in Figure 4.7-b. However, when running the nth-harmonic application, mRuby and microPython displayed the slowest performance, with 50x and 43x, respectively, slower than C. Meanwhile, HIVE, Lua, and Python were 9.5x, 13x, and 25x, respectively, slower than C.

We use the Linux 'size' command to retrieve the different frameworks' total compiled binary size (Static). Additionally, we use the Valgrind tool [94] to calculate each application's dynamic memory usage (Dynamic) on each framework. Figure 4.8 represents each application's size and memory usage in Kilo-Bytes. We note that HIVE's static values are not constant across the different applications; this is because the current implementation of HIVE embeds all HIVE scripts within the compiled binary, and the difference can be considered negligible. Using the results for the implementations in C as a benchmark, we observe that HIVE has a significantly smaller footprint than other frameworks; despite having a 11x larger static footprint than the benchmark.

Additionally, when considering the dynamic footprint, HIVE required 3x more memory than the benchmark and 8x less memory than the second-best framework,



(a) Print Footprint (b) Temperature Footprint



(c) Harmonics Footprint

(a) Print footprint (b) Temperature footprint (c) Harmonics footprint

Figure 4.8: HIVE footprint evaluation

namely, microPython. During our tests, we discovered that the microPython port for Unix had a memory leak issue when calculating a double-precision decimal value within a loop. This discovery is represented in Figure 4.8-c, in which the dynamic memory usage of microPython was 1MB.

4.4.7 Development Effort Evaluation

We evaluate the development effort for a subset of the applications outlined in Section 4.4.3 using HIVE and bare-metal implementations. However, since HIVE is considerably different from bare-metal, we subdivide our evaluation into predevelopment-related overhead, peripheral-related implementations, and application-related implementations. To demonstrate and evaluate a full CI/CD pipeline, we started by implementing each application on an Intel-based platform operating a MacOS 11 distribution that represents a developer's development machine. Using an Intel-based machine represents how accessible developing IoT applications can become when using HIVE. Since using an Intel-based personal machine to develop for bare metal is generally inefficient because of the lack of access to lower-level peripherals. We followed the Intel machine with an ARM®-based Raspberry Pi that represents a typical IoT gateway or edge-device, and finally, an embedded platform based on STM and ARM® to represent a resource-constrained IoT end-device. This flow aimed to model the typical development journey of an IoT application across the different platforms.

Table 4.1: Symbol definitions

Symbol	Remarks
α	Implementation Effort
γ	Porting Effort
Θ	HIVE
η	BareMetal
π	RPi
ϵ	ESP32
μ	MSP432

$$\begin{aligned} TotalEffort(x)_p = {} & OverheadEffort_p \\ & + \left(\sum_i^n PeripheralEffort_i \right)_p \\ & + \left(\sum_j^m ApplicationEffort_j \right)_p \end{aligned} \tag{4.1}$$

$$OverheadEffort(x)_p = \begin{cases} HEAL_p, & \text{If x} = \Theta \\ 0, & \text{If x} = \eta \end{cases} \tag{4.2}$$

$$\begin{aligned} PeripheralEffort(x)_p = {} & DatasheetEffort \\ & + TestingEffort_p \\ & + DriverEffort_p \end{aligned} \tag{4.3}$$

$$DriverEffort(x)_p = \begin{cases} \begin{cases} 0, & \text{If x} = \Theta \\ \gamma_C & \text{If x} = \eta \end{cases} & \text{If Available} \\ \begin{cases} \alpha_\Theta & \text{If x} = \Theta \\ \alpha_C & \text{If x} = \eta \end{cases} & \text{Otherwise} \end{cases} \tag{4.4}$$

We use Eq. (4.1) to calculate the total effort for each platform as the total development time consumed to implement all four applications. The overhead effort per platform outlined in Eq. (4.2) was calculated from the development time needed to implement the HAL, integrate the RTOS, and whether or not HEAL was implemented. However, since the effort for HAL and RTOS is shared between HIVE and bare-metal implementations, we omit them from our calculations. As represented in Eq. (4.3), to calculate the total development time required per peripheral on each platform, we combine the time required to read and understand the datasheet, develop/port the driver, and test the driver. Finally, we use Eq. (4.4) to estimate the total time spent implementing each peripheral driver for the different platforms. We note two circumstances, if the driver is available and if the driver is unavailable. In the former case, ZERO effort is required when using HIVE. However, some porting effort is still required when using bare metal. In the latter case, implementing the driver from scratch is required in both HIVE and bare-metal implementations.

$$\begin{aligned} TotalEffort(\Theta)_\pi = Overhead_\pi \\ + \quad \left(\sum_i^n PeripheralEffort_i\right) \\ + \quad \left(\sum_j^m ApplicationEffort_j\right) \end{aligned} \tag{4.5}$$

$$\begin{aligned} TotalEffort(\eta)_\pi = \quad \left(\sum_i^n PeripheralEffort_i\right)_\pi \\ + \quad \left(\sum_j^m ApplicationEffort_j\right)_\pi \end{aligned} \tag{4.6}$$

We use Eq. (4.5) and Eq. (4.6) to calculate the total effort spent for the Raspberry-Pi platform. However, we note that the Peripheral efforts using HIVE are independent of the underlying platform as opposed to bare-metal. Therefore, after the initial effort was spent on the implementation using HIVE, we were able to reuse it on subsequent platforms with ZERO additional effort. Furthermore, the application effort was implemented to be independent of the underlying platform for both HIVE and bare-metal, and therefore are reused with ZERO additional efforts on the following platforms. These observations are materialized in the ESP32 and MSP432 implementations with the former summarized in Eq. (4.7) and Eq. (4.8) for each HIVE and bare-metal respectively.

$$TotalEffort(\Theta)_\epsilon = Overhead_\epsilon \tag{4.7}$$

$$TotalEffort(\eta)_\epsilon = \left(\sum_{i}^{n} PeripheralEffort_i \right)_\epsilon \tag{4.8}$$

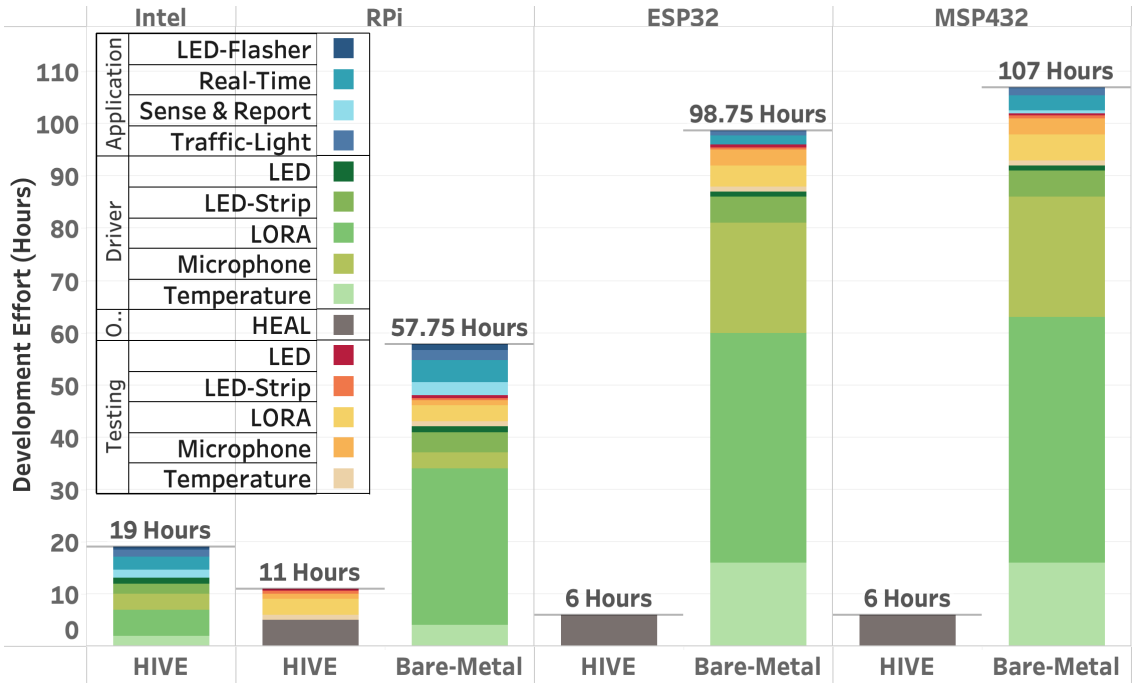


Figure 4.9: HIVE development effort evaluation

We measure the time taken in each of the four platforms' different parts. Our efforts attempt to utilize code-reuse whenever possible in both bare-metal and HIVE implementations. We started by implementing all applications and drivers using HIVE on the Intel-based Mac development machine[4]. For HIVE on the Raspberry Pi, we were able to reuse all of the applications and drivers developed on the Intel-based machine. Therefore, we implemented HEAL and were able to test all the applications. However, Bare-metal on the Raspberry Pi required implementing all drivers and applications before we could start testing. From this point forward, we had all the

[4]Due to the restricted access to underlying hardware peripheral pins, we were unable to implement HEAL as well as all bare-metal efforts on the Intel-based machine.

code we could reuse for any future platform; therefore, the HIVE effort on the ESP32 and MSP432 consisted of only implementing HEAL. At the same time, the effort for bare-metal on the ESP32 and MSP432 involved redeveloping the drivers, porting the applications, and retesting everything again.

Figure 4.9 illustrates the results of our effort in four different areas, Overhead, Application development, Driver Development, and Testing. Since the HEAL implementation in HIVE is required once per platform, we were able to reuse them in all subsequent applications. Additionally, HIVE peripheral drivers are platform-agnostic and were reused across all platforms. We believe that the high ratio of code-reuse provided by HIVE surpasses the once-per-platform overhead incurred for implementing HEAL. Finally, we note that the results reported by our efforts are not indicative of the actual effort an average developer will incur. Instead, we have striven to optimize for code reuse to present the best-case scenario in bare-metal implementations.

4.4.8 Observations

Considering the results from our evaluation efforts, we observe that despite HIVE being relatively as fast as Lua, we stress that Lua was not created with resource-constrained embedded devices in mind [41]. Additionally, the static size of Lua was 12x larger than that of HIVE. We further observe that when compared to microPython and mRuby, HIVE would be the framework of choice among IoT platform developers due to its faster performance and significantly smaller footprint. We finally concur that due to the modularity and separation of concerns achieved with HIVE, we were able to quickly prototype and test with different combinations of peripherals and platforms with near-ZERO effort.

Table 4.2: Minimum requirements for HIVE

Operating system	Any RTOS with threading capabilities
Disk space	15KB
RAM	2KB

4.5 Limitations and Open Issues

The current HIVE design is not free of limitations; the minimum hardware requirements for the current version of HIVE are outlined in Table 4.2. In addition to the limitations some issues remain open for future work. Allowing external code execution can have severe security risks; a peripheral can have a malicious driver embedded that can completely disrupt the operation of an IoT platform. Therefore, our first venture in the short term investigates the options of using a certificate of authenticity for peripheral drivers to validate a peripheral before executing its logic. Furthermore, we intend to investigate the energy efficiency of using HIVE compared to bare-metal and other scripting engines. This investigation will then serve as a seed for future work on implementing the OTA reprogramming engine. In order to reach a larger scale of adoption for HIVE, we believe that enabling extensibility in terms of crowd-created content is paramount. Therefore, we will investigate the effects of enabling a package management system for HIVE reminiscent of Python's Pip or Node's NPM [90].

4.6 Streamlined IoT Development with HIVE

Our primary goal with HIVE is to accelerate IoT platform software development through providing a resource-constrained tailored IoT containerization engine. Towards this end, we discuss how HIVE achieves this goal by enabling some of the requirements we outlined in chapter Chapter 3.

4.6.1 Platform Independence

Software portability is one of the core attributes desired in software development [111], [118]. Creating an executable version of a software program in a different environment than initially developed enables developers to focus more on development and less on reinventing the wheel [91]. However, porting IoT firmware is challenging due to the diverse number of platforms and sensing elements used in different applications. HIVE applications are developed in a platform-independent scripting language that is compiled into HIVE specific byte-code; therefore, IoT systems developed using HIVE are inherently portable to different platforms. However, contrary to other scripting engines, HIVE's VSoC was designed with resource-constrained devices and IoT applications in mind. A single HIVE container VM requires less than 15KBs of memory, rendering it compatible with more low-end IoT devices when compared to other VMs. Furthermore, when combined with a multithreaded IoT gateway solution [16], HIVE can run a large number of applications operating on a single resource-constrained IoT sensor node. Alternatively, since the different parts of the application are isolated, HIVE enables computation offloading in the fog computing paradigm to complement efforts such as [73], [107] and [49].

4.6.2 Peripheral Development

In addition to platform independence at the application level, peripheral drivers also share the same level of flexibility. A driver developed in HIVE does not require porting to new platforms, as is the case with bare-metal approaches. Due to this benefit, a peripheral driver can be embedded within the peripheral itself. Therefore, as proposed in PILoT, a shift in the driver development responsibility is possible with HIVE; from a shared-responsibility model to a manufacturer-centric model. Instead of creating a driver for one (or more) platform(s) and relying on users/developers to correctly port it to other platforms, peripheral manufacturers can embed HIVE-compatible peripheral drivers on the peripheral memory. Developers can then use the peripheral without spending any porting effort seamlessly. Furthermore, manufacturers can ensure that the driver implementation is operating as designed.

4.6.3 True Plug & Play

Although PnP is not a requirement for DevOps, we argue that it is essential to enable CI on IoT platforms. The ability to hot-plug a peripheral into any IoT platform, dynamically load its driver, and update the application logic accordingly; has the potential to disrupt the IoT space the same way the USB did with personal computing. However, this vision requires a twofold effort, seamless hardware bus connectivity and software operation. HIVE uses a concept similar to that of the IEEE 1451.2 standard. A peripheral driver in HIVE consists of the device manifest descriptor and the device logic; we call this combined approach a Peripheral Phantom. As illustrated in Figure 4.6(a), the LED phantom identifies details about the peripheral, in addition to three standard scripts containing the peripheral logic. This approach allows peripheral

drivers to become completely decoupled from the underlying hardware platform and thus satisfies the software effort of requirement 4. Furthermore, peripheral phantoms allow the underlying implementation of driver interfaces to be decoupled from the actual driver logic, thus, enabling the virtualization of full IoT hardware platforms. We envision that when combined with a hardware bus effort, such as discussed in Chapter 5, IoT developers will have complete peripheral flexibility and thus reduce risk and time to market.

4.6.4 Over The Air Updates

To complete the full-cycle of CI/CD, the ability to support IoT sensor nodes post-deployment is necessary. Towards this end, HIVE enables updating the operational logic of the node without the need for a reboot. The HIVE-VSoC can dynamically load/unload scripts during runtime, resulting in zero-downtime upgrades. With the implementation of an additional engine, the VSoC would be able to schedule the reception of new QUEEN scripts through a connected communication peripheral. Such capability enables an IoT based edge cloud deployments when combined with efforts such as [108].

4.6.5 Enabling modern Software Development Life Cycles

One of the core challenges in IoT platform software development is the limited support for modern SDLC methodologies, specifically DevOps. Towards achieving a full CI/CD pipeline, HIVE enables IoT developers to implement their application without physical access to a hardware platform. Additionally, it allows them to virtualize the platform on the cloud, automate testing & deployments, hot-plug peripherals, and access a

production node memory stack for debugging purposes if needed. With such capability, HIVE can satisfy the first requirement.

4.6.6 Virtual Development Kits as a Service

Towards realizing the Virtual Development Kit as a Service (VDKaaS) paradigm discussed in Section 3.3.3, we outline how HIVE enables the concept of SDDK.

A virtualized hardware requires a Hardware Definition Blueprint (HDB) to define the capabilities of the underlying hardware. This HDB is what HIVE uses to map and expose the underlying hardware capabilities to the VSoC's HEAL layer. In order for the HDB to be compatible with HIVE, the HEAL layer must first be implemented. The SDDK can support several predefined hardware platforms and allow the user to implement their own HEAL to support any custom platform. Furthermore, to assist the user with implementing HEAL, the SDDK can provide several predefined sub-modules. For instance, support for popular peripherals and SoCs.

Additionally, an SDDK requires virtualized sensors/peripherals. PILoT defines the concept of a Peripheral Definition Blueprint (PDB), while HIVE defines the peripheral phantoms to contain the same information. The device manifest descriptor and the device logic in the phantom can be used to virtualize the peripheral. The third requirement involves a method to provide the application with test cases to simulate the operation of the device. Towards this end, the implementation of HEAL can inject a list of values to each peripheral. Alternatively, the peripheral driver on the phantom can inject a list of values into the honey pot through the $INIT$ BEE. Finally, the Integrated Development Environment (IDE) to be used with the SDDK can initialize HIVE with different initial values in the honey pot, as well as feed the hardware

peripherals with a series of values predefined by the user.

An optional feature in the SDDK is the ability to generate the equivalent C code for HEAL to program the real platform once development is complete. Such a feature enables users to seamlessly migrate from the SDDK to testing on the physical hardware with minimal effort. Finally, the SDDK must be able to generate and export all relevant HIVE files, including the BEEs (custom and peripheral) and the QUEEN.

4.7 Conclusion

Towards realizing the dream of I4.0, significant improvement in the IoT development process is necessary. In this chapter, we outlined the primary requirements for a streamlined IoT development pipeline that enables modern SDLCs. We then proposed the HIVE approach for both development and production applications to act as a catalyst for this pipeline. HIVE employs a low-footprint VM to enable the virtualization of IoT platforms and peripherals. Thus, a developer can start developing and implementing an IoT application with zero access to the hardware. Furthermore, HIVE isolates driver and application development to enable TPnP for seamless risk-free prototyping and zero-downtime post-deployment OTA upgrades for accelerated low-risk delivery. We implemented a proof-of-concept and presented the performance analysis of HIVE compared to bare-metal and other scripting engines. Based on the different test scenarios we analyzed, HIVE outperformed all other scripting frameworks with an 8x smaller footprint and 5x faster processing. Additionally, HIVE had very similar performance metrics compared to bare-metal. We further present our evaluation of the development effort using HIVE compared to bare-metal and demonstrate the significant time-cost saving due to enhanced separation of concerns and efficient code (re)utilization. Therefore, we conclude that HIVE is suitable for high-portability, quick iteration, and low-risk IoT development when compared to the alternatives.

In the next chapter, we present our vision towards enabling an IoT optimized TPnP solution. WhiteBus offers the simplicity of USB with the flexibility of bare-metal in resource-constrained IoT devices.

Chapter 5

Towards having an IoT friendly hardware bus for enabling TPnP

In Chapter 2 we reviewed multiple efforts that seem to implement a partial solution towards PnP functionality optimized for IoT platforms. However, a significant burden is incurred on the system designer and the sensor manufacturers to implement the proposed solutions. Nonetheless, the efforts that seem to offer a full PnP capability are not optimized for IoT or embedded system applications. With USB being the most common PnP bus, unfortunately, it is very resource demanding, making it impractical for peripheral manufacturers [9]. Therefore, manufacturers often use a lower rate, simpler interface bus. Some of the popular choices among manufacturers are I^2C, SPI, UART, Analogue and Digital are some of the popular choices among manufacturers [109, 84, 24, 58]. Using one of these interfaces, manufacturers are restricting an IoT system developer with their choice of communication bus for peripheral interfacing. Hence, it complicates developing a new IoT system, reducing the flexibility of using different peripherals post-production and intensifying the risk of investment for shareholders. Therefore, having the ability to seamlessly plug and unplug a peripheral

without the need to reconstruct the system architecture and re-interface with the IoT devices' MCU would reduce the development time and stakeholders' risk significantly. Due to the diversity of interfaces and transducer manufacturers, a gap exists between PnP functionality and IoT applicability. However, PnP capability is essential for IoT as it will provide researchers with a simple prototyping need, thus accelerating the research and development of IoT to realize the vision of I4.0.

Towards this end, we propose our vision towards enabling an IoT optimized TPnP solution. We start by introducing WhiteBus to tackle the interfacing challenges and satisfy requirements 2 to 4, then discuss how PILoT's TPnP solution can be achieved by combining WhiteBus and HIVE.

5.1 The WhiteBus Concept

Before we propose and discuss WhiteBus, it is necessary to outline what TPnP requires from a hardware bus. PILoT presents the TPnP paradigm and outlines a number of requirements to realize it in Section 3.1.1. We reiterate these requirements and introduce additional hardware-related requirements in this section.

Towards supporting TPnP on an IoT node, the concept of embedding the peripheral driver on the peripheral module was discussed, and an enabling solution was proposed in 4.6.3. In addition to HIVE 's containerization and architecture towards TPnP, additional hardware requirements are imposed.

1. The IoT node must be able to detect when a peripheral module is connected or disconnected.

2. The peripheral module should have a driver installed on its memory chip.

3. The IoT node must be able to read and understand the driver installed on the memory chip embedded in the peripheral module.

4. The IoT node must be able to initialize and use the appropriate interfacing bus according to the definition found in the peripheral driver.

Towards satisfying those requirements, a temperature sensor module with an onboard memory chip might be created and connected to an IoT node as illustrated in Fig. 5.1. Although this design would satisfy the hardware requirements, supporting multiple peripherals would not be trivial. Furthermore, the pins are not optimized and would not allow many peripherals to be connected. Our proposed system (WhiteBus), aims to resolve the limitations of the example illustrated in Fig. 5.1 and complete the full E2E vision of the TPnP paradigm.

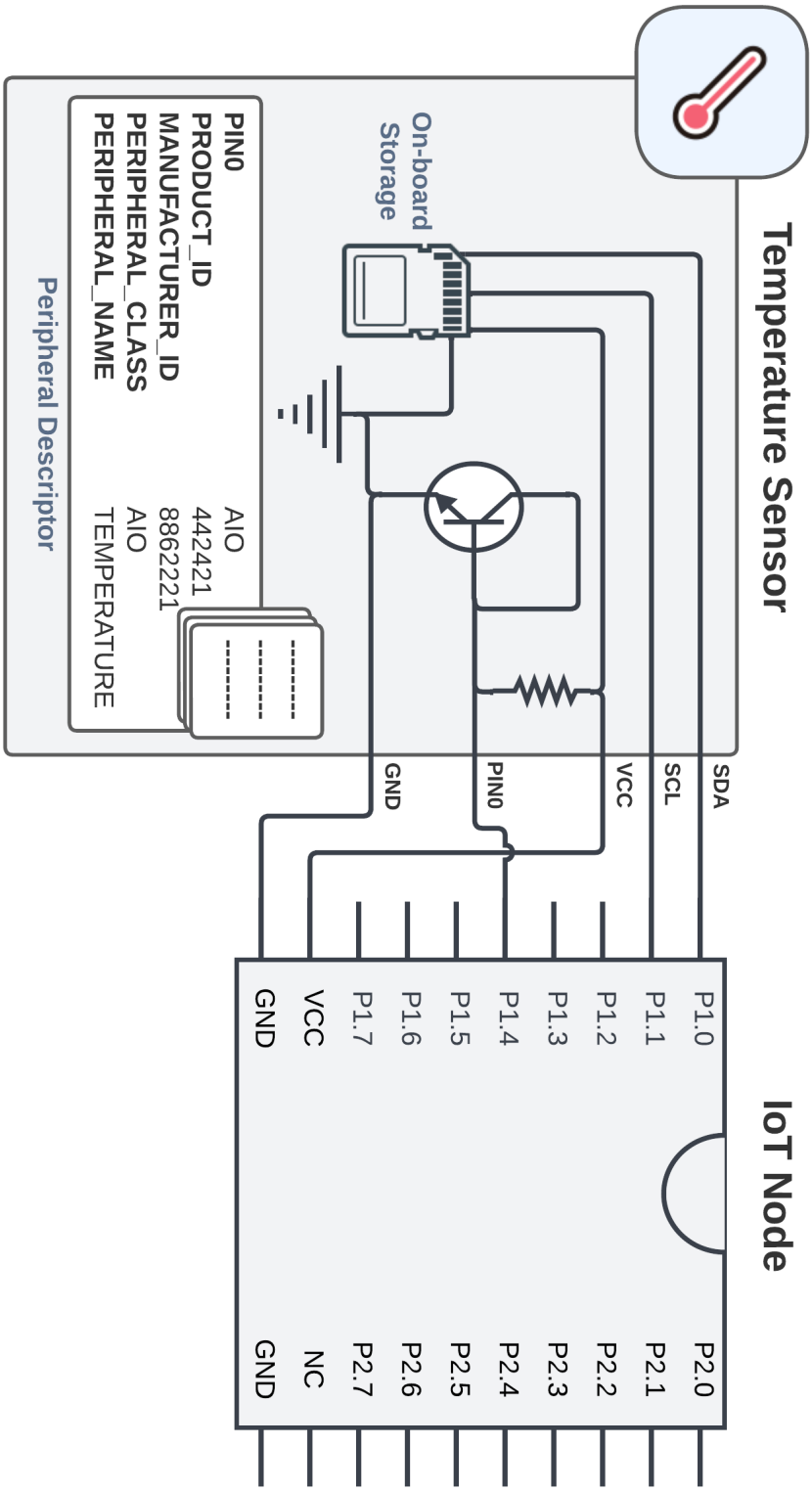


Figure 5.1: An example approach towards satisfying the hardware TPnP requirements

5.2 WhiteBus Architecture

To optimize pin usage and simplify TPnP support on IoT devices, we propose WhiteBus, an interface multiplexing and abstraction module tailored for TPnP functionality with PILoT. WhiteBus aims to expose the current standard interfaces to peripherals through a unified bus, significantly reducing the burden on the designers and manufacturers. Such property combines the simplicity of operation in USB with the simplicity and flexibility of design in bare-metal interfacing. Such flexibility enables TPnP functionality that is optimized and targeted at IoT applications while maintaining platform independence.

Our interface bus aims to minimize the burden on system developers and maximize the return on investment for the companies; this is possible due to the ability to develop a genuinely generic IoT device that can be customizable during run-time by connecting different sensors. Developers can use any peripheral on the market from any manufacturer and connect it to their IoT device with minimal configuration.

There are three main parts to WhiteBus, the dynamic WhiteBus Universal Serial Interface (WBUSI), the WBM, and the WhiteBus Master Serial Interface (WBMSI). In order to add WhiteBus support to peripherals and IoT nodes, a WBM needs to be used. Figure 5.2 illustrates how a WBM facilitates interfacing between a generic MCU and multiple peripherals through WBUSI and WBMSI. One of the motivations behind developing WhiteBus is to facilitate TPnP functionality. Therefore, peripheral identification is a crucial requirement. A concept similar to the TEDS from the IEEE 1451 standard is to be used in a peripheral. A memory chip attached to the peripheral contains a descriptor file (written by the manufacturer). WhiteBus uses the descriptor file to identify the type of peripheral, manufacturer, interface, and other metrics used

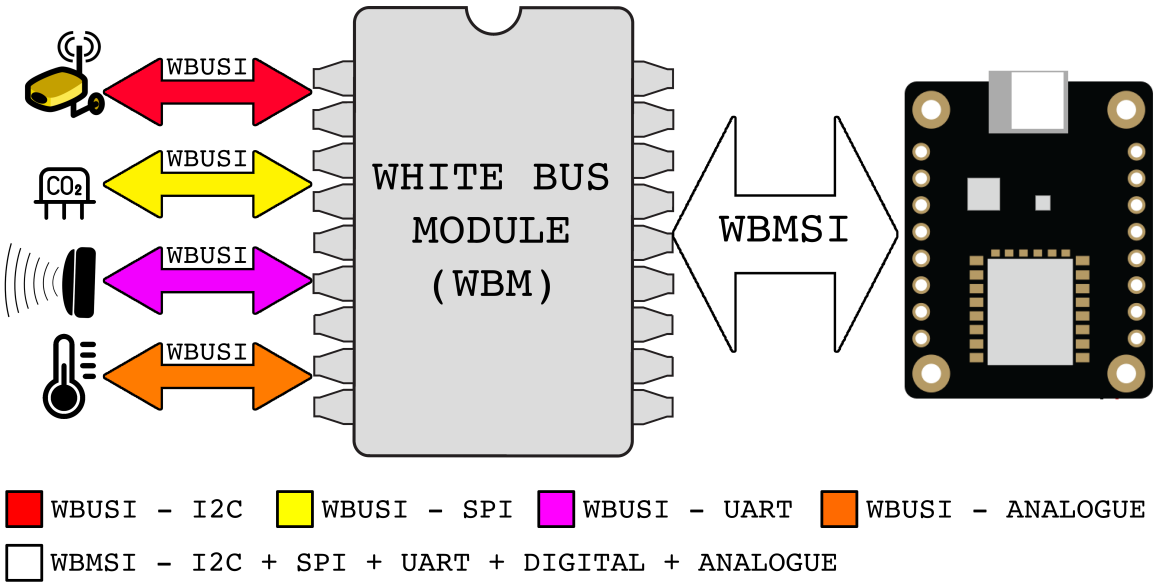


Figure 5.2: Illustration of the WhiteBus architecture concept

by the MCU. Such information is useful to map and address the correct interfaces to the connected peripheral. HIVE introduces the concept of peripheral phantoms that include this information.

5.2.1 The WhiteBus Serial Interface (WBMSI)

Finally, the WBMSI consists of nine lines. The interface connects the WBM with the common serial interfaces on an IoT node in order to connect the IoT device to any peripheral. The interfaces supported by a WBM are I^2C, SPI, UART, analogue and digital input/output. WBMSI allows unconnected interfaces, such that if an IoT device does not have one of the interfaces mentioned earlier, it is still able to utilize the benefits of WhiteBus.

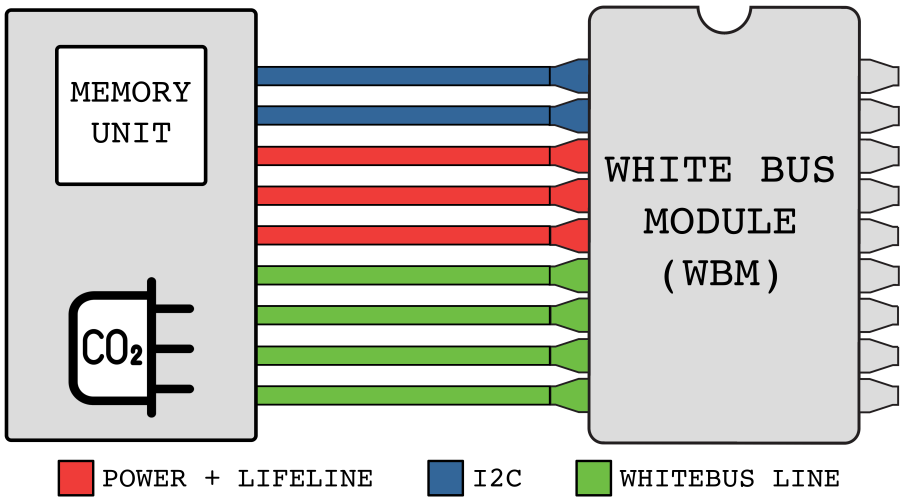


Figure 5.3: Details of a WhiteBus interface

5.2.2 The WhiteBus Universal Serial Interface (WBUSI)

A WBUSI concept consists of nine lines, illustrated in Fig. 5.3. Since each peripheral houses a memory chip, they must use an I^2C interface; hence the Serial DAta (SDA) and Serial CLock (SCL) lines are fixed for all peripherals and are used to identify the connected peripheral. The LifeLine is used to detect connection and disconnection of the peripherals, while four WhiteBus Lines (WBLs) are used to support the different modes of operation. Common power supply lines are included in operating the peripheral. We note that the physical connector for this bus is out-of-scope.

5.2.3 The WhiteBus Module (WBM)

The WBM is the core of the WhiteBus system. This module can be implemented using transistors or using Field-Programmable Gate Array (FPGA) controllers. Power efficiency and cost would need to be balanced for this module to become attractive to IoT developers. We propose and discuss an overview of how this module would be implemented using transistor logic. The WBM architecture consists of four core units:

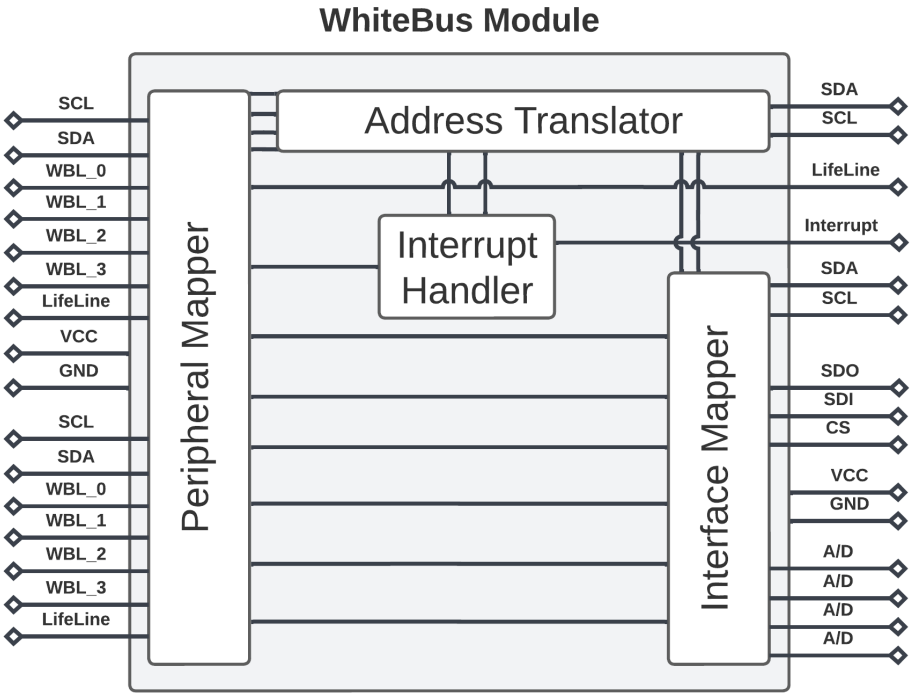


Figure 5.4: The internals of a White Bus Module. (does not show the full number of pins of the module)

address translator, interface mapper, interrupt handler and peripheral mapper. The total number of pins required in our initial design of the WBM is one-hundred and sixty. Figure 5.4 illustrates the internal architecture of a WBM. The function and operation of each unit are described in the following subsections.

Address Translator

The address translator receives serial data from the node and converts it into parallel data fed to the interface mapper and peripheral mapper. The translation operation divides the received address into different parts and uses a look-up table to match each part to its appropriate function. The current design uses four parallel lines for peripheral addresses, resulting in a total of sixteen peripherals. Since each peripheral

would be connected to the WBM through a single WBUSI, this would give us a total of one-hundred and forty-four pins that would exit from the WBM. Although the WBM could combine all the Common Collector Voltage (V_cc), Common Ground (GND), SCL and SDA lines into four lines, reducing the number of pins required on the WBM to 80; every WBUSI port would require the full nine lines. When a peripheral is first connected to one of the peripheral mapper ports, the peripheral mapper translates the port number to a four-bit code and stores it in the look-up table. When the node requests to be connected to a certain port, the address translator decodes the message and uses the look-up table to switch the routing on the peripheral mapper. The communication protocol between the node and the address translator involves the node initiating the communication. The first message from the node to the translator consists of a four-bit command, followed by an eight-bit port Identification (ID), followed by a two-bit interface mode ID, and finally terminated by two digital 0's. The response from the translator consists of a one-bit success flag, followed by a four-bit peripheral status code, and terminated by a three-bit number representing an interrupt.

Interface Mapper

The interface mapper is responsible for multiplexing the interface lines received from the node to the peripheral mapper. It receives multiple different interfaces from the IoT node to be connected with the appropriate peripherals and an interface mode code from the address translator. The interface mapper then selects the appropriate interfaces based on four distinct interfacing modes, magenta, orange, yellow, and red. Each of these modes involves a combination of the input interfaces to be connected to

the input of the peripheral mapper and discussed further in Section 5.2.4.

Peripheral Mapper

The peripheral mapper receives a single interface mode from the interface mapper and outputs multiple WBUSIs, each to be connected to a different peripheral. A peripheral map code received from the address translator is used to route one of the peripheral's WBUSI with the input interface mode pins.

Interrupt Handler

Each of the WBLs on the WBUSIs acts as an interrupt when operating in the digital Input/Output (IO) mode, and the port is not directly routed to the node. When an interrupt signal is detected, the interrupt handler accesses the peripheral look-up table and signals the address translator with the port that triggered the interrupt. The interrupt handler also triggers the interrupt pin that is directly connected to the node to notify it of a pending interrupt.

Once the node detects the interrupt signal on the interrupt pin, it should request a direct connection with the interrupting port. The node sends an interrupt query command to the address translator, which then responds with a message containing the peripheral associated with the interrupt. Finally, the node requests a direct connection with the peripheral to which the address translator initiates and responds with a success message.

5.2.4 Operation Concept

Any PnP system requires three features, the ability to detect connection and disconnection, the ability to identify the connected peripheral, and driver support for the connected peripheral [88]. Current IoT nodes embed the driver implementations in the firmware directly, but no method to easily plug and unplug (hot-plug) these sensors without a reboot or significant reprogramming.

An IoT system that uses WhiteBus would connect an IoT node with multiple peripherals using a WBM. A WhiteBus system is based on a star topology with the IoT node initiating and controlling all communications with the peripherals. In an IoT application. The node schedules the communication with all the current connected peripherals and ensures that the time slot each peripheral receives is fair. Figure 5.5 illustrates an example peripheral connected to an IoT node through a WBM.

The node keeps a list of all currently connected peripherals and schedules communication with each. To initiate the connection with one of the peripherals, the node requests the connection from the WBM, which maps the correct interfaces to the requested peripheral, providing the node with a direct link.

Connection/Disconnection Detection

The first stage of PnP is the detection of the status change of a peripheral. In WhiteBus, the node periodically polls the LifeLine of each peripheral port to detect connection/disconnection. Once a peripheral is detected, the internal peripheral map is updated.

Peripheral Identification

When a peripheral connection is detected, the node first checks the internal map to find if this peripheral has previously been identified. Then, the node requests a direct I^2C connection with the peripheral to read the peripheral's ID from its descriptor file. If the ID is unchanged, the node skips the driver exchange stage; however, if an ID never existed or has changed, the node will perform the driver exchange stage first. The descriptor file, similar to TEDS, contains information about the peripheral, including its ID, the manufacturer's name, and the interface required. A different concept of peripheral identification relies on the WBM to indicate to the node if there has been a change in the connected peripheral. The concept choice's trade-offs include the complexity of building the WBM versus additional effort on the IoT node developer.

Driver Exchange

Regardless of the concept used for identification, if the node needs to read the driver from a connected peripheral, it will request a direct I^2C connection from the WBM. Once the connection is established, the node can read the onboard memory unit's contents, including the driver and descriptor. If the peripheral does not have onboard memory, the developer may pre-load the node firmware with the peripheral driver or remotely download it during operation.

Communication

Finally, before the node can communicate with the peripheral, it decides on the interfacing mode required based on the peripheral descriptor. Our initial design

includes four interfacing modes of operation, Red, Yellow, Magenta, and Orange.

- **Red Mode** asks the WBM to create a direct I^2C and LifeLine connections. Thus, directly routing three lines (SDA, SCL, and the LifeLine).
- **Yellow Mode** asks the WBM to create a direct SPI, and LifeLine connections. Thus, directly routing five lines (Serial Data Out (SDO), Serial Data In (SDI), SCL, Chip Select (CS), and the LifeLine).
- **Magenta Mode** asks the WBM to create a direct UART, and LifeLine connections. Thus, directly routing three lines (Transmit (TX), Receive (RX) and the LifeLine).
- **Orange Mode** asks the WBM to create a direct A/D,and LifeLine connections. Thus, directly routing all A/D lines and the LifeLine.

Depending on the design of the WBM, it additionally routes A/D lines regardless of the mode; to optimize the use of all the WBUSI pins. Figure 5.6 illustrates the operation flow of a WhiteBus system.

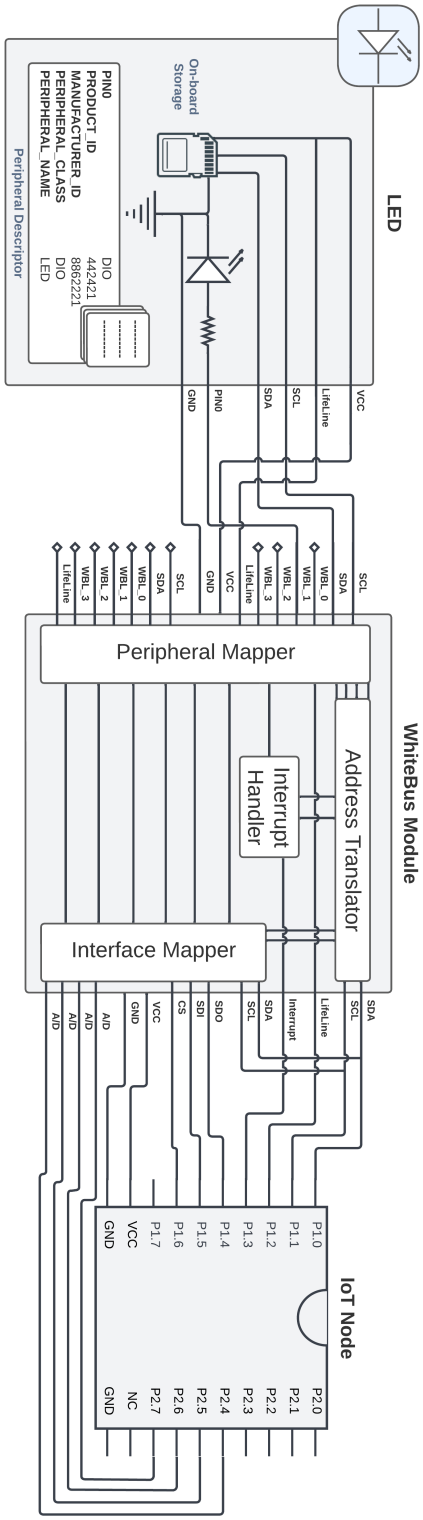


Figure 5.5: An example peripheral connected through a WBM

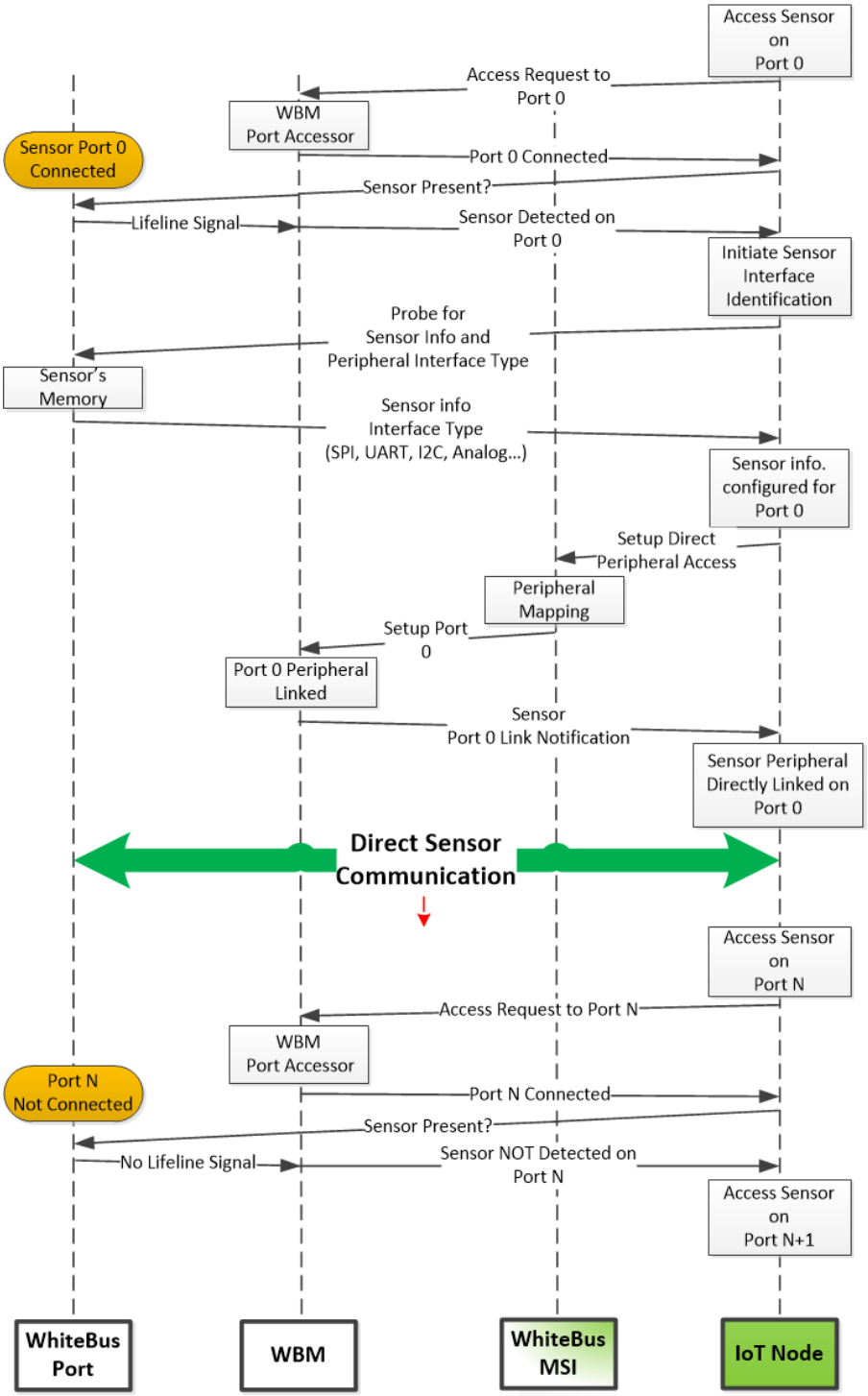


Figure 5.6: Operation flow of WhiteBus

Table 5.1: Energy Assessment of Selected Buses

	Bare-Metal	**USB**	**WhiteBus**	**mikroBUS**
Energy Impact	LOW	HIGH	MEDIUM	LOW

5.3 System Assessment

In order to ensure the applicability of WhiteBus, we use four fundamental properties to compare it with select PnP and interfacing efforts qualitatively. We focus on key differences such as the cost/complexity to integrate, the impact on the IoT node energy consumption, the integration/development process, the peripheral scalability, and TPnP support. It is critical to note that, unlike the other interfacing efforts, WhiteBus combines interfacing buses currently used in embedded systems and multiplexes between them to offer a dynamically changing bus.

5.3.1 Energy impact

One of the significant considerations for any IoT device is the energy consumption and lifetime of the device. Therefore, it is crucial to ensure that the benefits brought by using WhiteBus will not impact the overall lifetime of the device.

When a developer decides not to use a bus system, each peripheral is directly connected to the IoT device through one of the interfaces provided by the MCU. Such a bare-metal approach requires the circuitry used for each interface to be designed and implemented by the developer. Hence, the energy impact of interfacing a specific peripheral relies on the developer's hardware design. Our research assumes that the developer's hardware design is impeccable and does not involve current leakage issues. Therefore, assuming a highly optimized circuitry, directly connecting the peripheral

Table 5.2: Cost & Complexity Assessment of Selected Buses

	Bare-Metal	**USB**	**WhiteBus**	**mikroBUS**
Manufacturer	LOW	HIGH	MEDIUM	MEDIUM
System Designer	HIGH	HIGH	MEDIUM	MEDIUM

to the MCU serves as a benchmark for comparison with other interface bus systems.

MikroBus's energy impact depends on the manufacturer's hardware design of the sensor board. A best-case scenario would be equivalent to the bare-metal method in which the MikroBus sensor board is optimized regarding hardware design. On the contrary, to use USB on an IoT device, a microcontroller is required on the peripheral's side, which entails a higher energy impact. Furthermore, USB uses a 5v level instead of the common 3.3v used in IoT and embedded systems. Since the authors in [99] do not measure the energy impact of their proposed research, we assume that the use of an FPGA-based device would drastically elevate the energy consumption.

With WhiteBus, energy consumption in a best-case scenario is equivalent to that of the regular bare-metal implementation, along with the additional energy consumption incurred by the presence of the WBM. However, such an impact on energy consumption remains marginally less than that of USB because WhiteBus does not require additional microcontrollers.

5.3.2 Cost and complexity

The complexity sustained by an interfacing method directly affects the cost of implementation for both peripheral manufacturers and system designers. Therefore, it is essential to design an interface bus with reduced complexity in mind.

A bare-metal interfacing method has the lowest complexity for peripheral manufacturers. A manufacturer can design a peripheral using any interfacing system they prefer and delegate all the development complexity solely to the developer. However, this results in a daunting list of tasks on the system designers' side. Significantly shifting the complexity and cost away from the manufacturer towards the designer. Such a task list usually involves the design of the circuitry required for each type of interface.

By requiring a specific bus interface on both the peripheral and IoT system, mikroBUS and USB divide the complexity and cost between the system designers and the peripheral manufacturers. However, mikroBUS provides a combined bus interface with dedicated pins of the standard device peripherals. At the same time, USB uses a standardized bus along with drivers to provide a universal abstraction of data/control regardless of the sensors' native communication peripherals. Therefore, mikroBUS requires a large pin-out footprint while USB requires a microcontroller capable of its protocol on both the peripherals' side and the IoT system's side.

WhiteBus requires an EEPROM for identification purposes to be integrated on the peripheral, shifting most of the burden on the IoT system designer's side. Such design choice provides flexibility to the peripheral manufacturers while maintaining the same level of control on the designer's side. WhiteBus also reduces the burden on the designer using the WBM, in which the peripheral mapping is abstracted from the system designer and handled by the WBM. Another benefit of using a WBM is the reduction of pins compared to that of mikroBUS.

Table 5.3: Integration & Development Assessment of Selected Buses

	Bare-Metal	**USB**	**WhiteBus**	**mikroBUS**
Effort Required	HIGH	HIGH	LOW	MEDIUM

5.3.3 Integration/development process

The traditional integration process without a bus interface involves developing a set of drivers for each peripheral. These drivers are usually tightly coupled with the HAL, followed by the development of a Sensor Data Acquisition Algorithm (SDAA). The sole purpose of the HAL in a bare-metal setup is to link the pins on the microcontroller and its peripherals to the interface used by the sensor, while the SDAA uses the HAL to communicate with and operate the sensor.

Interfacing USB involves the use of an API that abstracts the protocol layer of the standard; the API is used to identify the connected peripheral and provide a high-level interaction with the programmer. However, the USB API is usually complicated and impractical for an IoT device due to its hardware requirements [99]. WhiteBus and mikroBUS are similar to the bare-metal integration regarding the HAL development requirements. However, mikroBUS provides the developers with a framework to simplify the development of the SDAA. In contrast, WhiteBUS, when combined with HIVE, provides the developer with a high-level API for identifying the connected sensor as well as abstracting an interaction layer for more straightforward development of the SDAA.

5.3.4 Sensor scalability

One of the significant benefits of using an interface bus is connecting a greater number of peripherals than the system hardware can support; this is one of the key reasons

Table 5.4: Scalability Assessment of Selected Buses

	Bare-Metal	**USB**	**WhiteBus**	**mikroBUS**
Scalability	LIMITED	HIGH	HIGH	LIMITED

behind the popularity of USB [83].

A bare-metal approach limits the peripheral scalability of an IoT device to the hardware interface capabilities. Therefore, using a bus system increases the number of peripherals the IoT device can interact. The mikroBUS standard enforces standardized pin-mapped headers on the peripheral and the IoT device. Some peripherals are not shared between sensor boards (i.e. I^2C); therefore, in a best-case scenario, the mikroBUS scalability is equivalent to bare-metal implementation.

USB and WhiteBus use a similar approach for scalability, in which a controller schedules different time-slots between the connected peripherals. Regarding the bus interface, USB uses hubs to extend the number of peripherals connected at a particular time. At the same time, WhiteBus utilizes a WBM that supports a limited number of connected peripherals at a time using the WBUSI ports. In order to increase the number of connected peripherals, a different WBM with a more significant number of WBUSI ports would be needed. Alternatively, WBMs can be combined akin to how USB hubs work.

5.4 Challenges & Future Directions

IoT microcontrollers are diverse and can have six pins to hundreds of pins [85]. Therefore, controllers with a low number of pins usually lack the support of all the peripherals supported by WhiteBus (e.g., lack of SPI or UART functionality). Moreover, some microcontrollers lack the support of peripheral logic due to design cost constraints regardless of the number of pins. Since WhiteBus is designed to provide peripheral mapping and multiplexing to sensor ports, it is limited by the available peripherals on the microcontrollers. Also, the support of I^2C is crucial to the operation of WhiteBus as it is used to access the sensors' memory units.

Fortunately, implementing the software-based I^2C driver is simple and has a minimal footprint on most microcontrollers. This implementation would allow WhiteBus to function effectively despite lacking other interfacing peripherals. The developers, in this case, can read the sensor interface information and determine the feasibility of supporting such a sensor. Developers can also incorporate other software-based peripheral drivers and then link the corresponding pins to the WhiteBus WBMSI, given that the microcontrollers can support them.

Another challenge lies in the diversity of voltage levels in sensors, the standard 3.3v level used in IoT is not supported by every sensor. Such a difference in voltage level requires a Power Management Unit (PMU) to supply the required supply voltages. We believe that the sensor manufacturer should implement the voltage level change circuitry into their sensors, increasing the cost; however, it remains a better option than incorporating a microcontroller for interfacing systems such as USB.

A third challenge relates to the advanced sensors that usually incorporate multiple feedback (i.e., interrupt) lines. WhiteBus limits the number of feedback lines available

to the sensor based on the peripheral mode supported by the sensor; for instance, the magenta and red modes provide only two lines for feedback. Sensors that require multiple feedback lines may require the manufacturer to integrate an I^2C to the I/O bridge to expose the feedback lines over the same I^2C bus used by the sensor's memory; however, with a different address. The sensor EEPROM will include, in this case, the identification and address for the I^2C to the I/O bridge. The IoT developers must pull the feedback status via the bridge to check for the sensor status.

5.5 Summary

In order to excel the vision of I4.0 rapid development in IoT is required. One of the factors that throttle the advancement in IoT is the lack of standardization in the industry. Therefore, a solution such as TPnP is in demand to overcome this heterogeneity.

In response to that demand, we proposed a dynamic peripheral interface (WhiteBus) that simplifies the integration of peripherals to any IoT system. WhiteBus combines the peripheral interfaces available on an IoT device and connects only the interfaces a specific peripheral uses, abstracting the actual interface to the developer and enabling TPnP functionality. Our research requires an EEPROM to be integrated with the peripheral similar to the IEEE 1454 standard, allowing the IoT device to identify the connected peripheral.

We compared WhiteBus to notable efforts in the literature and found that WhiteBus provides a good balance between scalability, complexity, and power consumption. We also noted that WhiteBus introduces some challenges; for instance, we proposed using software-defined peripherals when using limited-functionality microcontrollers. Using sophisticated peripherals might require extra digital feedback connections that WhiteBus does not support. Hence, we proposed using additional simple circuitry on the peripheral to support WhiteBus seamlessly.

Our research only scratches the surface of possibilities with WhiteBus; hence we consider some future directions that would improve the interface. It is possible to add self-configurable TPnP functionality to WhiteBus by integrating it with a HIVE-compatible IoT node. Another ambitious direction towards simplifying peripheral integration is implementing an interface translation unit within the WBM. The purpose

of this unit is to allow any peripheral to interface with any IoT device regardless of the available interfaces.

In the next chapter, we explore a practical use-case for an IIoT project that outlines the benefits reaped from using PILoT during development.

Chapter 6

Achieving Clairvoyance with PILoT

Monitoring and tracking assets is a tedious and inaccurate endeavour. With current systems, it is ubiquitous that no one can be held accountable for lost and damaged assets due to a lack of accurate tracking. Towards this end, a global shipping firm plans to employ an IIoT solution to keep track of shipping containers. The firm contracts an IoT solutions company (Smart Sensing Solutions (S3)) to design the customized system. In collaboration with Queen's University, the IoT company embarks on a five months journey to design, test and implement a PoC. A venture that traditionally takes more than twelve months. In this chapter, we describe our journey with the IoT company and the shipping firm to deliver the project using PILoT.

Clairvoyance - The supposed faculty of perceiving things or events in the future or beyond normal sensory contact

6.1 The Clairvoyant Solution

Despite the availability of different asset tracking solutions, none of the options satisfied the complex requirement of the shipping firm. As a result, a custom solution was necessary. In this section, we discuss the requirements of the system as provided by the firm. However, before discussing the requirements, we briefly overview the shipping process.

6.1.1 Overview

Almost 90% of all world trade is transported by shipping containers [22]. Throughout the container journey, four different parties are involved: the Sender, Shipper, Logistic Provider, and Service Provider.

- **Sender** is the company that requires shipping the cargo from one location to the other. In certain circumstances, the Sender can also be the Receiver. This party's entire supply chain is affected if a delay occurs during container shipping. Therefore, tracking the journey of containers can drastically reduce risk.

- **Shipper** is the company responsible for transporting the container from the source to the destination. The company needs to be able to track containers since it will be held liable if the trip was not successful.

- **Logistic Provider** is responsible for coordinating the process on behalf of the *Sender* by using different shipping providers.

Clairvoyant - A person who claims to have a supernatural ability to perceive events in the future or beyond normal sensory contact

- **Service Provider** is usually a firm responsible for helping the *Sender* manage their supply chain using multiple logistic providers.

For the purposes of this project, tracking is essential for both the *Sender* and *Shipper*; however, the other parties involved use the data generated by the tracking system heavily.

6.1.2 Requirements

1. Due to competitive and funding constraints, the shipping company requires a working industrial PoC within six calendar months.

2. The primary objective of the project is the ability to know the precise location of any shipping container throughout its journey. The precise location includes the global and local positions within a facility. This requirement necessitates that containers with an attached tracking module are identifiable and localizable.

3. Containers have different sizes and properties. Additionally, it is common for shipping containers to contain different types of items throughout their lifetime. A container shipping children's toys today could be shipping volatile chemicals next week. Therefore, the tracking module must allow customizable sensory peripherals with minimal effort. Preferably, sensor replacement is done by anyone in less than five minutes.

4. Containers on a freight ship must also be trackable. Therefore, satellite communication technology must be compatible with the system.

5. Different companies are responsible for the containers at different parts of their journey. Additionally, the firm collaborates with a container manufacturer. It

provides a "one-trip" [1] container service. Therefore, the tracking module must be easily removed/uninstalled by anyone (authorized) in less than one minute.

6. If a container is open during the journey, the system must be able to notify key personnel. Additionally, if the tracking module is tampered with, the system must alert a predefined manager.

7. Throughout the journey of a container, it is exposed to harsh environmental conditions. The tracking module must withstand these harsh conditions without affecting its performance.

8. Once the shipping firm receives an order, it either sends a container to the manufacturer or receives a container ready for shipping from the manufacturer. In both scenarios, the firm must attach the tracking module before proceeding. Until the firm delivers the container and renounces its responsibility for the container, the tracking module must remain active. Therefore, the battery life of the tracking module must be longer than four months. Furthermore, the batteries must be easily replaceable.

During the time of this project, the available solutions in the market were either based on GPS or short-range wireless communication (e.g. Bluetooth Low-Energy (BLE)). Both of which do not satisfy all the requirements of this system. For instance, GPS requires a clear line of sight to localize a container. Furthermore, it does not provide a precise location within the freight ship to satisfy requirement 2.

[1] A one-trip shipping container refers to containers manufactured then shipped with cargo to minimize the shipping cost of empty containers

6.1.3 Implementation

A custom IoT system was to be designed and implemented to satisfy the requirements outlined earlier. However, the conventional development methodology would not be suitable due to the strict timeline imposed by requirement 1. Furthermore, seamless collaboration was of utmost importance for the three stakeholders (shipping firm, IoT solutions company, and Queen's University). The shipping firm was located in East Asia, while both Queen's University and the IoT company were in Canada. However, the hardware engineer responsible for the hardware design was in the Middle East. Together, the team was able to build Clairvoyant in less than five calendar months using PILoT.

PILoT subdivides the development process into four phases previously discussed in 3.2. The following sections discuss how this project used each phase in PILoT to achieve its goals.

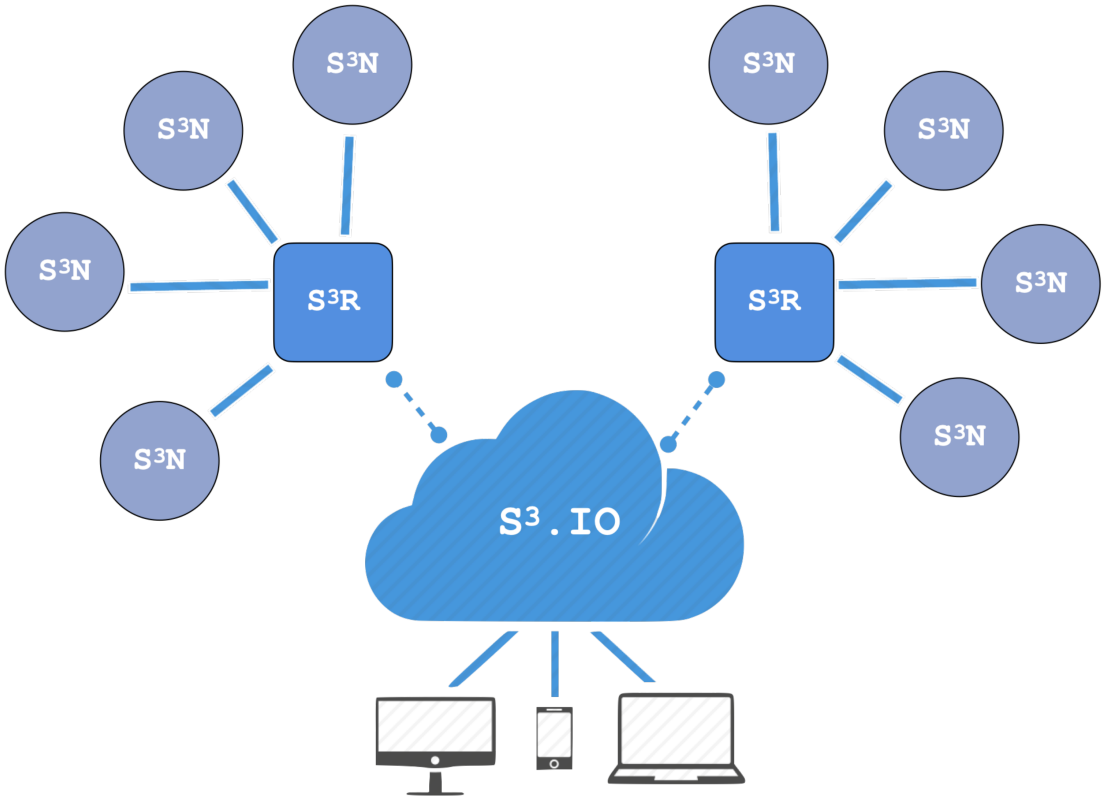


Figure 6.1: Clairvoyant network architecture

6.2 Design & Plan Phase

Clairvoyant comprises four core concepts working in tandem to achieve the requirements outlined earlier in this chapter. We discuss each of these concepts in the following sub-sections. In this phase, the core goal was to put together all the infrastructural blueprints needed to achieve the four core concepts of Clairvoyant. We first discuss these concepts and then outline the blueprints used to realize them. Figure 6.1 illustrates an overview of the Clairvoyant network architecture.

1. **Dynamic & Context-Aware Sensing** Due to the versatility of shipping containers, it is essential that the IoT nodes used on containers can dynamically

change operation according to the current context. Such feature requires OTA reprogramming driven by the router nodes, in addition to a zero-downtime/no-reboot capability on the sensor nodes. Finally, the nodes attached to the containers can allow a hot-plugging mechanism for sensor replacement.

2. **Reporting & Alerting** The requirements dictate a near-real-time operation of the IoT network. Despite this requirement, only a certain set of data/information needs to be reported immediately. Table 6.1 outlines the different data transfer involved and their reporting priority. We note that the dashboarding software (or user application) requires less than thirty seconds of latency between reports; however, the sensor data relayed from the nodes to the backend require less than an hour. When considering a few hundred or a few thousand container nodes, we can see that the dashboarding software would be able to view new data in less than thirty seconds and would not have to wait for an hour. However, in a worst-case scenario in which a low number of container nodes exist in the network and all report within a few seconds, it is possible that the dashboarding software would have latencies of up to one hour.

3. **Container Localization** A vital aspect of the Clairvoyant system is tracking containers across their trip. Due to the different layouts and structures of the locations in which a container passes, we used a zoning mechanism to identify which zone a container is in at a given period in time.

   Zoning involves the detection of the entry/exit of containers at the borders of certain areas in the pipeline, for instance, when a container is leaving the dock and onto a freight ship.

Table 6.1: The data generated and reported in Clairvoyant

Source	Destination	Data	Maximum Latency
Container Node	Router Node	Sensor Data	<1 hour
Container Node	Router Node	Zone Change Data	<10 seconds
Container Node	Router Node	Alert Data	<1 second
Router Node	Cloud Backend	Sensor Data	<1 hour
Router Node	Cloud Backend	Zone Change Data	<10 seconds
Router Node	Cloud Backend	Alert Data	<1 second
Cloud Backend	Dashboard	Sensor Data	<30 seconds
Cloud Backend	User Application	Sensor Data	<30 seconds
Cloud Backend	Alerting Systems	Alert Data	<1 second

In addition to identifying the zone in which a container is located, it is also essential to identify the exact location of the said container within the zone its located. It was necessary to use a combination of satellite positioning and proximity positioning to localize the containers within a zone. Satellite positioning alone would not suffice since containers are packed tightly together, preventing satellite communication modules from accurately positioning the container. Figure 6.2 shows an image of the containers taken during on-site deployment.

The first phase of PILoT (the design & plan phase) consists of four levels of abstractions, hardware, network architecture, deployment, and rules & policies. Towards achieving the concepts outlined earlier, we discuss each level as follows.

6.2.1 Hardware Abstraction

PILoT 's hardware abstraction describes the hardware used in the project. In Clairvoyant, it was subdivided into the cloud and a set of routers, nodes, and modules and outlined as follows.

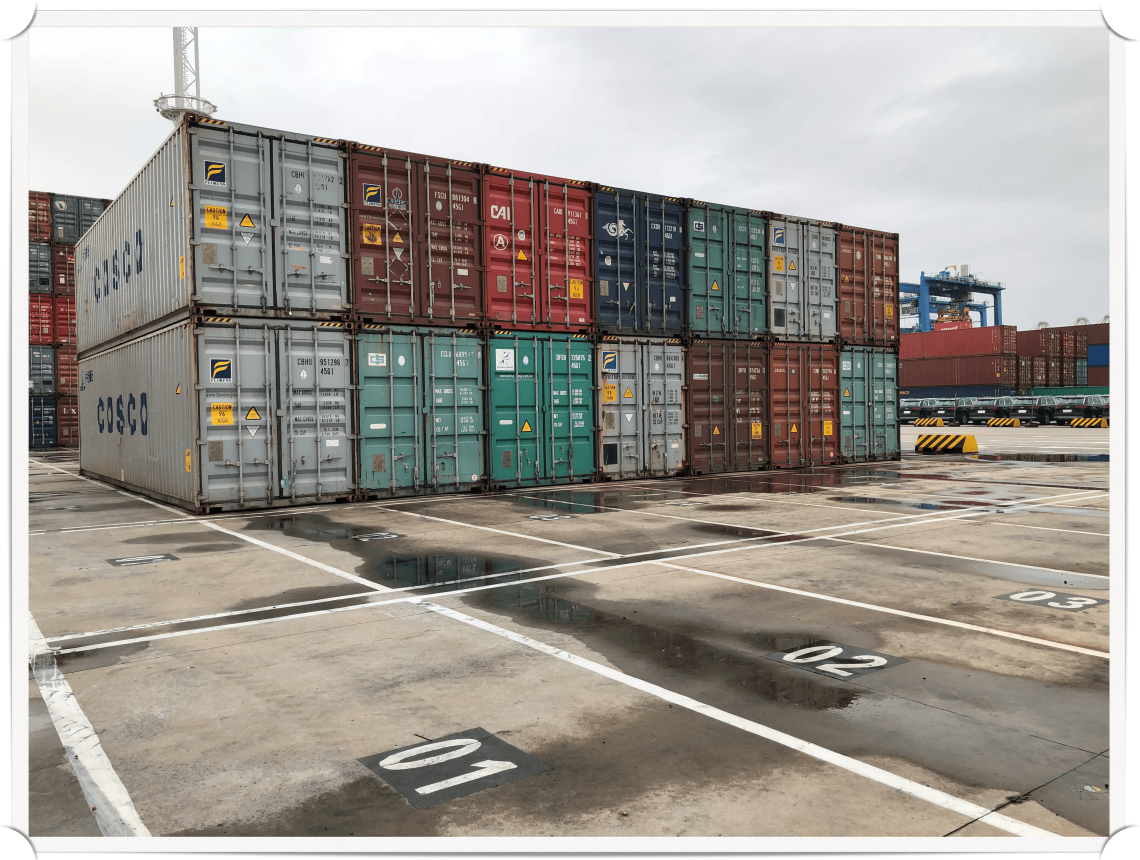


Figure 6.2: An image taken on-site of containers packed tightly together. It is common to have containers stacked up to six on top of one another

1. **S^3.IO Cloud (S^3.IO)** is the backend infrastructure of the project. At the time of development of the project, the prototype for the PILoT cloud infrastructure was not implemented. Therefore, the S^3.IO Cloud (S^3.IO) was built to implement some core elements and become a representative of its PILoT grandfather. The S^3.IO is responsible for collecting, processing and storing S^{3}N sensory data and hosting a dash-boarding infrastructure queried by the user application.

2. **Routers** are the units responsible for collecting information from nodes and relaying it to the S^3.IO. For our discussion, a node that collects information and

relays it to a router is not considered a router; we consider it a node with data collection capabilities. A GPS module can be connected to any S^3 Router (S^3R), allowing real-time geo-tracking if required in the deployed area. Clairvoyant had three types of S^3R as outlined below.

(a) **S^3R-Dock** is the type of router installed on the docks. Due to their location, they are equipped with both Ethernet and WiFi.

(b) **S^3R-Ship** is the type of router installed on freight ships. This router is fitted with a Beidou satellite transceiver module to provide end-to-end trip tracking.

(c) **S^3R-Truck** is the type of router installed on transport trucks. A GSM/LTE module is attached to supply the router with internet connectivity.

Algorithm 6.1 outlines the routers abstraction section in the Clairvoyant blueprint. One thing to note is that all routers have LoRa® communication capabilities to be able to receive reports from the nodes (S^3N). The core differences between the routers are the communication method used to relay the information back to the backend (S^3.IO).

3. **Nodes** are the core units that collect data and report it back to a router. The project involved three main types of nodes, as outlined below. Algorithm 6.2 outlines the nodes abstraction section in the blueprint.

(a) **S^3 Node (S^3N)** is the main IoT node to be attached to the container door. It is responsible for collecting all the information from the other nodes on the container and relaying it to a router node. The S^3N uses multiple wireless communication protocols, each for a specific purpose. ZigBee is

Algorithm 6.1 The router abstraction for Clairvoyant

```yaml
hardware_templates:
  - name: S3R-Dock
    type: router
    operational_logic: dock_router_logic
    features:
      - name: mS3L
        type: radio_controller
        driver: lora_gw_trx
        parameter_overrides: dock_lora_overrides
      - name: mS3E
        type: radio_reporter
        driver: ethernet_driver
        parameter_overrides: dock_eth_overrides
      - name: mS3W
        type: radio_reporter
        driver: wifi_driver
        parameter_overrides: dock_wifi_overrides
      - name: mS3Sat
        type: sensor
        driver: gps_driver
        parameter_overrides: dock_gps_overrides
  - name: S3R-Ship
    type: router
    operational_logic: ship_router_logic
    features:
      - name: mS3L
        type: radio_controller
        driver: lora_gw_trx
        parameter_overrides: ship_lora_overrides
      - name: mS3Sat
        type: radio_reporter
        driver: beidou_uplink_driver
        parameter_overrides: ship_beidou_overrides
      - name: mS3Sat
        type: sensor
        driver: beidou_driver
        parameter_overrides: ship_beidou_overrides
  - name: S3R-Truck
    type: router
    features:
      - name: mS3L
        type: radio_controller
        driver: lora_gw_trx
        parameter_overrides: truck_lora_overrides
      - name: mS34G
        type: radio_reporter
        driver: huawei_gsm_32x
        parameter_overrides: truck_gsm_overrides
    operational_logic: truck_router_logic
```

used to communicate with proximity nodes attached to the same container. While an energy harvesting circuit is used to harvest passive zone IDs as the container crosses between zones. Finally, a custom LoRa® MAC protocol (MAC on Time (MoT)) is used to communicate with the routers. The modules connected to the S^3N are the S^3 Guardian Module ($_mS^3G$), the S^3 Zoner Module ($_mS^3Z$), the $_mS^3N$ and the S^3 LoRa® Module ($_mS^3L$).

(b) **S^3 Proximity (S^3P)** is the node used for proximity detection and is attached to each side of the container. The node uses short-range Frequency Shift Keying (FSK) modulated signals to transmit and receive S^3N IDs. Additionally, it uses ZigBee to communicate with the S^3N and relay the ID it read. Thus the S^3N can localize itself in relation to the detected node direction. The S^3 Proximity (S^3P) does not have any modules connected.

(c) **S^3 All-Sense (S^3AS)** is the node attached inside the container. The node uses ZigBee to communicate with the S^3N and relay the data it reads. Modules connected to this node are the S^3 Sensing Module ($_mS^3S$) and the S^3 Tamper Module ($_mS^3T$).

(d) **S^3 Zoner (S^3Z)** is the type of node that creates an electromagnetic curtain to allow the S^3Ns to identify the zone in which it is entering/leaving. The node continuously transmits a directed electromagnetic field that contains the Zone-ID. Once an S^3N passes the curtain, it harvests the electromagnetic energy and triggers the circuitry to read the ID of the zone.

4. **Modules** are additional peripheral boards to be attached to any node to extend its capabilities. All modules are implemented as peripheral boards to be connected to one of the nodes. Thus, no abstraction is needed in the blueprints.

Algorithm 6.2 The nodes abstraction for Clairvoyant

```yaml
- name: S3N
  type: node
  features:
  operational_logic: s3n_logic
    - name: mS3S
      type: sensor
    - name: mS3T
      type: sensor
    - name: mS3G
      type: sensor
    - name: mS3L
      type: radio_reporter
      driver: lora_node_trx
      parameter_overrides: s3n_lora_overrides
    - name: zigbee_trx
      type: radio_controller
      parameter_overrides: xbee_proximity_overrides
    - name: zone_rx
      type: radio_controller
      parameter_overrides: zoning_overrides
- name: S3P
  type: node
  operational_logic: proximity_node_logic
  features:
    - name: zigbee_trx
      type: radio_reporter
      parameter_overrides: xbee_proximity_overrides
    - name: sr-fsk-trx
      type: radio_controller
      parameter_overrides: fsk_proximity_overrides
- name: S3AS
  type: node
  operational_logic: allsense_node_logic
- name: S3Z
  type: node
  operational_logic: zoner_node_logic
  features:
    - name: zone_tx
      type: radio_reporter
      parameter_overrides: zone_overrides
```

Furthermore, all the modules are designed to be TPnP compatible. We note that there was no need to create a ZigBee module since the controller used on the S^3N had a built-in 2.4 GHz radio.

(a) **S^3 Guardian Module ($_mS^3G$)** encrypts and stores the system and application logs for an extra layer of security. This log is hardware encrypted and can only be accessed using the $_mS^3G$ module. The stored logs can be

read by connecting the $_m$S^{3}G to any Personal Computer (PC) using a USB cable.

(b) **S^3 Zoner Module ($_m$S^{3}Z)** is a module to be attached to any S^{3}N, allowing it to communicate with a S^3 Zoner (S^{3}Z) unit and identify key zones it enters/leaves.

(c) **S^3 Sensing Module ($_m$S^{3}S)** is a generic sensing module that carries a different type of sensing element on board. This module could be described as a carrier board for any sensor/peripheral connected to an S^{3}N.

(d) **S^3 LoRa® Module ($_m$S^{3}L)** is a LoRa® radio module to provide LoRa® radio communication capabilities to the node it's connected to.

(e) **S^3 Ethernet Module ($_m$S^{3}E)** is an ethernet module provides internet capabilities to the node it is connected to.

(f) **S^3 WiFi Module ($_m$S^{3}W)** is a WiFi module to provide internet capabilities to the node it's connected to.

(g) **S^3 Sattelite Module ($_m$S^{3}Sat)** is a GPS/Beidou satellite module to provide global positioning and up-link capabilities to the node it's connected to.

(h) **S^3 4G Module ($_m$S^{3}4G)** is a Fourth Generation Wireless (4G)Global System for Mobile communications (GSM) module to provide internet capabilities to the node it's connected to.

(i) **S^3 NFC Module ($_m$S^{3}N)** is an NFC reader module to be connected to the main S^{3}N connected to the container. It allows a connected node to read an employee's ID card.

(j) **S^3 Tamper Module ($_mS^3T$)** is the tamper detection module. The module uses a magnetic switch to detect if the container door is open or closed.

6.2.2 Network Architecture Abstraction

The second level of abstraction involves describing the network architecture in our project. Clairvoyant consists of four different network architectures, as outlined below. While algorithm 6.3 illustrates the network section in the blueprint.

1. **Container** network is the type of architecture used in the context of a single container. The architecture consists of a single S^3N, a single S^3AS, and multiple S^3P nodes communicating together using ZigBee. The S^3P nodes communicate detected IDs with the S^3N in order to 3D-localize the container. In this specific network, we note that the S^3N acts as a router, while the S^3AS and S^3Ps act as nodes. For this network, each S^3N would require four S^3Ps to be able to localize itself in four directions (top, bottom, left, right). The S^3AS can host any sensing element or module, and its responsibility is to collect all information and report it to the S^3N.

2. **Dock** network is the one used to describe how containers communicate with router nodes on the dock. Additionally, since the dock is subdivided into different zones, the network consists of multiple S^3Zs. Note that the number of S^3Ns is set to -1, entailing an unknown number of this type of hardware in the network.

3. **Ship** network is identical to the dock network architecture with the difference in the number of hardware used for each type.

4. **Truck** network represents a truck transporting a container from point A to point

B. In this network, a single container (with a container sub-network) usually communicates with a single S^3R.

Algorithm 6.3 The network abstraction section in Clairvoyant's blueprint

```yaml
network_templates:
  - name: dock
    routers:
      - router: S3R-Dock
        count: 4
    nodes:
      - node: S3N
        count: -1
      - node: S3Z
        count: 6
  - name: ship
    routers:
      - router: S3R-Ship
        count: 6
    nodes:
      - node: S3N
        count: -1
      - node: S3Z
        count: 5
  - name: truck
    routers:
      - router: S3R-Truck
        count: 1
    nodes:
      - node: S3N
        count: 1
  - name: container
    routers:
      - router: S3N
        count: 1
    nodes:
      - node: S3P
        count: 4
      - node: S3AS
        count: 1
```

6.2.3 Deployment Abstraction

The third level of abstraction describes the different deployments in the system. Each deployment consists of a friendly name, a unique ID, coordinates and a reference to one or more network architectures. Algorithm 6.4 contains a snippet of the deployments used in the project.

Algorithm 6.4 The deployment abstraction in Clairvoyant's blueprint

```yaml
deployment_locations:
  - id: '1'
    name: Shanghai Dock
    coordinates: '1123, 41233'
    network_template: dock
  - id: '2'
    name: Ningpo Dock
    coordinates: '1123, 41233'
    network_template: dock
  - id: '3'
    name: The Expedition (Ship)
    coordinates: '1123, 41233'
    network_template: ship
```

6.3 Develop Phase

Once we were able to create the different abstractions, the next phase in PILoT 's process is implementing the logic. Clairvoyant consists of multiple hardware and software development initiatives. However, for this thesis, our focus is mainly on software implementations. We give minor focus to the hardware except when in the context of WhiteBus.

In practice, once PILoT is adopted by the industry, all peripheral drivers would not be implemented by us. Instead, each sensor/peripheral would have a driver on its onboard memory module. However, we developed all the peripheral drivers from scratch due to the current state of PILoT. Nevertheless, the time we spent developing the applications and drivers remained faster and more efficient than the traditional development approach.

During development, we were able to parallelize the work on each task across three different geographical regions. Furthermore, due to the versatility and modularity of PILoT, many parts of the system were developed once and deployed to different hardware architectures. For instance, the LoRa® driver for the ${}_m$S^{3}L was developed once and used on the S^{3}Ns (running on a cc2650 chip) and the router nodes (running on raspberry pi's).

Since WhiteBus and the WBM were mere concepts at the time of developing this project, and towards satisfying requirement 3 and supporting TPnP, we implemented a simplified version of WhiteBus using Ethernet ports. Even though we did not use a WBM, we achieved full TPnP. By embedding the peripheral phantoms on the modules and unifying the use of the SPI bus for communication, all modules and nodes fully supported PILoT 's TPnP. Figure 6.3 illustrates a schematic of an ethernet port. As

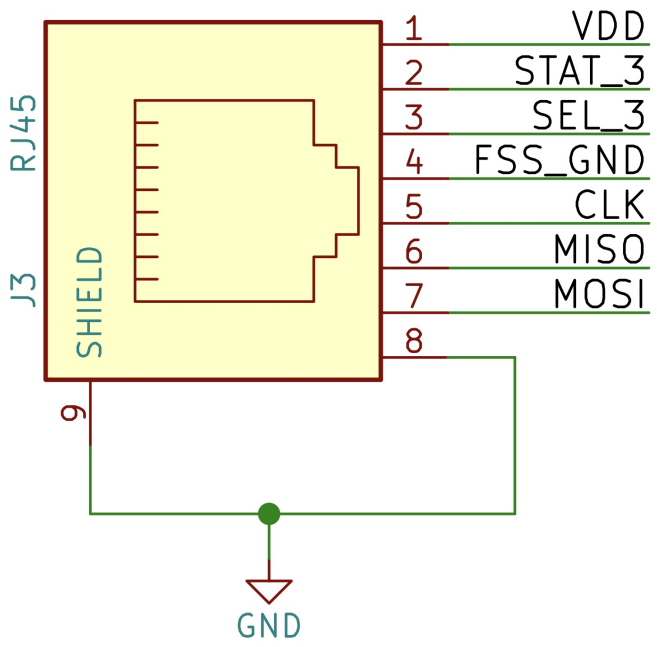


Figure 6.3: A schematic of a simplified WhiteBus connector for the third port on a S^3N

indicated in the figure, four pins are used for SPI, and the **STAT_3** pin represents the WhiteBus LifeLine. The S^3N, S^3AS and the S^3R had four WhiteBus ports.

In the following subsections, we discuss the implementation of a selection of the tasks outlined earlier. The choice of tasks to discuss is based on complexity and the attempt to illustrate a wide range of use-cases and benefits of PILoT.

6.3.1 Cross-Continent Virtual Development

Clairvoyant's software and hardware development were performed across three regions: North America, the Middle East, and Asia. Traditionally, this cross-continental team would need to use a high level of communication and product engineering to reach an efficient cadence of collaboration. However, with PILoT, we were able to simplify

our project management and centralize our virtualized hardware to work in complete autonomy and synchronicity.

Using HIVE to create the IoT containers allowed the teams to develop and test the software long before the hardware materialized. All three teams used a Raspberry Pi 4 as the testing embedded device. We were able to virtualize the S^3N and S^3AS on the Pi and test all the modules weeks before the hardware arrived. Since all the modules used the same port (Ethernet WhiteBus), we designed a similar port for the Raspberry Pi. Thus, once a team finalized a module design and committed it to the repository, the other teams were able to test and integrate the module within their pipeline.

Development in North America

The team in Canada was the epicentre of development and coordinated the project management with the other teams. In addition, the team's core responsibilities were software development and hardware development of the ${}_mS^3S$ and ${}_mS^3T$ modules. The final stage assembly, testing, and packaging were also performed in Canada.

The main development machine was running a MacOSx on an Intel I9 processor. The first testing stage was done on this machine before being deployed on the Raspberry Pi for validation testing.

Development in The Middle East

The team in Jordan was responsible for the hardware design of the S^3N, S^3AS, and S^3P. Once this team was done with the design and implementation of the S^3N for instance, they were able to connect it to the Raspberry Pi and run the Pi as one of the

modules (virtualized) that they did not have physical access to. Thus, they tested and validated the different nodes' correct operations before proceeding to full production and shipping them to Canada.

Development in Asia

The team's primary responsibility in China was Quality Assurance (QA) and validation. Without access to any hardware, they were able to deploy all code on the Raspberry Pi and run the validation tests against it.

Some modules, such as $_m$S^{3}L rely on wireless radio communication, to which the virtual module would receive input parameters that simulate a wireless channel and run the tests against it.

6.3.2 Custom LoRa MAC Protocol

Due to the low-power and long-range properties of the LoRa® modulation, we decided to use it for our node-to-router communications. Using Long Range Wide Area Network (LoRaWAN®) however, was unsuitable for this application.

MoT [53] was designed to mitigate the shortcomings of LoRaWAN®. Instead of ALOHA-based random access, MoT is based on TDMA across multiple channels. The protocol defines connection, uplink, and downlink phases in each frame.

Implementing the protocol in PILoT was the most challenging part of the develop phase. Primarily due to the choice between implementing it as part of the RTOS or in the application layer using HIVE. Embedding it as part of the OS has the benefit of superior performance. In contrast, the HIVE route has the benefits of multi-platform support, faster development, and the absence of the need to have physical access to the

hardware for testing. With the traditional route, we would have needed to implement the protocol twice, once for the S^3R (Raspberry Pi) and once for the S^3N (CC2650). To reduce the development effort and accelerate delivery, we decided to use HIVE to implement the protocol in the application layer.

There were three primary choices when implementing the protocol with HIVE, either we implement it as part of the QUEEN behaviour, implement it as custom BEEs, or implement it as a peripheral driver. To remain consistent and follow the SoCo principles, we have decided to use custom BEEs. Implementing as part of the QUEEN behaviour would not allow the reuse of the protocol on different devices since the QUEEN behaviour would be different. Implementing the protocol as a driver would allow the protocol behaviour to be contained within the radio modules. However, since this is a custom protocol, the manufacturer would not typically implement the driver with the custom protocol in place.

6.3.3 Development of the S3R

All router types in Clairvoyant do not require hot-plugging different peripherals. Therefore, the hardware interfaces (ports/pins) can be hard-coded when implementing the core application. However, since we have three types of routers, we decided to avoid hard-coding our interfaces and use TPnP to minimize our code-base.

Hardware

The router hardware consists of a Raspberry Pi 4 with a PiJuice module attached to provide power if the main power source is compromised. Instead of designing and implementing the S^3 Ethernet Module ($_mS^3E$), S^3 WiFi Module ($_mS^3W$), and S^3 4G

Module ($_m$S^{3}4G), we decided to use the onboard ethernet port and WiFi capabilities of the Raspberry Pi in addition to a SIM7600X 4G Hat. Since we are already creating the $_m$S^{3}L for the other nodes, we did not need to use a commercially available LoRa® hat. However, since the router would communicate with multiple nodes on different channels, the $_m$S^{3}L housing the SX1262 would not work. The router needs an SX1301 LoRa® module. To this end, we decided to use the Raspberry Pi LoRa® gateway hat. Finally, instead of implementing the S^3 Sattelite Module ($_m$S^{3}Sat), we used the Raspberry Pi MAX-7Q GNSS hat. Since all the modules are commercially available as Raspberry Pi hats, there was no need to create the modules from scratch. However, we implemented all the phantoms for the hats and stored them on the Raspberry Pi. Therefore, the core application can use the peripherals as if they were PILoT TPnP modules.

Software

Regardless of the router type, the algorithm's core loop is the same. Handle a new connection request, read a report from a node, send remote upgrade (if needed), and relay data back to the cloud.

6.3.4 Development of the S3N & S3AS

Both nodes use the same hardware; however, the connected modules are what decide the type of node. Traditionally, achieving this would require two software implementations and slightly different underlying hardware. PILoT 's TPnP with WhiteBus and IoT containerization with HIVE enabled us to reduce the hardware and software requirements significantly.

Hardware

The initial iteration of the S^3N and S^3AS hardware used a CC2650 controller based on the ARM® Cortex M3 architecture. The choice of this controller was based on the fact that it included an embedded ZigBee radio chip and a sensor controller peripheral system. The sensor controller system is TI 's effort to simplify peripheral development. The node boards also host four ethernet ports that act as the WhiteBus ports. Figure 6.3 illustrated a schematic of the port, while Fig. 6.4 illustrates the full schematic of a S^3N/S^3AS node.

Software

All IoT nodes share a similar core loop, wake up, read sensors, decide if there is a need to report, read radio data (if needed), then go back to sleep.

Our implementation for the S^3N node followed the same algorithm. However, since the S^3N does not use house sensors directly (except for the ${}_mS^3N$), it communicates with the S^3Ps and S^3AS to gather the information needed. In this context, we consider the data received from the S^3Ps and S^3AS as sensory data to be reported back to an S^3R.

In addition to collecting the data to send and reporting them, the node also needs to decide if action needs to be taken based on the collected data. If the ${}_mS^3T$ of the S^3AS indicates that the container door has been open, the S^3N checks if a scan of a known employee ID has been completed within the few seconds before opening the door. If a scan has not been completed successfully, the S^3N triggers the alarm state, which involves a loud audible alarm and an emergency report to the nearest S^3R.

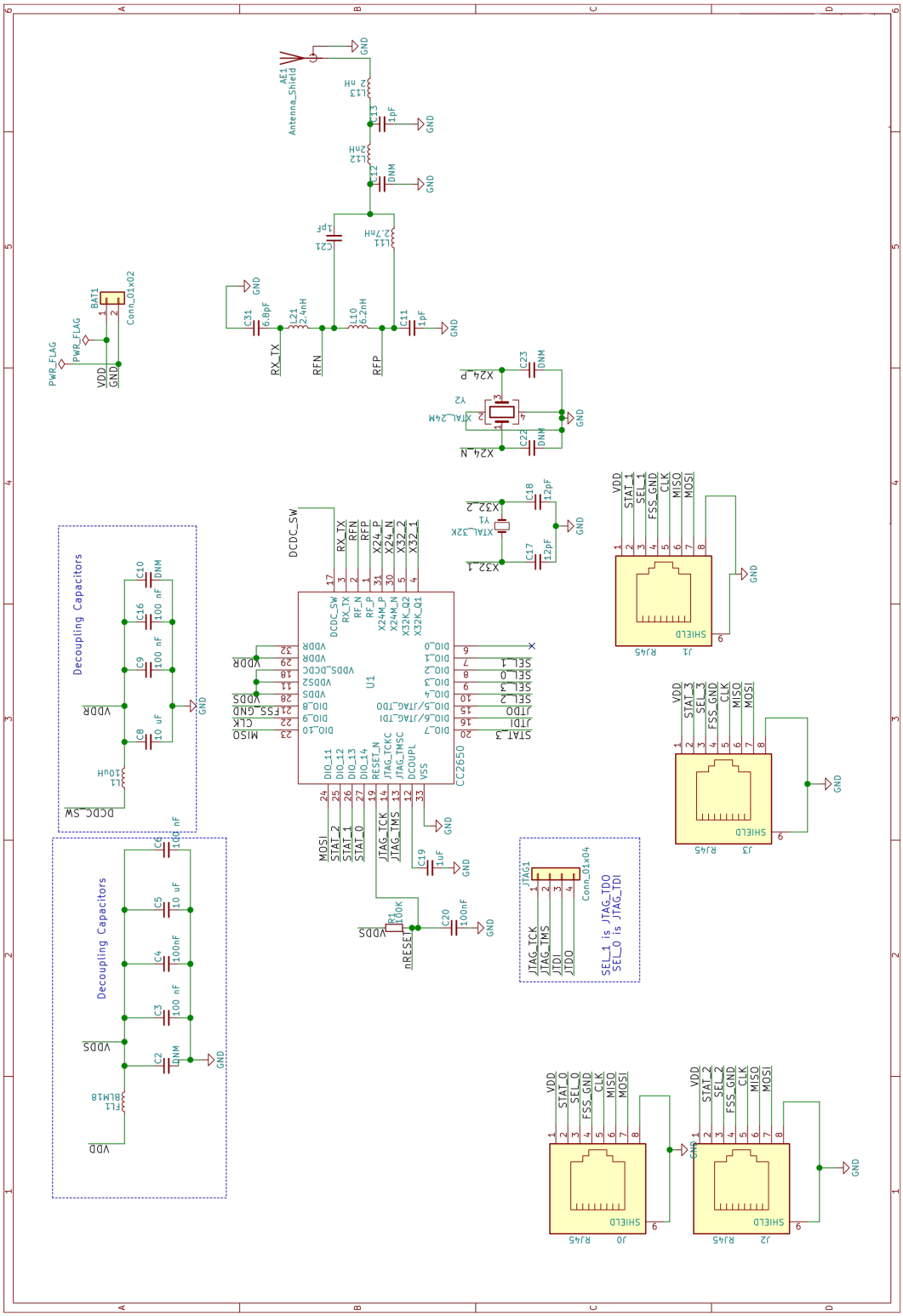


Figure 6.4: The full schematic of the S^{3}N/S^{3}AS nodes

The WhiteBus protocol was implemented on the S^3N and S^3AS. When implementing HEAL for the different nodes, we set the LifeLine as an interrupt. Therefore, the nodes do not have to poll the LifeLine. Furthermore, if a change is detected (module disconnected/connected), the node can re-read the module phantom from its memory. Thus, reducing energy consumption. The implementation of pin-mapping was handled on HEAL and mapped each of the four ports to the HIVE HEAL pins.

6.3.5 Development of the S3P

The node has a CC1101 chip that provides short-range wireless communication. However, the central controller was a CC2650 similar to the chip used on the S^3N. The CC1101 is used to detect the presence of a nearby container by communicating with its S^3P. Then report this information to the main S^3N node. We note that one possible approach was to reuse the same hardware as the S^3N and develop a module for the proximity sensor. However, creating a separate hardware board for the S^3P was more efficient due to physical footprint and cost constraints.

6.3.6 Hardware Development of the Modules

To support TPnP on the modules, we integrated an SPI 1 Mega-bit (Mb) Flash memory chip [79] on all modules. Furthermore, we developed a programming module specifically to write the peripheral phantoms on the memory chips of the modules.

$_m\mathbf{S^3T}$

The tamper module consists of two separate enclosures, the first encapsulating the logic board with the WhiteBus port and the second housing a dual-sided magnet. The

operation concept of the tamper detection sensor is similar to the traditional magnetic door chime; however, tailored for an industrial environment. The logic board uses an ATTINY1617 as the central controller and a reed switch circuit to detect the presence of the magnet in the other enclosure.

$_m$S^{3}N

The NFC module employed an MSP430G2553 chip as the main controller, with a TRF7970 as the NFC radio chip. Since the MSP430G2553 did not support SPI by default, we used a SC16IS740 to convert from UART to SPI for the WhiteBus port communication. The schematic for the $_m$S^{3}N is illustrated in 6.5.

The module also employed two LEDs. The red LED would flash when the scanned card is denied access, while the green would flash when allowed access. In order to decide on authorization, the Database (DB) is read from an Secure Digital (SD) card reader unit installed on the module. This card allows different stages of the shipping pipeline to switch the list of authorized personnel. To avoid this SD card being switched by a malicious user, a user with admin access must first scan their card twice in a row, which would trigger the DB change mode. If this mode was not entered before the SD card is changed, the module will not attempt to use the new DB. Finally, to avoid a malicious user removing the SD card to prevent everyone from accessing the module, a list of admin users is hard-coded and provided on a separate SD card with a security signature. Furthermore, once connected to an S^{3}N or S^{3}AS, an admin user can remotely override any setting, and it would persist on the attached flash storage.

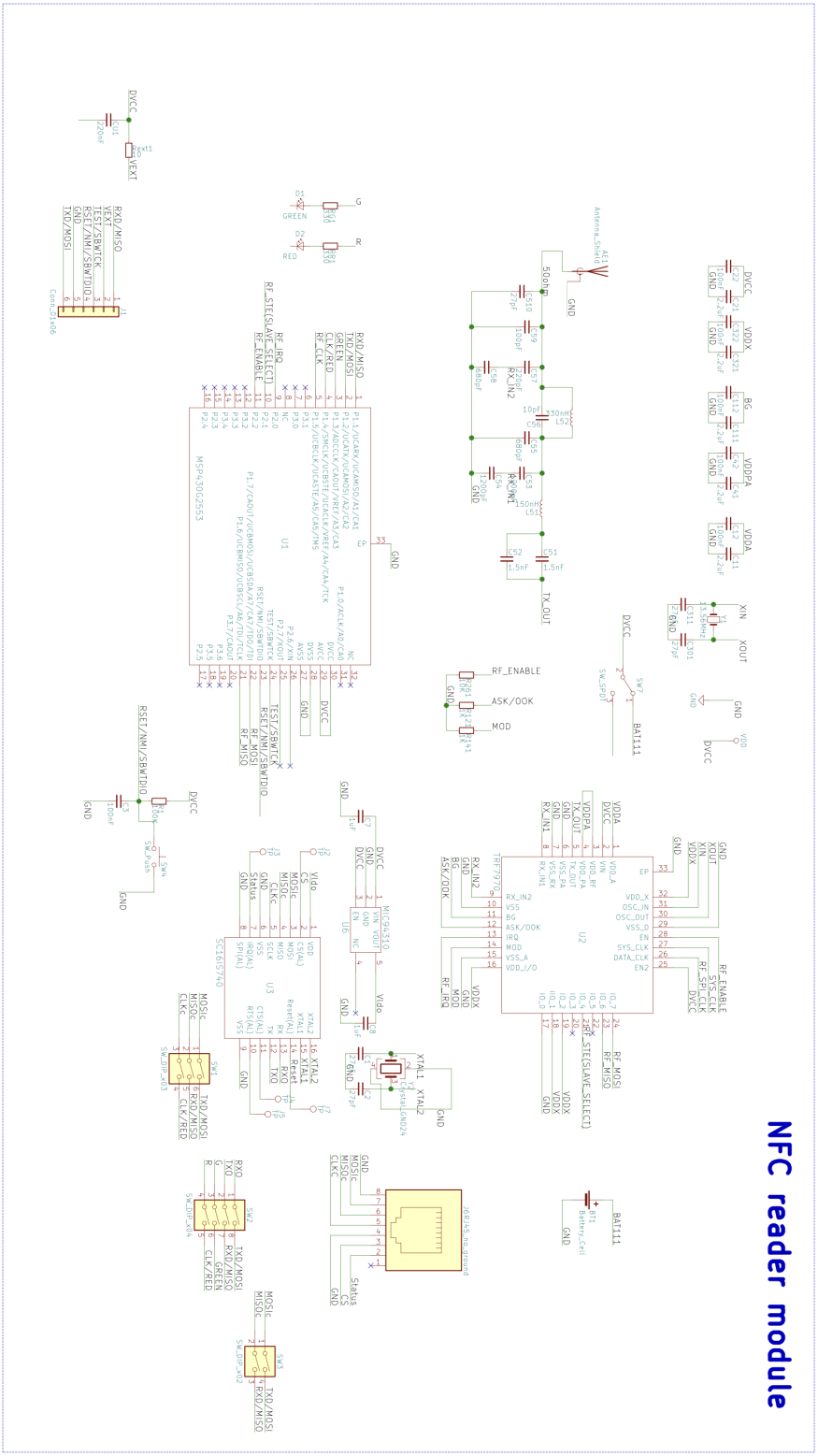


Figure 6.5: A part of the schematic of the $_m$S^{3}N module

$_m$S^{3}S

We have created a single $_m$S^{3}S for prototyping purposes that housed a BMA253 accelerometer and a SI7006A20IM1 humidity and temperature sensor; controlled by an ATTINY1617.

6.3.7 Development of Module Drivers

Despite the responsibility of the driver implementation being proposed to shift to the manufacturer, we implemented all the drivers required for this project. We discuss the implementation of radio drivers in this subsection, specifically, the development of the Sx1262 LoRa® driver. The LoRa® module was straightforward; since no logic was needed to run on the module, the logic board only contained the SX1262 chip and its circuitry.

We note that it is common practice for a manufacturer to choose to implement the driver in one of the multiple ways. The most straightforward to implement (*raw-interaction*) is not the most user-friendly. However, it gives the user the most flexibility. While the most user-friendly (*functional-interaction*) is not as straightforward to implement and might not be as flexible for some users. The *raw-interaction* method involves the manufacturer exposing the bare minimum to the user (e.g. direct register manipulation), essentially translating the data sheet to a library. In contrast, the *functional-interaction* method exposes a more extensive set of (more straightforward) interfaces (e.g. transmit and receive in a radio module) that encapsulate a more complex underlying functionality.

Specific to PILoT, the choice of implementation for the driver would be how to implement the **Initialize**, **Read**, **Write** and **Processing** methods. For instance, the

manufacturer may choose to implement the **Read** and **Write** as direct manipulation of the internal registers of the radio module, or implement them as the **Receive** and **Transmit** radio functionality respectively. Regardless of the direction the manufacturer decides to proceed, PILoT supports *custom* methods in addition to the internal constructs, the manufacturer may choose to implement additional functionality using these *custom* methods.

In our implementation we opted for using the **Read** and **Write** for radio receiving and transmitting respectively. The **Processing** for our custom MAC protocol. Moreover, the module's internal registers have multiple **Custom** methods for configuring, reading, and writing. To use the driver, the user would override some register values. The core application would call the *Initialize* routine, prepare some data to send, call the *Processing* routine, then either call the *Write* or *Read* according to the use-case.

6.4 Deploy Phase

With different independent working pieces, we started creating our E2E CI/CD pipeline to make sure our system worked as expected. We first discuss how we performed testing virtually for some of our nodes, router, and modules. Then, we outline how we set up our CI/CD pipeline and finally discuss our trip to deploy the PoC in one of the ports in China.

6.4.1 Virtualized Testing

As mentioned earlier, we used a Raspberry Pi 4 to virtualize the modules and test our HIVE logic. The Pi would act as a node or router and communicate with the modules (physical or virtual). The first step was to port HIVE to the Raspberry Pi 4, specifically, HEAL.

mS3L

We first started by creating the LoRa® module since this was the primary communication method between nodes. We implemented the phantom as discussed in Section 6.3.7. However, since the hardware was not yet ready, we tested the module by creating a mock on the Pi that mimicked the operation of the LoRa® module. Furthermore, since the module was essentially a bridge between different nodes when running a multi-node test, we created fixtures for the communication portion of the tests. To create the fixtures and mocks, we created a middleware layer between HEAL and the HIVE VSoC that injects test data and mocks the operation of certain interfaces.

S3N

Using the Pi to virtualize the S^3N involved executing the S^3N's QUEEN and custom BEEs. The Pi used the nodes abstraction blueprint created in the Design phase and illustrated in algorithm 6.2 to define the logic needed to operate.

However, testing the operation of the S^3N involved testing the behaviour with different modules connected and disconnected. For instance, we tested when no modules were connected and when the virtual ${}_mS^3S$ was connected. Since we expect the S^3N logic to collect sensory data from the modules and send them to the S^3R over a radio module, we used the virtual ${}_mS^3L$ module, ran the test cases and monitored the printed output.

S3R

Since the routers are to be built on the Raspberry Pi, there was no need to virtualize the hardware. However, since the Raspberry Pi is not an efficient development environment, we used the local machines to develop and deploy the code to the Pi.

Testing, however, required virtualized peripherals until the hats were delivered. Since we were planning on creating HIVE drivers for the hats, we were able to start the development of the peripheral phantoms and use the mocking method outlined earlier for testing.

mS3T

Since the Tamper detection module involved a binary response (1/0), the test cases for this module were straightforward. For the system tests, we mocked the input using the middleware discussed earlier to inject our tests to the peripheral phantom code

and run the S^3N container on the Raspberry Pi. Unit tests were run as a QUEEN that executes each peripheral BEE in the phantom, and the middleware validates the expected output.

mS3S

Similar to the $_mS^3T$ module, the sensing module phantom used mocks for the system tests and a specialized QUEEN for unit tests. Figure 6.6 illustrates how an example test was performed. The testing middleware illustrates a set of tests that control the execution of the containers and asserts the expected results.

6.4.2 CI/CD Pipeline

Once we had all the tests in place, automating the CI pipeline involved replicating the Raspberry Pi set up on a Docker container. We used the Balena Raspberry Pi docker image [15] to containerize the Pi; however, we could have also used a standard Linux-based container and port HEAL to this distro. Despite using the Balena image, we had to do some minor changes to HEAL on the docker image, mainly to inject hooks to notify the GitLab executor.

The CD pipeline was more challenging than the CI one, mainly because testing on physical devices was required before we deployed to production. To this end, we set up a staging environment that involved one of each S^3N, S^3R, S^3AS, and a S^3P. All nodes were physically connected to our central server and remotely programmed by the pipeline. In addition to the nodes, we also attached one of each module to the respective nodes. Thus, creating a full twin of the production environment. The primary goal of the staging environment was to replicate (as close as possible) the

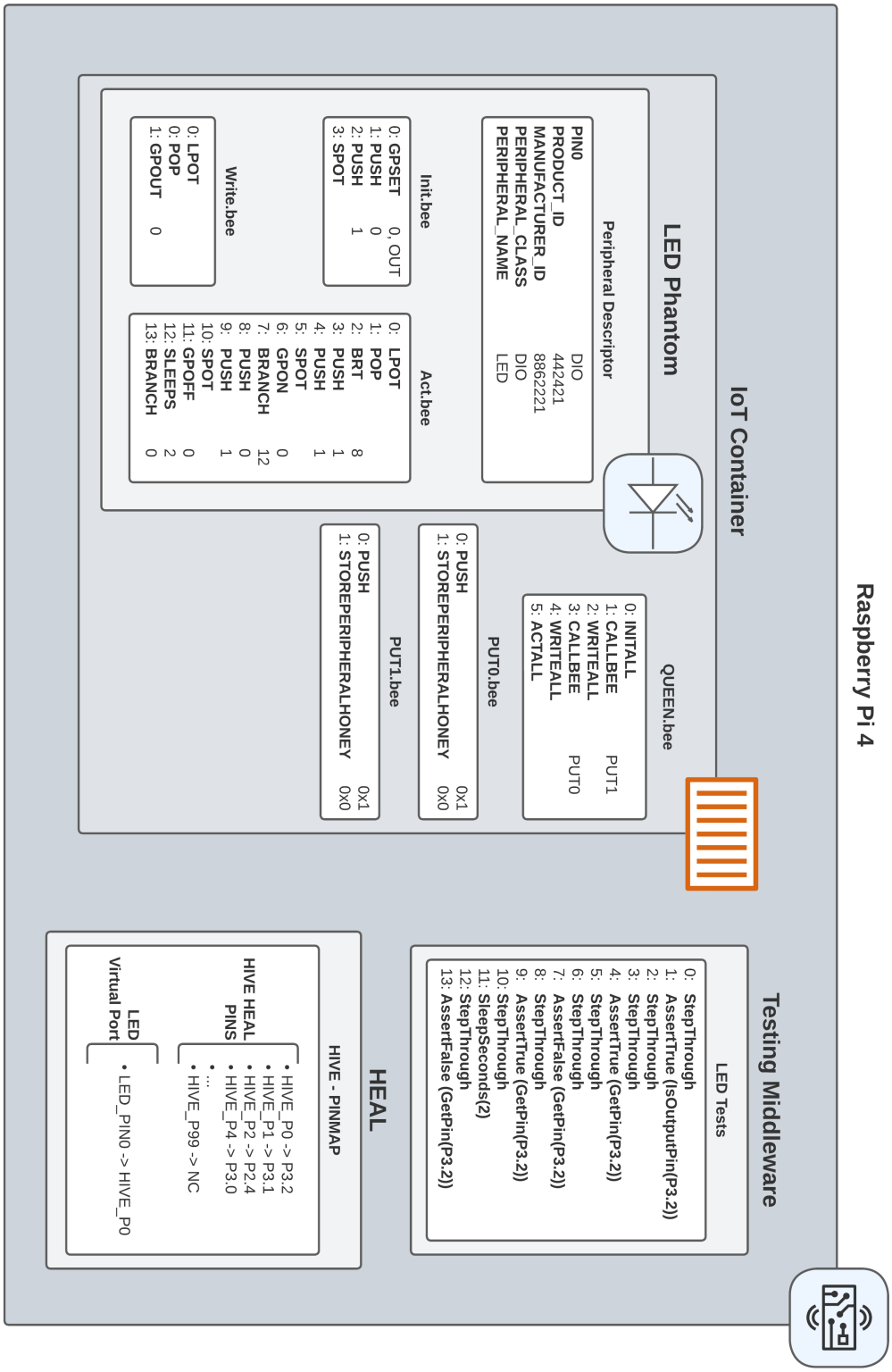


Figure 6.6: An example test setup running on the Raspberry Pi 4

system in production and prevent any potential issues from getting released.

Once the tests on the staging environment passed, we employed a manual step to gate the deployment to the production devices. However, since the system was not yet deployed, we disabled this stage until the system was deployed. This stage would iterate over all deployment abstractions and target each deployed network's S^3R. A payload with all upgraded IoT containers would be sent to the routers, offloading them to the nodes. However, an upgrade to one of the phantoms would get stored on the respective node, overriding the phantom on the module's memory.

6.4.3 On-Site Deployment

With the PoC tested and ready and all the code deployed on the devices, we set off to a container port in Ningpo, China. The requirements for the deployment were to install the system on two containers, with two S^3Rs deployed in different areas of the port. Since this was a PoC deployment for investors, the goal was to deploy a working E2E system. Therefore, we used two S^3P on each container, thus localizing from the sides only.

The node enclosures were 3D-printed and fixed with neodymium magnets. The S^3N had the ${}_mS^3L$ and ${}_mS^3N$ modules attached. While the S^3AS had the ${}_mS^3T$ and ${}_mS^3S$ modules attached. Figure 6.8(a) and Fig. 6.8(b) illustrate how the S^3N and S^3AS were attached to the container. While Fig. 6.7 illustrates the internals of the S^3N before attachment to the container.

Figure 6.7: A picture of the S3N before sealing the enclosure

Proximity Testing

A mobile application was developed to consume the data collected by the S^3R from the nodes. The application communicates with the S^3.IO backend and displays a GUI reflecting the data collected. Figure 6.9 illustrates a screenshot of the application during the proximity test. During this test, a second container with a S^3P was attached and was moved towards the container under test. We then check the mobile application to observe the digital twin replicating the new container approaching the first. Afterwards, we move the containers away from each other and observe the digital twins again replicating the action. After performing the test ten times, the average

(a) S3N on the door of a shipping container

(b) S3AS on the inside of the door of a shipping container

Figure 6.8: Deployed location of both S3N and S3AS

Figure 6.9: A picture of the mobile application during the proximity test

latency of the action replication was recorded to be ∼3 seconds.

Personnel Access Testing

Before a container door can be open, an authorized employee must first scan their name tag. To achieve this, we use a combination of both the ${}_m$S^{3}T and ${}_m$S^{3}N modules. As illustrated in Fig. 6.8(b), the ${}_m$S^{3}T is attached to the inside of the container door. Once the door is open, the module sends a signal to the S^{3}AS, which relays it to the S^{3}N. The S^{3}N would check if an authorized employee scanned their card first. To test this behaviour, we first open the container door without scanning the access card and observe the digital twin showing an alert.

Additionally, the S^{3}N sounds an audible alarm accompanied by a visual red LED flash. To test the expected behaviour, we first scan the access card, wait for the

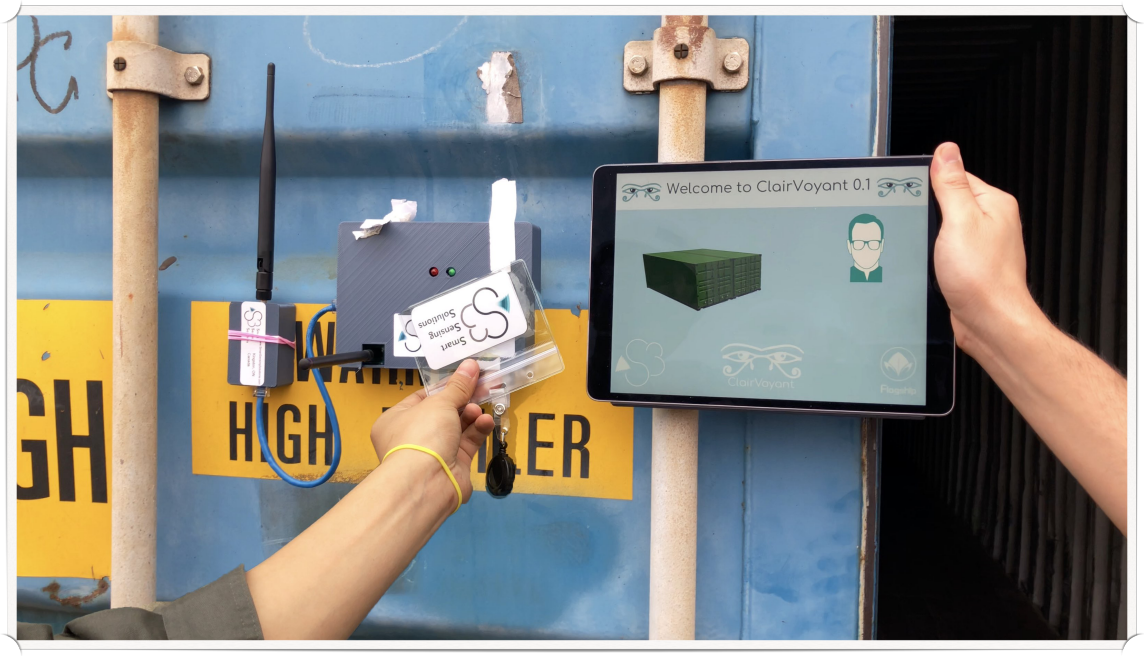


Figure 6.10: A picture of the mobile application during the personnel access test

green LED to light up and observe the employee's image appear on the digital twin. Afterwards, we open the container door and ensure the S^3N does not sound the alarm. Figure 6.10 displays an image taken during the expected behaviour test.

Wireless Range Testing

Finally, we ran several wireless range tests in different scenarios to ensure stable connectivity. All range tests involved keeping one item stationary while the other was mobile, and we would record the Received Signal Strength Indicator (RSSI) and distance along the way.

Communication between the S^3Ns and the S^3R is achieved over LoRa®. To test the wireless link's robustness, we performed four tests involving a different location for the S^3N being attached to the container. We would move the S^3R a few meters for

each test and measure the RSSI. The S^3R was running a QUEEN that periodically transmits a sequenced packet through the connected $_mS^3L$ module. The S^3R was running a QUEEN that measures the packet-loss ratio and RSSI. With the help of a HopeRF Software-Defined Radio (SDR), we were able to validate the presence of the LoRa® signal and rule out any possible malfunction with the S^3R. The most optimal location was found on the container's outside door; however, the top and sides of the container had comparable communication performance. Communicating from within the container was successful; however, it would not be efficient if long-distance communication was needed. For instance, to reduce the number of S^3Rs.

6.5 Summary

The operate phase is when a system is fully deployed in production and being used/maintained. However, since our collaboration with the company was for a PoC, our efforts ended at the deploy phase. Our journey concluded with a fully operational deployed PoC in less than six months since conception.

The PILoT approach allowed us to realize our vision of the product more than 3x faster than the traditional approach. Furthermore, we were able to avoid a waterfall development methodology and instead parallelize our efforts across three continents. HIVE 's containerization engine was paramount for the time-saving witnessed. We were able to abstract, develop, and test our applications long before the hardware was fabricated. Furthermore, we were able to set up a CI pipeline with automated testing using HIVE containers. Finally, once the hardware was delivered, transitioning from the virtual versions to the physical ones was an immediate zero-effort process.

Integrating HIVE and a simplified version of WhiteBus, we were able to adopt PILoT 's vision of TPnP in Clairvoyant. All our modules were designed with an ethernet port that connects to WhiteBus' lines. The bus used SPI for communication and a LifeLine to detect connection/disconnection. Furthermore, all modules included a memory chip to house the peripheral phantom. The S^3N and S^3AS were able to use these phantoms to understand which peripheral is connected and how to communicate, configure, and use the peripheral. Finally, the TPnP feature allowed us to use a single hardware board for both the S^3N and S^3AS by changing the type of connected modules.

This chapter presented a snippet of our endeavour to create a full IoT solution from scratch using PILoT. Clairvoyant was the embodiment of our research presented

across the different chapters of this thesis. We believe that PILoT has the potential to democratize the IoT development industry.

Chapter 7

Summary and Future Directions

Today, an IoT developer is presented with a fractured market of diverse peripheral manufacturers, incompatible off-the-shelf platforms, and a wide range of IoT cloud providers. To say the choices are overwhelming is an understatement. Furthermore, once the choices are locked-in, a significant amount of effort is needed to port some software to the selected platform. In addition, another significant amount of effort is needed to allow the chosen set of equipment to become cross-compatible and just work. If we are to realize the full I4.0 vision in this decade, a catalyst for IoT development is necessary.

7.1 Summary

The primary goal of our thesis is to democratize and accelerate IoT platform development. To this end, we started by investigating the current IoT development paradigm in Chapter 2. Additionally, we investigated the challenges of adopting modern SDLCs in the IoT realm. Our findings indicated that adoption in the lower layers of IoT development could be attributed to the hardware access requirement. After exploring the concept of sandboxing and containerization, we outlined the challenges related to its lack of adoption in IoT. Furthermore, we discussed related efforts and highlighted their shortcoming. Finally, we looked into solutions such as USB and their role in the popularity of the PnP concept.

Inspired by the shortcomings of the efforts described in Chapter 2, we re-envisioned the IoT platform development process by proposing PILoT in Chapter 3. In addition to enabling modern SDLCs in the IoT realm, PILoT proposed three core paradigms that if enabled, would democratize IoT development. We discussed each paradigm and outlined the requirements needed to realize each. PILoT's architecture was presented by the capabilities achieved in each of the four phases of development, *Design & Plan*, *Develop*, *Deploy*, and *Operate*. In the first stage, PILoT enabled IaC for IoT development; we outlined the abstraction blueprint capabilities and provided examples of the different sections. For the *Develop* stage, we discussed using the IoT containers and outlined our designs for how it would be implemented. Then, we discussed how PILoT enables the adoption of modern SDLCs through describing a CI/CD pipeline in the *Deploy* stage. In the *Operate* stage, we presented the different post-deployment operation capabilities enabled by PILoT. Finally, we imagined and recounted a world with PILoT as the mainstream development methodology.

The paradigm of IoT containers renders IoT development, platform agnostic. In Chapter 4 we proposed the architecture and discussed the implementation along with the PoC results of HIVE, the IoT containerization engine proposed by PILoT. We compared HIVE's performance with bare-metal and popular VM-based languages. The evaluation used several applications to target CPU, memory, and disk usage. We remark that HIVE was 5x faster with an 8x smaller memory footprint than the other languages. Additionally, HIVE had comparable metrics with bare-metal. Finally, HIVE enables part of the TPnP paradigm. To realize the full vision, we proposed the shift of responsibility of driver development to the manufacturer and away from the developer. The manufacturer would create a manifest along with a driver and embed them on a memory chip on the peripheral unit. We dub this combination of files a peripheral phantom.

Combined with HIVE's IoT containers, peripheral phantoms stored on the peripheral are the first steps to enable TPnP. In Chapter 5, we proposed WhiteBus, an interface multiplexing and abstraction module. Inspired by the IEEE 1454 standard, the module provides the simplicity of USB with the flexibility of bare-metal interfacing. We discussed WhiteBus' architecture and operational concept, then assessed our designs qualitatively against other efforts.

In Chapter 6 we described our journey to create an IIoT container tracking solution from scratch using PILoT. We outlined the steps we took in each of the development phase and illustrated some of the designs and scripts implemented. We indicated that the endeavour that traditionally takes more than twelve months took us less than six months using PILoT. We outlined how we were able to adopt a version of WhiteBus that allowed us to reuse our hardware and reduce our hardware requirements. Finally,

we presented our CI pipeline and discussed our deployment visit along with the testing and results.

7.2 Future Directions

Our objective from day one is to accelerate and democratize IoT platform development. Although our thesis proposed efforts to enable this vision, our journey with PILoT is still beginning. In addition to publishing our research and igniting the open-source community around PILoT, we plan on engaging the IEEE community to investigate the possibility of expanding on the 1454 standard. All the details and the cloud service of PILoT will be available on iot-pilot.com and iotpilot.io, while HIVE is available on thingshive.io.

Even though HIVE has demonstrated superior performance over alternative efforts, the undertaking is still in its infancy. Towards making HIVE more appealing to developers, we first plan on investigating the different scripting languages that would be suitable to be compiled into HIVE's byte-code. We believe that compiling a language such as Python to run on our VM would significantly accelerate the adoption of HIVE among IoT developers. Once a compiler is created, we then plan on running several developer tests and surveys to further understand their pain points with HIVE.

Some challenges remain unaddressed in HIVE's core, and we plan on investigating and addressing them with the help of the open-source community. HIVE's current proposed design does not have multiple levels of software interrupt. Thus, it is not as flexible to create very complex applications. Our first effort on the core is to propose, implement, and test an updated version of the core that enables multi-level customizable software interrupts. While ensuring that the added capability does not affect the execution time and footprint of HIVE, we shall also attempt to improve the run-time. Since the proposed run-time PoC uses a switch-dispatch pattern, it does not fully utilize branch prediction. We plan to investigate and implement a different

pattern that fully utilizes modern CPU features, including branch prediction and multiple cores.

Our proposed design on WhiteBus still carries some challenges. For instance, our prototype was implemented using ethernet and without the WBM presented in Chapter 5. Therefore, our first effort will be to investigate the implementation options. Between the proposed design and a design using FPGA, the investigation will involve cost and power efficiency, with the latter carrying a more considerable weight on the implementation decision. Furthermore, our proposed design lacks customizable interrupt features. We plan on investigating how to enable a peripheral to customize and describe how interrupt lines would work when interfacing with the IoT node.

Combining HIVE and WhiteBus to achieve the TPnP paradigm proposed by PILoT is not free of challenges. The first improvement planned is creating a GUI tool that allows developers to map the physical pins to the HIVE virtual pins. This tool would then generate the pin-map files needed to be added to HEAL when porting to a new platform. Furthermore, our proposal has not fully investigated security in peripheral phantoms. Per the current designs, a malicious phantom can impersonate a peripheral through a man-in-the-middle attack. Therefore, we plan on studying the effect of adding a hash signed (using a private/public key pair) by the manufacturer to the phantom descriptor. This hash would allow an IoT node to verify the authenticity of the driver before executing it.

Bibliography

[1] Shalabh Agarwal, Sayantan Majumdar, Abhisek Maiti, and Asoke Nath. Security and privacy issues of internet of things: Challenges and threats. *International Journal of Advanced Technology in Engineering and Science*, 3(11):89–98, 2015.

[2] Mohd Shahdi Ahmad, Nur Emyra Musa, Rathidevi Nadarajah, Rosilah Hassan, and Nor Effendy Othman. Comparison between android and ios operating system in terms of security. In *Information Technology in Asia (CITA), 2013 8th International Conference on*, pages 1–4. IEEE, 2013.

[3] Farshad Ahmadighohandizi and Kari Systä. Application development and deployment for iot devices. In *European Conference on Service-Oriented and Cloud Computing*, pages 74–85. Springer, 2016.

[4] Sarah A Al-Qaseemi, Hajer A Almulhim, Maria F Almulhim, and Saqib Rasool Chaudhry. Iot architecture challenges and issues: Lack of standardization. In *2016 Future technologies conference (FTC)*, pages 731–738. IEEE, 2016.

[5] M Zulfiker Ali, Jelena Mišić, and Vojislav B Mišić. Performance evaluation of heterogeneous iot nodes with differentiated qos in ieee 802.11 ah raw mechanism. *IEEE Transactions on Vehicular Technology*, 68(4):3905–3918, 2019.

[6] Jérémie Allard, V Chinta, S Gundala, and Golden G Richard. Jini meets upnp: an architecture for jini/upnp interoperability. In *2003 Symposium on Applications and the Internet, 2003. Proceedings.*, pages 268–275. IEEE, 2003.

[7] Amazon. Overview of amazon web services. https://d1.awsstatic.com/whitepapers/aws-overview.pdf, accessed on 02.28.2021.

[8] Charles Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, 2015.

[9] Don Anderson and Dave Dzatko. *Universal Serial Bus System Architecture*. MindShare INC, second edition, 2001.

[10] Simon D Duque Anton, Daniel Fraunholz, Daniel Krohmer, Daniel Reti, Hans D Schotten, Franklin Selgert, Marcell Marosvölgyi, Morten Larsen, Krishna Sudhakar, Tobias Koch, et al. Creating it from scratch: A practical approach for enhancing the security of iot-systems in a devops-enabled software development environment. In *International Conference on Computer Safety, Reliability, and Security*, pages 266–281. Springer, 2020.

[11] Arduino. About arduino, 2018. https://www.arduino.cc/en/about, accessed on 02.20.2022.

[12] E. Baccelli, J. Doerr, S. Kikuchi, F. A. Padilla, K. Schleiser, and I. Thomas. Scripting over-the-air: Towards containers on low-end devices in the internet of things. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 504–507, 2018.

[13] Emmanuel Baccelli, Joerg Doerr, Ons Jallouli, Shinji Kikuchi, Andreas Morgenstern, Francisco Acosta Padilla, Kaspar Schleiser, and Ian Thomas. Reprogramming low-end iot devices from the cloud. In *2018 3rd Cloudification of the Internet of Things (CIoT)*, pages 1–6. IEEE, 2018.

[14] Jeongju Bae, Chorwon Kim, and JongWon Kim. Automated deployment of smartx iot-cloud services based on continuous integration. In *International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1076–1081. IEEE, 2016.

[15] Balenalib. balenalib/raspberry-pi. https://hub.docker.com/r/balenalib/raspberry-pi, accessed on 05.28.2022.

[16] Fatemeh Banaie, Jelena Misic, Vojislav B Misic, Mohammad Hossein Yaghmaee Moghaddam, and Seyed Amin Hosseini Seno. Performance analysis of multithreaded iot gateway. *IEEE Internet of Things Journal*, 6(2):3143–3155, 2018.

[17] Thomas W Barr and Scott Rixner. Owl embedded python environment: microcontroller development for the modern world. *; login:: the magazine of USENIX & SAGE*, 38(1):48–51, 2013.

[18] Richard Barry et al. Freertos. *Internet, Oct*, 2008.

[19] Charles Bell. Introducing micropython. In *MicroPython for the Internet of Things*, pages 27–57. Springer, 2017.

[20] Jacob Beningo. Building reusable device drivers for microcontrollers, 2013.

[21] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[22] Jiang Bian, Remzi Seker, Srini Ramaswamy, and Nuri Yilmazer. Container communities: Anti-tampering wireless sensor network for global cargo security. In *2009 17th Mediterranean Conference on Control and Automation*, pages 464–468. IEEE, 2009.

[23] Eric W Biederman and Linux Networx. Multiple instances of the global linux namespaces. In *Proceedings of the Linux Symposium*, volume 1, pages 101–112. Citeseer, 2006.

[24] BOSCH. *Intelligent 9-axis absolute orientation sensor*, 6 2016. Rev. 1.4.

[25] Malte Brettel, Niklas Friederichsen, Michael Keller, and Marius Rosenberg. How virtualization, decentralization and network building change the manufacturing landscape: An industry 4.0 perspective. *International journal of mechanical, industrial science and engineering*, 8(1):37–44, 2014.

[26] Nicola Bui, Michele Rossi, Cristiano Tapparello, and Michele Zorzi. Reprogramming over the air and sensor island management through synapse++. In *2009 6th IEEE Annual Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks Workshops*, pages 1–3. IEEE, 2009.

[27] Antonio Celesti, Davide Mulfari, Maria Fazio, Massimo Villari, and Antonio Puliafito. Exploring container virtualization in iot clouds. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–6. IEEE, 2016.

[28] Qingping Chi, Hairong Yan, Chuan Zhang, Zhibo Pang, and Li Da Xu. A reconfigurable smart sensor interface for industrial wsn in iot environment. *IEEE transactions on industrial informatics*, 10(2):1417–1425, 2014.

[29] Komkrit Chooruang and Kraison Meekul. Design of an iot energy monitoring system. In *2018 16th International Conference on ICT and Knowledge Engineering (ICT&KE)*, pages 1–4. IEEE, 2018.

[30] Malcolm Cohorn, Susie Cummings, Ankur Laroia, and Eskander Yavar. Covid-19 is accelerating the rise of the digital economy, 2021. "https://www.bdo.com/insights/business-financial-advisory/strategy,-technology-transformation/covid-19-is-accelerating-the-rise-of-the-digital-e", accessed on 09.19.2021.

[31] European Comission. Industry 5.0, 2022. https://research-and-innovation.ec.europa.eu/research-area/industry/industry-50_en, accessed on 09.29.2022.

[32] Soumya Kanti Datta and Christian Bonnet. An edge computing architecture integrating virtual iot devices. In *2017 IEEE 6th Global Conference on Consumer Electronics (GCCE)*, pages 1–3. IEEE, 2017.

[33] Scott Davidson and Bruce D Shriver. An overview of firmware engineering. *Computer*, 11(5):21–33, 1978.

[34] Claudio M de Farias, Italo C Brito, Luci Pirmez, Flavia C Delicato, Paulo F Pires, Taniro C Rodrigues, Igor L dos Santos, Luiz FRC Carmo, and Thais

Batista. Comfit: A development environment for the internet of things. *Future Generation Computer Systems*, 75:128–144, 2017.

[35] David Déharbe, Stephenson Galvao, and Anamaria Martins Moreira. Formalizing freertos: First steps. In *Brazilian Symposium on Formal Methods*, pages 101–117. Springer, 2009.

[36] William Dunham. The bernoullis and the harmonic series. *The College Mathematics Journal*, 18(1):18–23, 1987.

[37] Prabal Dutta, Jay Taneja, Jaein Jeong, Xiaofan Jiang, and David Culler. A building block approach to sensornet systems. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 267–280, 2008.

[38] Eclipse. Open source for iot, 2022. https://iot.eclipse.org, accessed on 02.20.2022.

[39] Holly Ellyatt. Supply chain chaos is already hitting global growth. and it's about to get worse. *CNBC*, Oct 2021. https://www.cnbc.com/2021/10/18/supply-chain-chaos-is-hitting-global-growth-and-could-get-worse.html, accessed on 12.04.2022.

[40] eLua. The elua project, 2021. "http://www.eluaproject.net/", accessed on 02.28.2021.

[41] eLua. The elua project - overview, 2021. "http://www.eluaproject.net/home/overview", accessed on 02.28.2021.

[42] Espressif Systems. *ESP32 Series Datasheet*, January 2021. "Version 3.5", http://esp32.net.

[43] Sheikh Ferdoush and Xinrong Li. Wireless sensor network system design using raspberry pi and arduino for environmental monitoring applications. *Procedia Computer Science*, 34:103–110, 2014.

[44] Open JS Foundation. Low-code programming for event-driven applications, 2018. https://nodered.org, accessed on 02.20.2022.

[45] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. Ultra lightweight javascript engine for internet of things. In *Companion Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 19–20, 2015.

[46] G. B. Ghantous and A. Gill. The devops reference architecture evaluation : A design science research case study. In *IEEE International Conference on Smart Internet of Things (SmartIoT)*, pages 295–299, 2020.

[47] Grafana. Operational dashboards for your data here, there, or anywhere, 2014. https://grafana.com, accessed on 02.20.2022.

[48] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[49] Hongzhi Guo, Jiajia Liu, Jie Zhang, Wen Sun, and Nei Kato. Mobile-edge computation offloading for ultradense iot networks. *IEEE Internet of Things Journal*, 5(6):4977–4988, 2018.

[50] Ligia Georgeta Guşeilă, Dragoş-Vasile Bratu, and Sorin-Aurel Moraru. Continuous testing in the development of iot applications. In *International Conference on Sensing and Instrumentation in IoT Era (ISSI)*, pages 1–6. IEEE, 2019.

[51] Galal Hassan. Things hive - a better way to create your next iot project, 2021. https://thingshive.io, accessed on 12.15.2021.

[52] Galal Hassan, Mohamed ElMaradny, Mohamed A. Ibrahim, Abdulmonem M. Rashwan, and Hossam S. Hassanein. Energy efficiency analysis of centralized-synchronous lora-based mac protocols. In *14th International Wireless Communications Mobile Computing Conference (IWCMC)*, pages 999–1004, 2018.

[53] Galal Hassan and Hossam S. Hassanein. Mot: A deterministic latency mac protocol for mission-critical iot applications. In *14th International Wireless Communications Mobile Computing Conference (IWCMC)*, pages 588–593, 2018.

[54] Galal Hassan, Abdelmoneim Rashwan, and Hossam Hassanein. Sandboxer: A self-contained sensor architecture for sandboxing the industrial internet of things. In *IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6, 2019.

[55] Matthew Henschke, Xinzhou Wei, and Xiaowen Zhang. Data visualization for wireless sensor networks using thingsboard. In *2020 29th Wireless and Optical Communications Conference (WOCC)*, pages 1–6. IEEE, 2020.

[56] Mario Hermann, Tobias Pentek, and Boris Otto. Design principles for industrie 4.0 scenarios. In *2016 49th Hawaii international conference on system sciences (HICSS)*, pages 3928–3937. IEEE, 2016.

[57] Dixys Hernández-Rojas, Tiago Fernández-Caramés, Paula Fraga-Lamas, and Carlos Escudero. A plug-and-play human-centered virtual teds architecture for the web of things. *Sensors*, 18(7):2052, 2018.

[58] HONEYWELL. *Heavy Duty Pressure Transducers - Sensing and Internet of Things*, 2018. Issue B.

[59] HOPERF. *Low Power Long Range Transceiver Module*, 2018. Version 2.0, https://www.hoperf.com/modules/lora/RFM95.html.

[60] Kaa IoT. Elevate your iot experience with business-ready iot dashboards, 2017. https://www.kaaiot.com/, accessed on 02.20.2022.

[61] Nasser Jazdi. Cyber physical systems in the context of industry 4.0. In *2014 IEEE international conference on automation, quality and testing, robotics*, pages 1–4. IEEE, 2014.

[62] Muhamad Fitra Kacamarga, Bens Pardamean, and Hari Wijaya. Lightweight virtualization in cloud computing for research. In *International Conference on Soft Computing, Intelligence Systems, and Information Technology*, pages 439–445. Springer, 2015.

[63] Shawn Knight. Chip delivery times reach record levels of 20 weeks as shortage drags on. *TechSpot*, Aug 2021. https://www.techspot.com/news/90756-chip-delivery-times-reach-record-levels-shortage-drags.html, accessed on 02.13.2022.

[64] Ahmad El Kouche, Hossam S. Hassanein, and Khaled Obaia. Wsn platform plug-and-play (pnp) customization. In *IEEE Ninth International Conference*

on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), pages 1–6, 2014.

[65] Tamás Kovácsházy, Gábor Wacha, Tamás Dabóczi, Csanád Erdős, and Attila Szarvas. System architecture for internet of things with the extensive use of embedded virtualization. In *2013 IEEE 4th international conference on cognitive infocommunications (CogInfoCom)*, pages 549–554. IEEE, 2013.

[66] Ram Kumar, Eddie Kohler, and Mani Srivastava. Harbor: software-based memory protection for sensor nodes. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 340–349. ACM, 2007.

[67] Er-Rajy Latifa et al. Android: Deep look into dalvik vm. In *2015 5th World Congress on Information and Communication Technologies (WICT)*, pages 35–40. IEEE, 2015.

[68] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[69] Philip Alexander Levis, David Edward Gay, and David E Culler. *Bridging the gap: Programming sensor networks with application specific virtual machines*. Computer Science Division, University of California, 2004.

[70] Z. Li, X. Peng, L. Chao, and Z. Xu. Everylite: A lightweight scripting language for micro tasks in iot systems. In *IEEE/ACM Symposium on Edge Computing (SEC)*, pages 381–386, 2018.

[71] Gaohua Liao, Quanguo Lu, and Weizhong Zhang. Design of reusable software for usb host driver in embedded system. In *Computing, Control and Industrial Engineering (CCIE), 2010 International Conference on*, volume 1, pages 312–315. IEEE, 2010.

[72] Torben R Licht. The ieee 1451.4 proposed standard. *IEEE Instrumentation & Measurement Magazine*, 4(1):12–18, 2001.

[73] Liqing Liu, Zheng Chang, Xijuan Guo, Shiwen Mao, and Tapani Ristaniemi. Multiobjective optimization for computation offloading in fog computing. *IEEE Internet of Things Journal*, 5(1):283–294, 2017.

[74] Lora-net. Lorawan end-device stack implementation and example projects, 2022. https://github.com/Lora-net/LoRaMac-node, accessed on 02.13.2022.

[75] Lora-net. Sx126x driver, 2022. https://github.com/Lora-net/sx126x_driver, accessed on 02.13.2022.

[76] Mike Loukides. *What is DevOps?* " O'Reilly Media, Inc.", 2012.

[77] Lucy Ellen Lwakatare, Teemu Karvonen, Tanja Sauvola, Pasi Kuvaja, Helena Holmström Olsson, Jan Bosch, and Markku Oivo. Towards devops in the embedded systems domain: Why is it so hard? In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5437–5446. IEEE, 2016.

[78] M. A. López-Peña, J. Díaz, J. E. Pérez, and H. Humanes. Devops for iot systems: Fast and continuous monitoring feedback of system availability. *IEEE Internet of Things Journal*, 7(10):10695–10707, 2020.

[79] Macronix International. *MX25V1006F - 1M-BIT [x 1/x 2] CMOS SERIAL NOR FLASH*, 04 2017. Rev. 1, https://www.macronix.com/Lists/Datasheet/Attachments/8673/MX25V1006F,%202.5V,%201Mb,%20v1.0.pdf, accessed on 02.25.2022.

[80] Mirjana Maksimovic, Vladimir Vujovic, Nikola Davidović, Vladimir Milosevic, and Branko Perisic. Raspberry pi as internet of things hardware: Performances and constraints. In *Proceedings of 1st International Conference on Electrical, Electronic and Computing Engineering IcETRAN, ISBN 978-86-80509-70-9*, 06 2014.

[81] Yukihiro "Matz" Matsumoto. mruby github repository, 2021. https://github.com/mruby/mruby.

[82] Yukihiro "Matz" Matsumoto. Sleep module for mruby, 2021. "https://github.com/matsumotory/mruby-sleep".

[83] Paul Merolla, John Arthur, and J Wittig. The usb revolution. *The Neuromorphic Engineer*, 2(2):10–11, 2005.

[84] MICROCHIP. *Thermal Sensor with SPI Interface*, 2012.

[85] MICROCHIP. *6-Pin, 8-Bit Flash Microcontrollers*, 2014.

[86] MicroPython. The micropython project, 2021. https://github.com/micropython/micropython.

[87] Microsoft. Microsoft azure iot reference architecture. http://aka.ms/iotrefarchitecture, accessed on 02.28.2021.

[88] Konstantin Mikhaylov, Tomi Pitkaäho, and Jouni Tervonen. Plug–and–play mechanism for plain transducers with wired digital interfaces attached to wireless sensor network nodes. *International Journal of Sensor Networks*, 14(1):50–63, 2013.

[89] Charlie Miller, Dion Blazakis, Dino DaiZovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philip Weinmann. *iOS Hacker's Handbook*. John Wiley & Sons, 2012.

[90] André Miranda and João Pimentel. On the use of package managers by the c++ open-source community. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 1483–1491, New York, NY, USA, 2018. Association for Computing Machinery.

[91] James D Mooney. Bringing portability to the software process. *Dept. of Statistics and Comp. Sci., West Virginia Univ., Morgantown WV*, 1997.

[92] John Moore, Gerd Kortuem, Andrew Smith, Niaz Chowdhury, Jose Cavero, and Daniel Gooch. Devops for the urban iot. In *Proceedings of the Second International Conference on IoT in Urban Space*, pages 78–81. ACM, 2016.

[93] Anand Nayyar and Vikram Puri. A review of arduino board's, lilypad's amp; arduino shields. In *3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1485–1492, 2016.

[94] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[95] NXP. *Digital temperature sensor and thermal watchdog*, 2 2015. "Rev. 6.1" https://www.nxp.com/docs/en/data-sheet/LM75B.pdf.

[96] Institute of Electrical and Electronics Engineers. IEEE standard for a simple 32-bit backplane bus: Nubus: Includes ANSI/IEEE std 1101-1987. In *IEEE Standard for Mechanical Core Specifications for Microcomputers*. Institute of Electrical and Electronic Engineers, 1988.

[97] Francisco Ortin, Jose Manuel Redondo, and J Baltasar García Perez-Schofield. Efficient virtual machine support of runtime structural reflection. *Science of computer Programming*, 74(10):836–860, 2009.

[98] John K Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[99] Michele Pancani. Architecture for plug-and-play modular technology. *2015 NCUR*, 2015.

[100] Particle. Reprogram the world, 2022. https://www.particle.io, accessed on 02.20.2022.

[101] David S Peterson, Matt Bishop, and Raju Pandey. A flexible containment mechanism for executing untrusted code. In *Usenix Security Symposium*, pages 207–225, 2002.

[102] Stefan Pitzek and Wilfried Elmenreich. Plug-and-play: Bridging the semantic gap between application and transducers. In *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, volume 1, pages 8–pp. IEEE, 2005.

[103] Sebastian Plamauer and Martin Langer. Evaluation of micropython as application layer programming language on cubesats. In *ARCS; 30th International Conference on Architecture of Computing Systems*, pages 1–9. VDE, 2017.

[104] A. RADOVICI, C. RUSU, and R. ŞERBAN. A survey of iot security threats and solutions. In *2018 17th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–5, Sep. 2018.

[105] Ammar Rayes and Samer Salam. Fog computing. In *Internet of Things From Hype to Reality*, pages 155–180. Springer, 2019.

[106] Vasja Roblek, Maja Meško, and Alojz Krapež. A complex view of industry 4.0. *Sage Open*, 6(2):2158244016653987, 2016.

[107] Tiago Koketsu Rodrigues, Jiajia Liu, and Nei Kato. Application of cybertwin for offloading in mobile multi-access edge computing for 6g networks. *IEEE Internet of Things Journal*, 2021.

[108] Tiago Koketsu Rodrigues, Katsuya Suto, and Nei Kato. Edge cloud server deployment with transmission power control through machine learning for 6g internet of things. *IEEE Transactions on Emerging Topics in Computing*, 2019.

[109] ROHM Semiconductor. *Digital 16bit Serial Output Type Color Sensor IC*, 5 2016. Rev. 2.

[110] Michael Rüßmann, Markus Lorenz, Philipp Gerbert, Manuela Waldner, Jan Justus, Pascal Engel, and Michael Harnisch. Industry 4.0: The future of productivity and growth in manufacturing industries. *Boston Consulting Group*, 9(1):54–89, 2015.

[111] Stephen R Schach. *Object-oriented and classical software engineering*, volume 6. McGraw-Hill New York, 2007.

[112] Bruce Schneier. The internet of things is wildly insecure and often unpatchable., 2014. https://www.schneier.com/essays/archives/2014/01/the_internet_of_thin.html, accessed on 06.13.2016.

[113] SEMTECH. *LoRa Modem*.

[114] Semtech. *SX1261/2 Long Range, Low Power, sub-GHz RF Transceiver*, 12 2021. Rev. 2.1, https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/2R000000Un7F/yT.fKdAr9ZAo3cJLc4F2cBdUsMftpT2vsOICP7NmvMo, accessed on 02.13.2022.

[115] Nasif B. Shafi. *Efficient Over-the-air Remote Reprogramming of Wireless Sensor Networks*. PhD thesis, Queen's University, 2012. https://www.proquest.com/dissertations-theses/efficient-over-air-remote-reprogramming-wireless/docview/1514470831/se-2?accountid=6180, accessed on 02.23.2021.

[116] Nasif Bin Shafi, Kashif Ali, and Hossam S. Hassanein. No-reboot and zero-flash over-the-air programming for wireless sensor networks. In *9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 371–379, 2012.

[117] Omer Shwartz, Yael Mathov, Michael Bohadana, Yuval Elovici, and Yossi Oren. Opening pandora's box: Effective techniques for reverse engineering iot devices. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research*

and Advanced Applications, pages 1–21, Cham, 2018. Springer International Publishing.

[118] Ian Sommerville. Software engineering 9th edition. *ISBN-10*, 137035152:18, 2011.

[119] AIOTI WG03-loT Standardisation. High level architecture (hla). *Technical specification*, 2017.

[120] Manuel Suárez-Albela, Paula Fraga-Lamas, Tiago M Fernández-Caramés, Adriana Dapena, and Miguel González-López. Home automation system based on intelligent transducer enablers. *Sensors*, 16(10):1595, 2016.

[121] Tarek Sultan Al Essa. 5 ways the covid-19 pandemic has changed the supply chain, 2022. Wolrd Economic Forum, "https://www.weforum.org/agenda/2022/01/5-ways-the-covid-19-pandemic-has-changed-the-supply-chain/", accessed on 02.01.2022.

[122] Takayuki Suyama, Yasue Kishino, and Futoshi Naya. Abstracting iot devices using virtual machine for wireless sensor nodes. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 367–368. IEEE, 2014.

[123] Antero Taivalsaari and Tommi Mikkonen. A roadmap to the programmable world: software challenges in the iot era. *IEEE Software*, 34(1):72–80, 2017.

[124] Antero Taivalsaari and Tommi Mikkonen. On the development of iot systems. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 13–19. IEEE, 2018.

[125] Kazuaki Tanaka, Avinash Dev Nagumanthri, and Yukihiro Matsumoto. mruby – rapid software development for embedded systems. In *15th International Conference on Computational Science and Its Applications*, pages 27–32, 2015.

[126] Texas Instruments. *MSP432P401R, MSP432P401M SimpleLink™ Mixed-Signal Microcontrollers datasheet*, 6 2019. "Rev. H", https://www.ti.com/lit/ds/symlink/msp432e401y.pdf.

[127] Texas Instruments. *CC26xx, CC13xx Sensor Controller Studio Version 2.8*, 01 2021. "Version. 2.8", https://www.ti.com/lit/ml/swru439l/swru439l.pdf, accessed on 02.26.2022.

[128] ThingsBoard. Open-source iot platform, 2016. https://thingsboard.io, accessed on 02.20.2022.

[129] R. Tjoa, K. L. Chee, P. K. Sivaprasad, S. V. Rao, and J. G. Lim. Clock drift reduction for relative time slot tdma-based sensor networks. In *2004 IEEE 15th International Symposium on Personal, Indoor and Mobile Radio Communications (IEEE Cat. No.04TH8754)*, volume 2, pages 1042–1047 Vol.2, Sept 2004.

[130] Liam Tung. Global supply chains are struggling. the iot could help. *ZDNet*, Sep 2021. https://www.zdnet.com/article/global-supply-chains-are-struggling-the-iot-could-help, accessed on 12.04.2022.

[131] UniTN-Mechatronics. mruby-raspberry github repository, 2021. https://github.com/UniTN-Mechatronics/mruby-raspberry.

[132] Adrien Van den Bossche, Réjane Dalcé, and Thierry Val. Openwino: An open hardware and software framework for fast-prototyping in the iot. In *2016 23rd International Conference on Telecommunications (ICT)*, pages 1–6. IEEE, 2016.

[133] Shivangi Vashi, Jyotsnamayee Ram, Janit Modi, Saurav Verma, and Chetana Prakash. Internet of things (iot): A vision, architectural elements, and security issues. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*, pages 492–496. IEEE, 2017.

[134] Svein Vetti, Jonas Olsson, and Jeanna Copley. 'sub-1 ghz long-range communication and smartphone connection for iot applications. *Texas Instrum., Dallas, TX, USA, White Paper*, pages 1–9, 2016.

[135] Dorothea Wagner and Roger Wattenhofer. *Algorithms for Sensor and Ad Hoc Networks: Advanced Lectures*. Springer-Verlag, Berlin, Heidelberg, 2007.

[136] A Wallgren. As the iot expands, agile is the way forward. https://techbeacon.com/app-dev-testing/iot-expands-agile-way-forward, accessed on 02.28.2021.

[137] A Wallgren. Rise of the internet of things (iot), 2021. https://www.techradar.com/news/rise-of-the-internet-of-things-iot, accessed on 04.25.2021.

[138] E. Wen and G. Weber. Wasmachine: Bring iot up to speed with a webassembly os. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 1–4, 2020.

[139] Miao Wu, Ting-Jie Lu, Fei-Yang Ling, Jing Sun, and Hui-Ying Du. Research on the architecture of internet of things. In *2010 3rd International Conference*

on Advanced Computer Theory and Engineering (ICACTE), volume 5, pages V5–484. IEEE, 2010.

[140] Qiang Xie, Jinfeng Liu, and Pai H Chou. Tapper: a lightweight scripting engine for highly constrained wireless sensor nodes. In *Proceedings of the 5th international conference on Information processing in sensor networks*, pages 342–349. ACM, 2006.

[141] Fan Yang, Nelson Matthys, Rafael Bachiller, Sam Michiels, Wouter Joosen, and Danny Hughes. μ pnp: plug and play peripherals for the internet of things. In *Proceedings of the tenth European conference on computer systems*, page 25. ACM, 2015.

[142] Muneer Bani Yassein, Mohammed Q Shatnawi, Shadi Aljwarneh, and Razan Al-Hatmi. Internet of things: Survey and open issues of mqtt protocol. In *2017 international conference on engineering & MIS (ICEMIS)*, pages 1–6. Ieee, 2017.

[143] Aexandros Zaharis, Adamantini I Martini, Leonidas Perlepes, Georgios Stamoulis, and Panayotis Kikiras. Live forensics framework for wireless sensor nodes using sandboxing. In *Proceedings of the 6th ACM workshop on QoS and security for wireless and mobile networks*, pages 70–77. ACM, 2010.

[144] K. Zandberg and E. Baccelli. Minimal virtual machines on iot microcontrollers: The case of berkeley packet filters with rbpf. In *9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN)*, pages 1–6, 2020.

[145] Keliang Zhou, Taigang Liu, and Lifeng Zhou. Industry 4.0: Towards future industrial opportunities and challenges. In *12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pages 2147–2152. IEEE, 2015.

Appendix A

Blueprints

A.1 PILoT Blueprint Example

```yaml
company: Syncrude
project: Shovel Monitors
deployment_locations:
  - id: '1'
    name: Shovel 1
    coordinates: '1123, 41233'
    network_template: standard
  - id: '2'
    name: Shovel 2
    coordinates: '1123, 41233'
    network_template: standard
network_templates:
  - name: standard
    routers:
      - router: sink-1
        count: 1
    nodes:
      - node: sensor-node-1
        count: 15
  - name: advanced
    routers:
      - router: sink-1
        count: 1
```

```
24      nodes:
25        - node: sensor-node-unknown-features
26          count: -1
27  hardware_templates:
28    - name: sink-1
29      type: router
30      features:
31        - name: lora_trx
32          type: radio_controller
33          driver: lora_gw_trx
34          parameter_overrides: sink1_lora_overrides
35        - name: gsm_trx
36          type: radio_reporter
37          driver: huawei_gsm_32x
38          parameter_overrides: sink1_gsm_overrides
39      operational_logic: sink1_logic
40    - name: sensor-node-1
41      type: node
42      features:
43        - name: temperature
44          type: sensor
45          driver: temperature_tsys01
46          parameter_overrides: node1_temp_overrides
47        - name: heater
```

```yaml
48          type: actuator
49          driver: heater_custom
50        - name: lora_trx
51          type: radio_reporter
52          driver: lora_node_trx
53          parameter_overrides: node1_lora_overrides
54      operational_logic: node1_logic
55    - name: sensor-node-unknown-features
56      type: node
57      operational_logic: node1_logic
58  policies:
59    - name: flexible_network_policy
60      type: network_policy
61      allow_node_registration: true
62
63    - name: fixed_network_policy
64      type: network_policy
65      allow_node_registration: false
66      known_nodes:
67        - D0E45C4A-56FA-11EC-8683-FE4AFB1BFCCB
68        - D0E45D4E-56FA-11EC-8683-FE4AFB1BFCCB
69        - D0E45D76-56FA-11EC-8683-FE4AFB1BFCCB
70        - D0E45DA8-56FA-11EC-8683-FE4AFB1BFCCB
71        - D0E45DB2-56FA-11EC-8683-FE4AFB1BFCCB
```

```
72
73    - name: flexible_sharing_policy
74      type: sharing_policy
```

Appendix B

Peripheral Examples

B.1 LED Strip

```yaml
peripheral:
  - name: LED_STRIP
    class: DIO
    product_id: 223432
    vendor_id: 553421
    certificate: 352F5B0BCBFA6E1ABEE263927DB42D40
    pins:
      - name: red_led
        type: GPIO_PIN
      - name: blue_led
        type: GPIO_PIN
      - name: yellow_led
        type: GPIO_PIN
workers:
  - type: "Initialization"
    code: |
      SetOutput(red_led)
      SetOutput(blue_led)
      SetOutput(yellow_led)
  - type: "Processing"
    parameters_in:
      - redStatus: OFF
        blueStatus: OFF
```

```
24          yellowStatus: OFF
25          sleepDuration: 1
26      code: |
27        RunForever{
28          redStatus = Inverse(redStatus)
29          blueStatus = Inverse(blueStatus)
30          yellowStatus = Inverse(yellowStatus)
31          PutGPIO(red_led, redStatus)
32          PutGPIO(blue_led, blueStatus)
33          PutGPIO(yellow_led, yellowStatus)
34          SleepForSeconds(sleepDuration)
35        }
36    - type: "Write"
37      parameters_in:
38        - redStatus: OFF
39          blueStatus: OFF
40          yellowStatus: OFF
41      code: |
42        PutGPIO(red_led, redStatus)
43        PutGPIO(blue_led, blueStatus)
44        PutGPIO(yellow_led, yellowStatus)
```

B.2 Temperature Sensor

```yaml
peripheral:
  - name: TEMPERATURE_SENSOR
    class: DIO
    product_id: AD7314ARMZ-REEL7
    vendor_id: Analog Devices Inc.
    certificate: F2BC0A6E2652F1AEC8EB6A78F1918CAD
    pins:
      - name: CE
        type: SPI_CS_PIN
      - name: SCLK
        type: SPI_CLK_PIN
      - name: SDI
        type: SPI_MOSI_PIN
      - name: SDO
        type: SPI_MISO_PIN
workers:
  - type: "Initialization"
    code: |
      SPI_Settings(125000, MSB_FIRST, SPI_MODE1)
  - type: "Read"
    parameters_out:
      - raw_data
    code: |
```

```
24        # read 16 bits
25        raw_data = READ_SPI_REG16(0x00)
26    - type: "Processing"
27      parameters_in:
28        - raw_data: 0
29        - temperature_unit: C
30      parameters_out:
31        - sensor_data
32      code: |
33        # remove rightmost 5 bits from LSB
34        raw_data = raw_data >> 5
35        # remove leftmost 6 bits from MSB
36        raw_data = raw_data & 0x7FF
37
38        sign_bit = raw_data & 0x400
39        sign_bit = sign_bit >> 10
40
41        if (sign_bit == 1)
42        {
43          raw_data = (~raw_data & 0x7FF) + 1
44        }
45
46        sensor_data = raw_data & 0x3FF
47        sensor_data = sensor_data / 4
```

```
48
49        if (sign_bit == 1)
50        {
51          sensor_data = sensor_data * -1
52        }
53
54        if (temperature_unit == "F")
55        {
56          sensor_data = (sensor_data * (9/5)) + 32
57        }
```

Appendix C

MoT

MAC on Time (MoT) is designed to resolve multiple dilemmas related to MAC protocols: namely, reduce energy consumption, support good scalability, ensure fairness, and maximize channel utilization. The strength of MoT relies on overcoming clock drifts while using a channel-access method based on centralized-scheduling protocols.

An MoT network uses a star topology architecture, in which nodes periodically communicate information to a centralized BS, that then relays this information to a back-end server. Between each communication, a node idles for an extended period, the duration of which is determined by the centralized BS and conveyed to each node in the acknowledgment of each report. We expect a large number of nodes to be implemented in a single network; scalability is an important consideration.

In the following subsections, we discuss how MoT maximizes channel utilization while providing deterministic latency.

C.1 MoT Design Details

MoT can be described as a hybrid protocol, one that combines contention-based and TDMA-based protocols. It is a centralized, guaranteed-access, wireless MAC protocol using a BS for coordinating the channel access time-slots in a semi round-robin manner. Hence a star topology network is required. Every report is acknowledged by the BS to confirm that message delivery is reliable. MoT is designed to run on top of a LoRa® Physical Layer (PHY) layer with the following six modem settings: Spreading Factor of 10, Bandwidth of 125 kHz, Coding Rate of 4/5, with an explicit header, Cyclic Redundancy Checking (CRC) enabled, and a preamble of 8 symbols. These settings give us a link budget of 152 dB and receiver sensitivity of -132 dBm.

One of the main features of MoT is the scheduling mechanism used by the BS to utilize the maximum capacity of a single channel, allowing MoT to produce superior throughput. The BS is capable of receiving multiple packets at the same time over different logical and physical channels to ensure that a single node can transmit packets as frequently as possible without violating the duty cycle limitations. A physical channel is one that is defined by a difference in carrier frequency; multiple logical channels would use a single physical channel, but then with different modulation parameters, this is possible through the modulation/demodulation technique used by the PHY layer. The BS divides a single channel into a reporting phase and a connection phase. The reporting phase is subdivided into multiple time-slots with each time-slot divided into multiple sub-slots. Each sub-slot represents a report packet of a fixed size. At the end of a time-slot, the BS acknowledges all the packets received during this time-slot. During the connection phase, the BS individually approves/denies each connection request that it received during the reporting phase.

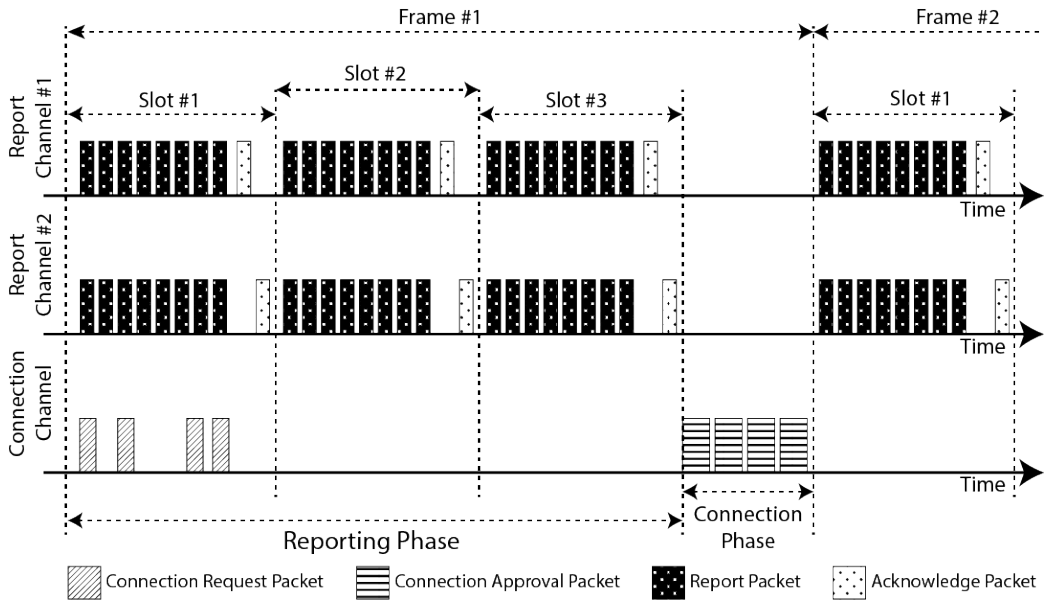


Figure C.1: Basic MoT allocation scheme

MoT allows one node to report only once per frame to safeguard the deterministic latency unique to MoT. However, MoT allows a variable packet size for each node. Consequently, one node can occupy one or more sub-slots based on the size of the payload required to be transmitted. Once a node is connected to the network, it gets assigned a number of sub-slots by the BS; this is initiated by the node transmitting a connection request packet on the connection channel indicating the number of sub-slots required. If the node needs to change the size of the payload at any time, it disconnects and reconnects to the network. During each time-slot, each node will transmit a packet only during its assigned sub-slot(s), rendering the reporting phase free of collisions between packets. The basic scheme is depicted in Fig. C.1.

C.2 Time and Scheduling

One challenge of most TDMA protocols is resynchronizing clocks between the different nodes and the BS due to clock drifts [129]. This drift is caused by the dissimilar hardware used for the different types of nodes as well as the environmental factors in which this node is operating. A case in point, the crystal oscillators of a node would cause a time drift of up to 0.18 seconds every hour [135]. MoT does not require clock synchronization between the different nodes and the BS. Instead, a BS in MoT calculates a time delay for each node between reports. To further protect against time drifts, a tolerance value Tol is applied to different parts of a frame.

When a BS is first initialized, it starts calculating the frame schedule based on the different parameters set by the application; this includes the acknowledge time on air T_{ack}, payload time on air T_{pl}, duration of one time-slot $T_{time-slot}$, and the number of sub-slots per time-slot n_{ss}. These values remain constant over the lifespan of the network. There are some values that are recalculated by the BS during network operation; these values include the current number of slots per frame n_{slots}, duration of the reporting phase T_{rp}, duration of the connection phase T_{cp}, and time until the start of time-slot.

- **The Time On Air**

  This is the time required for one packet of a certain size to be transmitted over the PHY layer. This calculation is based on the PHY layer modulation parameters and described in (C.1-C.4) [113]. The LoRa® PHY layer modulates the PayLoad bytes (PL) into multiple symbols $n_{payload}$ based on the Spreading Factor (SF), BandWidth (BW), Coding Rate (CR), the presence of an explicit Header (H), and whether or not the low Data-rate optimization Enable (DE) is

enabled. The time on air of one symbol T_{sym} can be calculated using the SF and BW; using T_{sym} along with the number of preamble symbols $n_{preamble}$ allows us to calculate the time on air for the preamble $T_{preamble}$, this, in turn, gives us the ability to calculate the time on air for both T_{ack} & T_{pl}.

$$T_{sym} = \frac{2^{SF}}{BW} \tag{C.1}$$

$$T_{preamble} = (n_{preamble} + 4.25) \times T_{sym} \tag{C.2}$$

$$\begin{aligned} n_{payload} &= 8 + max(\\ &ceil((\frac{8PL - 4SF + 28 - 16 - 20H}{4(SF - 2DE)})(CR + 4), 0)) \end{aligned} \tag{C.3}$$

$$T_{oa} = T_{preamble} + (n_{payloadsymbols} \times T_{sym}) \tag{C.4}$$

- **The Duration of One Time-Slot**

  To ensure fair access to the BS on a certain channel, the T_{ack}, including the tolerance value Tol, should be a certain percentage of the total $T_{time-slot}$. This percentage is the Duty Cycle (DC) limitation of the region in which the MoT network is operational. Eq. (C.5) allows us to calculate $T_{time-slot}$ in the presence of a DC.

$$T_{time-slot} = \frac{T_{ack} \times Tol}{DC} \tag{C.5}$$

- **The Number of Time-Slots per Reporting Phase**

  For the BS to ensure the least amount of time between each frame, it utilizes the $n_{channels}$ available before increasing the number of time-slots in each frame. Based on the number of currently connected nodes n_{nodes}, the BS can calculate n_{slots} using (C.6).

$$n_{slots} = \frac{n_{nodes}/n_{ss}}{n_{channels}} \tag{C.6}$$

- **The Duration of the Connection Phase**

  The connection phase is only part of a frame when the BS receives connection requests during the Connection Request Period CR_P. The CR_P is the period of time between the start of the acknowledgement packet of the first slot of this frame and the start of the acknowledgement packet of the first slot of the previous frame, this is illustrated in Fig. C.2. Eq. (C.7) is used to calculated T_{cp} factoring in the number of requests $n_{requests}$ received during CR_P, as well as the time on air of the approval packet T_{app}.

$$T_{cp} = T_{app} \times n_{requests} \tag{C.7}$$

- **The Duration of the Reporting Phase**

  This is dependent on both the n_{slots} and the $T_{time-slot}$, assuming no connection requests were received; as a result, the reporting phase duration consumes the full duration of the frame.

- **The Duration of the Frame**

  This is the total of the T_{rp} & Tcp; it is also defined as the latency of an MoT network as it represents the time until a node can send another packet.

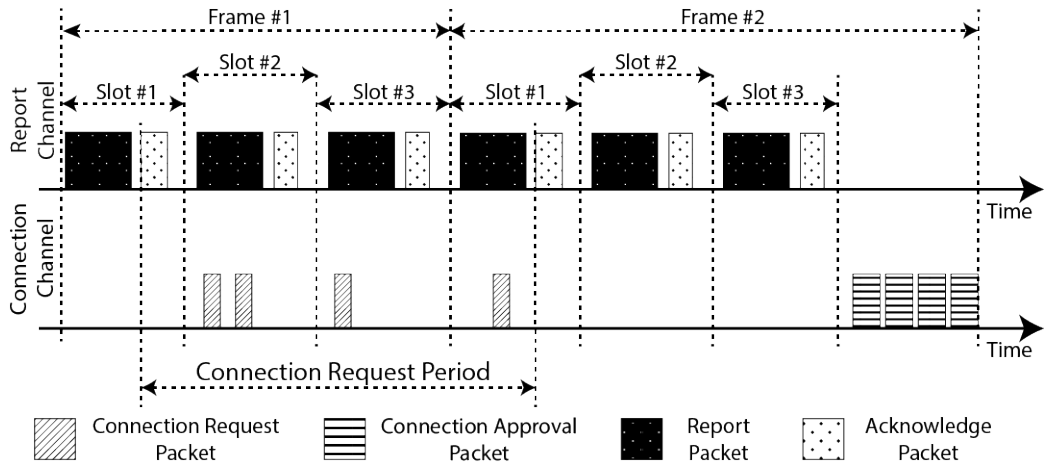


Figure C.2: Connection phase timing calculation

- **Time Until a Time-Slot Restarts**

  The BS calculates the remaining time until a time-slot starts, this happens on two occasions: when a new node joins the network T_α and at the end of each time-slot T_β. These values are then used by each node along with its sub-slot index SS_i to calculate the time at which it can transmit a report packet. The slot index $Slot_i$ is the position of a time-slot inside the reporting phase. Eq. (C.8) is used at the beginning of the connection phase to initialize the remaining duration of the connection phase T_{rcp}, followed by (C.9) and (C.10), which are used once for each connection approval packet to be sent. Eq. (C.11) is used at the beginning of each frame to initialize the remaining duration of the frame T_{rf}, followed by (C.12) and (C.13), which are used once for each slot in the time frame.

$$T_{rcp} = T_{cp} \tag{C.8}$$

$$T_{rcp} = T_{rcp} - T_{ack} \tag{C.9}$$

$$T_{\alpha} = T_{rcp} + (T_{time-slot} \times Slot_i) \tag{C.10}$$

$$T_{rf} = T_{frame} \tag{C.11}$$

$$T_{rf} = T_{rf} - (T_{time-slot} \times (Slot_i + 1)) \tag{C.12}$$

$$T_{\beta} = T_{rf} + (T_{time-slot} \times Slot_i) \tag{C.13}$$