# A Cloud Infrastructure as a Service for an Efficient Usage of IoT Capabilities

**Jasmeen Kaur Ahluwalia**

A Thesis

in

the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

November 2020

**CONCORDIA UNIVERSITY**

**SCHOOL OF GRADUATE STUDIES**

This is to certify that the thesis prepared

By: Jasmeen Kaur Ahluwalia

Entitled: "A Cloud Infrastructure as a Service for an Efficient Usage of IoT Capabilities"

and submitted in partial fulfillment of the requirements for the degree of

**Master of Computer Science**

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair

Dr. Y.-G. Guéhéneuc

_____ Examiner

Dr. Y.-G. Guéhéneuc

_____ Examiner

Dr. J. Rilling

_____ Supervisor

Dr. R. Glitho

Approved By: _____

Dr. Mourad Debbabi

Interim Dean, Faculty of Engineering and Computer Science

_____20__

_____

Dr. Mourad Debbabi

Interim Dean,

Faculty of Engineering and Computer Science

# Abstract

A Cloud Infrastructure as a Service for an Efficient Usage of IoT Capabilities

Jasmeen Kaur Ahluwalia

The Internet of Things comprises of a system of devices (or objects) connected to the Internet and interacting with each other to satisfy various tasks or goals. These objects could be sensors, actuators, smart phones, smart appliances, etc. With the ever-increasing demand of IoT in daily life as well as in the industry, and billions of devices being connected over the internet, most IoT applications aim for cost and energy efficiency, scalability, and minimal latency in terms of resource provisioning.

To fulfill these requirements, Cloud Computing might prove beneficial. Cloud Computing provides on demand access to configurable computing resources (servers, memory, network, etc.) in the cloud, which require minimal management by the end user. It comprises of three service models, which are: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The cloud IaaS aims at an efficient usage of resources. In the specific case of IoT, these resources are the sensing and actuation capabilities. However, there are still many challenges that the design and implementation of an IoT IaaS faces. Some examples are the heterogeneity of the sensors and actuators, orchestration, provision of bare metal access, and also publication and discovery of the capabilities of IoT devices.

This thesis aims at the design and implementation of an architecture for IoT IaaS. First, it lays down a set of requirements essential to the architecture. This is followed by a thorough review of the state of the art. Next, it proposes an architecture for IoT IaaS that utilizes node level virtualization for an efficient usage of IoT capabilities. Functional entities are proposed as well as

interfaces relying on RESTful Web services. The interfaces include a low-level interface for homogenously accessing all the heterogenous capabilities of IoT devices, as well as high level interfaces which allow the IoT cloud users (e.g. PaaS or individual applications) to access these capabilities in an efficient manner. We have implemented a prototype using real-life as well as simulated Temperature sensors & Humidity sensors, and EV3 LEGO Mindstorms robots. The architecture is validated by concrete measurements on the prototype and by extensive simulations.

# Acknowledgements

First of all, I would like to give my sincere thanks to my supervisor Dr. Roch Glitho, without whose guidance and constant support I would not have reached this far. I wholeheartedly thank him for accepting me as his student and for helping me stay on track and progress well by providing me with his expertise. His patience, knowledge, and motivation truly helped me in not only enhancing my abilities but also in overcoming challenging tasks that seemed impossible initially.

I would also like to express my sincere gratitude to Dr. Yann-Gaël Guéhéneuc and Dr. Juergen Rilling for serving as the members of my thesis committee. I would also like to thank Dr. Yann-Gaël Guéhéneuc for serving as the chair at my thesis defence.

I would also like to thank my colleagues at Concordia University for their guidance, help, and encouragement. I would like to especially thank Carla Mouradian for helping me and guiding me at every stage of the thesis, and Vahid Maleeki for his encouragement and support. I would also like to thank Nazmul Alam for providing me with his valuable advice whenever I faced issues. I would further like to thank my colleague and friend Ujjwal Khanna for motivating me and boosting my confidence.

Finally, I am forever indebted to my parents and my sister Divleen, without whose constant motivation, encouragement, and support I would not have reached this far in life and accomplished so much. I can never thank them enough for supporting me through every step of my journey and always believing in me incessantly.

# Contents

# List of Figures

# List of Tables

# Acronyms and abbreviations

**AMD-V** Advanced Micro Dynamics – Virtualization.

**Amazon EC2** Amazon Elastic Compute Cloud.

**ANSI** American National Standards Institute.

**API** Application Programming Interface.

**AWS** Amazon Web Services.

**CAN** Controller Area Network.

**CPU** Central Processing Unit.

**CRUD** Create Read Update Delete.

**EPC** Electronic Product Code.

**EV3** Evolution 3.

**GRE** Generic Routing Encapsulation.

**HTTP** HyperText Transfer Protocol.

**HVAC** Heating Ventilation and Air Conditioning.

**IaaS** Infrastructure as a Service.

**IDE** Integrated Development Environment.

**$I^2C$** Inter-Integrated Circuit.

**Intel-VT** Intel Virtualization Technology.

**IoT** Internet of Things.

**IR** Infrared.

**IT** Information Technology.

**JSON** JavaScript Object Notation.

**JVM** Java Virtual Machine.

**LAN** Local Area Network.

**nesC** Network Embedded Systems C.

**NFC** Near Field Communication.

**NFV** Network Functions Virtualization.

**NFVI** Network Functions Virtualization Infrastructure.

**OMNeT++** Objective Modular Network Testbed in C++.

**OPNFV** Open Platform for Network Function Virtualization.

**OS** Operating System.

**PaaS** Platform as a Service.

**PC** Personal Computer.

**PIR** Passive Infrared.

**REST** Representational State Transfer.

**RFID** Radio Frequency Identification.

**SaaS** Software as a Service.

**SDN** Software Defined Networking.

**Se-aaS** Sensing as a Service.

**SLA** Service Level Agreement.

**SOA** Service Oriented Architecture.

**SPI** Serial Peripheral Interface.

**URI** Uniform Resource Identifier.

**USB** Universal Service Bus.

**UID** Unique Identification.

**UUID** Universally Unique Identifier.

**VM** Virtual Machine.

**VMM** Virtual Machine Monitor.

**VSN** Virtual Sensor Network.

**WSN** Wireless Sensor Network.

# Chapter 1

# Introduction

This chapter consists of a definition of the key terminologies and concepts which are related to this thesis. These definitions are then followed by a description of the motivation and problem statement, and the contribution of this thesis. The last section of this chapter consists of an overview of how the rest of the thesis is organized.

## 1.1. Definition

In the section to follow, the definitions of key terms essential to this thesis are provided. The terms included in this section are Internet of Things, IaaS (Infrastructure-as-a-Service), and Cloud Computing. These are the terminologies that are most crucial to this thesis.

### 1.1.1. Internet of Things (IoT)

The Internet of Things refers to a paradigm that "*enables physical objects to see, hear, think and perform jobs by having them "talk" together, to share information and to coordinate decisions. The IoT transforms these objects from being traditional to smart by exploiting its underlying technologies such as ubiquitous and pervasive computing, embedded devices, communication technologies, sensor networks, Internet protocols and applications*" [1]. These physical 'objects' involved in IoT are essentially devices that can perform some type of computation and communication, and all these devices can have differing capabilities. The main components that make up the Internet of Things include these objects, a network for communicating data, such as the Internet, and backend servers to handle and process this data. In

fact, the Internet of Things is a very broad domain that is now gaining immense popularity and providing efficient solutions in nearly all sectors and industries, such as health care, traffic systems, education, retail, etc. Some examples of the various IoT devices include microcontrollers such as ESP8266, Arduino, to which external sensors can be attached, sensors and actuators such as Virtenio Preon32, Lego EV3 Mindstorms, DHT22, Advanticys TelosB Skymotes, etc.

### 1.1.2. Cloud Computing

Cloud Computing provides on demand access to configurable computing resources (servers, memory, network, etc.) in the cloud, which require minimal management by the end user. It *"refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services. The services themselves have long been referred to as Software as a Service (SaaS). The datacenter hardware and software is what we will call a Cloud*" [2]. The resources provided by the cloud platforms are provided on a pay-per-use basis and allow the users to easily carry out dynamic provisioning on a seemingly infinite pool of computing resources [2]. When these cloud services are made available to the general public using the pay-per-use model, it is referred to as the *Public cloud.* When these resources, such as datacenters, are not made available to the public and are instead utilized internally by an organization, it is referred to as the *Private Cloud*. Cloud computing includes three service models which are: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS is when cloud providers give users access to servers, networking, storage, and other such computing resources; PaaS is when the cloud providers provide the users with software development environments where they can build their own applications; and SaaS is when the cloud service providers give users access to specific applications or software for them to use.

### 1.1.3. Infrastructure as a Service (IaaS)

"*Infrastructure as a service (IaaS) is an instant computing infrastructure, provisioned and managed over the internet*" [3]. In simpler words, it is a category of cloud computing that allows the users to leverage resources like storage, processing, network, computing, and other "Infrastructural" resources on a pay-per-use or on-demand basis. These resources can be used by the users to run applications requiring different degrees of computational power, or other software. It allows the users to cut on the costs of purchasing high cost physical resources such as servers, operating systems, and other datacentre resources, and instead use the infrastructure provided by these IaaS Cloud service providers on demand. Examples of some providers of IaaS include Microsoft Azure, Amazon Web Services (AWS), Google Compute Engine, IBM Cloud etc.

## 1.2. Motivation and Problem Statement

The emergence of Cloud Computing has made it somewhat possible to provision the heterogenous IoT devices (sensors, actuators, etc.) in a manner that is scalable, energy efficient, and suffers minimal latency to some extent. However, the conflicting properties of the cloud and IoT infrastructure pose many challenges to the successful integration of Cloud computing and IoT. More specifically, "*IoT is generally characterized by real world small things, widely distributed, with limited storage and processing capacity, which involve concerns regarding reliability, performance, security, and privacy*" [4]. Thus, IoT devices are usually resources constrained and expensive. On the contrary, "*Cloud computing has virtually unlimited capabilities in terms of storage and processing power, is a much more mature technology, and has most of the IoT issues at least partially solved*" [4]. To address this particular challenge and bridge the gap between these incompatibilities in the two infrastructures, it is essential to decouple the IoT device services from

the physical IoT devices. The IoT devices (sensors and actuators) can be virtualized through node level virtualization, which will enable these IoT devices to become multi-purpose by concurrently running several applications on a single node [5]. The virtualization of these devices will allow the applications to share the capabilities of these devices and have access to them in a manner similar to the rest of the cloud infrastructure. It will not only improve the costs, but also provide better flexibility in terms of IoT device access. Currently there are several IoT devices in the market that support virtualization, for example, Virtenio Preon 32 Shuttle with Varisen Module, Advanticsys CM5000 and XM1000 sensors, etc.

However, in order to utilize the IoT infrastructure in a manner similar to the Cloud infrastructure, it is essential to design and implement an IoT IaaS. This turns out to be a very challenging task due to the heterogenous nature of the IoT devices and their capabilities. Each IoT device supports different modes of communication and different types of tools and platforms. In fact, some IoT devices might not support virtualization at all. Hence the first challenge encountered in the design of an IoT IaaS is the creation of a high-level interface to access the IoT IaaS, as well as a low-level interface to access the different IoT devices in a homogenous manner. The second challenge is to build a mechanism for allowing the publication and discovery of the various IoT devices and their capabilities. The third challenge would be the need for an orchestration mechanism to orchestrate the different device capabilities based on the application's requirements. The fourth challenge would be to allow virtualization as well as bare metal access to devices. This will also be helpful in situations when certain devices do not support virtualization at all. The fifth and last challenge would be to allow the automatic triggering of certain IoT devices based on the outputs obtained from other IoT devices, for example, dispatching fire-fighting robots automatically when fire is detected by sensors.

## 1.3. Thesis Contributions

The contributions made by this thesis are as follows:

- A set of requirements essential to the IoT IaaS architecture.
- Review of the state of the art and its evaluation based on our derived requirements.
- An architecture for the IoT IaaS, with IoT devices such as sensors and actuators as part of the infrastructure, and virtualized, as well as Bare metal access to the IoT devices.
- High level interfaces to allow access to the IoT IaaS.
- Low level interfaces for accessing the heterogenous IoT devices in a uniform manner.
- Implementation of the prototype and evaluation of its performance metrics.
- A Simulation using Contiki Cooja to measure the scalability performance of the architecture.

## 1.4. Thesis Organization

The remaining thesis is organized in the following manner:

Chapter 2 focusses on the background and key concepts associated with this thesis, where each concept is discussed in detail.

Chapter 3 presents the motivating scenario and also lays down the set of requirements essential to the IoT IaaS architecture. The state of the art is also reviewed and assessed against these requirements.

Chapter 4 focuses on the description of the proposed IoT IaaS architecture. Each component of the architecture is explained along with the various interfaces utilized.

Chapter 5 describes the tools and platforms used for realizing the proof-of-concept prototype. The implemented architecture is also thoroughly explained followed by detailed explanations of the performance metrics for the architecture's evaluation.

Chapter 6 provides a conclusion to the thesis. The overall thesis contributions are summarized and the future research directions for the proposed architecture are identified.

# Chapter 2

# Background

The goal of this chapter is to explain all the terms and concepts related to this thesis. The chapter begins by providing a detailed overview of the Internet of Thing (IoT) including its enabling technologies and application areas. This is followed by a review of the concept of virtualization and its various techniques, with special focus on virtualization in IoT devices. This includes node-level virtualization, which is essential for this thesis. Then, we describe Cloud Computing, its features and service models with a special focus on IaaS, which is crucial to this thesis. We also provide a brief description of Bare Metal Provisioning before concluding the chapter.

## 2.1. The Internet of Things (IoT)

In this section, we present an overview of the internet of things. First, a general definition along with the simplified structure of the IoT is presented. This is followed by a brief overview of the key enabling technologies of IoT and its application areas.

### 2.1.1. General Definition of the Internet of Things (IoT)

The Internet of Things refers to a paradigm that "*enables physical objects to see, hear, think and perform jobs by having them "talk" together, to share information and to coordinate decisions. The IoT transforms these objects from being traditional to smart by exploiting its underlying technologies such as ubiquitous and pervasive computing, embedded devices, communication technologies, sensor networks, Internet protocols and applications*" [1]. It can further be defined as *"An open and comprehensive network of intelligent objects that have the capacity to auto-organize, share information, data and resources, reacting and acting in face of*

*situations and changes in the environment"* [6]. These physical 'objects' involved in IoT are essentially devices that can perform some type of computation and communication, and all these devices can have differing capabilities. The main components that make up the Internet of Things include these objects, a network for communicating data, such as the Internet, and backend servers to handle and process this data. In fact, the Internet of Things is a very broad domain that is now gaining immense popularity and providing efficient solutions in nearly all sectors and industries, such as health care, traffic systems, education, retail, manufacturing etc.

Figure 1 [7] shows the main components that constitute the Internet of Things. The first essential component is the IoT devices, represented as 'Smart devices', which make up the entire infrastructure for the Internet of Things. These devices or 'things' are capable of interacting with other devices on the network as well as with users. These devices must be capable of computation and some forms of communication. Moreover, these 'things' must be context aware, meaning that they must be able to dynamically adapt to their changing environments or contexts and accordingly take suitable measures for self-configuration. Some examples of these 'things' are sensors, actuators, smart phones, etc. The second component of the Internet of Things is the Network Infrastructure, which refers to all the resources that make up this network over which these devices can communicate. For example, the internet. This network allows these devices to send and receive data to and from other devices, other servers, or other platforms connected to the network. The third component is the Cloud and back end servers [7].

Sensors and actuators, the 'things' or 'smart devices' which make up an essential part of the IoT IaaS proposed in this thesis, are described below.

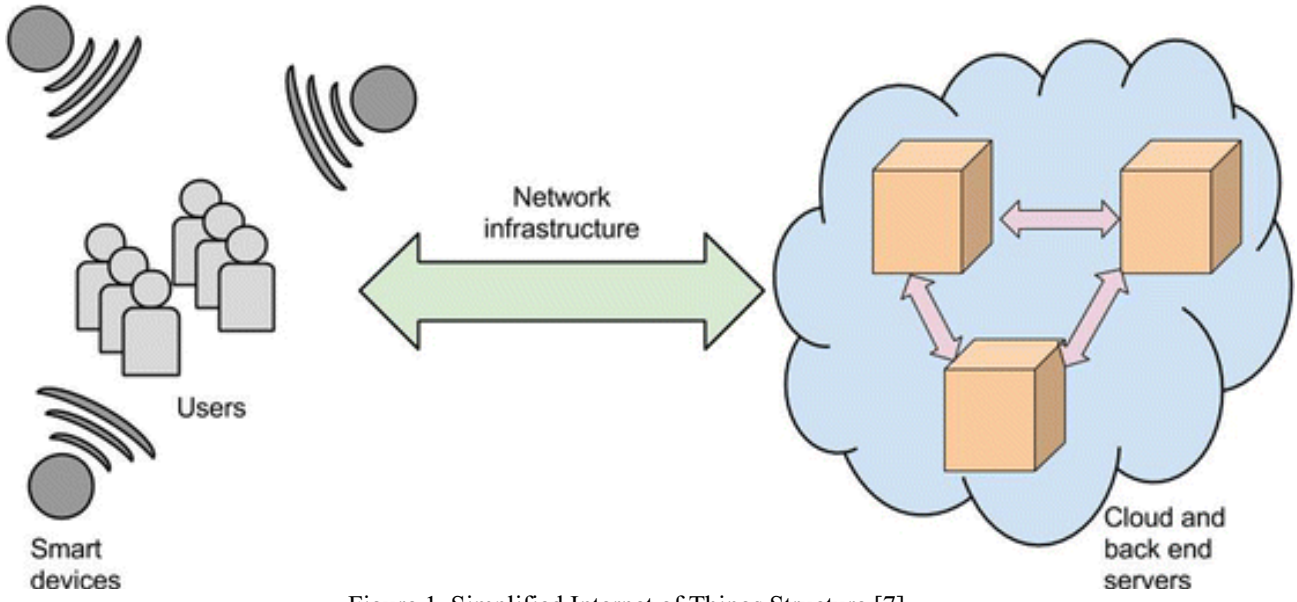


Figure 1. Simplified Internet of Things Structure [7]

#### 2.1.1.1. Sensors

Sensors are devices, modules, or machines that can sense the environment and the changes taking place within it, communicate with other devices or systems to send this data, and occasionally, perform basic computations on the data being collected [8]. They are always used with other computing systems/electronics. Nowadays, sensors are used in nearly every system and have become of wide importance. They are key enablers for IoT and are used in everyday objects such as Air Conditioning and Heating system thermostats, smart phones, smart home systems, security systems etc. Some examples of sensors are Virtenio Preon32 with VariSen module, Advanticsys SkyMote sensors like XM1000 and CM5000, DHT22 temperature-humidity sensor, PIR motion sensor, Arduino, etc.

#### 2.1.1.2. Actuators

*"Actuators perform actions to change the behavior of the environment or physical systems"* [9]. For instance, a robot starts moving when some command is sent to it. *"In many situations,*

*actuator nodes typically have stronger computation and communication powers and more energy budget that allows longer battery life"* [9]. In order to properly function, actuators need an energy source and a control signal. This signal can come from various other sources, for example, devices such as sensors, PCs, other IoT devices, or other events taking place in the external environment of the actuators. The actuators, thus, perform certain actions when triggered. Examples of various actuators are Lego Mindstorms Robots, relay motors, solenoids, etc.

### 2.1.2. Enabling Technologies of IoT

In this section, the key enabling technologies of IoT are described. According to [10], there are several capabilities that allow these smart 'things' to interact with and understand their environments. These capabilities are the key enablers on which IoT relies. In [10], these enabling technologies have further been divided into 3 categories: hardware, software, and architectures.

#### 2.1.2.1. Hardware

The hardware infrastructure on which IoT is built primarily includes RFID, NFC, Sensor Networks.

##### 2.1.2.1.1. RFID (Radio Frequency Identification)

*"RFID is a short-range communication technology where an RFID tag communicates with an RFID reader via radio-frequency electromagnetic fields"* [10]. The RFID tags contain some form of data. It allows for tracking as well as identification of the objects with the RFID tags attached on it. This proves beneficial for IoT. For IoT applications, the data contained in these RFID tags is mostly Electronic Product Code (EPC). This EPC is essential to IoT as it allows each device in the IoT network to be uniquely identified, as the EPC is unique for each 'thing' [10].

#### 2.1.2.1.2. NFC (Near Field Communication)

NFC is a communication technology that has evolved from RFID and is a communication standard over short-range. NFC allows devices to communicate via radio communication when kept in close proximity to each other. Similar to RFID, each device with NFC capability contains a tag that uniquely identifies the device, also called Unique Identification (UID). Currently, most smart phones contain the NFC technology and are capable of transferring data to each other quickly, when kept within few centimetres from each other. In fact, NFC devices can also connect with objects containing NFC tags that are not powered up, or are passive, for example, smart posters with NFC tags containing relevant data [10]. The NFC chips have very low power consumption, and thus NFC is currently one of the most efficient ways of wireless communication.

#### 2.1.2.1.3. Wireless Sensor Networks

Sensors are devices, modules, or machines that can sense the environment and the changes taking place within it, communicate with other devices or systems to send this data, and occasionally, perform basic computations on the data being collected [8]. Whenever a collection of several sensors is utilized as a network, where they are capable of interacting with each other as well as with the external environment, it is called a wireless sensor network (WSN). *"Wireless sensor networks contain the sensors themselves and may also contain gateways that collect data from the sensors and pass it on to a server"* [10]. It is also possible to have sensor-actuator networks as part of the WSN. This can allow the networks to sense the environment as well as perform some action, or interact with the environment in response, which is an essential objective of IoT. For example, in the case of smart irrigation systems, sensor-actuator networks can allow detection of low water levels in crops through temperature and humidity sensors, which will in

turn signal the relay motor to turn on in order to water the plants. Thus, sensor-actuator networks are frequently utilized.

#### 2.1.2.2. Software

Since the IoT hardware and infrastructure comprises of several different types of devices, i.e. heterogenous devices, it is essential to constantly have new software available in order to allow interoperability between these devices, as well as the ability to search the data being generated by these different IoT devices [10]. The key enabling technologies for IoT with regards to software are described in this section.

##### 2.1.2.2.1. Middleware

In order to provide abstraction to the IoT applications from IoT devices, it is essential to have an application-independent software in between, known as middleware. The middleware enables interoperability between heterogenous IoT devices. It "*sits between the IoT hardware and data and the applications that developers create to exploit the IoT*" [10]. It facilitates the connection between the IoT devices and applications irrespective of the underlying networks, hardware, operating system, etc. This gives the flexibility to developers to simply focus on creating and deploying new IoT applications instead of worrying about writing different application code for different types of IoT device platforms [10]. Some examples of middleware solutions that exist include Hydra, Impala, Lime, MiLAN etc.

##### 2.1.2.2.2. Searching/Browsing

IoT devices tend to generate a large amount of information and data. Moreover, due to their capabilities to adapt dynamically to their environment, often the information generated by these devices keeps changing. Therefore, similar to the search engines that currently exist for the World

Wide Web, there needs to be a similar search engine for IoT devices, which can constantly search the information that these devices generate. Moreover, similar to the current internet browsers, there is also a dire need for a browser for the IoT, which can interact with the IoT devices, identify them, and discover their capabilities [10].

#### 2.1.2.3. Architecture

There are several architectures that have been proposed in order to represent, organize, and structure the IoT [10]. This section aims at exploring these architectures that can support the IoT devices and their services. In [10], the architectures for IoT have been classified into hardware/network, software, process, and general. The hardware/network architectures are proposed in order to handle and provide support for the distributed computing environments for the heterogenous IoT devices. Examples include peer-to-peer architecture. In order to access and share services of the IoT smart devices, various software architectures have also been proposed. Some examples include Service Oriented Architectures (SOA), Representational State Transfer (REST) model, etc. Certain process-based architectures have also been proposed in order to provide a structure for the business processes that make use of IoT, for instance, architectures for structuring workflows. The last category of architectures, general/requirements, includes those architectures that are generalised, not based on specific categories, since currently there is no single architecture that is the best fit for IoT. This includes various architectural design concepts that have been proposed and currently exist in the literature [10].

### 2.1.3. Application Areas for IoT

With the advancement in technology and the ever-increasing demand for IoT, the number of domains within which IoT applications can now be used are limitless. There are several categories

within which the IoT applications can be categorized. In [10] and [11], several applications of IoT are proposed, which are summarized below.

#### 2.1.3.1. Healthcare

IoT applications are being extensively used in the healthcare industry, especially in aiding in assisted living. Certain tasks such as making decisions based on patients' symptoms, monitoring body fluid levels and other bodily changes are now being handled by several IoT applications. Patients' monitoring equipment often contain smart sensors that collect the patients' health data and make them available to doctors, as well as provide treatments during certain circumstances. Examples of some smart sensors in the healthcare sector include blood glucose level sensors, blood pressure sensors, sensors for detecting heart attacks etc. The information collected by these sensors is sent on to the cloud from where it can be made available as needed [10].

#### 2.1.3.2. Supply Chains/Logistics

Supply chains and logistics industry has been making use of RFID and sensor networks for tracking and tracing products in manufacturing as well as other parts of the supply chain process. Currently, IoT applications are improving the processes within the supply chain and logistics sector by providing up-to-date information in an efficient and reliable manner [10]. IoT applications are being used for inventory management and tracking in warehouses, monitoring transportation of the items, decision making processes and analysis in this industry etc.

#### 2.1.3.3. Smart Transportation

Smart transportation, i.e. Intelligent transportation systems, is another area of research which is being extensively explored. IoT applications are being built to enable smart transportation systems. Smart transportation refers to a network of smart vehicles, interconnected with each other and

capable of communicating, as well as smart traffic signals, etc. Smart transportation aims at improving the current transportation system through the use of cloud computing and IoT in order to make the system more secure, reliable, and efficient for the citizens [11]. Google's self-driving cars, vehicles developed by Tesla Inc. are some examples of smart vehicles that are currently being tested. The main idea is to allow vehicles to use IoT and cloud services to share data with other vehicles nearby, as well as to analyse the traffic data within its vicinity. One such example can be a smart car which communicates with the smart vehicles present within a range of a few kilometres and can thus judge which routes would be less congested than others. It can then suggest the driver the best path to reach the destination making it possible to avoid getting stuck in traffic congestion.

#### 2.1.3.4. Smart Infrastructure

*"Integrating smart objects into physical infrastructure can improve flexibility, reliability and efficiency in infrastructure operation"* [10]. The IoT technology is now being utilized to enhance the infrastructure of homes, industries, offices, parking lots, public spaces etc., thus, invoking the concept of 'smart cities'. Smart homes are gaining increased popularity, where the homes are fitted with smart infrastructure, such as smart HVAC systems, smart security systems, smart lighting, smart appliances, smart energy consumption systems etc., which make homes more secure, giving the resident a superior experience. Similar systems are also being incorporated into offices to enhance the security and reduce the costs and power consumption. To enable the realization of 'smart cities', all of its sub-applications (which include smart transportation, smart healthcare, smart infrastructure, etc.) must be incorporated. The city of Padova in Italy is one such example where a smart city framework has been deployed [11].

#### 2.1.3.5. Social Applications

IoT technology is also being increasingly incorporated into social media applications for several functionalities such as detecting friends, social events, or activities taking place nearby, or fetching information about the whereabouts and activities of an individual etc. For example, social media applications such as snapchat now allow people to detect their friends present within a range of a few kilometers. Another example can be the transfer of data (through NFC or other IoT technologies) between several smart devices just by bringing the devices in the vicinity of each other [10]. All these are examples of IoT technologies being integrated into the various social networking platforms, providing users with a better experience and service.

## 2.2. Virtualization

This section provides detailed overview of virtualization, which is a key concept utilized throughout the IoT IaaS proposed in this thesis. First, a general definition of virtualization is provided, followed by explanations of the various types of virtualizations that currently exist. This is followed by a brief overview of the techniques to virtualize IoT devices.

### 2.2.1. Definition

The term Virtualization refers to a technology that "*promises a reduction in cost and complexity through the abstraction or emulation of physical resources (e.g., servers, network links, and host bus adapters) into logical units*" [12]. Virtualization, thus, allows decoupling between the hardware infrastructure and the software and applications running on the machine. This allows complete and more efficient utilization of the hardware resources, thus minimizing the overall costs. To perform this decoupling between the hardware resources and applications running on the machine, a Hypervisor (also called Virtual Machine Monitor-VMM) is used. A hypervisor allows

one host computer to support multiple guest VMs (Virtual Machines) by virtually sharing its resources, such as memory and processing [13]. The hypervisor software, which is installed directly on the system hardware, divides/partitions the hardware resources as needed amongst the various virtual machines running on top of it. These virtual machines run independently from each other, in an isolated manner, while the hypervisor manages how they share the underlying hardware resources. According to [14], virtualization can be of several types, which are as described below.

#### 2.2.1.1. Server Virtualization

In Server Virtualization, special software is used to virtualize one physical server into many virtual servers. The virtual servers run in isolation from each other and can run different operating systems. This allows optimum CPU utilization as the resources are not underutilized, since multiple virtual servers are running on it, thus reducing the CPU idle time. Server virtualization is further divided into three types.

##### 2.2.1.1.1. Full Virtualization

In Full Virtualization, the hypervisor has complete control over the resources of the physical server and is responsible for providing them, as needed, to the different virtual servers. The *"hypervisor creates isolated environment between the guest or virtual server and the host or server hardware"* [15]. Privileged instructions sent by the virtual servers are trapped by the hypervisor. Moreover, the overall performance of the server is quite slow since the hypervisor needs some processing power for itself as well. However, each virtual machine server is provided with complete isolation and maximum security in full virtualization. It also makes the migration and portability relatively easily as the guest OS can be migrated and run as-is on other virtualized or

physical servers. Some examples of products which provide full virtualization include VMWare ESXi, Microsoft Virtual Server, etc.

#### 2.2.1.1.2. Para-Virtualization

*"Para virtualization modifies the OS kernel in order to replace the non-virtualizable instructions with hyper calls which can communicate directly with the virtualization layer i.e., hypervisor"* [14]. The guest OS and hypervisor are thus able to communicate with each other. Moreover, the various guest servers are also not fully isolated (but are partially isolated) and are able to work together in a more efficient manner. Furthermore, the guest Operating System is modified to be able to run on the hypervisor, and the hypervisor does not need as much processing power for itself as it needed in full virtualization. The privileged instructions sent by the guest OS are not trapped by the hypervisor. Para-virtualization provides much better performance compared to full virtualization but is not as good when it comes to migration and portability since the guest OS is modified to be compatible with the hypervisor. An example of a para virtualization project is the Xen Windows GPLPV.

#### 2.2.1.1.3. Hardware Assisted Virtualization

Hardware Assisted Virtualization is when the features for virtualization are built into the hardware (CPUs) in order to simplify the virtualization techniques. It thus enables virtual machines to be run without any modification, and with less overhead compared to full virtualization. Moreover, "*the privileged and sensitive calls are set to automatically trap to the hypervisor and removes the requirement for either binary translation or paravirtualization"* [14]. Examples of some processors that include these features include AMD-V where Virtual Control Blocks are used for storing the guest state, and Intel-VT-x where Virtual Machine Control Structures are responsible for storing the guest state [14].

#### 2.2.1.2. Desktop Virtualization

*"Desktop virtualization refers to the virtualization of the computer desktop in order to achieve security and flexibility of the desktop usage"* [16]. In Desktop virtualization, the desktop environment is completely decoupled from the computing device that is utilized to access it (i.e. physical client device). This poses an advantage that the user can access their desktop from any client device. Moreover, desktop virtualization, as a result, also *"reduces the need for duplicate hardware and has other economical aspects"* [14]. Some examples of desktop virtualization software include Citrix XenDesktop, Microsoft Remote Desktop Services, etc.

#### 2.2.1.3. Virtual Networks

Virtual Networks, which can at times also be called Virtual Private Networks allow the user to believe that they are directly connected to a company's network or other resources, even if there is no direct physical link present. The users can utilize any internet network to connect to a virtual private network that can provide them access to the company's resources [14].

### 2.2.2. Virtualization of IoT Devices

Virtualization of IoT devices is more challenging compared to the virtualization of traditional nodes such as servers, computers, etc. This is because IoT devices possess limited processing capacity, storage, and might also be battery operated. Their resource-constrained nature requires more efforts in terms of carrying out effective virtualization in order to realize the true potential of these IoT devices. It is possible to virtualize the IoT device nodes at node-level and at network level [17].

These two approaches of virtualizing IoT devices are described in this subsection. Furthermore, the key differences between these two approaches are also explored within this subsection.

#### 2.2.2.1. Node-Level Virtualization

When virtualization is utilized to allow several applications to run concurrently on a single sensor node or other IoT device, it is referred to as Node-level virtualization [5]. This technique, thus, allows efficient utilization of the devices by rendering them multi-purpose. The two main ways in which node level virtualization is carried out are: Sequential execution and Simultaneous execution [5]. *"Sequential execution can be termed a weak form of virtualization, in which the actual execution of application tasks occurs one-by-one (in series)"* [5]. Sequential execution is, thus, much easier to implement but is not as efficient since the applications have to wait in order to execute their tasks and utilize the device resources. Whereas, in the case of simultaneous execution, *"application tasks are executed in a time-sliced fashion by rapidly switching the context from one task to another"* [5]. Even though simultaneous execution is much more complex to implement, it is beneficial as the waiting times for applications are reduced as context switching at time intervals allows each application to carry out its task without having to endlessly wait for more time-consuming applications to finish their execution. Node-level virtualization, thus, essentially allows applications to share the devices' resources and capabilities, which is specifically beneficial in the case of IoT devices (sensor nodes, actuators etc.). In this thesis, node-level virtualization has been incorporated.

#### 2.2.2.2. Network-Level Virtualization

In Network-Level Virtualization, Virtual Sensor Networks (VSN) are formed. In this type of virtualization, a subset of the nodes in the IoT network (Wireless Sensor Network) are used to form VSNs. At a given time, the VSN can be dedicated exclusively to one application, thus providing it isolation [5]. Moreover *"enabling the dynamic formation of such subsets ensures*

*resource efficiency, because the remaining nodes are available for different multiple applications"* [5]. There are two ways of creating VSNs, one way is to use the same IoT network infrastructure and create several VSNs over them, or the second way is to have a VSN composed of nodes from administratively different IoT networks [5].

#### 2.2.2.3. Node-Level vs Network-Level Virtualization

Both node-level and network-level virtualization aim to increase resource utilization and efficiency within a network of physical IoT devices. According to [5], the general architecture of node-level virtualization is shown in figure 2, while figure 3 shows the general architecture for the two types of network-level virtualizations.

While node-level virtualization aims to enable *"multiple applications to run their tasks concurrently on a single sensor node, so that a sensor node can essentially become a multi-purpose device"* [5], network-level virtualization aims to enable the formation of dynamic Virtual Sensor Networks (VSNs) over the Wireless Sensor Networks (WSNs). This dynamic formation of the VSNs over a subset of the WSN's nodes can allow efficient utilization of resources as the remaining nodes in the network can then be available for other applications. Thus, it can be inferred that while node-level virtualization targets the increase in the utilization of a single IoT device through virtualization, network-level virtualization aims to increase the efficiency of a network of physical IoT devices through virtualization.

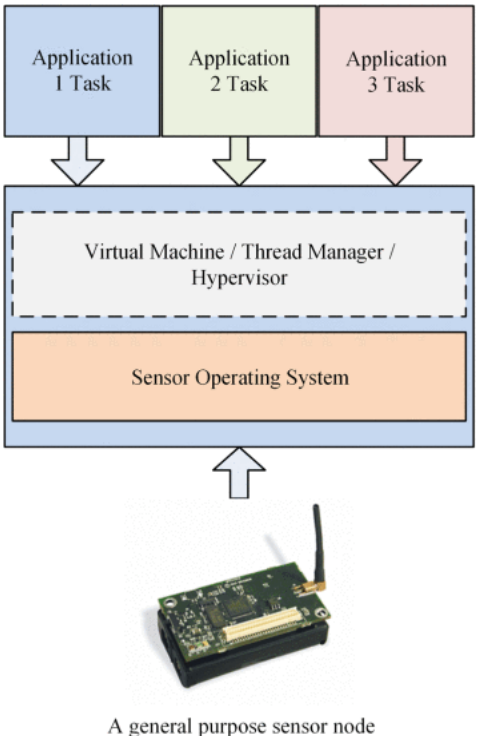


Figure 2. Node Level Virtualization: Execution of multiple applications in a general purpose WSN node [5]

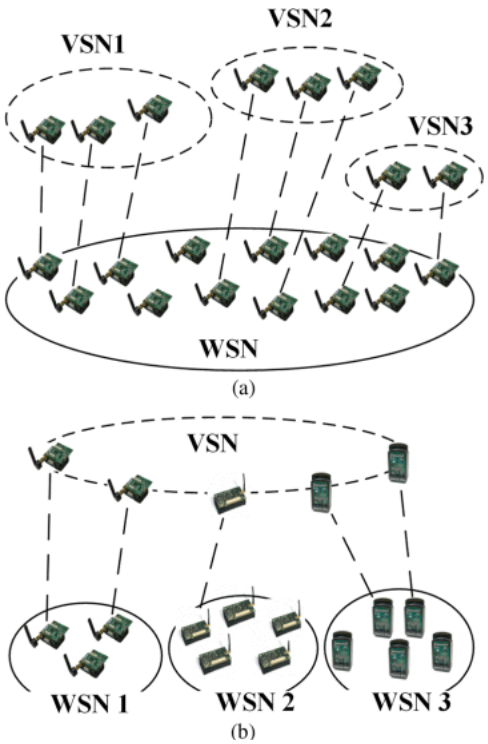


Figure 3. Network-Level Virtualization: (a) Multiple VSNs over single WSN (b) Single VSN over multiple WSNs [5]

Node-level virtualization consists of two approaches: sequential and simultaneous execution of the application task, while the two approaches of network-level virtualization are: creating multiple VSNs over the same underlying WSN infrastructure, or having a VSN which is composed of WSN nodes from different administrative domains (depicted in figure 3 (a) and (b)). Although network-level virtualization increases resources efficiency, it is still possible that the individual devices within the networks are underutilized, since at a given point in time the device is only a part of one virtual network and can thus only cater to the needs of the application using this network. This issue can only be tackled through node-level virtualization, which can allow several applications to run simultaneously on a single physical device. Thus, often a hybrid of the two techniques is used to achieve maximum resource, as well as cost efficiency [5].

## 2.3. Cloud Computing

This section aims at describing Cloud Computing. The section first gives an overview of Cloud computing, followed by a brief description of some of its unique characteristics. This is followed by a brief discussion of the advantages and disadvantages of Cloud Computing. It then covers the service models of Cloud Computing, with special focus on IaaS (Infrastructure-as-a-Service), which is a crucial concept for this thesis.

### 2.3.1. Definition

Cloud Computing provides on demand access to configurable computing resources (servers, memory, network, etc.) in the cloud, which require minimal management by the end user. It *"refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services. The services themselves have long been referred to as Software as a Service (SaaS). The datacenter hardware and software is what we will call a Cloud*" [2].

The resources provided by the cloud platforms are provided on a pay-per-use basis and allow the users to easily carry out dynamic provisioning on a seemingly infinite pool of computing resources [2]. When these cloud services are made available to the general public using the pay-per-use model, it is referred to as the *Public cloud.* When these resources, such as datacenters, are not made available to the public and are instead utilized internally by an organization, it is referred to as the *Private Cloud*. When the cloud infrastructure is made available for a specific community of users, it is called *Community Cloud,* for example a group of universities interconnecting their infrastructure to provide cloud services to their students, faculty, etc. [18]. Another type of cloud is the *Hybrid Cloud* where *"the computing infrastructure is a combination of two or more distinct*

*entities, namely, private cloud, public cloud or community cloud. Each entity remains distinct, but they are bound together by standardized protocols that permit data and application portability"* [18]. For example, when an organization is unable to handle the customer load on its private cloud, it may decide to use the services of the public cloud to manage the load. In this case, both private and public cloud are used.

### 2.3.2. Characteristics of Cloud Computing

Cloud Computing has several unique characteristics that make it particularly enticing to organizations as well as independent users. This section aims at briefly describing the characteristics of Cloud Computing that make it a distinct paradigm today.

#### 2.3.2.1. Multi-tenancy

Cloud Computing supports multi-tenancy. This implies that multiple users share the same resources provided by the cloud service providers. However, each user (tenant) uses the cloud platform in an isolated manner, and the data of each user remains separate from the other users. All the users are unaware of the fact that they are sharing the same resources.

#### 2.3.2.2. Scalability

*"Scalability is the ability of a system to sustain increasing workloads with adequate performance provided that hardware resources are added"* [19]. In simpler words, scalability refers to the ability to process increased workloads on the current infrastructure (scale up) or on the current plus some additional infrastructural resources (scaling out) without having drastic impacts on the performance, and without any interruptions. In order to maintain the performance when the load increases, cloud computing can allow either vertical scaling, i.e. scaling up within the existing infrastructure, or horizontal scaling, i.e. scaling out to additional infrastructure. The

infrastructural resources are usually of pre-planned capacity. When the application demands less resources, the IT manager can scale down the resources statically and thus reduce costs. Thus, in scalability, the resource allocation is such that it can suffice the maximum predicted workload without suffering significant performance degradation. Furthermore, as mentioned in [19], *"The scalability of a system including all hardware, virtualization, and software layers within its boundaries is a prerequisite in order to be able to speak of elasticity".*

#### 2.3.2.3. Elasticity

*"Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible"* [19]. In simpler terms, Elasticity refers to the ability to dynamically add or remove resources based on the needs of the users. These resources could be any of the cloud computing resources such as servers, storage resources, network resources, etc. Elasticity is a main feature of several public cloud platforms that rely on the pay-per-use model. It is closely associated with cloud solutions that provide horizontal scaling (scale-out). Elasticity allows cost efficiency as the users only pay for what they use. When the workloads are high, more resources are added and the users pay more, whereas when the workload is less, the users pay less since resources are removed. Thus, elasticity gives the users the illusion of there being an infinite pool of resources at their dispersal.

#### 2.3.2.4. Pay-per-use Model

Cloud Service Providers provide a pay-per-use or pay-as-you-go model to the customers. This essentially means that the users only pay for the services and infrastructure that they use. The

customers are billed based on several criteria, such as the number of hours or usage, workload, type of resources being used, etc.

#### 2.3.2.5. Dynamic Provisioning of Resources

*"One of the key features of cloud computing is that computing resources can be obtained and released on the fly"* [20]. Cloud computing makes it very easy for service providers to obtain resources based on the current demand. If the demand increases, they can easily acquire more resources, or release them when the demand decreases, thus lowering the cost of operating [20].

### 2.3.3. Advantages and Disadvantages of Cloud Computing

With the rapid growth of Cloud Computing, more and more users and organizations are starting to rely on its services for better managing their costs and enhancing the overall functioning of their organizations. In this section, we review the advantages that Cloud Computing poses, as well as its disadvantages.

#### 2.3.3.1. Advantages of Cloud Computing

According to [18], Cloud Computing has the following advantages:

- It allows organizations to reduce their capital expense as they do not need to purchase their own new infrastructure but can instead use the Cloud infrastructure at reasonable costs.
- Cloud computing provides a plethora of software systems and other services on a pay-per-use basis.
- Cloud Computing provides the users access to scalable and elastic infrastructure on demand. This allows organizations to increase their computing power by requesting the cloud resources as needed, thus, giving the illusion of infinite availability of resources.

This can be especially advantageous for start-up companies that can lower their investment costs by making use of the cloud resources and requesting more resources as needed.

- Cloud Service Providers and organizations sign Service Level Agreements (SLAs) which assure the quality of the service provided by the service providers to the organizations.
- Cloud services can be used by organizations for automatic backing up of data. In the case of server crashes or data corruption this will prove beneficial for immediate recovery of data.

#### 2.3.3.2. Disadvantages of Cloud Computing

There are several advantages in addition to those mentioned above. However, there are also certain disadvantages or risks that Cloud Computing can pose. These are listed below.

- Since access to the Cloud Computing infrastructure requires a constant connection, such as connection to the internet, etc., if the connection gets disrupted then the access to the Cloud services also gets cut off. This can be mitigated by having backup independent modes of connection to the Cloud Service Provider's services.
- The exchanging of data back and forth between the user and the Cloud Service over a public connection, such as the internet, can lead to security threats, such as eavesdropping by malicious third parties, corruption or stealing of data while it is being sent, etc. To avoid this issue, strong data encryption techniques need to be utilized along with other forms of security in order to make the access to the Cloud Services and exchange of data as secure as possible.
- Due to the lack of standardization of the services provided by different Cloud Service Providers, it becomes tedious for an organization to back-up and move all of their data onto a different Cloud platform in case the vendor they currently use is unable to provide the

desired quality of service, or declares bankruptcy. Such a situation can be mitigated if the organization considers vendors whose services are similar to the standards used by other vendors, thus allowing minimum software rewriting in case the vendor needs to be changed.

- Another disadvantage is when the Cloud Service providers and the users/organization using the services are both located in different countries. In this case, if the data gets corrupted or stolen, legal problems can arise. To avoid such a situation, the Service Level Agreements should clearly state what laws should apply during such a situation.

### 2.3.4. Service Models in Cloud Computing

Cloud computing includes three service models which are: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS is when cloud providers give users access to servers, networking, storage, and other such computing resources; PaaS is when the cloud providers provide the users with software development environments where they can build their own applications; and SaaS is when the cloud service providers give users access to specific applications or software for them to use.

#### 2.3.4.1. IaaS (Infrastructure-as-a-Service)

IaaS is the main focus of this thesis. This section provides a brief overview of IaaS. First, we define Infrastructure as-a Service, followed by a detailed overview of its architecture and the layers into which it is organized.

##### 2.3.4.1.1. Definition

*"Infrastructure as a service (IaaS) is an instant computing infrastructure, provisioned and managed over the internet"* [3]. In simpler words, it is a category of cloud computing that allows

the users to leverage resources like storage, processing, network, computing, and other "Infrastructural" resources on a pay-per-use or on-demand basis. These resources can be used by the users to run applications requiring different degrees of computational power, or other software. It allows the users to cut on the costs of purchasing high cost physical resources such as servers, operating systems, and other datacentre resources, and instead use the infrastructure provided by these IaaS Cloud service providers on demand. The role of the Cloud IaaS providers is to only provide the required hardware of appropriate capacity. In addition, several users utilize the hardware provided by the vendors at the same time. Thus, *"as different customers may deploy their own operating systems and applications running on them, the servers are enveloped by a layer of software which makes them behave like the hardware system demanded by the user"* [18]. This implies that virtualization is required, where several different types of virtual machines can be supported. This is achieved by running the hypervisor. IaaS allows multitenancy, and companies usually provide an Application Programming Interface (API) for users to easily be able to access the hardware [18]. Examples of some providers of IaaS include Microsoft Azure, Amazon Web Services (AWS), Google Compute Engine, IBM Cloud etc.

#### 2.3.4.1.2. IaaS Cloud Architecture

According to *Moreno-Vozmediano et at (2012)* [21], the IaaS Cloud Architecture is divided into distinct layers, which are shown in figure 4. The Cloud IaaS comprises of two main layers, which are the Physical Infrastructure and the Cloud Operating System. This section briefly describes the components present within this IaaS Cloud Architecture, which is responsible for managing the infrastructure (both physical and virtual), as well as the provisioning of resources.

#### 2.3.4.1.2.1. Physical Infrastructure

The physical infrastructure comprises of all the resources present in the data centre, which are servers, networks, storage units, etc. It is the lowest layer in the architecture.

#### 2.3.4.1.2.2. Cloud OS Drivers

This layer includes the 2 components: The Physical Infrastructure Drivers and the Cloud Drivers, as shown in the figure 4. These drivers and adapters are utilized by the Cloud Operating System to interact with the different virtualization technologies that permit the abstraction of the underlying infrastructure. These virtualization technologies could include the hypervisor, storage drivers, etc.

#### 2.3.4.1.2.3. Cloud OS Core

The Cloud OS Core comprises of several components such as the VM Manager, Network Manager, Storage Manager, etc. These components utilize the underlying Cloud OS drivers to manage the virtual infrastructure, deploy the virtual infrastructure as well as manage it after deploying. The components within this layer are as follows:

- **Virtual Machine Manager:** The Virtual Machine Manager manages the entire life cycle of the Virtual Machines. It is responsible for performing the VM actions such as deploy, resume, suspend, shut down, migrate etc. based on the commands sent by the user. *"To perform these actions, the VM manager relies on the hypervisor drivers, which expose the basic functionality of underlying hypervisors, and VMware to avoid limiting the cloud OS to a specific virtualization technology*" [21].
- **Network Manager:** The Network manager is responsible for managing the instantiation of all the possible networks on the physical network infrastructure. It uses the network

drivers for the provisioning of the virtual networks and allows the services to be accessible by external users through the management of the interconnection of various service components.

- **Storage Manager:** *"The storage manager's main function is to provide storage services and final-user virtual storage systems as a commodity"* [21]. The storage services provided to the user need to be scalable, reliable, available, easily manageable, and high performing. In order to suffice these requirements, the Storage Manager uses the storage drivers, which provide abstraction and allow the storage resources to appear as one, thus, enabling the storage manager to manage them easily based on the users' needs.
- **Image Manager:** The Image Manager is responsible for efficiently managing all the different VM images used by different users. These VM images have different configurations, different operating systems, etc. The Image Manager is also responsible for ensuring the security of the VM images.
- **Information Manager:** The Information Manager monitors the state of each VM, servers, and other infrastructural components and collects information on it. It, thus, ensures that each component is functioning normally and maintaining the expected level of performance.
- **Authentication and Authorization:** This component is responsible for authenticating the users and administrators in order to create a secure cloud environment, as well as give the customers access to authorized resources.
- **Accounting and Auditing:** This component is responsible for providing the billing information for each user (accounting), as well as monitoring the users' activities in the

cloud resources indicating which resources were accessed when and what operations were performed on these resources.

- **Federation Manager:** This component allows the Cloud OS to access remote and partner cloud infrastructures managed by similar or public cloud providers. It provides the basic mechanisms such as deployment, authentication, runtime management, etc. as well as several advanced features depending on the design and capabilities of the Federation Manager.

#### 2.3.4.1.2.4. Cloud OS Tools

The Cloud OS Tools layer comprises of components that allow the IaaS to be accessible by external users and organizations. It includes several components such as Administrator Tools, Service Manager, Scheduler, and Cloud Interfaces, which are briefly described below.

- **Scheduler:** The scheduler is responsible for managing the scheduling within the cloud infrastructure. Its role is to decide which VM will get access to the system resources, and which physical CPUs and other resources will be assigned to the VMs, as well as which VM will be deployed on which physical server.
- **Administrative Tools:** This component is responsible for providing the different interfaces and tools that will allow the users and administrators to perform several tasks on the Cloud OS. For example, tasks such as shutting down or starting servers; deploying, shutting down, suspending VMs etc.
- **Service Manager:** *"The cloud OS should be able to manage and support virtualized multitier services. A multitier service can comprise several component/tiers with some intrinsic dependencies among them"* [21]. The role of the Service Manager is to accept/reject services depending on the resources available and the requirements of the

service, and manage the entire life cycle of the accepted services, which would include deployment, canceling, suspending, etc.

- **Cloud Interfaces:** Cloud interfaces allow the services of the Cloud to be exposed to the users and organizations. This would involve standardized APIs, which the users can use to access the cloud services. Most cloud service providers provide their own APIs such as Amazon's EC2, etc.

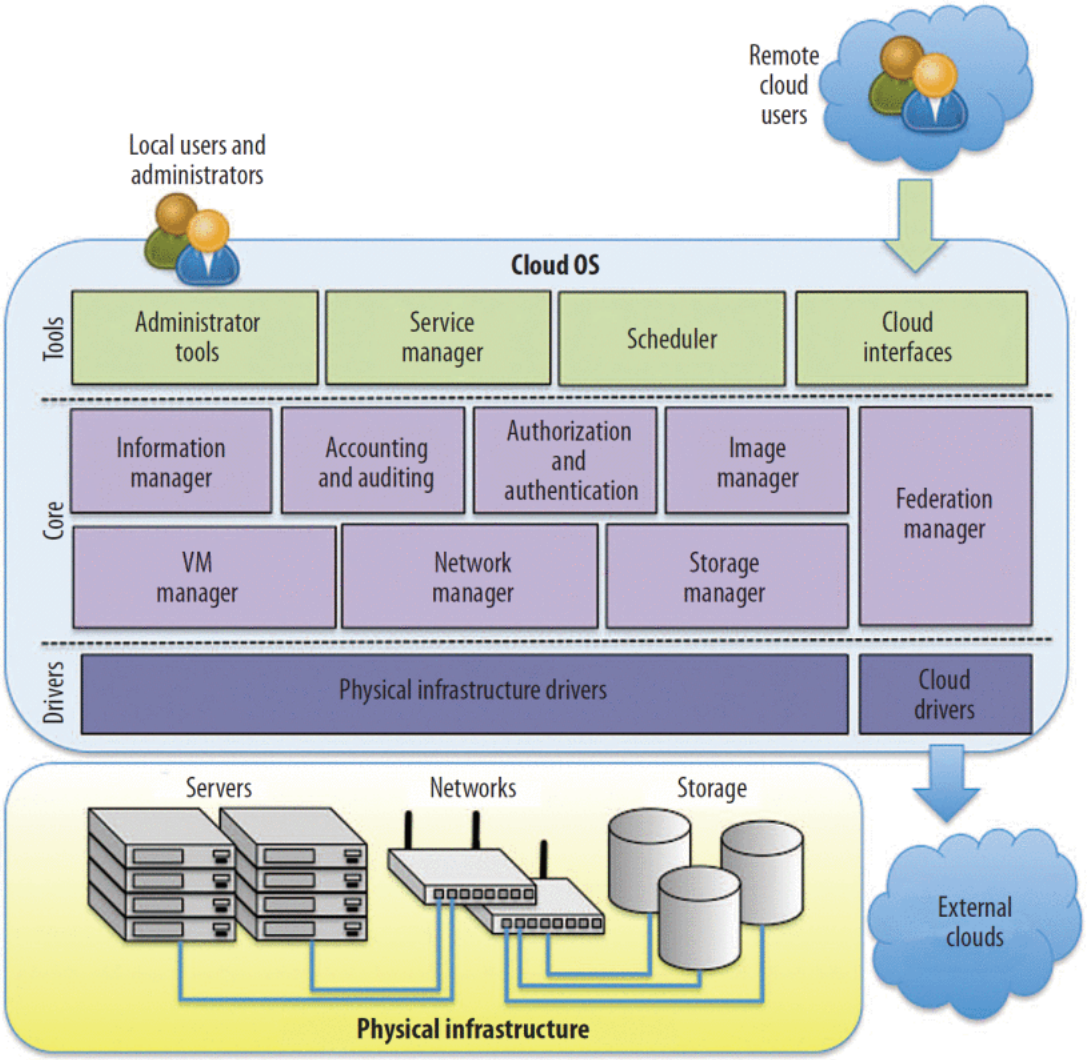


Figure 4. IaaS cloud architecture with its three layers: drivers, core components, and high-level tools [21]

#### 2.3.4.2. PaaS

The PaaS is built on top of the IaaS. In PaaS, the vendors provide the users with an environment/platform where they can develop their applications. This includes providing the operating system, deployment tools, programming languages, and the application program development tools[18]. Some examples of PaaS include Windows Azure by Microsoft, IBM's SmartCloud, Google App Engine, etc.

#### 2.3.4.3. SaaS

In SaaS, the vendors build applications that run on the IaaS or on their own servers. These applications are then provided to the customers on a pay-per-use basis. The customers use this application software simultaneously by connecting with it through the internet. For instance, the email applications provided by various companies like Google, Yahoo, etc. are one such example. Several other examples include third party customer relationship management software on the cloud, etc. [18].

## 2.4. Bare Metal Provisioning

Bare metal provisioning refers to allocating the entire server hardware to the user/organization/application. *"Consequently, applications can run natively on the host and fully utilize the underlying hardware. However, this is a single tenant option as unused hardware resources cannot be shared or re-used by others within the data center"* [22]. Bare metal provisioning does not allow any form of virtualization, and the hardware remains fully dedicated to the user/application. Furthermore, since the provisioning of a server as bare metal allows the direct installation of an operating system, it is specifically advantageous for running latency sensitive tasks that require significant processing power. Moreover, since the user is the only tenant

on the bare metal server, it gives the user better control over the resources. However, the single tenancy of the bare metal resources often leads to decreased overall utilization rates of the hardware, and thus, it is often not cost effective [22].

## 2.5. Conclusion

In this chapter, we discussed all the essential terms and concepts crucial to this thesis. We began by providing an overview of the Internet of Things (IoT), its enabling technologies and application areas. We then described virtualization and covered its definition, types and techniques. We also discussed briefly the techniques to virtualize IoT devices. Next, we discussed Cloud Computing, its characteristics and service models with special focus on IaaS. Lastly, we provided a brief overview of Bare Metal provisioning before concluding the chapter.

The next chapter presents the motivating scenario and also lays down the set of requirements essential to the IoT IaaS architecture. The state of the art is also reviewed and assessed against these requirements.

# Chapter 3

# Use Case and State of the Art

In this chapter we first provide a motivating use case for the IoT IaaS. The requirements of the IoT IaaS are then derived with the help of this use case. Finally, the chapter focuses on evaluating the current state of the arts against these derived requirements and obtaining conclusive results.

## 3.1. Use Case

The use case considered in this thesis is a 'Smart Factory' scenario. The goal is to make product manufacturing factories, such as pharmaceutical manufacturing factories, or cellular devices manufacturing factories smarter and more efficient. Applications of IoT can enhance the functioning of these factories by improving their performance, making them more cost and energy efficient, and also enhance their risk/hazard handling systems.

In particular, the case of Pharmaceutical factories is considered. The pharmaceutical industry is essential in enhancing the health care of any country and, thus, making it more efficient is necessary. Making the factory smart would allow automation of the manufacturing process, as well as better monitoring and control over the equipment. The idea is to use IoT technologies to enhance the performance of the pharmaceutical factories in a number of ways, for instance, by monitoring the performance of the equipment, by efficiently tracking the products, by ensuring the proper storage of cold chain products, by improving the efficiency of fire and hazard handling systems, by enhancing the security systems etc.

In order to improve the pharmaceutical factory's performance, as well as handle risks in a better and efficient manner, several IoT applications, along with the required infrastructure (such as

sensors and actuators) can be deployed into factories. The Smart Factory Scenario is depicted in figure 5. The following types of IoT applications can be utilized for this purpose:

### 3.1.1. Monitoring of Cooling Systems

Pharmaceutical factories always contain or manufacture certain cold chain products. These products are those that require very low or refrigerated temperatures in order to remain usable. These products could include certain temperature sensitive medicines, vaccines, etc. In order to enhance the working of these cooling systems and make them more efficient, IoT sensors can be deployed to monitor the cooling systems in areas where these products are kept. This would allow continuous monitoring of these spaces and immediate control of the cooling system without much human interference. This application could require IoT devices for temperature sensing, humidity sensing, and a mechanism to control the HVAC systems. For example, the sensors could constantly detect the temperature and humidity levels within the specific area where environment sensitive products are kept, and when the environment conditions in the area become unsuitable for the products, the HVAC systems are immediately controlled to mitigate the situation.

### 3.1.2. Anti-Fire Systems

Pharmaceutical factories contain certain flammable items and materials such as acetone, plastic powder dust etc. Due to the presence of these substances it is essential to have an efficient Anti-Fire System in place to allow early detection and proper mitigation of these situations. The Anti-Fire Systems application could require temperature sensing, humidity sensing, smoke detection, firefighting robots. For example, the sensors can be utilized to keep checking for potential fire hazards or detect fires. When a fire is detected, firefighting robots can be immediately dispatched to extinguish the fire.

### 3.1.3. Items Tracking Systems

It is important to keep track of all the products and items present in the factory, where they are kept, where they have been moved to etc. This is essential to avoid misplacing or losing the products, since these products could incur significant costs to the pharmaceutical factory. Real-time tracking of these products within the factory is thus necessary. This Items Tracking application could require RFIDs (tags and readers), and access to databases. For example, the location of the items in the factory can be tracked by RFID tags in real time.

### 3.1.4. Inventory Management Systems

In a factory, it is essential to have an effective inventory management system. Inventory management includes updating the databases with the quantities of each item, most up-to-date information on each product, etc. In pharmaceutical factories, this is essential too. Having an efficient inventory management system will allow the factory to know exactly which products are sufficiently available, and which need to be produced or ordered as they are low in stock. This will, in turn, allow cost-savings as no excess products will be produced/ordered since the most up-to-date information will be available on each item. This application would require RFIDs (tags and readers), database access and management. For example, the RFID tags associated with each item will contain information on that item, which can be updated or modified. These tags will then allow the databases to have the most up-to-date information on each item in the inventory.

### 3.1.5. Smart Security Systems

An efficient security system is essential in any factory to ensure that unauthorized personnel cannot access the factory, as well as to ensure no thefts, etc. IoT Applications can be utilized to enable a smart and efficient security system that can detect such unauthorized access and

immediately issue alerts, notify the respective authorities, etc. Especially in the pharmaceutical factories, where expensive and specialty drugs/vaccines etc. are prepared, which may not be available to the general public or have limited distribution, the security of these products is essential. Thus, a smart security system is essential in the pharmaceutical factories. The Smart Security Systems application could require motion sensors, alarms, and other devices for issuing alerts or notifying authorities. For example, motion sensor-alarm systems can be deployed. The motion sensors can be activated when the factory operating hours end. If any unauthorized motion is detected, alarm modules can be triggered, and alerts can be sent to the respective authorities.

### 3.1.6. Smart Energy Systems

In order to reduce the costs of the factories, energy efficient systems need to be enabled. In pharmaceutical factories, there are several equipment that are used for preparing medicines and other products, which consume a lot of energy and incur a lot of costs to the factory. Moreover, lots of workers work in these factories and, thus, proper lighting and ventilating systems are also in place, which incurs additional costs. Thus, a smart energy system is needed that can detect when one section of the factory is not being used and automatically turn of the lights, ventilation, and other non-essential electric equipment in that area. This will not only reduce overall costs but will also allow the factories to be more environment friendly and consume less energy. This Smart Energy Systems application would require motion sensors, access to systems for controlling electricity, etc. For example, the motion sensors at the factories can detect motion in the areas of the factories where workers are present. When no motion is detected, i.e. no worker is present, a signal can be sent to turn off the lights, and other non-essential electronic equipment in that area. This can allow lower energy consumption as well as lower electricity costs.

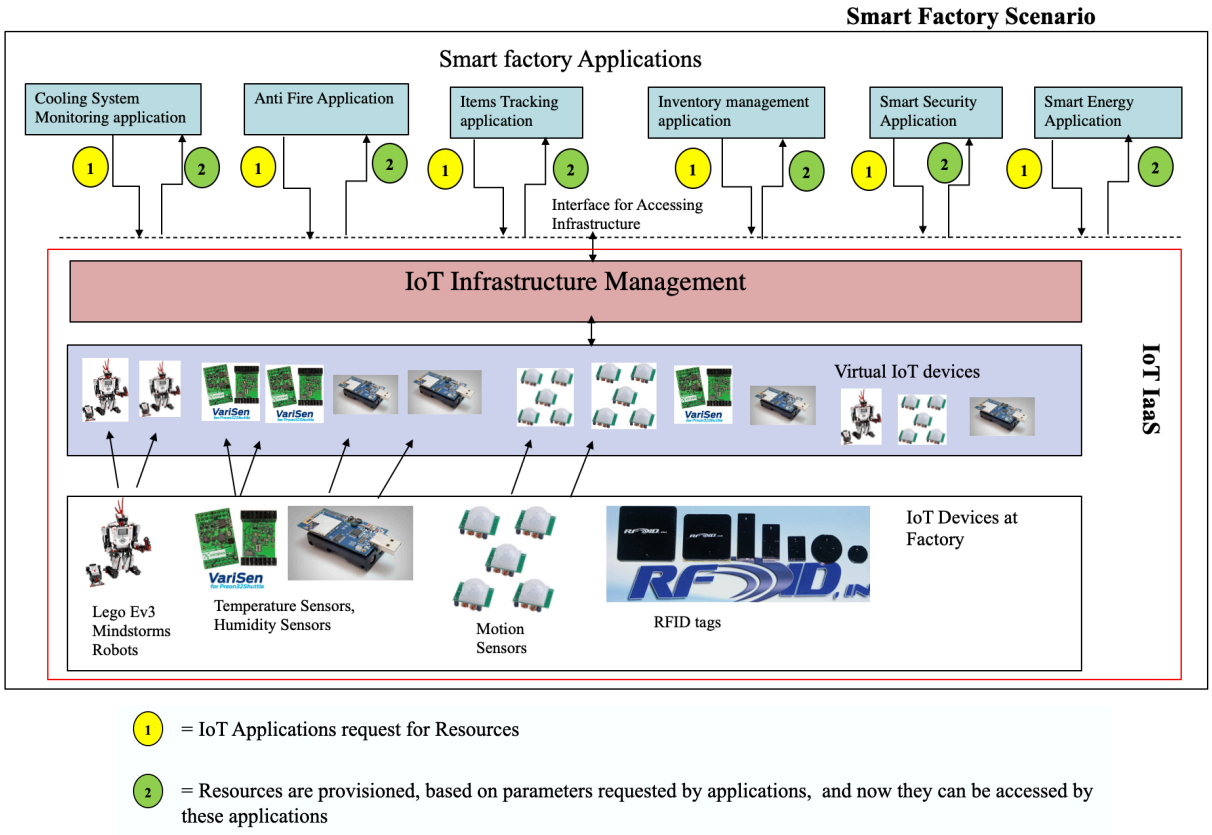


Figure 5. Motivating Scenario: Smart Factory use case

## 3.2. Requirements

**The first requirement that our IoT IaaS must fulfill is node level virtualization of the IoT devices in order to allow better and more efficient utilization of the resources.**

Node level virtualization will prove beneficial in our use case as well, since most of the applications in our 'Smart Factory' use-case require IoT devices that are common (sensors, actuators). To increase the efficiency of the factory and to reduce the costs spent on the infrastructure, it is essential that many applications share the same physical IoT devices. The applications can send requests for virtualizations of the physical IoT devices and then use these virtual devices. For instance, the 'Anti Fire Systems' application and the 'Monitoring of Cooling Systems' application can share the temperature and humidity sensors, since both the applications

involve detecting the temperature and humidity of the environment after certain time intervals in order to take specific actions. Similarly, the motion sensors can be shared by the 'Smart Security Systems' and the 'Smart Energy Systems' application. The 'Smart Security Systems' can use these motion sensors to identify motion after factory operating hours in order to detect unauthorized access, while the 'Smart Energy Systems' can use these motion sensors to detect motion in the specific factory areas and thus, turn the lights and equipment off when no motion is detected for a considerable period. Node level virtualization can enable this sharing, as the physical IoT devices will be virtualized for each application depending on the parameters specified by the application, and the application can then use the specific virtualization created for it. These virtualizations on the IoT devices are run simultaneously. Thus, these applications should be able to use the particular IoT sensors or actuators concurrently, and in order to support this scenario the IoT IaaS must support node level virtualization.

**The second requirement of the IaaS would be to have a publish and discovery mechanism that can allow the IoT devices' capabilities to be stored and queried as needed.**

As is evident from the Smart Factory scenario, in order to efficiently handle all the deployed infrastructure and use it efficiently, there is a need for a Repository which will contain information about all the physical devices available in the IaaS. This is essential in order to create virtualizations on, or reserve devices that can fulfill the requirements of the applications requesting the resources. For instance, if the IaaS receives requests from the applications in the Smart Factory, it must be able to search for an appropriate physical device which can meet the needs of application, and further reserve it or create virtualizations on it. In order to achieve this, the information about the capabilities of each device can be published in a database, i.e. Repository,

from where the IaaS can easily discover this device. Thus, our second requirement is necessary for the IoT IaaS.

**The third requirement of the IaaS is to have an orchestration mechanism to allow the orchestration of different device capabilities.**

This is critical for our use case as well, since some applications in the Smart Factory scenario require the services of several IoT devices for their functioning. An orchestration mechanism needs to be in place to allow an application to get access to several devices as needed. Moreover, this orchestration mechanism is essential to ensure that the various IoT devices work together as one unit to achieve the task of the IoT application.

For example, the 'Anti-Fire Systems' application requires the temperature sensor and humidity sensor to sense potential fire hazards. It also needs access to the robot which will be dispatched to extinguish fires in case the temperature falls outside a threshold. To allow the 'Anti-Fire Systems' application to use all of these devices, an orchestration mechanism is needed, which can allow for the creation of virtualizations for each of these devices and orchestrate them to be used by the application.

Similarly, the 'Monitoring and Cooling Systems' application requires temperature sensing, humidity sensing, and access to the HVAC system in order to control it. Thus, the orchestrator would need to virtualize the temperature sensor and humidity sensor to be used by the application. The orchestrator would also need to provide the application with the HVAC control actuator. All of these services would need to be orchestrated and given to the application to be used in conjunction.

Another example is the 'Smart Security Systems' application, which would need access to the motion sensor and the alarm module. Thus, the orchestrator would again need to orchestrate these

two services and provide it to this application. All these applications would not be able to function without the provision of an orchestration mechanism within the IaaS.

**The fourth requirement that our IoT IaaS must fulfill would be to have bare metal access to the IoT devices.**

At times it is possible that an application might need hardware that remains fully dedicated to the specific application and is not virtualized and shared with any other application. In such a situation, this requirement becomes critical for the IoT IaaS. For instance, in our 'Smart Factory' scenario, the 'Anti-Fire Systems' application would require exclusive access and complete control over the fire-fighting robots in order to immediately dispatch them during hazardous situations. However, if these robots are being utilized by other applications, then the 'Anti-Fire Systems' application might not be able to utilize the full capabilities of the robot, or the robots might not perform efficiently to mitigate the fire hazard since other applications are also using them simultaneously. Hence, in such a scenario, there needs to be a mechanism to provision the physical devices as bare metal, thus allowing the application to use the device as-is and have complete control over it. Moreover, since the physical device is used as is without any middleware for virtualization, bare metal provisioning poses an additional advantage of supporting tasks that are latency sensitive. Thus, the 'Anti-Fire Systems' application, which is a latency sensitive application, would benefit from this bare metal provisioning since the robots will be dispatched immediately without significant delay.

**Finally, the fifth and last requirement of the IaaS would be the ability to control and use actuators as needed by the application.**

This requirement becomes critical for the IoT IaaS when applications request for actuation capabilities from it. For instance, the applications within the Smart Factory scenario require both

sensing as well as actuation capabilities. The 'Anti-Fire Systems' application would require firefighting robots, which would need to be dispatched in case of fire hazards. Similarly, the 'Monitoring and Cooling Systems' application would require access to an actuator to control the HVAC system in situations when the temperatures increase beyond the specified threshold in the cooling areas. Therefore, it is essential that besides providing sensing capabilities, the IaaS also provide actuation capabilities to allow the applications to perform certain controlling/actuating tasks.

## 3.3. State of the Art

In this subsection, the current state of the art is analysed and evaluated against our proposed requirements. For this purpose, the state of the art is divided into two categories. First the state of the art involving complete architectures for the IoT IaaS is analyzed and summarized. This is followed by an analysis and summary of the state of the art involving models and frameworks that can aid the IoT IaaS.

### 3.3.1. Architectures for IoT IaaS

There currently exist very few architectures for the IoT IaaS that have been analyzed in this subsection. Each of these works is first discussed in detail and then evaluated against our set of derived requirements.

In the work titled *"Cloud Based IoT Network Virtualization for Supporting Dynamic Connectivity among Connected Devices"*, Ullah *et al.* [23] propose a concept for building a dynamic virtualized IoT network over the cloud environment. These IoT devices can belong to different domains, are interconnected, and their virtual objects are utilized for the creation of this network over the cloud. Figure 6 shows the architecture proposed in this work. To begin the

process, the IoT devices first register themselves by posting their profile information to a pre-configured virtualization server. To do this, a registration request is sent by the devices. Next, the virtualized objects are created for these registered IoT devices which are stored in this virtualization server. Whenever a user sends a request based on certain criteria and with the desired settings, this request is sent to the *Controller* component present within the *Virtualization Layer* of the architecture. This *Controller* is responsible for retrieving the virtual objects from the virtualization server that match the criteria specified in the users' requests, as well as manipulating these virtual objects to obtain the required network settings. To obtain data from the virtual objects an activation command can be sent to them. Moreover, a mapping list is maintained that can be used to retrieve the mapped virtual object to which an actuation command can be sent. The paper presents three use cases: Automatic Door Opening Application, Fire Safety Application, and Indoor Environment Application. OMNeT++ is used for simulating the virtualization networks, which include the client, server, and the IoT device nodes. A local gateway node is also present in this simulation that allows the IoT device nodes to be connected to the virtualization server.

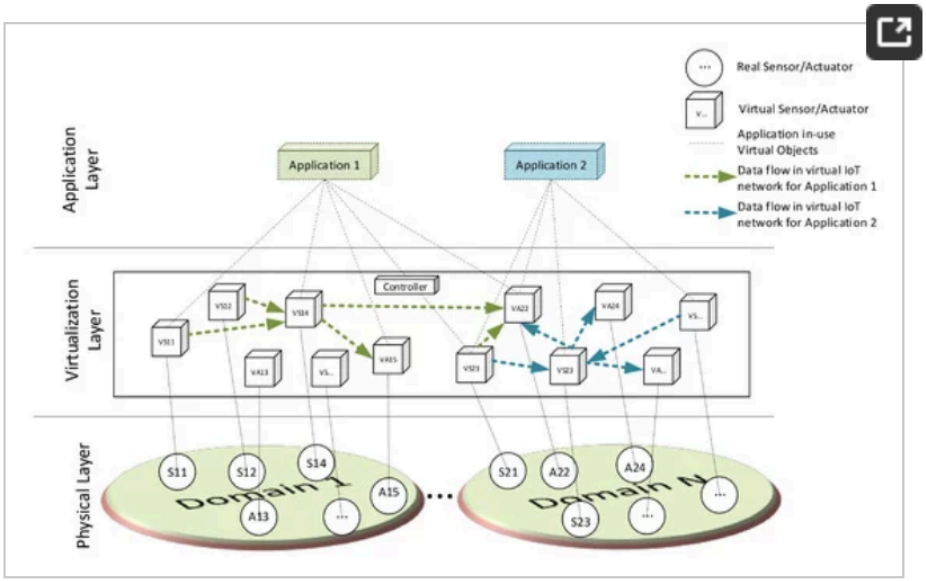


Figure 6. Layered architecture of the proposed IoT network virtualization [23]

In addition, there is also a dedicated interface that permits the applications to express the number of IoT devices they require, along with the specifications and settings of these devices. However, there is no mention of any high-level APIs for interacting with the gateway node or any sort of cloud access interface.

On evaluating the architecture proposed in this work against our derived requirements, the following points were observed:

1. **Node Level Virtualization:** In this work, for every IoT device, a corresponding virtual object is created. Moreover, a concept is proposed to build a dynamic network of these virtual objects, based on the needs of the application. The virtual networks on top of the physical IoT devices are then utilized by the application. This implies that this work makes use of network level virtualization, and not node level virtualization. Thus, our first requirement of node level virtualization is not met.
2. **Publication and Discovery Mechanism:** In this work, since initially each IoT device registers and posts its specification by sending the server a registration request, and the *Controller* can later discover these device profiles, the second requirement of a 'Publication and Discovery mechanism for IoT devices' is met.
3. **Orchestration Mechanism:** The *Controller* module, proposed in the architecture specified in this work, is responsible for orchestrating the various virtual objects to dynamically form a network based on the application's specifications. Thus, the third requirement for an orchestration mechanism is met by this work.
4. **Bare metal access to IoT devices:** The main focus of this work is to create a virtual network using the virtual objects of the IoT devices. There is no mention of any provision

to access the physical IoT devices as bare metal. Hence, the fourth requirement of having bare metal access to IoT devices is not met.

5. **Ability to control and use Actuators:** The proposed work highlights the ability to control virtualized actuators in a similar manner to the virtual sensors. Moreover, the *Controller* module contains a set of simple rules to determine whether to send the actuation command to the actuator or not based on the obtained sensor output. Hence, the fifth requirement pertaining to the ability to control and use actuators is met in this work.

Guerreiro *et al.* [24] present a resource allocation model for assigning sensor and cloud resources to the clients/applications. The Sensing as a Service (Se-aaS) paradigm used in this work allows multi-client access to these sensor resources as well as multi-supplier deployment. The work also proposes a heuristic algorithm based on Se-aaS.

As shown in figure 7, the registered physical sensors are virtualized to allow management and customization of the IoT devices according to the needs of the applications/clients/consumers. Many virtual sensors can be grouped together to achieve the applications'/clients' needs.

One of the main contributions of this work is to enable the software components to have bindings to mashups managed in the cloud. Each of these mashups consists of a workflow that combines one or more devices. This allows the events to be processed and actuation commands to be triggered in the cloud, based on the workflows of the mashup. Only the final data is then delivered to the application. The sensors and data are shared by the clients/applications through the dedicated instances of each distinct type of sensor.

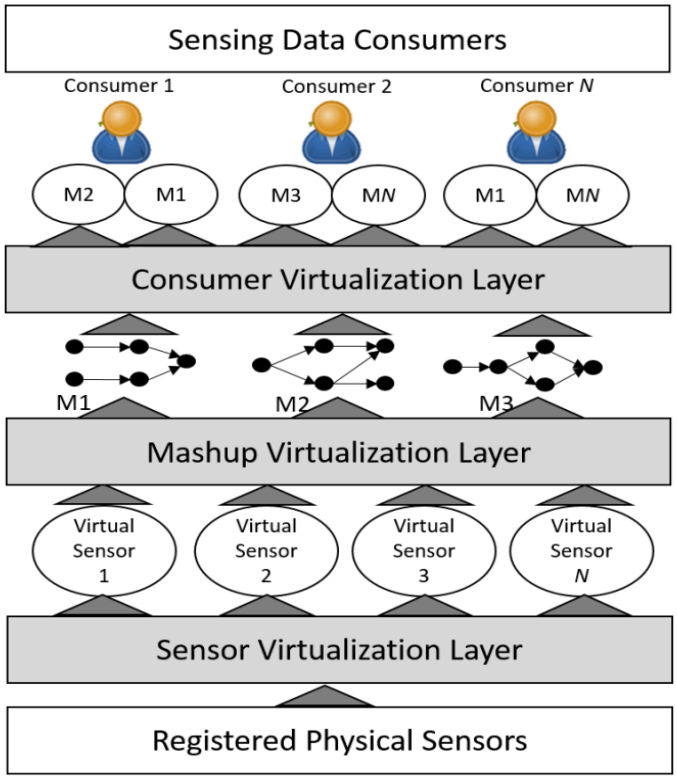


Figure 7. Virtualization layers in Se-aaS [24]

The proposed resource allocation model was evaluated against our requirements and the following points were observed:

1. **Node Level Virtualization:** Since the resource allocation model allows several applications/clients to share the same sensor devices (through their virtualizations), the first requirement of node level virtualization is met by this work.
2. **Publication and Discovery Mechanism:** In this work, there is no mention of any kind of registry service for publishing the capabilities of the IoT devices or any discovery service. It is assumed that the physical devices are registered in the cloud. Hence the second requirement of a publication and discovery mechanism is not met by this work.
3. **Orchestration Mechanism:** The 'mashups' seen in the architecture of this work somewhat mimic the role of an orchestrator. The mashups (workflows), for instance, allow for

actuations to be triggered based on the outputs of certain virtual sensors. This allows for a proper mechanism to orchestrate (mashup) these virtual devices and only deliver the final data of interest to the applications/clients. Thus, the third requirement of an orchestration mechanism is met by this work.

4. **Bare metal access to IoT devices:** The work only proposes virtualizing all the registered physical sensors to be used by applications/clients through the mashups in the cloud. However, there is no mention of any mechanism to allow these IoT devices to be used as bare metal by these applications/clients. Thus, the fourth requirement of having bare metal access to the IoT devices is not met in this work.
5. **Ability to control and use actuators:** The architecture proposed in this work is Se-aaS (Sensor as a Service). Its main focus is to virtualize the registered physical sensors for usage. Even though the 'mashups' proposed in the work do mention triggering actuations, there is no mechanism mentioned to virtualize the physical actuators in a manner similar to the physical sensors. Hence the fifth requirement for handling and using actuators is not met.

Atzori *et al.* [25] in their work titled "*SDN&NFV contribution to IoT objects virtualization",* aim to provide IoT devices "as a Service" i.e. "Smart Devices as-a-Service" (SDaaS) to the users through virtual images, similar to [24]. However, the main aim of this work is to design a novel infrastructure and paradigm to enable the "*deployment of new personal IoT services inside the infrastructure provider premises*" [25]. The idea is to provide the users with the IoT services through the virtual objects of the IoT devices present at the service provider's end, instead of physically placing the IoT devices in the users' homes/premises, such as network-attached storage servers, sensors, set-top boxes etc. The design proposed in this work utilizes Network Functions

Virtualization (NFV) and Software Defined Networking (SDN) in order to create virtual overlay networks for each user. These virtual overlay networks allow the users to connect to the virtual images of the IoT devices required for the services they need and provides the level of isolation and security similar to LAN (Local Area Network). On evaluating this work with our requirements, the following were observed:

1. **Node Level Virtualization:** In this work, a virtual overlay network is associated with each user. This virtual network is deployed on top of the network of physical IoT devices. Moreover, each physical IoT device is associated with only one virtual object, and there is no mention of any possibility of running several virtualizations of the same physical device simultaneously. Thus, several applications cannot run concurrently on this physical device. Thus, it can be said that the work proposed by Atzori *et al.* [25] does not satisfy our first requirement of node level virtualization.
2. **Publication and Discovery Mechanism:** The proposed work does not explain any mechanism for publication and discovery of the physical IoT devices' specifications. Although a 'Discover' operation is mentioned in the LwM2M enabler interface, it isn't explained in any further detail. Hence the second requirement of a publication and discovery mechanism is not satisfied.
3. **Orchestration Mechanism:** The solution proposed in this work supports the grouping together of several Virtual Objects in a PaaS instance to create a virtual overlay network depending on the needs of each user. Hence, a mechanism for the orchestration of several services is present and our third requirement is satisfied.
4. **Bare metal access to IoT devices:** One of the main features of this work is to provide added security by not allowing direct access to the physical IoT devices. Instead, only the

virtual objects of these devices are utilized in the overlay networks. Each user makes use of the virtual overlay network created as per their needs, and the sensors/actuators cannot be accessed for specific functions directly by any external entity. Hence the fourth requirement of having bare metal access to the physical IoT devices is not met.

5. **Ability to control and use actuators:** The proposed system includes virtualization of IoT services that includes all its associated sensors and actuators. Hence actuator control is carried out by the system in a manner similar to sensor control and control of other IoT devices. Thus, the fifth requirement of controlling and using actuators is met by this work.

In [26] Alam *et al.* propose a full-fledged architecture for an IaaS for the Internet of Things, which can permit the low-cost provisioning of IoT applications as well as their decoupling from the underlying physical devices. In addition, the work also proposes high-level APIs that allow the applications and users to interact with the IoT IaaS and utilize the virtualized resources as per their needs. Low level APIs are also proposed for the management of the physical IoT devices. The work stresses on sharing the capabilities of the physical devices via node level virtualization and depicts this through an Anti-Fire application and a Smart HVAC application as use cases. The work proposes an architecture that allows for more efficient resource utilization. The architecture proposed in the work is shown in figure 8.

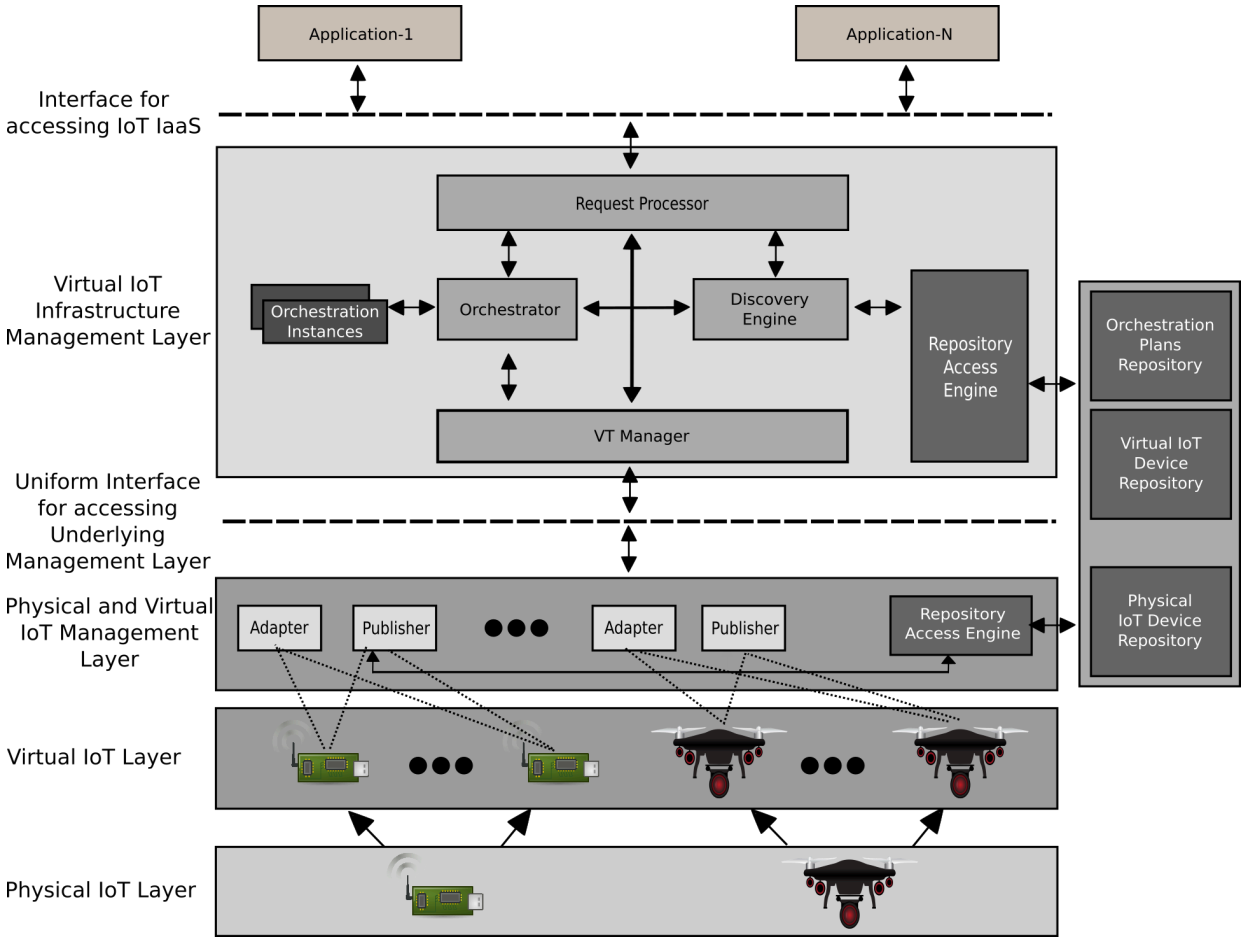


Figure 8. The Architecture of the IoT IaaS [26]

In the proposed architecture, the *Physical IoT Layer* contains all the physical IoT devices (sensors, actuators etc.), while the *Virtual IoT Layer* consists of a logical representation of the virtualized IoT devices. The *Physical* and *Virtual IoT Management Layer* allow the underlying heterogenous devices to be managed and used in a homogenous manner. Furthermore, the *Virtual IoT Infrastructure Management Layer* manages the virtual IoT infrastructure and is responsible for providing the applications with the required virtual resources as per their needs. It makes use of the uniform interface to interact with and control the virtual IoT resources. On evaluating the work against our requirements, the following points can be observed:

1. **Node Level Virtualization:** The proposed architecture utilizes node-level virtualization to allow several applications to simultaneously run on the physical IoT devices through their virtualizations. Thus, this work meets our first requirement.
2. **Publication and Discovery Mechanism:** The proposed architecture contains a *Publisher* module, which is responsible for publishing the information of each device into the repository, as well as a *Discovery Engine* to discover the devices with the capabilities required by the application. Thus, this work meets our second requirement of a mechanism for publication and discovery of the devices' services.
3. **Orchestration Mechanism:** The authors describe a detailed mechanism for the orchestration of several services in this work. The architecture contains an *Orchestrator* which is responsible for this task. The Orchestration Plans are further stored in the repository for usage later on. Thus, this work fulfills the third requirement for an orchestration mechanism.
4. **Bare metal access to IoT devices:** The work only focuses on creating virtual instances of the physical IoT devices and provides no mechanism to have bare metal access to a physical IoT device. Thus, this work does not meet our fourth requirement of having bare metal access to an IoT device.
5. **Ability to control and use actuators:** Although the work mentions orchestration plans consisting of actuation tasks, it does not provide any detailed mechanism for the provision of, or usage of actuators. The work focuses primarily on sensor devices. Thus, this work does not meet our fifth requirement of the ability to control and use actuators.

Khan *et al.* propose a novel architecture for WSN Virtualization in [27], which is shown in figure 9. The work focuses on tackling the issue of redundant WSN deployment by using

virtualization techniques. The architecture makes use of node level as well as network level virtualization to enhance cost and resource efficiency. In this work, the architecture allows a single WSN to be shared by several applications. The architecture makes use of the constrained application protocol and consists of four layers, as shown in the figure 9.

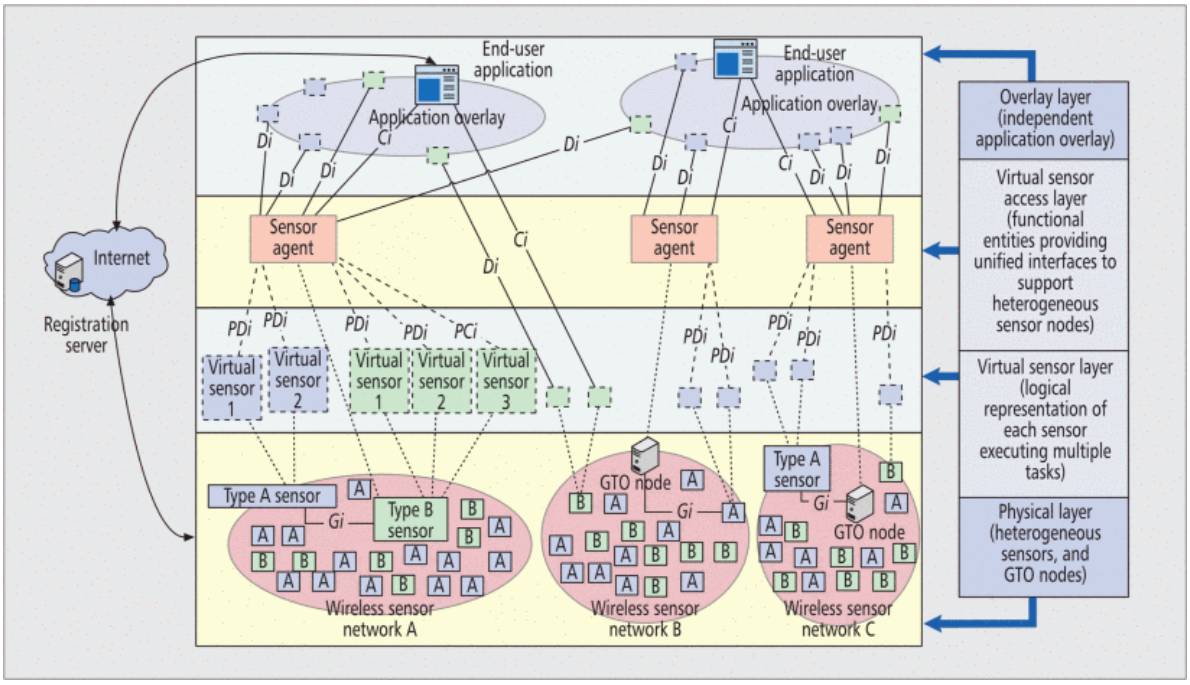


Figure 9. Multi-layer WSN virtualization architecture [27]

The *Physical Layer* contains various independent WSNs, while the *Virtual Sensor Layer* consists of a logical representation of each of the physical sensors. The *Virtual Sensor Access Layer* contains sensor agents that control and retrieve data from the virtual sensors and interact accordingly with the overlays, thus, ensuring platform independence. The last layer, the *Overlay Layer* contains the different overlay networks created on the basis of the specifications of each application. The work is further evaluated against our proposed requirements and the results are summarized below.

1. **Node Level Virtualization:** The architecture proposed in this work allows the physical sensor nodes to simultaneously execute several tasks by allowing their virtualizations to be a part of several application specific overlay networks at a given time. Thus, this work meets our first requirement of node level virtualization.
2. **Publication and Discovery Mechanism:** Although the proposed architecture contains a discovery service for finding the appropriate devices, it assumes that the sensor and GTO owners have already published their nodes to a central repository. Hence, a dynamic mechanism does not exist for the publication and discovery of the devices' services, and thus, this work does not fully satisfy our second requirement of a publication and discovery mechanism for the devices' services.
3. **Orchestration Mechanism:** Although the overlay networks for each application contain several virtual sensors for fulling the application's tasks, there is no mechanism described in this work for orchestrating the services of the different IoT devices. The overlays are simply a network of several different virtual devices and do not depict these devices as one orchestrated virtual device for the application. Thus, this work does not meet our third requirement of an orchestration mechanism for orchestrating the services of several IoT devices together as needed.
4. **Bare metal access to IoT devices:** The architecture does not provide any mechanism for accessing the physical devices as bare metal, and thus, it does not meet our fourth requirement of having bare metal access to the devices.
5. **Ability to control and use actuators:** The architecture primarily focuses on sensors as physical devices. It does not provide any mechanism for controlling or using actuators, which implies that our fifth requirement is not satisfied by this work.

### 3.3.2. Summary of the State of the Art of Architectures for IoT IaaS

Table 1 summarizes the state of the art of the architectures for the IoT IaaS against the evaluation of our requirements. In this table a '✓' means that the requirement is met by the particular work, whereas a '×' means that the requirement is not met by the work.

Table 1. Summary of the Related Works involving Architectures for the IoT IaaS

| Papers | Requirements | | | | |
|---|---|---|---|---|---|
| | **Node Level Virtualization** | **Publication and Discovery Mechanism** | **Orchestration Mechanism** | **Bare metal access to IoT devices** | **Ability to control and use actuators** |
| Ullah *et al.* [23] | × | ✓ | ✓ | × | ✓ |
| Guerreiro *et al.* [24] | ✓ | × | ✓ | × | × |
| Atzori *et al.* [25] | × | × | ✓ | × | ✓ |
| Alam *et al.* [26] | ✓ | ✓ | ✓ | × | × |
| Khan *et al.* [27] | ✓ | × | × | × | × |

### 3.3.3. Models and Frameworks for aiding the IoT IaaS

There are few works in the current state of the art that propose frameworks or service models that can aid in the development of an IoT IaaS by overcoming the issues that it currently faces. In this section, some of these current works have been analyzed. First, each of the works is discussed in detail followed by an evaluation against our set of derived requirements.

In [28] and [29], Gupta *et al.* propose and provide an implementation of virtual sensors at the IaaS level respectively. The implementation in [29] focusses on representing the physical sensors as virtual sensors and provides a soft sensor API to handle these sensor objects at the IaaS level.

The aim is to enable on-demand, pervasive, and shared access to these physical sensors through their abstractions (virtual sensors). The authors propose a distributed architecture for the virtual sensor system abstraction. This architecture consists of a total of 5 layers: *Physical Sensor Devices* layer, *Sensing System* layer, *Processing System* layer, *Storage System* layer, and *Communication System* layer. There is also a *Monitoring System* in the architecture. Basic APIs are also proposed to facilitate interaction between the layers as well as to expose the virtual sensor system abstraction to the cloud. The *Physical Sensor* layer contains the physical sensor devices, while the *Sensing System* acquires the sensor data, converts it into the appropriate form and analyzes it. The *Sensing System* contains appropriate APIs for collecting and interpreting the sensor data. The *Processing System* is responsible for modifying the sensor data to suit the needs of the user. It contains APIs to efficiently handle the individual functions and allows the processing tasks to be executed in the cloud. It contains APIs capable of terminating or creating virtual sensors, reading and writing to sensor objects, etc. *Storage System* is responsible for handling the storage constraints of the physical sensors by providing virtual sensor storage in the cloud. It has APIs for providing specifications such as storage types, i.e. heap, database, etc., allocation and deallocation of memory space, as well as a handler to access the stored data. The last layer is the *Communication System* which is responsible for transmitting the data received from the physical sensors to the appropriate destinations, as well as for receiving any data sent to the virtual sensor objects. It handles tasks such as protocol specification for effective communication and identifying the sensor data source. It consists of appropriate APIs to achieve these tasks. The *Monitoring System* is responsible for providing error free sensor data by interacting with user and physical sensors to ensure that the data is correct and is provided in an uninterrupted manner. The architecture extracts the data from the physical sensors and provides it to the virtual sensors. Modifications are

performed on these readings directly fetched from the sensor based on the needs of the users. The work proposed in this paper is further analyzed against our proposed requirements as follows:

1. **Node Level Virtualization:** The architecture proposed in this paper proposes virtual sensors that simply request the same sensor data, which is further modified at the higher layers to fit the user specified configurations of the virtual sensor. There is no provision in this architecture to run several applications concurrently on the same physical sensor node. Thus, the first requirement of node level virtualization is not met in this work.
2. **Publication and Discovery Mechanism:** The proposed work does not contain any mechanism for publishing the capabilities or discovering the capabilities of the physical sensors. In addition, there is no repository or database present in the architecture that can store the information of each device. Thus, the second requirement for a publication and discovery mechanism is also not met.
3. **Orchestration Mechanism:** In the proposed architecture in this work there is no mention of any orchestration mechanism for orchestrating several services provided by the physical sensors. The Virtual Sensor System abstraction architecture only allows for abstracting the sensor services from the underlying physical devices. The data fetched directly from the sensors is simply provided to the virtual sensors, which can be further utilized by the users directly, other applications, or cloud services. However, the services are not orchestrated together. Thus, our third requirement for an orchestration mechanism is not met by this work.
4. **Bare metal access to IoT devices:** The aim of this work is to provide a virtual sensor system for abstracting the services of the physical devices. There is no provision for

accessing the physical sensors as bare metal in this work. Thus, the fourth requirement of bare metal access to the physical IoT devices is not met.

5. **Ability to control and use actuators:** The architecture proposes abstraction for virtual sensors and provide no mechanism for controlling or using actuators. Hence, this work does not meet our fifth requirement of the ability to control and use actuators.

In [30], Mattos *et al.* propose a network function virtualization infrastructure for the Internet of Things that is effective and agile. The paper mainly proposes the development of a gateway node that can effectively virtualize the domains to which the physical IoT devices connect. The proposed gateway in this work allows for the creation of virtual interfaces that behave as different virtual access points for different domains of the connected IoT devices. The physical resources/devices of the network are abstracted into virtual resources in the NFVI. Furthermore, the software-based virtual network functions are chained together in the virtual environment to provide the required network service. Moreover, outsourcing the network functions to the proposed virtualized infrastructure also allows for minimum load on the gateway. The gateway is, thus, mainly responsible for providing access to the IoT devices through the virtual interfaces. The entire NFVI along with the gateway and virtual interfaces is presented in figure 10.

The gateway and NFVI communicate via the GRE (Generic Routing Encapsulation) tunnel, and once the devices are associated, all the frames are forwarded to the NFVI through this tunnel. For the network virtualization infrastructure OPNFV (Open Platform for Network Function Virtualization) 4 is utilized, while the management of the virtualization layer is carried out through OpenStack5. Virtual network functions are implemented as virtual machines running Linux Ubuntu 16.04.

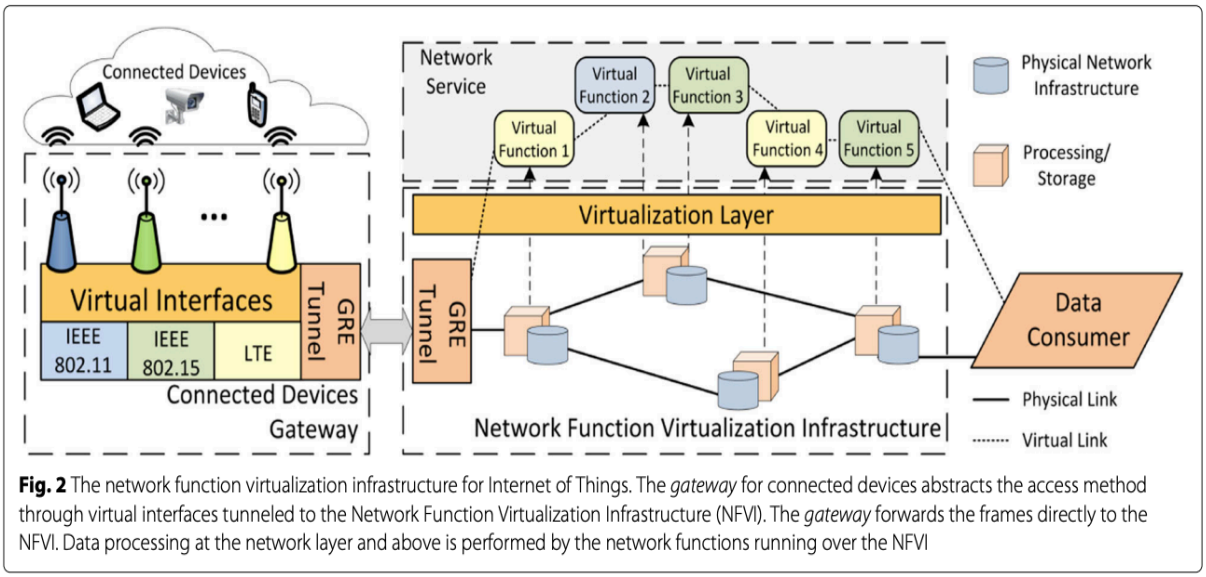


Figure 10. The complete architecture showing the NFVI for IoT and the connected devices Gateway. [30]

The proposed work is evaluated against our requirements and the results are summarized below.

1. **Node Level Virtualization:** In the proposed work the physical IoT devices are used and connected through the virtual interfaces. However, there is no mechanism mentioned for running several applications together on the same IoT device. The devices are a part of different virtual networks and the work does not address the possibility of a device being used in several virtual networks at the same time and concurrently running several applications on it. Thus, this work does not meet our first requirement of node level virtualization.
2. **Publication and Discovery Mechanism:** The proposed work does not mention any mechanism for publishing and discovering the capabilities of the different IoT devices. Hence, our second requirement of having a mechanism for publication and discovery of the IoT device services is not met.
3. **Orchestration Mechanism:** Although, in this work, there is a mention of orchestration of the various network services together, as such a proper orchestration mechanism has not

been described. Hence, the third requirement for an Orchestration Mechanism is not met by this work.

4. **Bare metal access to IoT devices:** This work only focuses on a gateway node that allows for the creation of virtual interfaces for connecting the physical IoT devices to NFVI. Thus, the physical devices are only accessible through these virtual interfaces and there is no means to provide bare metal access to these physical IoT devices. Thus, the fourth requirement is also not met in this work.
5. **Ability to control and use actuators:** The proposed work provides a gateway node to access IoT devices, such as both sensors and actuators. However, there is no specific mechanism mentioned pertaining to controlling and using actuators. Thus, the fifth requirement of having the ability to control and use actuators is not completely met by this work.

In [31], Mandal *et al.* propose a service model for IoT services, called 'Things as a Service' (TaaS) that allows the users to be exposed to the capabilities of the IoT devices, and efficient utilization of the IoT resources. The architecture framework for the TaaS is shown in figure 11.

The primary constituents of the proposed service model are the 'things', i.e. sensors and actuators. The service gateways present in the framework allow the IoT devices to expose their service interfaces and publish it onto the cloud, from where it can be accessed by users and applications. The cloud is responsible for maintaining a registry containing information of all these devices and providing the users with the appropriate services. It is also responsible for event and process management. The overall architecture also utilizes REST based services for the interaction between the various layers.

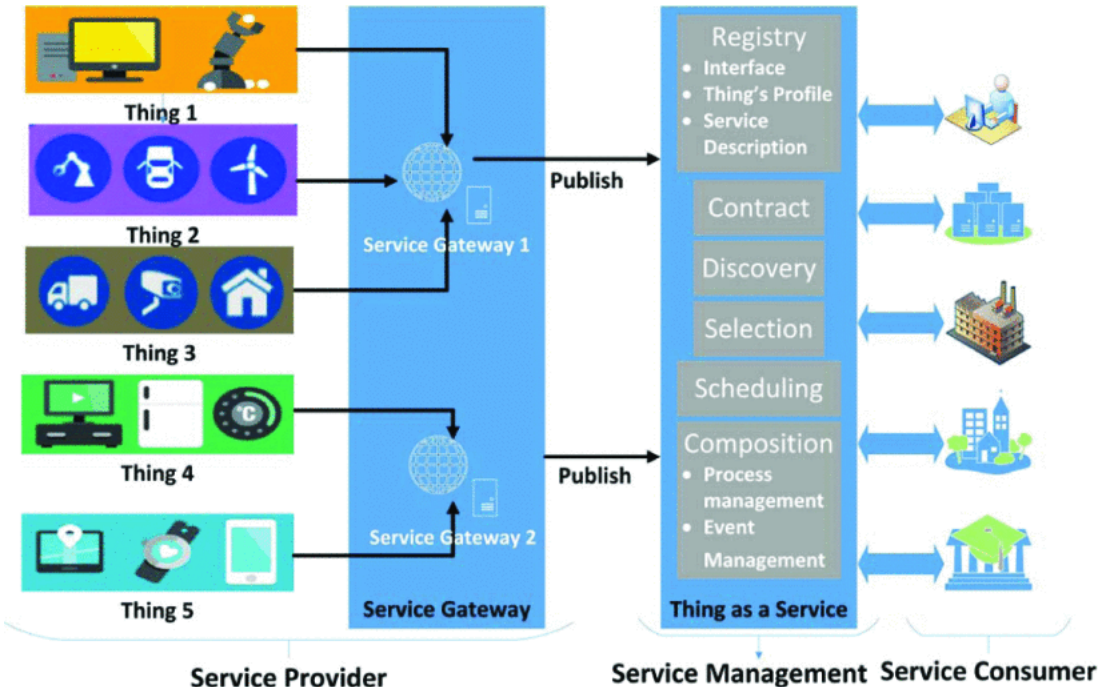


Figure 11. Architectural Framework for Things as a Service [31]

The proposed framework is further evaluated against our proposed requirements as follows:

1. **Node Level Virtualization:** The proposed framework simply exposes the services provided by the IoT devices to the cloud but does not provide any mechanism for running several applications concurrently on the IoT devices. The work is more 'data' driven, i.e. the data collected by the IoT devices is retrieved and provided to the applications as needed. Thus, our first requirement of node level virtualization is not met by this work.
2. **Publication and Discovery Mechanism:** The framework focuses on publishing the services of the IoT devices to the cloud registry. Moreover, the cloud is responsible for discovering the service needed by the application and providing it. Thus, there exists the implementation of a proper publication and discovery mechanism which fulfills our second requirement.

3. **Orchestration Mechanism:** In this work, there is no explicit definition of any kind of orchestration mechanism for combining the services of several devices as per the applications' needs. Thus, the third requirement for an orchestration mechanism is not met by this work.
4. **Bare metal access to IoT devices:** In this work, the services of the devices are published in the cloud and simply provided to the applications as needed. However, there is no mechanism described to allow an application to access an IoT device exclusively as bare metal. Thus, our fourth requirement for bare metal access to IoT devices is not met in this work.
5. **Ability to control and use actuators:** The framework proposed in this work involves handling actuators. It treats both the sensors and actuators as 'things' with certain capabilities, which are published onto the cloud. Further, the cloud layer is responsible for handling actuator events, i.e. it carries out event management. Thus, our fifth requirement of controlling and using actuators is met by this work.

K.P.S. *et al.* propose a software framework for WSN (Wireless Sensor Network) virtualization, i.e. virtualization of a network of IoT sensors, in [32]. *"This new Framework presents few conventions that help application developers to create applications without requiring it to understand underlying hardware and hardware developers can provide plug-and-play modules to the virtualization layer"* [32].

The framework proposed in this work is shown in figure 12. The lowest layer comprises of the various sensors. These devices first need to be registered to be utilized in this framework. There is a special registration server for this purpose in the framework. After registration the Middleware Layer is responsible for implementing each functionality that the hardware layer may require. It

also provides the respective APIs to access the functionalities of each sensor. Furthermore, there is also Session Management present in this Middleware Layer that allows management of each session such as data acquisition, actuation, etc. As we go up the layers in the framework, it becomes possible to provide application developers or other clients/platforms with high level Java APIs which they can use in their own applications and interact with the devices.

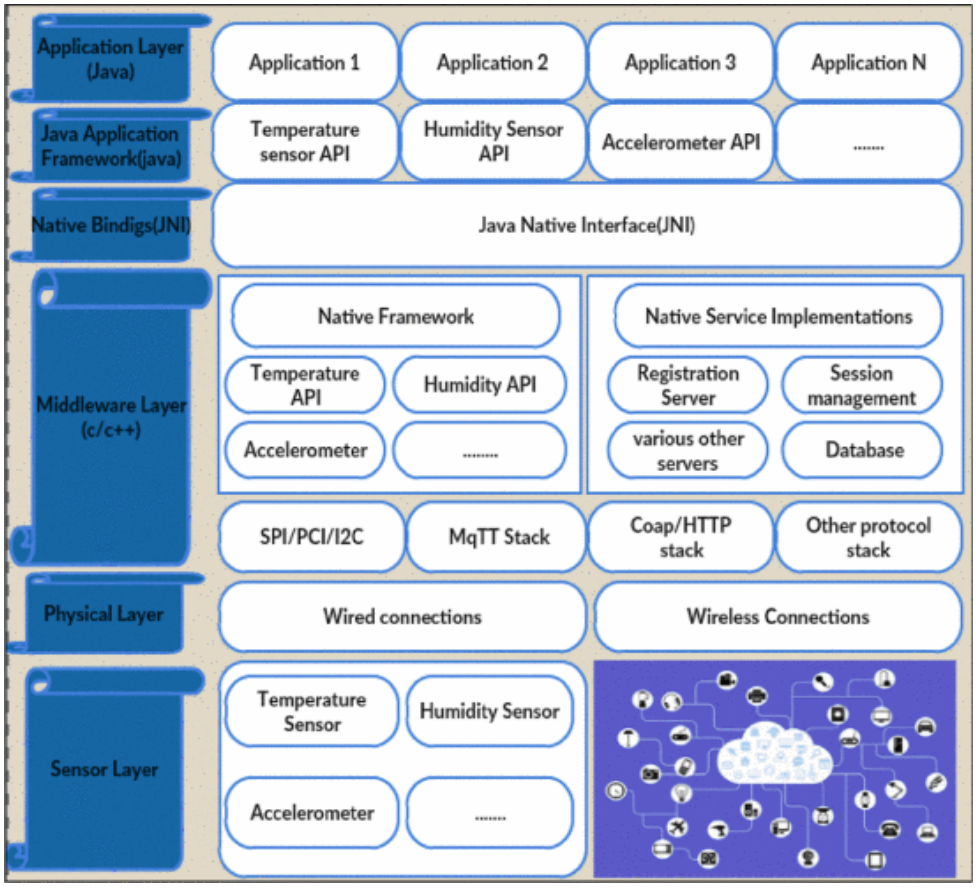


Figure 12. Software Framework for WSN virtualization [32]

This framework is further evaluated against our requirements, and the results are summarized below:

1. **Node Level Virtualization:** The framework proposed in this paper contains a session manager that ensures that the no two devices can access the same device at the same time. Thus, the concept of node level virtualization is not possible in this work since a device

cannot have many concurrent sessions on it. Thus, this first requirement is not satisfied by this work.

2. **Publication and Discovery Mechanism:** The framework contains a special Registration Server where the all the new devices are registered. This Registration Server contains a database that stores the relevant information of each device, such as location, module supported data types, etc. This database is further utilized for finding appropriate functionalities. Thus, this framework contains a mechanism for publication and discovery of the devices, which fulfills our second requirement.
3. **Orchestration Mechanism:** The framework does not explicitly state any mechanism for orchestrating the services of several devices. This implies that our third requirement for an orchestration mechanism is not met by this work.
4. **Bare metal access to IoT devices:** This framework does not provide any mechanism for having access to the devices as bare metal. It only provides virtualization of the WSN. Thus, our fourth requirement of having bare metal access to the IoT devices is also not met by this work.
5. **Ability to control and use actuators:** As mentioned in the framework, actuation tasks are handled by the Session Management. However, there is no explicit mechanism or APIs described for the management of actuators. Thus, this requirement for controlling and using actuators is not completely met by this work.

### 3.3.4. Summary of the State of the Art of Models and Frameworks for aiding the IoT IaaS

Table 2 summarizes the state of the art of the models and frameworks for aiding the IoT IaaS against the evaluation of our requirements. In this table a '✓' means that the requirement is met by the particular work, whereas a '×' means that the requirement is not met by the work.

Table 2. Summary of the Related Works involving the models and frameworks for aiding the IoT IaaS

| Papers | Requirements | | | | |
|---|---|---|---|---|---|
| | **Node Level Virtualization** | **Publication and Discovery Mechanism** | **Orchestration Mechanism** | **Bare metal access to IoT devices** | **Ability to control and use actuators** |
| Gupta *et al.* [28] and [29] | × | × | × | × | × |
| Mattos *et al.* [30] | × | × | × | × | × |
| Mandal *et al.* [31] | × | ✓ | × | × | ✓ |
| K.P.S *et al.* [32] | × | ✓ | × | × | × |

## 3.4. Conclusion

In this chapter, we first provided a motivating use case for the IoT IaaS. This use case was then discussed and used to derive the requirements of the IoT IaaS. This was followed by a thorough analysis of the current state of the art. We divided the state of the art into two categories: architectures for the IoT IaaS, and models and frameworks to aid the IoT IaaS. For the works in each of these categories, we first discussed each work in detail and then evaluated the work against the set of our derived requirements. Summary tables were also presented to show which

requirements were fulfilled by which state of the art. It was observed that none of the current state of the art was able to meet all of our derived requirements.

In the next chapter, the proposed architecture of the IoT IaaS is discussed in detail. All the components, functionalities, procedures, and interfaces are thoroughly discussed in it.

# Chapter 4

# The Architecture of the IoT IaaS

This chapter involves providing a description of the architecture of the IoT Infrastructure-as-a-Service proposed in this thesis. First, a high-level view of the architecture is provided, followed by the detailed description of the various architectural modules. The detailed description also includes the interfaces present within the architecture. Next, we provide a detailed description of the procedures pertaining to the architecture along with sequence diagrams for their illustration. Finally, the proposed architecture is evaluated against our derived requirements, followed by a conclusion to the chapter.

## 4.1. High-Level View of the IoT IaaS Architecture

Figure 13 shows the high-level view of the proposed architecture for the IoT IaaS. The architecture consists of the *IoT Devices Layer*, the *IoT Capabilities Management Layer*, the *IoT Cloud Management Layer,* the *Repository,* and several interfaces. The *IoT Capabilities Management Layer* and the *IoT Devices Layer* have access to the *Repository*. Moreover, each layer contains a *Front End* that is responsible for processing the incoming request and forwarding this request to the appropriate managers within the layer.

The topmost layer of the architecture is the *IoT Cloud Management Layer* which is responsible for parsing all the incoming requests. This layer includes the *Cloud Front End* and the *Cloud Manager*. The *Cloud Front End* forwards the received request to the *Cloud Manager, which* can handle application requests requiring a single IoT device or several IoT devices. It also specifically handles the cases when applications request both sensing as well as actuation capabilities.

Next is the *IoT Capabilities Management Layer,* which is responsible for handling the sensing capabilities and the actuation capabilities requested by the applications. This layer includes a *Capabilities Front End*, the *Sensing Capabilities Manager* and the Actuation *Capabilities Manager*. The *Capabilities Front End* receives the request and sends it to either the *Sensing Capabilities Manager* or the *Actuation Capabilities Manager*. The *Sensing/Actuation Capabilities Managers* handle the provisioning of the sensing or actuation IoT devices. The *Sensing Capabilities Manager* is responsible for handling application requests for sensing devices, such as temperature sensors, humidity sensors, motion sensors, etc., while the *Actuation Capabilities Manager* is responsible for handling requests for actuation devices, such as Lego Mindstorms robots, motors, etc. Both these managers are responsible for querying the *Repository* to find the appropriate IoT devices with the requested capabilities, and then handle the provisioning of these devices as per the applications' needs.

The lowest layer is the *IoT Devices Layer,* which consists of all the physical IoT devices, the virtual IoT devices created on top of the physical devices, and the modules to handle these IoT devices. These devices have sensing and/or actuation capabilities and are all heterogenous and resource constrained. The virtual IoT devices are basically abstractions of the properties of the physical IoT devices. These virtual devices can be orchestrated together to fulfill the needs of various applications. The applications that request virtual devices have complete control over the provisioned virtual IoT devices. In addition, a physical IoT device can only support a certain maximum number of virtualizations running on top of it. For instance, the Advanticsys SkyMote can support a maximum of 4 virtualizations on it without running out of memory. The *IoT Devices Layer* is further responsible for handling the creation of virtual devices, as well as the bare metal provisioning of the physical IoT devices. It also has the provision to add/update/delete information

to/from the *Repository* based on the status of the IoT devices. This layer includes the *IoT Devices Front End*, *Virtual/Bare-Metal Device Manager*, and the physical/virtual IoT devices. The *IoT Devices Front End* sends the received request to the *Virtual Device Manager* or the *Bare Metal Device Manager*, which are responsible for handling the provisioning of the virtual and physical IoT devices, respectively.

There also exists a *Repository* in the architecture, to which two layers of the architecture have access. This Repository includes all the information about the physical as well as the virtual devices, including their device IDs, capability descriptions, type of device (sensor/actuator), and status (idle/busy). The capability description includes the number of virtual devices the physical device can support (if any). For instance, the *Repository* includes the information of a physical Virtenio sensor with its capabilities: temperature sensing, humidity sensing, illuminance, air pressure, and acceleration; the virtual sensors created on top of it, their IDs, and their status.

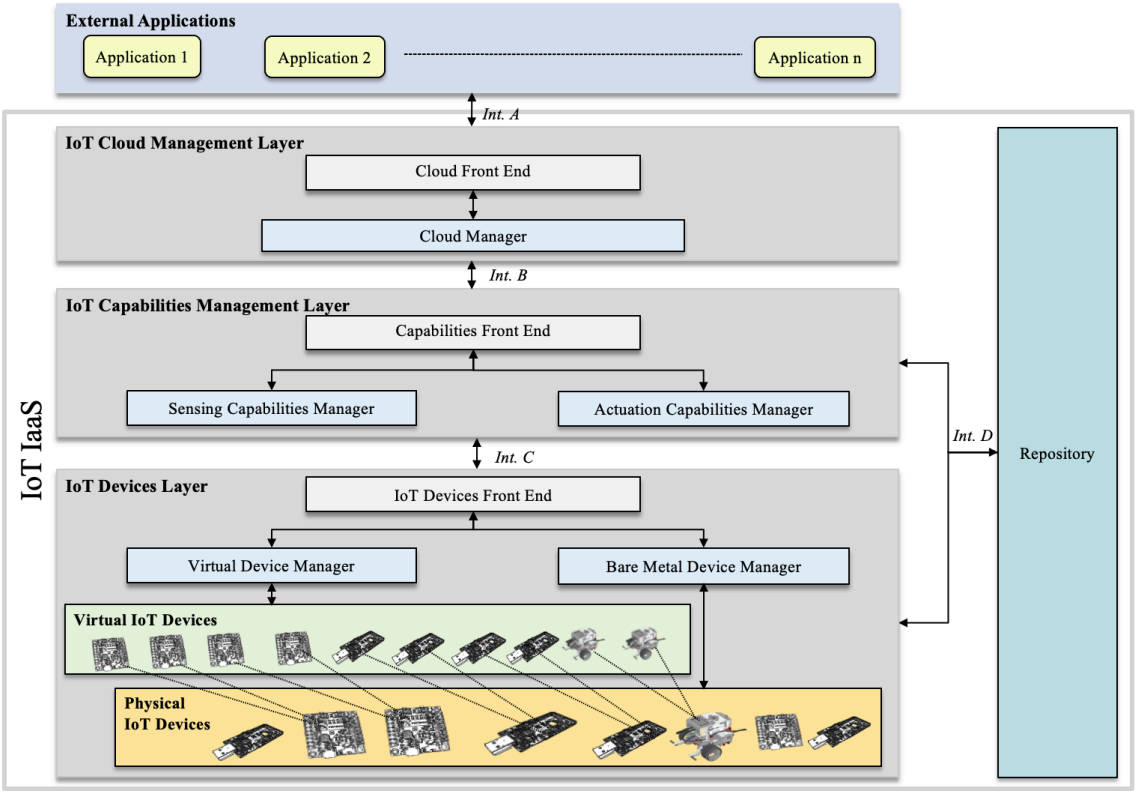


Figure 13. High Level View of the Architecture of the IoT IaaS

The architecture also contains several REST based interfaces, which include a high-level interface (*Int. A)* and several inter-layer interfaces. The high-level interface, *Int. A*, exposes the IoT IaaS to be used by various applications, which are shown above this interface in the figure as the *External Applications* layer. However, in addition to these applications, various PaaS can also access this IoT IaaS through this interface. In addition, several other interfaces, denoted by *Int. B, Int. C,* and *Int. D,* are also present between the different layers in order to facilitate their interaction.

## 4.2. Detailed View of the IoT IaaS

This section provides a detailed description of the various modules present in the architecture. More specifically, the details of the internal components of the Managers within the architecture, the *Repository*, as well as the interfaces are provided. A detailed view of the *Cloud Manager,* the *Sensing/Actuation Capabilities Manager,* and the *Virtual/Bare Metal Device Manager* are shown in figure 14 (a), (b), and (c) respectively. In order to describe these modules in detail, the internal components have been categorized into various entities based on their functions within these modules. These categories include the *Coordinators, Orchestrators, Publication/Discovery Entities,* and the *Interface Mappers.* Each of these entities are described in detail in the following subsection, followed by a description of the *Repository.* Finally, this section provides a brief description of the interfaces, especially the high-level interface (*Int. A*) that exposes the IoT IaaS to external applications and PaaS, and the inter-layer interface *Int. C* that allows the virtual and bare metal provisioning of the IoT devices.

### 4.2.1. Coordinators

As shown in figure 14 (a), (b), and (c), each of the managers contains a *Coordinator* entity. The main responsibility of the *Coordinators* is to send the request to the appropriate modules within

these managers in addition to facilitating the coordination within the managers' modules. These coordinators have similar, yet slightly different roles in each manager. This section describes in detail the role of each *Coordinator* within the *Cloud Manager,* the *Capabilities Manager,* and the *Device Manager.*

#### 4.2.1.1. Cloud Coordinator

The *Cloud Coordinator* is a part of the *Cloud Manager* in the *IoT Cloud Management Layer.* This component is primarily responsible for directing all the applications' requests to the appropriate component within the *IoT Cloud Management Layer* or to the underlying layers. All the application requests forwarded to the *Cloud Manager* are first received by the *Cloud Coordinator*. The responsibility of the *Cloud Coordinator* is to decide whether to forward this request to the *Cloud Orchestrator* within the *Cloud Manager,* or to the underlying *IoT Capabilities Management Layer,* depending on whether the application request requires the orchestration of the services of several IoT devices for sensing and actuation, or only a single device.

#### 4.2.1.2. Capabilities Coordinator

The *Capabilities Coordinator* is present in the *Sensing Capabilities Manager* and *Actuation Capabilities Manager* within the *IoT Capabilities Management Layer.* All the requests sent to any of the *Capabilities Managers* first reach its *Capabilities Coordinator*. This module is responsible for checking if the request needs orchestration or not, and then taking appropriate measures to handle this request. If only a single device is needed, then the *Capabilities Coordinator* accesses the *Discovery Engine* for the information of such a matching device and sends a command to the underlying *IoT Devices Layer* for reserving the device to be used as bare metal or creating a virtualization of this device. If more than one sensing or actuation devices are needed, in which

case orchestration of the capabilities of several devices is needed, it forwards the obtained request to the *Capabilities Orchestrator* within the *Capabilities Manager*, which further handles the request.

#### 4.2.1.3. Device Coordinator

The *Device Coordinator* is present within the *Virtual Device Manager* and the *Bare Metal Device Manager* within the *IoT Devices Layer.* It is responsible for coordinating between and interacting with the *Device Interface Mapper* and the *Publication Engine* in order to provision the devices and publish the most up-to-date information into the *Repository.* In order to provision the devices as bare metal, or create virtual devices, the *Device Coordinators* of the respective managers, i.e. *Bare Metal Device Manager* and *Virtual Device Manager,* interact with the respective *Device Interface Mappers* so that the devices can be provisioned. Once these devices are provisioned, the coordinators further interact with the respective *Publication Engines* within these managers to update the *Repository* with the information pertaining to the devices.

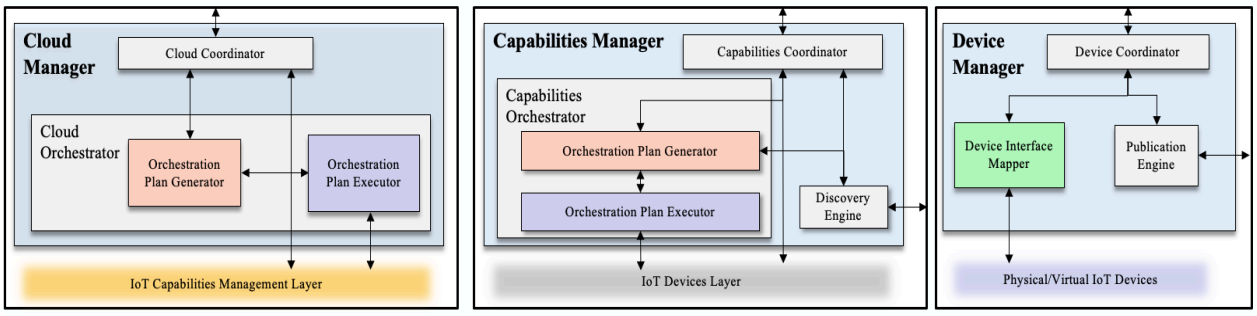


Figure 14. Detailed View of (a) *Cloud Manager* in the *IoT Cloud Management Layer,* (b) *Capabilities Manager* in the *IoT Capabilities Management Layer*, (c) *Device Manager* in the *IoT Devices Layer*

### 4.2.2. Orchestrators

The *Orchestrators* are present within the *Cloud Manager*, the *Sensing Capabilities Manager* and the *Actuation Capabilities Manager*. They are responsible for the orchestration of the services

of several IoT devices and contain the *Orchestration Plan Generator* and *Orchestration Plan Executor*. This subsection describes the *Cloud Orchestrator* present in the *Cloud Manager,* and the *Capabilities Orchestrator* present in the *Sensing/Actuation Capabilities Managers* in detail. Both these orchestrators have similar roles, with one key difference being that the *Cloud Orchestrator* orchestrates the services of both sensing and actuation devices, while the *Capabilities Orchestrator* only orchestrates the services of specific types of devices (i.e. sensing or actuation). This implies that the *Capabilities Orchestrator* within the *Sensing Capabilities Manager* orchestrates the services of several sensing devices, while the *Capabilities Orchestrator* within the *Actuation Capabilities Manager* orchestrates the services of several actuation devices.

#### 4.2.2.1. Cloud Orchestrator

The *Cloud Orchestrator* is present within the *Cloud Manager* in the *IoT Cloud Management Layer.* It is responsible for handling the orchestration of the services of several IoT devices when the application needs both sensing and actuation capabilities. It is also responsible for monitoring the outputs of the provisioned IoT devices and taking appropriate measures, such as firing actuation triggers, whenever the outputs of the devices exceed the specified thresholds. The *Cloud Orchestrator* further contains the *Orchestration Plan Generator* and the *Orchestration Plan Executor,* which are described below in detail.

##### 4.2.2.1.1. Orchestration Plan Generator

The *Orchestration Plan Generator* within the *Cloud Orchestrator* is responsible for handling the requests when both sensing, and actuation devices are needed by the application. Based on the sensing and actuation devices requested, it generates an appropriate *Orchestration Plan* which will meet the needs of the application. This *Orchestration Plan* also includes the threshold values for

the IoT devices, and the actions to be taken when these thresholds are exceeded. This *Orchestration Plan* is further sent to the *Orchestration Plan Executor* which is responsible for executing the plan.

A flowchart of the sample *Orchestration Plan* generated by the *Orchestration Plan Generator* in the *Cloud Orchestrator* is shown in figure 15 and described below.

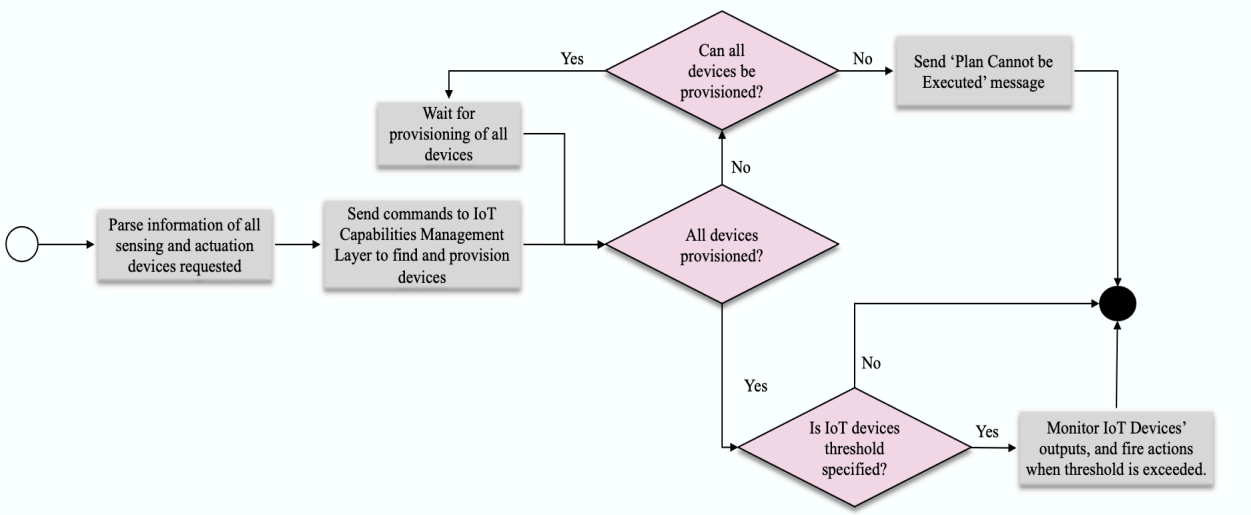


Figure 15. *Orchestration Plan* generated by the *Orchestration Plan Generator* in the *Cloud Orchestrator*

On getting the application's request for both sensing and actuation devices, as well as the thresholds for these devices' outputs, the *Orchestration Plan Generator* within the *Cloud Orchestrator* generates an *Orchestration Plan*.

The *Orchestration Plan* is as follows:

- On receiving the request for sensing and actuation devices, the request is first parsed for the specifications of all the sensing and actuation devices, and their thresholds.
- Next, commands are sent to the underlying *IoT Capabilities Management Layer* to find the suitable sensing and actuation devices and provision them.

- If matching devices are not found, and thus the provisioning of all devices cannot take place, then the respective message is sent to the *Orchestration Plan Generator* indicating that the plan cannot be executed and thus the application's request cannot be met. This terminates the plan.
- If all matching devices are found and provisioned, the IoT devices' outputs can now be monitored for the specified thresholds, if provided by the application. When the IoT devices' outputs exceed the thresholds, the corresponding action trigger is fired. For example, in the case of the 'Anti-Fire Systems' application, when the obtained temperature sensor value is above a certain specified threshold, the command is sent to dispatch the firefighting robots in that area to extinguish the fires.

  This Plan is sent to the *Orchestration Plan Executor*, which executes all the steps in this Plan.

#### 4.2.2.1.2. Orchestration Plan Executor

The *Orchestration Plan Executor* simply executes the *Orchestration Plan* that it receives from the *Orchestration Plan Generator.* This *Orchestration Plan* includes all the steps required for fulfilling the needs of the application. This, first and foremost, involves checking the request for the different devices requested by the application, and further sending the request to the underlying *IoT Capabilities Management Layer* to handle the provisioning of the sensing and actuation devices.

The *Orchestration Plan* also includes the information about the threshold values for the outputs of the IoT devices, and the actions to be taken when these thresholds are exceeded. Thus, the *Orchestration Plan Executor* is also responsible for monitoring these IoT devices once they are provisioned. For example, it monitors the outputs of the temperature sensors being used by the application. When the outputs of these sensors exceed the specified threshold (indicating a possible

fire hazard), it sends commands to the underlying layers to immediately dispatch the fire-fighting robots provisioned for this application.

#### 4.2.2.2. Capabilities Orchestrator

The *Capabilities Orchestrator* is present in the *Sensing Capabilities Manager* and the *Actuation Capabilities Manager* present within the *IoT Capabilities Management Layer.* It is responsible for the orchestration of the services of several IoT devices, when these services belong to the same type of device, i.e. sensing or actuation. Thus, the *Capabilities Orchestrator* within the *Sensing Capabilities Manager* is responsible for orchestrating the services of several sensing devices, while the *Capabilities Orchestrator* within the *Actuation Capabilities Manager* handles the orchestration of several actuation devices. The *Capabilities Orchestrator* consists of the *Orchestration Plan Generator* and the *Orchestration Plan Executor*, which are described below.

##### 4.2.2.2.1. Orchestration Plan Generator

The *Orchestration Plan Generator* within the *Capabilities Managers* is responsible for generating an orchestration plan for orchestrating the services of the various sensing devices or actuation devices needed as per the specifications of the request, as well as for monitoring these devices. For instance, when several sensing devices are requested, it parses the request for specifications of each sensing device. It then accesses the *Discovery Engine* to check if the required devices with the specifications provided in the application's request are available for provisioning or not. Once it receives this information, if all the matching devices required for the application's request are available, it generates an *Orchestration Plan* and forwards this plan to the *Orchestration Plan Executor* to be executed. However, if any of the devices matching the

specifications requested by the application are not available, an *Orchestration Plan* cannot be generated since the needs of the application cannot be met.

The sample *Orchestration Plan* generated by the *Orchestration Plan Generator* within the *Capabilities Orchestrator* of the *Sensing Capabilities Manager* is shown in figure 16 and explained below.

After storing the information of all the sensing devices that match the devices' specifications requested by the external application, the *Orchestration Plan Generator* within the *Capabilities Orchestrator* generates an *Orchestration Plan*, which can be executed by the *Capabilities Orchestration Plan Executor*. The *Orchestration Plan* is shown in figure 16.

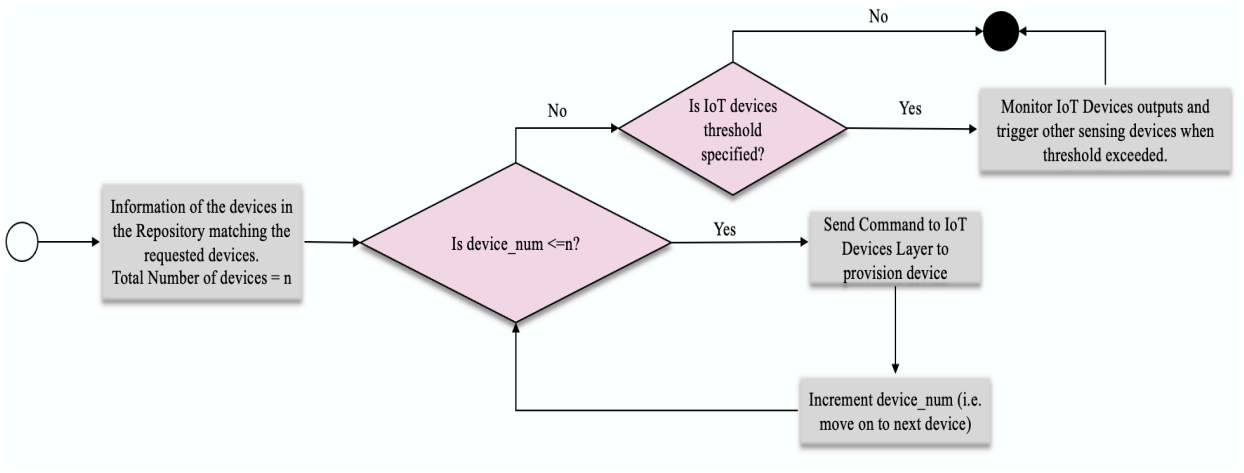


Figure 16. *Orchestration Plan* generated by the *Orchestration Plan Generator* in the *Sensing Capabilities Orchestrator*

The *Orchestration Plan Generator* generates an *Orchestration Plan* which is as follows:

- Let us say that 'n' sensing devices were found in the Repository that match the application's request and need to be provisioned.
- For each device, a command is sent to the underlying *IoT Devices Layer* to provision this device.

- The above step is carried out until commands are sent for all the sensing devices needed by the application (i.e. device_num is incremented until it is equal to 'n').
- Once all the sensing devices are provisioned, the specific IoT devices' outputs are now monitored. If the outputs exceed the threshold values specified by the application, the corresponding measures are taken. For example, in the case of several sensors, this could mean starting another sensor when one sensor's output exceeds the threshold value. This concludes the plan.

This plan is sent to the *Orchestration Plan Executor* which carries out all its steps.

#### 4.2.2.2.2. Orchestration Plan Executor

The *Orchestration Plan Executor* is responsible for receiving the *Orchestration Plan* from the *Orchestration Plan Generator* and executing it. Based on the *Orchestration Plan*, it sends commands to the underlying *IoT Devices Layer* for the creation of virtual devices and/or reservation of devices to be used as bare metal by the applications. Similar to the *Cloud Orchestrator*, it performs all the actions present in the *Orchestration Plan*, which can also include monitoring the outputs of the provisioned IoT devices for thresholds and taking appropriate measures based on these outputs. For example, when the threshold of a particular sensor is exceeded, it sends commands to start/stop another sensor as per the *Orchestration Plan*.

### 4.2.3. Publication/Discovery Entities

The publication and discovery modules present within the *IoT Capabilities Management Layer* and the *IoT Devices Layer* allow for the publication of the IoT devices information into the *Repository,* as well as for the discovery of the suitable devices based on the applications' needs.

This section describes the *Discovery Engine* in the *Sensing/Actuation Capabilities Managers* and the *Publication Engine* in the *Virtual/Bare Metal Device Managers* in detail.

#### 4.2.3.1. Discovery Engine

The *Discovery Engine* is present in the *Sensing Capabilities Manager* and the *Actuation Capabilities Manager* within the *IoT Capabilities Management Layer*. It is responsible for querying the *Repository* to find suitable information about the IoT devices as requested by the *Orchestration Plan Generator* in the *Capabilities Orchestrator,* or the *Capabilities Coordinator*. It interacts directly with the *Repository* and fetches the information as needed.

#### 4.2.3.2. Publication Engine

The *Publication Engine* is present in the *Virtual Device Manager* and the *Bare Metal Device Manager* within the *IoT Devices Layer.* It is responsible for updating the Repository with the most relevant information on the physical and virtual IoT devices. Whenever a new physical device is added to the *Physical IoT Devices* layer or a new virtual IoT device is created over the physical IoT devices, this component updates the *Repository* with the information pertaining to these new devices. Similarly, it also updates the *Repository* with the relevant information whenever a device is removed from the infrastructure or a virtual device is deleted. If a physical device is reserved to be used as bare metal or released by the application using it as bare metal, it again updates the *Repository* with this latest information. It gets the command to do so from the *Device Coordinator*.

### 4.2.4. Interface Mappers

The *Device Interface Mappers* are present in the *Virtual Device Manager* and the *Bare Metal Device Manager* within the *IoT Devices Layer.* They interact with the proprietary interfaces of the IoT devices, and essentially play the role of a ‘mapper’ between these proprietary interfaces and

the uniform interfaces. Although the roles of both these *Device Interface Mappers* are similar, they differ in handling the virtual devices and physical devices respectively. The *Bare Metal Device Interface Mapper* and the *Virtual Device Interface Mapper* are described in this section.

#### 4.2.4.1. Bare Metal Device Interface Mapper

The *Bare Metal Device Interface Mapper* within the *Bare Metal Device Manager* is responsible for controlling the physical devices by directly interacting with them through their proprietary interfaces. It acts as a mapper between the uniform bare metal REST based interface, and the proprietary interfaces of the IoT devices. Whenever the *IoT Devices Layer* gets the command via the uniform interface to reserve a device to be used as bare metal, this component interacts directly with the physical device (through the proprietary interface of this physical device) and reserves it for the application. Similarly, it also releases the physical device being used as bare metal when the application is done using it. For example, the *Bare Metal Interface Mapper* uses nesC for interacting with the advanticsys sensor when it needs to be reserved as bare metal.

#### 4.2.4.2. Virtual Device Interface Mapper

The *Virtual Device Interface Mapper* is responsible for creating virtualizations of the physical IoT devices by interacting with them through their proprietary interfaces. Whenever the *IoT Devices Layer* gets the command via the uniform interface to create a virtual device, this component interacts with the physical devices and creates their virtualization as per the application's specifications. Similarly, it also deletes the virtual device when it receives the command to do so, when the application is done using the virtual device. Similar to the *Bare Metal Interface Mapper,* it acts as a mapper between the uniform and independent REST based interface for creating virtual devices, and the proprietary interfaces of the IoT devices. For example, the

*Virtual Device Interface Mapper* uses Java for creating virtualizations on top of the Virtenio sensor.

### 4.2.5. Repository

The *Repository* is essentially a database responsible for storing the latest and most up-to-date information pertaining to the physical and virtual IoT devices. There are two types of repositories utilized within this IoT IaaS. These repositories are: *Physical IoT Device Repository,* and the *Virtual IoT Device Repository.* These repositories are described in detail in the subsections to follow.

#### 4.2.5.1. Physical IoT Device Repository

The *Physical IoT Device Repository* contains the information about all the physical IoT devices present within the infrastructure. It contains information about their capabilities/functionalities, type of device (sensor/actuator), device ID, and their status (idle/busy) i.e. whether they have been reserved to be used as bare metal or not. The capability description would also include the number of virtual devices the physical device could support (if any). For instance, the *Repository* would include the information of a physical Virtenio Preon32 Shuttle + VariSen sensor with its capabilities, i.e. temperature sensing, humidity sensing, illuminance, air pressure, and acceleration, and the virtual sensors created on top of it, their IDs, and status.

#### 4.2.5.2. Virtual IoT Device Repository

The *Virtual IoT Device Repository* contains information about all the virtualized devices created on top of the physical devices. It contains information about their capabilities/functionalities, type of device (sensor/actuator), device ID, and status.

### 4.2.6. Interfaces

For the interaction between the layers of the proposed architecture, several interfaces have been designed. The general principle used to design these interfaces is the use of the REpresentational State Transfer (REST) architectural style. The interfaces between the layers expose CRUD (Create, Read, Update, Delete) operations. For instance, the interface between the applications and *IoT Cloud Management Layer* (Int. A) allows the applications to send a request to the *IoT Cloud Management Layer* to create a sensor + actuator module with given sensor and actuator parameters (e.g., service-type, location, sampling rate, etc.), whenever a combination of sensors and actuators is required by the application. It also allows the applications the get the sensors data and a list of actuator actions. Table 3 demonstrates the uniform interface we propose for Interface A. Each sensor+actuator module is identified by a unique *ID*.

Table 3. Summary of the API to access the IoT IaaS (Interface A)

| Plane | Operation | Explanation | Focus Point | Example Values | Resource URL |
|---|---|---|---|---|---|
| Control | create_sam (sensor actuator module) | Create a sensor_actuator module with given sensor and actuator parameters. | Method: | POST | *<BASE_URI>* |
| | | | Parameters | {sensor: [{service-type, location, sampling-rate, data-mode}], actuator:[{service-type, location}]} | |
| | | | Success | 200 OK *<ID>* | |
| | | | Failure | Error message/code (e,g – 404 Not Found, 422 Unprocessable Entity) | |
| Control | delete_sam | Delete the sensor_actuator module with ID | Method: | DELETE | <BASE_URI/ID> |
| | | | Parameters | | |
| | | | Success | 200 OK <ID> | |

| | | | | | |
|---|---|---|---|---|---|
| | | | Failure | Error message/code (e,g – 404 Not Found) | |
| Data | get | Gets the sensor output/data and list of actuator actions (actionIDs for each action the actuator can perform) | Method: | GET | <BASE_URI/ID/sam_data> |
| | | | Parameters | | |
| | | | Success | 200 OK | |
| | | | Failure | Error message/code (e,g – 404 Not Found) | |
| Data | post | Specify the threshold value for the sensor and its corresponding actuator action | Method: | POST | <BASE_URI/setter> |
| | | | Parameters | {mappings: [{samID:xx, sensor_ID: xx, sensor_threshold : [{type:range/const, val:xxx,…}], actuator_ID:xx, action_URI: xx},…]} | |
| | | | Success | 200 OK | |
| | | | Failure | Error message/code (e,g – 404 Not Found, 403 Forbidden) | |

Similarly, the interfaces for provisioning of the IoT devices as bare metal, and creation of virtual devices are shown below. The *Int. C (Interface C)* comprises of these interfaces for the provisioning of devices. Table 4 shows the RESTful API for provisioning of the bare metal devices. Each physical device has a unique *UUID*. The table shows the request types for reserving a device as bare metal, getting list of devices which are available to be provisioned as bare metal, releasing the device being used as bare metal. Similarly, table 5 and 6 show the APIs for provisioning virtual actuators and sensors respectively, identified by unique *UUIDs*.

The interfaces *Int. B* and *Int. D* are simply concerned with exchanging the information pertaining to the applications' requests between the *IoT Cloud Management Layer* and the *IoT*

*Capabilities Management Layer*, and the exchange of IoT devices information between the *Repository* and the *Publication/Discovery Engines* respectively. Thus, they are not as full-fledged as the interfaces summarized below.

Table 4. Summary of the API for the Bare Metal Provisioning of IoT Devices.

| **Plane** | **Operation** | **Explanation** | **Focus Point** | **Example Values** | **Resource URL** |
|---|---|---|---|---|---|
| Control | reserve_biot | Reserve the physical IoT device to be accessed as bare metal for the application requesting it | Method | POST | <BASE_URI/UUID> |
| | | | Parameters | {status (e.g. busy)} | |
| | | | Success | 200 OK <UUID> | |
| | | | Failure | Error message/code (e,g – 404 Not Found on giving uuid that does not exist) | |
| Control | get | Get list of UUIDs of IoT devices (to access as bare metal) | Method: | GET | *<BASE_URI/LIST_IOT>* |
| | | | Parameters | | |
| | | | Success | 200 OK | |
| | | | Failure | Error message/code (e,g – 404 Not Found) | |
| Control | release_biot | Release the physical IoT device once application is done using it | Method | POST | <BASE_URI/UUID> |
| | | | Parameters | {status (e.g. idle)} | |
| | | | Success | 200 OK <UUID> | |
| | | | Failure | Error message/code (e,g – 404 Not Found on giving uuid that does not exist) | |

Table 5. Summary of the API for Creating a Virtual Actuation Device.

| Plane | Operation | Explanation | Focus Point | Example Values | Resource URL |
|---|---|---|---|---|---|
| Control | create_act | Create virtual actuator with given parameters | Method: | POST | <*BASE_URI*> |
| | | | Parameters | {service-type, location} | |
| | | | Success | 200 OK <*UUID*> | |
| | | | Failure | Error message/code (e,g – 404 Not Found, 422 Unprocessable Entity) | |
| Control | delete_act | Delete actuator identified by the UUID | Method: | DELETE | <*BASE_URI/UUID*> |
| | | | Parameters | | |
| | | | Success | 200 OK <*UUID*> | |
| | | | Failure | Error message/code (e,g – 404 Not Found) | |
| Control | get | List all actions that the actuator can perform | Method: | GET | <BASE_URI/UUID/actions> |
| | | | Parameters | | |
| | | | Success | 200 OK | |
| | | | Failure | Error message/code (e,g – 404 Not Found) | |
| Data | post | Post request trigger for firing an action in the actuator | Method: | POST | <BASE_URI/UUID/actions/ actionid> |
| | | | Parameters | {event_start_time, status (eg. on/off), action_duration etc.} | |
| | | | Success | 200 OK | |
| | | | Failure | Error message/code (e,g – 404 Not | |

| | | | | | |
|---|---|---|---|---|---|
| | | | | Found, 422 Unprocessable Entity) | |
| Control | get | List the device's specifications (eg. location, manufacturer, serial number) and device's configurable settings (eg. uptime, task-mode, (other device specific settings)) | Method: | GET | <BASE_URI/UUID> |
| | | | Parameters | | |
| | | | Success | 200 OK | |
| | | | Failure | Error message/code (e,g – 404 Not Found) | |
| Control | post | Configure device parameters or settings that can be configured by user | Method: | POST | <BASE_URI/UUID> |
| | | | Parameters | {set-task-mode=XX (specific to device, for eg. Standard mode/advanced mode for particular action), set-velocity=XX (eg. In case of robots set their velocity or other device specific parameters) etc.} | |
| | | | Success | 200 OK | |
| | | | Failure | Error message/code (e,g – 404 Not Found, 422 Unprocessable Entity) | |

Table 6. Summary of the API for Creating a Virtual Sensing Device

| **Plane** | **Operation** | **Explanation** | **Focus Point** | **Example Values** | **Resource URL** |
|---|---|---|---|---|---|
| Control | create_sen | Create virtual sensor with | Method: | POST | *<BASE_URI>* |

| | | | | | |
|---|---|---|---|---|---|
| | | given parameters | Parameters | {service-type, location, sampling-rate, data-mode} | |
| | | | Success | 200 OK <*UUID*> | |
| | | | Failure | Error message/code (e,g – 404 Not Found, 422 Unprocessable Entity) | |
| Control | delete_sen | Delete sensor identified by the UUID | Method: | DELETE | *<BASE_URI/UUID>* |
| | | | Parameters | | |
| | | | Success | 200 OK <*UUID*> | |
| | | | Failure | Error message/code (e,g – 404 Not Found) | |
| Data | get | Get the data from the virtual sensor device | Method: | GET | <BASE_URI/UUID/data> |
| | | | Parameters | | |
| | | | Success | 200 OK | |
| | | | Failure | Error message/code (e,g – 404 Not Found) | |
| Control | get | List the device's specifications (eg. location, manufacturer, serial number), capabilities, and device's configurable settings (if any) | Method: | GET | <BASE_URI/UUID> |
| | | | Parameters | | |
| | | | Success | 200 OK | |
| | | | Failure | Error message/code (e,g – 404 Not Found) | |

## 4.3. Procedures

The proposed architecture consists of three sub-procedures within the layers of the IoT IaaS, as well as two procedures that span several layers and may contain the specified sub-procedures. This

section describes each of these procedures in detail. These main procedures spanning several layers of the architecture are: *IoT Devices Provisioning,* and *IoT Devices Monitoring.*

### 4.3.1. Procedures within the Layers of the Architecture

The procedures within the layers of the IoT IaaS architecture include orchestration, device capabilities management, and virtual device creation/device reservation. Each of these procedures are described in this subsection. These procedures may further act as sub-procedures for the procedures that span several layers, and which are described in the upcoming subsections in this chapter.

#### 4.3.1.1. Orchestration

The orchestration procedure is essential to the IoT IaaS when the orchestration of the services of several IoT devices is required as per the applications' requests. The *Orchestrators* present within the respective Managers, i.e. *Capabilities Manager* and/or the *Cloud Manager,* are responsible for carrying out this orchestration. The orchestration procedure starts with a request being forwarded to the orchestration component to orchestrate the services of 2 or more devices. On receiving this request, the *Orchestration Plan Generator* within the *Orchestrator* of the appropriate Manager, creates an *Orchestration Plan* for orchestrating the services of the devices, and monitoring the devices. The *Orchestration Plan Executor* of this Manager further handles the execution of this *Orchestration Plan*. If the application's request is for a combination of sensing and actuation devices, the *Cloud Orchestrator* handles this orchestration, whereas if the application's request is for a combination of only sensing devices or only actuation devices, then the *Capabilities Orchestrator* present within the respective *Capabilities Manager,* i.e. *Sensing Capabilities Manager* or *Actuation Capabilities Manager*, handles the orchestration of the

devices' services. Furthermore, the orchestration procedure would also include monitoring the outputs of the devices based on specified thresholds once the provisioning is completed. Figures 18 and 19 highlight the IoT devices provisioning procedure, which is described in the upcoming subsections. The sequence diagrams in these figures also show the sequence of steps when the request reaches the *Cloud Orchestrator* or the *Capabilities Orchestrator* respectively.

In the motivating 'Smart Factory' scenario presented in chapter 3, the 'Monitoring of Cooling Systems' application will use this procedure when requesting for temperature and humidity sensors. Since this application will require only sensing devices, the *Capabilities Orchestrator* within the *Sensing Capabilities Manager* will handle the orchestration of the temperature and humidity sensing devices. Similarly, in the case of the 'Anti-Fire Systems' application, the *Cloud Orchestrator* within the *Cloud Manager* will handle this orchestration since both sensing and actuation capabilities would be needed.

#### 4.3.1.2. Device Capabilities Management

The *Device Capabilities Management* procedure is carried out by the *Sensing Capabilities Manager* and the *Actuation Capabilities Manager* within the *IoT Capabilities Management Layer.* This procedure handles the management of the sensing and actuation capabilities of the IoT devices. Primarily, it involves querying the *Repository* to find the appropriate and available sensing or actuation devices as per the needs of the application. For instance, in the case of the 'Smart Factory' scenario, all the applications would require access to sensing or actuation devices. In this case, the *Device Capabilities Management* procedure would be used to find the most appropriate devices that can fulfill the applications' needs. Furthermore, this procedure would not only be used for applications requiring single IoT devices, but also for applications requiring several IoT

devices, in which case the *Orchestration* procedure would require the devices' data obtained from the *Device Capabilities Management* procedure.

#### 4.3.1.3. Virtual Device Creation/ Device Reservation

This procedure is carried out by the *Virtual Device Manager* or the *Bare Metal Device Manager* in the *IoT Devices Layer.* This procedure involves two distinct functions, which include interacting with the proprietary interfaces of the physical IoT devices to virtualize them or provision them to be used as bare metal, and publish the most relevant information pertaining to these physical and virtual IoT devices into the repository. Furthermore, once the devices are provisioned based on the applications' requests, the appropriate 'device reserved/virtualization created' messages must also be returned. The *Virtual Device Manager* is responsible for creating virtual devices on top of the physical IoT devices and for publishing their information into the repository, while the *Bare Metal Device Manager* does the same for the bare metal provisioning of the physical IoT devices.

Within the 'Smart Factory' use case mentioned in chapter 3, the 'Anti-Fire Systems' application will require bare metal access to the fire-fighting robots in order to reduce latency. In order to achieve this, the *Device Reservation* procedure would be utilized by the *IoT Devices Layer* to provide bare metal access to the robots.

### 4.3.2. Procedures spanning several Layers of the Architecture

The procedures that span several layers of the architecture are described in this subsection. These include the *IoT Devices Provisioning* and the *IoT Devices Monitoring* procedures.

#### 4.3.2.1. IoT Devices Provisioning

The *IoT Devices Provisioning* procedure involves the provisioning of the various sensing and actuation devices as bare metal or as virtual devices on top of the physical IoT devices. This

procedure, which involves the provisioning of the required devices (sensors, actuators), would further include the sub-procedures, which are *Orchestration*, *Device Capabilities Management,* and *Virtual Device Creation/Device Reservation* explained in the previous subsection. This procedure spans all the layers of the architecture, since the request must pass through each layer in order to finally allow the IoT devices to be reserved or virtualized.

The request received by the IoT IaaS is first processed and if several devices are requested, i.e. orchestration is needed, then it is performed by the appropriate *Orchestrator*, i.e. the *Cloud Orchestrator* or the *Capabilities Orchestrator*. The *Device Capabilities Management* procedure is further used to find the appropriate device or devices that can be provisioned. The last step of this procedure involves the *Virtual Device Creation/Device Reservation*, i.e. the virtual devices are created on top of the physical IoT devices, or the physical IoT devices are reserved to be used as bare metal. The sequence diagrams for the *IoT Devices Provisioning* procedure are shown in figures 17, 18 and 19. Figure 17 shows the provisioning of a single IoT device, where the sequence diagram depicts step-by-step how the request passes through the appropriate components in each layer. For provisioning a single IoT device orchestration is not required. Similarly, figure 18 shows the sequence diagram for provisioning a combination of several sensors and actuators, while figure 19 shows the sequence of steps for provisioning several devices of only one specific type (i.e. sensing or actuation), in this case, several sensing devices. These diagrams depict how these different types of requests pass through the various layers in the architecture, and which components are responsible for handling them.

In the motivating 'Smart Factory' scenario, this procedure proves critical as every application must send a request to the IoT IaaS to provision the devices it requires. For instance, the 'Smart Energy Systems' application will send the request to the IoT IaaS, which will use the *IoT Devices*

*Provisioning* procedure to provide motion sensors and actuators for controlling electricity systems to the application.

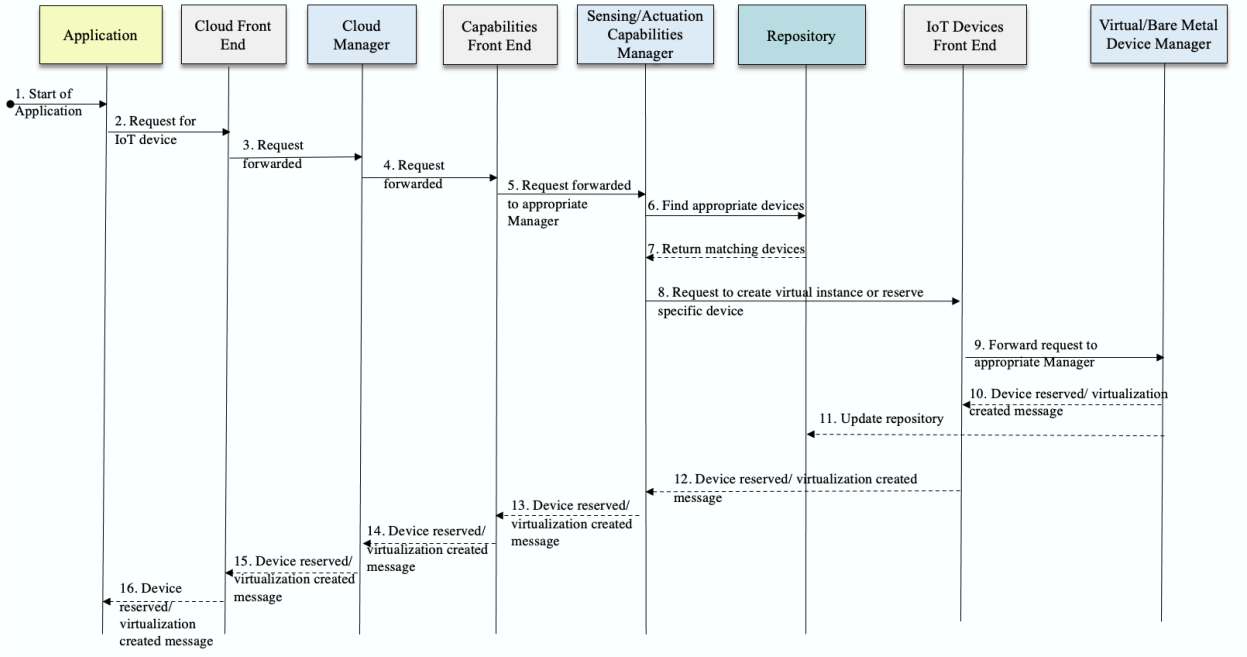


Figure 17. Sequence Diagram for Provisioning a Single IoT Device

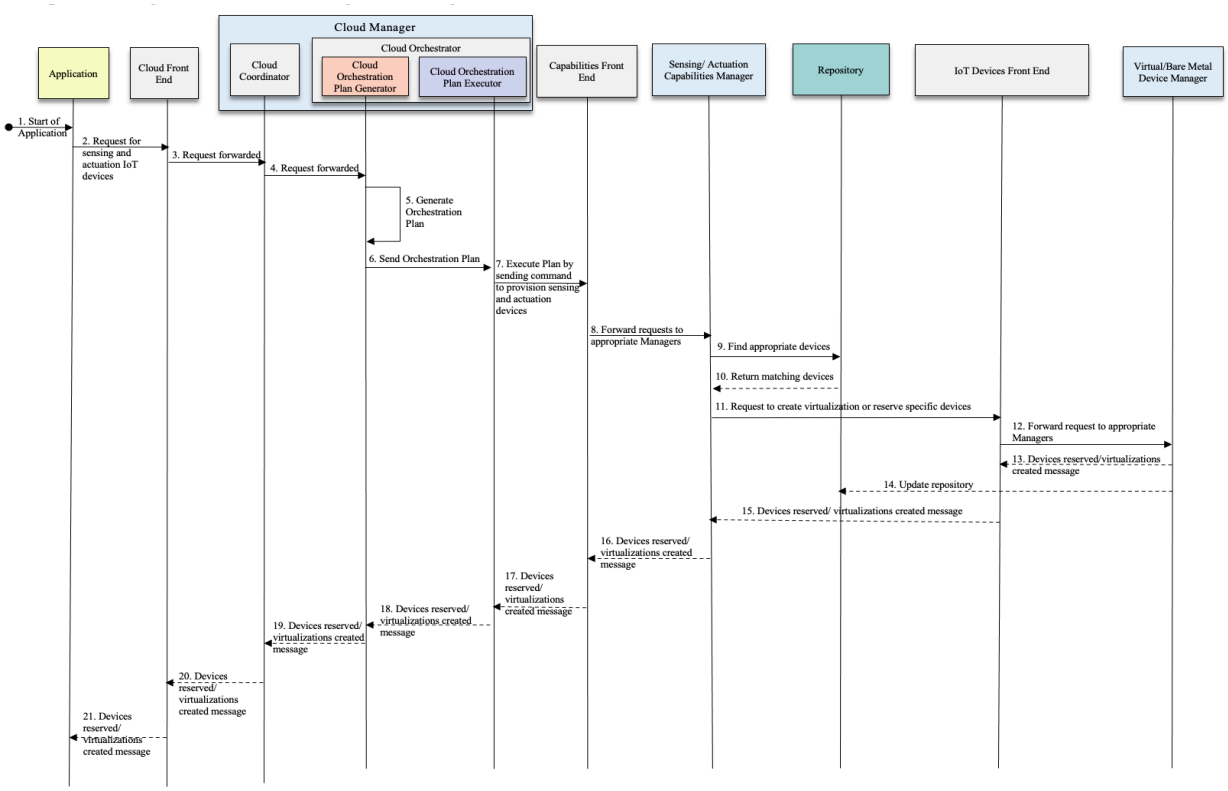


Figure 18. Sequence Diagram for Provisioning of Sensing and Actuation IoT Devices

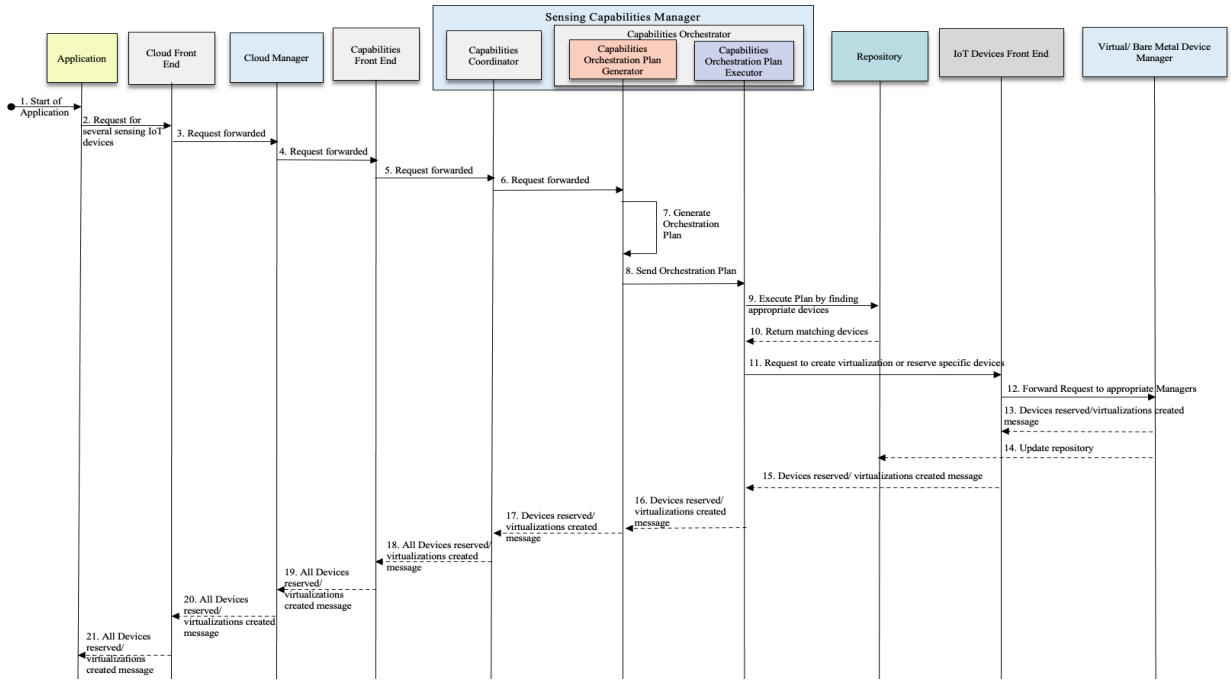


Figure 19. Sequence Diagram for Provisioning of Several Sensing Devices

### 4.3.2.2. IoT Devices Monitoring

The *IoT Devices Monitoring* procedure involves monitoring the outputs of the IoT devices to detect the specified thresholds. Once the thresholds are exceeded, it is further responsible for taking appropriate actions as per the applications' specifications. Thus, this procedure is not only responsible for monitoring the IoT devices, but also for automatically sending control commands to other IoT devices whenever the threshold values are exceeded. The *Orchestrators* present within the *Cloud Manager* and the *Capabilities Manager* handle this monitoring of the devices' outputs. The threshold values, mentioned above, are incorporated into the *Orchestration Plan*, which is generated by the *Orchestration Plan Generator* and executed by the *Orchestration Plan Executor* of the appropriate managers. The *Orchestration Plan Executor* then constantly checks the IoT devices for the outputs matching or exceeding the threshold values, and further sends commands to the underlying layers to fire the appropriate actuation triggers or control other sensing IoT devices. Figure 20 shows the sequence diagram for the *IoT Devices Monitoring* procedure.

Within the ‘Smart Factory’ use case, the ‘Anti-Fire Systems’ application will use this procedure to monitor the values of the temperature and humidity sensors. Once these sensors exceed the specified thresholds, the fire-fighting robots would automatically be launched.

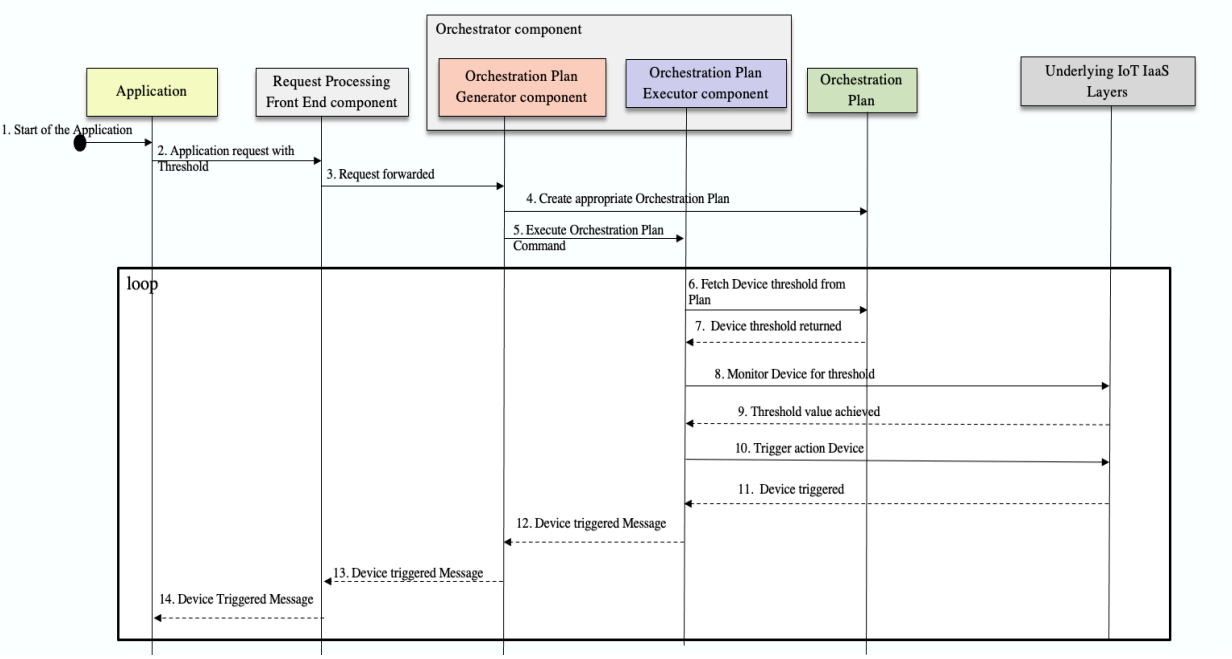


Figure 20. Sequence Diagram for IoT Devices Monitoring Procedure

## 4.4. Evaluation of the Proposed Architecture against the Requirements

It is essential to evaluate our proposed architecture against the requirements derived from the motivating use case. In fact, it is observed that this architecture fulfills all the requirements.

The provision of several virtual devices on top of the same physical device based on the applications’ needs aim to increase resource utilization by running several applications simultaneously on the physical device. **Thus, the architecture relies heavily on node level virtualization, which fulfills our first requirement.**

The proposed architecture consists of a *Publication Engine* in both the *Virtual Device Manager* and *Bare Metal Device Manager*, which are responsible for publishing the most relevant information on the virtual devices and the physical devices respectively into the *Repository.* This

includes information about addition of new devices, deletion of devices, and their current status. In addition, the *Sensing Capabilities Manager* and *Actuation Capabilities Manager* both consist of a *Discovery Engine* which is responsible for querying the *Repository* and fetching information about the matching devices based on the request received.

**Thus, there exists a mechanism for the publication and discovery of the devices' services, which satisfies our second requirement.**

The proposed architecture consists of a mechanism for orchestration. The *Cloud Manager* contains a *Cloud Orchestrator*, which is responsible for handling the orchestration of the capabilities of the sensing and actuation devices. In addition, the *Sensing Capabilities Manager* and *Actuation Capabilities Manager* both contain the *Capabilities Orchestrator*, which is responsible for handling the orchestration of the capabilities of several sensing devices or of several actuation devices respectively.

**Thus, it can be seen that there exists a proper orchestration mechanism to orchestrate the capabilities of the different IoT devices. Hence, our third requirement is fulfilled by the proposed architecture.**

The proposed architecture supports the bare metal provisioning of IoT devices. The *Bare Metal Device Manager* present in the *IoT Devices Layer* is responsible for reserving devices to be used as bare metal by the applications. It achieves this by interacting with the proprietary interfaces of the physical IoT devices.

**Thus, the architecture supports the bare metal provisioning of the IoT devices, which satisfies our fourth requirement.**

Finally, the architecture consists of a mechanism for using and controlling actuators. The architecture contains the *Actuation Capabilities Manager*, which is responsible for the

provisioning and control of actuation devices by sending appropriate commands to the underlying layers. To facilitate this, the architecture contains RESTful APIs for interacting with the actuator devices.

**Thus, our fifth requirement of the ability to control and use actuators is fulfilled by the proposed architecture.**

## 4.5. Conclusion

In this chapter, we provided a detailed description of the architecture of the IoT Infrastructure-as-a-Service proposed in this thesis. We began by providing a high-level view of the architecture, followed by a detailed view of the various architectural modules and interfaces. Next, we described the procedures pertaining to the architecture along with several detailed sequence diagrams for their illustration. Finally, we provided a justification of our proposed architecture by evaluating it against our derived requirements.

In the next chapter, we will present the implemented prototype and describe the tools and platforms used. In addition, we will also analyze the performance metrics for the architecture's evaluation and derive conclusive results.

# Chapter 5

# Validation of the Architecture

This chapter begins by providing an overview of the implemented prototype architecture. This overview includes a brief description of the implemented scenario and the implemented prototype, followed by the details of the hardware and software used. We then provide the detailed description of the prototype architecture, which includes the description of each implemented layer of the prototyped architecture using the bottom-up approach, and a validation summary. This is followed by a detailed section on the performance evaluations, which includes the description of the performance metrics, the experimental setup, and a thorough analysis of the results obtained. In addition, the performance evaluation also includes testing the scalability of the IoT IaaS by provisioning a large number of devices through extensive simulations. Finally, we conclude the chapter by providing its summary.

## 5.1. Prototype Architecture Overview

In this section we first describe the implemented scenario, followed by a high-level description of the working of the prototype. Finally, we present a brief description of the hardware and software used for the implementation of this prototype.

### 5.1.1. Implemented Scenario

The implemented scenario consisted of a subset of our motivating use case. The ‘Anti-Fire Systems’ and ‘Monitoring of Cooling Systems’ applications were implemented for this purpose. For the ‘Anti-Fire Systems’ application, both the sensing capabilities (temperature sensing and humidity sensing) and actuation capabilities (firefighting robot movement) were considered. For

the ‘Monitoring of Cooling Systems’ application, only the sensing capabilities (temperature sensing and humidity sensing) were considered. The actuation capabilities for this application were excluded. The sensors (for temperature and humidity sensing) used were from two different vendors, Advanticsys (CM5000 TelosB SkyMote) and Virtenio (Preon32 Shuttle and VariSen module), while the actuator (robot) used was the LEGO Mindstorms EV3 robotics kit. For both the applications, the sampling rate for temperature and humidity sensing was set to 1 sample/second. In the case of the ‘Anti-Fire Systems’ application, the requirement was to start the robot as soon as the temperature and humidity sensing values exceeded a pre-defined threshold. For trial purposes this threshold was set to 25ºC for the temperature sensor and 50% relative humidity for the humidity sensor. The threshold of 25ºC for the temperature sensor was simply chosen for the purpose of experimentation. In the case of the ‘Monitoring of Cooling Systems’ application, the requirement was to raise an alert message whenever the temperature and sensing values exceeded beyond the pre-defined threshold.

Both the applications required orchestration of several services. While the ‘Anti-Fire Systems’ application required the orchestration of both sensing and actuation capabilities, the ‘Monitoring of Cooling Systems’ application only required the orchestration of multiple sensing capabilities. More specifically, the ‘Anti-Fire Systems’ application required sending a request to the IoT IaaS for temperature sensor, humidity sensor, and EV3 LEGO robot, along with arbitrary thresholds for values of temperature and humidity sensors. Since these are several devices, orchestration was needed, and the virtualizations of these devices or bare metal access to these devices needed to be provisioned. Next, the temperature and humidity sensor readings were to be constantly monitored and as soon as the value of the temperature and humidity sensors crossed the specified threshold, the EV3 robot was to be automatically started. A similar procedure was required for the

'Monitoring of Cooling Systems' application, except that this application did not require an actuator, instead it required that an alert message be issued to the application when thresholds were exceeded. Both the applications required access to common IoT devices (temperature sensor and humidity sensor). The devices were, thus, to be shared by creating virtualizations that would be accessed independently by these applications or by provisioning some devices as bare metal for an application, depending on the application's request.

### 5.1.2. Description of the Implemented Prototype

In order to access the IoT IaaS, a REST API (Application Programming Interface) has been implemented that allows the applications to utilize the services of the IoT IaaS in a uniform manner. This further enables the programming interface to be platform/language independent. The interfaces for provisioning of virtual sensors and actuators, and bare metal provisioning of IoT devices, as well as interaction between the different components in the architecture, are also sets of REST APIs.

The captured application request is parsed by the IoT IaaS and on successful completion of the request the results are returned to the application with the appropriate URIs to access the devices, whether virtualized, composite, or reserved for bare metal access. In the case of monitoring IoT devices for thresholds, the responses returned include messages indicating that the threshold is exceeded, as well as messages indicating the successful completion of actions triggered on threshold surpassing. In order to match devices with the applications requests, the application requests are parsed for parameters such as type of device/devices (i.e. sensor/actuator), functionalities of devices (eg. temperature sensing, humidity sensing, movable robot), virtualized or bare metal access to the devices, sampling-rate (in case of temperature and humidity sensors).

In order to validate the prototype, we conducted experiments using the following physical devices: an Advanticsys TelosB CM5000 SkyMote (temperature and humidity sensing), a Virtenio Preon32 Shuttle with VariSen module (temperature and humidity sensing), and a LEGO EV3 Mindstorms robot (actuation). In addition, Contiki Cooja was used to simulate SkyMotes (temperature and humidity sensing) up to a limit of 1000 devices to test the scalability of the IoT IaaS. First, each of the single physical devices were individually used for bare metal provisioning as well as creation of virtual devices. Next, for validating the ability of the IoT IaaS to monitor the outputs of the IoT devices (in this case, sensing device) and immediately take appropriate actions (in this case, sending command to actuator) based on the applications' needs, the EV3 robot and Virtenio sensor were provisioned to get both the actuation and sensing capabilities respectively. Finally, in order to test the scalability of the IoT IaaS, provisioning of 2, 4, 8, 10, 16, 32, 100, 200, 400, and 1000 SkyMotes was carried out using Contiki Cooja Simulator. This setup was also utilized to validate the orchestration of the services of these devices.

### 5.1.3. Software and Hardware Used

In this section, we briefly describe the software and hardware used while implementing the proof-of-concept prototype.

#### 5.1.3.1. Advanticsys TelosB SkyMote – CM5000

The Advanticsys CM5000 TelosB sensor is a wireless IoT device which has limited processing power and memory, and is IEEE 802.15.4 compliant. It is based on the TelosB platform, which is designed by University of California, Berkeley and is open source. This SkyMote comes with three sensing capabilities, which are temperature sensing (within the range -40 ~ 123.8 ºC), humidity sensing (within the range 0 ~ 100% RH), and light intensity sensing. The capabilities utilized in

our implementation prototype only included temperature and humidity sensing. The sensor contains a USB interface that can be used to connect with the device and program it. The programming language required to program this device is C-like, called NesC. It can be used wirelessly by inserting 2xAA batteries or can be used by powering up through the USB connector. It is compatible with TinyOS and ContikiOS [33]. We used ContikiOS to connect with, and program the device.

Figure 21. The Advanticsys TelosB SkyMote (©Advanticsys™)

### 5.1.3.2. Virtenio Preon32 Shuttle with VariSen Module

The Virtenio Preon32 Shuttle is a radio module with a 32-bit microcontroller and IEEE 802.15.4 compatibility. It contains interfaces such as USB, SPI, $I^2C$, CAN. It does not possess any sensing capabilities but has an expansion module, the VariSen Module, which is the ultimate sensor extension for it. This VariSen Module contains several different sensing capabilities, which are temperature (in the range -40°C to +105°C), humidity (in the range 0 to 100 %RH), illuminance, air pressure, and acceleration. However, only temperature sensing and humidity sensing were used in our prototype. The Preon32 shuttle can be programmed in Java and consists of a JavaVM for embedded sensors developed by Virtenio on it. This JavaVM contains Java libraries that are modified for these sensors and not the same as the standard Java libraries [34,35].

The Virtenio documentation for these Java Libraries is provided along with concrete examples on how to utilize the interfaces and access the sensors. The Virtenio Pren32 Shuttle is more capable compared to the Advanticsys CM5000 and can be battery operated or connected to a power source.


Figure 22. Virtenio Preon32 Shuttle and VariSen Module (©Virtenio™)

### 5.1.3.3. LEGO Mindstorms EV3

We used LEGO Mindstorms EV3 robot as part of the prototype. The EV3 is the third generation of robots in the LEGO Mindstorms series and was released in 2013. The programmable robotics kit of EV3 contains a programmable brick that can be programmed using the original firmware which comes pre-installed on it, or in Java using the LeJOS firmware, which is a replacement to the original firmware and includes a Java Virtual Machine [36]. In our prototype, we installed the leJOS firmware on the brick to program it. The kit includes the brick, three servo motors, color, touch, and IR sensors [37]. The kit contains several pieces that can be assembled to make several types of robots. In our prototype we built the basic EV3 ‘Botticelli’ robot. The robot built in our

prototype in the lab is shown in figure 23. The robot can be powered up using the rechargeable batteries provided in the kit or using 6xAA batteries. It can be connected to the laptop/PC via USB or via Bluetooth. It is easy to program and reusable since the same kit can be utilized to make several robots with different functionalities.

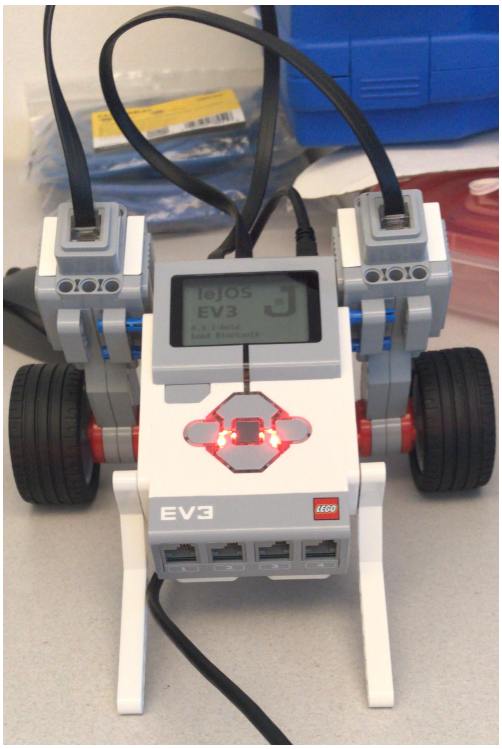


Figure 23. EV3 Robot Built in our Lab for the Proof-of-Concept Prototype from the EV3 LEGO Mindstorms Kit

### 5.1.3.4. Contiki Cooja

Contiki operating system is an open source OS for memory and resource constrained microcontrollers and other IoT devices. It allows efficient applications to be developed for these IoT devices, which can fully utilize the hardware and provide low-power wireless communication [38]. Cooja is a network-simulator application for Contiki OS. It permits the emulation of real IoT devices with special focus on the simulation of wireless sensor networks. It supports several standards such as IEEE 802.15.4, Contiki-RPL, uIPv6 stack, etc. [39]. It can simulate various

sensor motes such as SkyMote, Z1 mote, Wismote mote etc. without using any actual physical motes. We used Contiki Cooja for simulating several Sky Motes to test the scalability performance of the IoT IaaS. In our implementation, we used Instant Contiki with ContikiOS version 3.0. Instant Contiki allows easy installation and usage of ContikiOS since it is a virtual machine having all the required software and toolchains.

#### 5.1.3.5. JVM

A Java Virtual Machine (JVM) makes it possible to run java bytecode on a hardware processor. It is essentially a virtual machine that allows java programs, or programs from other languages compiled into bytecode, to run on machines irrespective of the underlying platforms or operating systems. Thus, it enables interoperability. In our infrastructure, the Virtenio Preon32 Shuttle comes with the PreonVM installed on it, which is a modified version of the JVM for the embedded systems. It allows java bytecode to run on the Virtenio sensor, and includes all data types, several libraries and drivers.

#### 5.1.3.6. Python-Flask

Flask is a microframework for python that does not require specific libraries or tools. Flask makes it easy to add several different types of functionalities to application through the various extensions that it supports. For instance, in our implementation, Flask was used to build REST APIs with ease. The Flask-RESTful extension adds support for building REST APIs quickly. Although, even without this RESTful extension Flask can easily allow building of REST APIs and handling of HTTP requests.

#### 5.1.3.7. Python Requests Library

The python Requests library makes it easy to send HTTP requests easily. Simple methods for ‘get’, ‘post’, ‘put’, ‘delete’ etc. can be utilized for sending requests. Moreover, a response object gets returned for every request, which has data such as status, encoding, etc. [40]. In our implementation, the Requests library was used by the applications to issue requests to the IoT IaaS. It was also utilized by the various components in the architecture to forward requests and thus interact.

#### 5.1.3.8. MySQL

MySQL is one of the most utilized open source relational database management system. It allows the storage of data in a structured manner. SQL is used for sending commands to the database to perform operations such as adding data, modifying data, deleting data, etc., through the MySQL server. Since this comprises of relational databases, all the data is stored in tabular form, with several tables having relationships between them such as pointers, one-to-many, one-to-one etc. MySQL is open source, fast, reliable, and easy to use [41]. Moreover, python’s MySQL driver can allow python programs to have access to and control databases.

#### 5.1.3.9. Programming Languages and IDE Used

For programming the IoT devices, the languages used were Java and nesC. The IDE used was the Eclipse IDE. The leJOS plugin was installed on the Eclipse IDE in order to enable the programming of the EV3 LEGO Mindstorms robot. In order to implement the REST APIs, the applications, and the various components of the architecture, python was used. Primarily, python-flask along with libraries such as Requests, MySQL driver were used. The IDE used for python programming was Visual Studio Code.

## 5.2. Prototype Architecture

The architecture of the prototype is shown in figure 24. As shown in this architecture, the 'Anti-Fire Systems' application and 'Monitoring of Cooling Systems' application are able to utilize the services of the IoT IaaS by sending requests for the provisioning of devices.

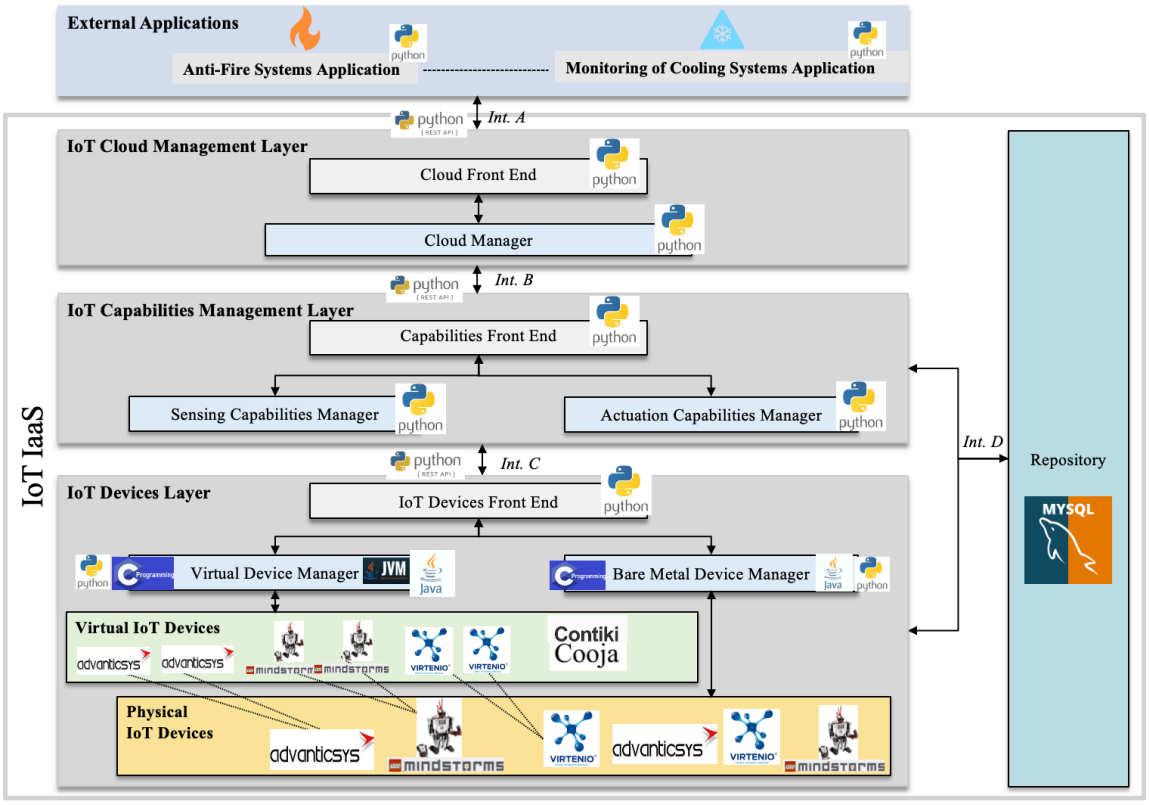


Figure 24. Prototype Architecture for the IoT IaaS

In the subsections to follow, the details of the IoT IaaS implementation prototype are discussed along with a description of the interfaces and the applications. Finally, a short summary of the prototype validation is provided.

### 5.2.1. IoT IaaS Prototype

In this section, we describe each implemented layer of the prototype using the bottom up approach. For each of these layers we discuss the components implemented, or excluded, and how

the implementation was carried out. The *Repository* implementation and the implementation of the applications and interfaces is also discussed.

#### 5.2.1.1. IoT Devices Layer

For the implementation of the prototype, the physical sensing devices used were from two different vendors. These devices were the Virtenio Preon32 Shuttle with the VariSen module and the Advanticsys CM5000 TelosB SkyMote. Both these devices had temperature and humidity sensing capabilities. In addition, Contiki Cooja was used to simulate several TelosB SkyMotes with simulated temperature and humidity sensing. The actuation device used was the LEGO Mindstorms EV3 robot. The virtual sensing and actuation devices were run on top of the physical IoT devices. The simulated sensors in Contiki Cooja also supported running several virtual instances on each simulated sensor.

The Virtenio Preon32 Shuttle with VariSen module was programmed in Java with the help of the modified Java libraries for embedded systems developed by Virtenio. The Advanticsys CM5000 TelosB SkyMote and the simulated SkyMotes on Contiki Cooja were programmed in a C-like language known as nesC. Lastly, the EV3 robot was programmed in the Java language using the leJOS firmware, which was installed on the robot and provided libraries for controlling the robot. The *IoT Devices Front End* was implemented using python-Flask. The components within the *Bare Metal Device Manager* and the *Virtual Device Manager* were all implemented, with the *Device Coordinators* being implemented using python. The *Device Interface Mappers* present within these managers received the command from the *Device Coordinators*, and then handled these requests by directly interacting with the proprietary interfaces of the IoT devices, which included utilizing Java and nesC programs for controlling the devices. The implementation of the *Publication Engine* present within these managers required the use of python and the MySQL

driver of python to interact with the *Repository* and publish information pertaining to the IoT devices on it. For implementing the publishing capability of the *Publication Engine* in a simplified manner, some details such as functionality of the specific device, type of the device were set to predefined values for the specific devices, so that whenever these devices would connect to the IoT IaaS, their information would be added to the *Repository.* For example, when the EV3 robot would connect to the IoT IaaS, its information would be published into the repository with predefined value "moveable" for functionality of device, and predefined value "actuator" for type of device.

#### 5.2.1.2. Interface C (*Int. C*)

The interface for interacting with the *IoT Devices Layer* (*Int. C*) was implemented using the REST architectural style. For virtual devices provisioning, operations for creation, deletion of virtual devices, fetching data, fetching capabilities, fetching and modifying the configurable settings were implemented. In addition, for virtual actuators, extra operations were implemented for fetching the list of actions that the actuator could perform, and for firing actuation action triggers. For bare metal provisioning of devices, operations such as getting list of physical devices available for bare metal provisioning, reserving device for bare metal access, and releasing device from bare metal access were implemented. Python-flask was used for creating the REST API for the implementation of this interface.

#### 5.2.1.3. IoT Capabilities Management Layer

Within the *IoT Capabilities Management Layer*, the *Capabilities Front End,* the *Sensing Capabilities Manager* and *Actuation Capabilities Manager* were all implemented using python. The *Capabilities Front End* was implemented with python's Flask being used to receive the

incoming requests. The *Capabilities Coordinators* within the managers were also implemented using python. Within the *Capabilities Orchestrators,* the *Capabilities Orchestration Plan Generator* was implemented in python and simply generated a json file, instead of a graph, that could later be parsed by the *Orchestration Plan Executor*. The *Capabilities Orchestration Plan Executor* and the *Discovery Engine* were also implemented using python, and the *Discovery Engine* could query the *Repository* using the MySQL driver for python. The Requests library of python was used for issuing requests to the underlying layers by the *Capabilities Orchestration Plan Executor.*

#### 5.2.1.4. Interface B (*Int. B*) and Interface D (*Int. D*)

The implementation of the interface *B* included very basic commands to parse the incoming requests from *IoT Cloud Management Layer* and forward them to the appropriate component in the underlying *IoT Capabilities Management Layer*. It was implemented using python's Flask and Requests library.

Furthermore, Interface D (*Int. D)* did not need to be implemented as a separate API since the implemented publication and discovery entities within the architecture made use of the MySQL driver for python to send requests to the MySQL based *Repository.*

#### 5.2.1.5. IoT Cloud Management Layer

In order to implement the *IoT Cloud Management Layer,* all its components were implemented with the *Orchestration Plan Generator* specifically being implemented in a simplified manner. The *Cloud Front End* was a request processing python program implemented using python's Flask, and Requests libraries. Within the *Cloud Manager,* the *Cloud Coordinator* was also implemented in python. Furthermore, within the *Cloud Orchestrator,* the *Orchestration Plan Generator* python

program gave the output for the orchestration plan as a simple json file, instead of a graph, which could then be parsed by the *Cloud Orchestration Plan Executor.* The *Cloud Orchestration Plan Executor* also used the python-requests library to further send commands to the underlying interface.

#### 5.2.1.6. Interface A (Int. A)

*Interface A* was the interface that exposed the IoT IaaS to external applications. The REST API was implemented using Python-flask for the implementation of this interface. This interface included operations through which the applications could request for the provisioning of sensor and actuator combinations, with given parameters for sensors and actuators, such as service-type, location, sampling rate. It also included operations for deleting the provisioned devices, getting data from the sensors, and specifying the threshold values for sensors and the corresponding actions.

#### 5.2.1.7. Repository

The *Repository* was implemented using MySQL. Both the *Physical IoT Device Repository* and the *Virtual IoT Device Repository* were implemented as MySQL tables in which the records of the physical and virtual IoT devices could be created, queried, updated, or deleted by the different layers of the IoT IaaS. The records in MySQL stored information pertaining to each device, which included, device ID, type of device (sensor/actuator), functionality of device, status of device (idle/busy), action URI (in the case of actuators).

#### 5.2.1.8. Anti-Fire Systems Application

The 'Anti-Fire Systems' application was implemented using python. Since an anti-fire system requires constant and regular monitoring, the sampling rate used was 1 sample/sec for this

application. The python application used the Requests library to initiate the request for temperature sensing, humidity sensing, and robot devices, and displayed distinct messages when it received confirmation of devices provisioning, confirmation of threshold exceeding, and confirmation of actuation action being triggered. The application also constantly pulled the data from the IoT IaaS and displayed it. For testing the application and using it with the IoT IaaS, the threshold for temperature sensing was set to 25°C and for humidity sensing was set to 50%.

#### 5.2.1.9. Monitoring of Cooling Systems Application

The 'Monitoring of Cooling Systems' application was also implemented in python in a similar manner to the 'Anti-Fire Systems' application. However, this application only required temperature and humidity sensing capabilities, and not any actuation capability. The sampling rate for this application was set to 1 sample/second for both the sensing capabilities. The threshold set for trial purposes was 25°C for temperature sensing and 50% for humidity sensing. This application displayed messages for devices provisioning, and also displayed the data being received. Whenever the threshold was exceeded, it displayed an alert message to alert the user of the same.

### 5.2.2. Summary

The prototype implementation allowed the applications to provision several virtual devices on top of the physical IoT devices. This was made possible through the sharing of the same physical devices for running several virtualizations. Thus, this validated *Node Level Virtualization* in the proof-of-concept implementation. Furthermore, in order to implement the publishing of devices' information, some predefined values for the devices, such as functionality, type of device, were added automatically to the *Repository* as soon as the devices were available within the IoT IaaS.

The implemented *Discovery Engine* was also able to query this *Repository* and fetch the information of the appropriate devices as needed. Thus, the *Publication and Discovery* mechanism for finding the appropriate devices and keeping up to date information on each device, was validated. In addition, it was possible for the applications to send requests for provisioning multiple devices, for which the orchestration of the capabilities of several different IoT devices was required. It was handled successfully by the orchestration components of the prototype, which further validated the *Orchestration Mechanism*. Moreover, the *Int. C* included the implementation of an API for the bare metal provisioning of the IoT devices. In addition, the discovery mechanism involved checking the *Repository* for the information of the physical IoT devices matching the capabilities required by the application and available to be reserved for bare metal usage. The *IoT Devices Layer* then handled the interaction with the proprietary interface of the physical device in order to reserve it to be solely used as-is by the application. This validated that the prototype allowed *bare metal access* to the physical IoT devices. Lastly, the infrastructure for the prototype included actuators, in this case the EV3 Mindstorms robot. It was possible to use the robot and control it, for instance, to make it move. The application was able to request for actuators, which could further be accordingly provisioned. This validated the ability of our proof-of-concept to *control and use actuators*. Thus, we can conclude that the architecture for the IoT IaaS proposed in this work is validated by the implemented prototype.

## 5.3. Performance Evaluations

In this section, we first describe the performance metrics to evaluate the performance of the proposed architecture. This is followed by a description of the experimental setups for this evaluation. Finally, we provide an end to this section by an analysis of the obtained results.

### 5.3.1. Performance Metric

Three performance metrics were selected in order to evaluate the performance of our proposed architecture. These were as follows:

**IoT Device Provisioning Delay:** It is measured from the time a request is sent by the application for the provisioning of devices, to the time the acknowledgement message of the devices having been provisioned is received. This includes the time taken for the processing of the application's request, orchestration of the services of several devices, and the creation of virtual devices or reservation of the devices to be used as bare metal. The *IoT Devices Provisioning Delay* is measured for the various individual heterogenous devices available within the infrastructure, as well as for a combination of several devices. The measurements on the individual devices were taken for provisioning virtual devices on these physical devices, as well as for bare metal provisioning of these physical device. In addition, the number of devices requested for provisioning were gradually increased up to 1000 devices to test for scalability.

**Orchestration Delay:** The orchestration delay is measured whenever 2 or more capabilities are requested by the application, in which case, the orchestration of the services of several IoT devices is needed. The *Orchestration Delay* is defined as the time between the orchestrator component receiving the request for orchestration, and the time the acknowledgment message of orchestration completion is received. The *Orchestration Delay* was measured for the orchestration of the services from 2 to up to a 1000 IoT devices. This also allowed testing the scalability of the architecture.

**Sensor Threshold – Actuation Trigger Delay:** The *Sensor Threshold – Actuation Trigger Delay* is defined as the time taken from the detection of the IoT devices' output exceeding the specified threshold, to the time the appropriate actuation action trigger is fired.

### 5.3.2. Experimental Setup

For the experimental setup, each physical IoT device was programmed to run a maximum of 4 virtual devices on top of it. This was because the Advanticsys TelosB SkyMote could only support a maximum of 4 virtual devices running on top of it without running out of memory. Hence, to maintain uniformity, each device in the infrastructure was programmed to support up to 4 virtual instances.

In order to evaluate the first performance metric, the following setups, Setup 1 and Setup 2, were considered.

**Setup 1:** In this setup, virtual device provisioning, as well as bare metal device provisioning was done individually for the following physical IoT devices within the infrastructure: the Advanticsys CM5000 SkyMote, Virtenio Preon32 Shuttle with VariSen module, and the LEGO Mindstorms EV3 Boticelli robot. For each of these individual experiments, the tests were repeated 10 times in order to capture the average delays. In addition, virtual device provisioning, as well as bare metal device provisioning was also done for a combination of sensor and actuator, in this case the Virtenio sensor and the EV3 robot, with the test being repeated 10 times.

**Setup 2:** In this setup, Contiki Cooja simulator was incorporated into the infrastructure to simulate several TelosB SkyMotes. Experiments were conducted where 2, 4, 8, 10, 16, 32, 100, 200, 400, and 1000 virtual devices were provisioned from the infrastructure. For each of these experiments, an average of ten (10) iterations was taken. The number ten (10) was chosen because beyond these ten iterations there was no significant difference in the measurements.

In order to evaluate the second performance metric, the Setup 2 was considered.

For evaluating the third performance metric, the following setup, Setup 3, was considered.

**Setup 3:** In this setup, the EV3 robot and the Virtenio sensor were provisioned in order to detect the delay between the sensor output's threshold detection and the automatic starting of the robot action. Both the devices were provisioned, and a threshold of 25ºC was provided for the temperature sensor. A total of 10 iterations were conducted for this experiment in order to capture the average delay. The number of experiments was arbitrarily chosen to be 10, since after 10 iterations there was no significant difference observed in the measurements.

## 5.3.3. Results and Analysis

In this subsection, the performance results are discussed based on the specified metrics. First, the *IoT Device Provisioning Delay* is analyzed for the setups 1 and 2 in separate subsections. Next, the *Orchestration Delay* is analyzed for setup 2, and finally, the *Sensor Threshold – Actuation Trigger Delay* is analyzed for setup 3.

### 5.3.3.1. IoT Device Provisioning Delay for Setup 1

Figure 25 shows the time taken for creating a virtual device over the Virtenio sensor as well as for its bare metal provisioning over 10 iterations. The average device provisioning delay for creating a virtual sensor on top of the Virtenio sensor is 606.5 ms (milliseconds), while the average delay for the bare metal provisioning of the Virtenio sensor is 338.6 ms. Similarly, the figures 26 and 27 show the time taken for creating a virtual device and the time taken for bare metal provisioning of the Advanticsys sensor and the EV3 Mindstorms robot respectively, taken over 10 iterations. For the Advanticsys sensor, the average time for provisioning a virtual device on top of the physical sensor is 154 ms, whereas the time taken for the bare metal provisioning of the device

is 79.2 ms. The average time taken to create a virtualization on top of the EV3 robot is 518.7 ms, whereas the average time taken for the bare metal provisioning of the robot is 233.2 ms.

It can be observed that for all these physical devices, the time taken to create a virtual instance on top of the physical device is nearly double the time taken to provision the device as bare metal, with a factor of 1.79 for the Virtenio sensor, 1.94 for the Advanticsys sensor, and 2.22 for the EV3 robot.

In the case of the EV3 robot, there was one limitation encountered while creating several virtual devices on top of it. These virtual devices would run successfully on top of the physical robot as long as their actions did not clash, i.e. require access to the same parts of the robot, or overlap. For instance, while trying to run two virtual devices on top of the robot, where both the devices required controlling the motors of the robot to move it for 2 seconds and 4 seconds respectively, it was noticed that the robot would only move for 2 seconds and then stop. However, the messages for both the actions would be displayed. This was because for the first virtualization, the command to stop the robot would come after 2 seconds, while for the second virtualization the command to stop the robot would come after 4 seconds. However, since the 2 actions would overlap, the command to stop the robot after 2 seconds would reach it and control it first, and it would thus remain in the stopped position even when it received another stop command 2 seconds later. However, such issues could be avoided to some extent by handling the edge cases while programming.

In addition, on comparing the Advanticsys CM5000 sensor and the Virtenio Preon32 Shuttle + VariSen Module, it can be observed that for virtualization, the Virtenio sensor takes nearly 3.93 times the time taken by the Advanticsys sensor. In order to provision the devices as bare metal, the Virtenio Sensor takes nearly 4.27 times the time taken by the advanticsys sensor. Thus, on an

average, for both the provisioning operations, the Virtenio sensor takes nearly 4 times more time than the Advanticsys sensor. One reason that can possibly explain this phenomenon is the fact that the Advanticsys sensor makes use of C based libraries for running programs on it, whereas the Virtenio sensor uses the Java based libraries, which causes some overhead leading to additional overall delays. It is a known fact that C based code, in general, runs faster than java as it usually provides better startup performance for machines for which it is compiled. Moreover, Java requires JVM to convert byte code to machine code which causes more delay.

Figure 28 shows the time taken for the virtualization and bare metal provisioning of the Virtenio sensor and EV3 robot combination. The test is repeated 10 times, and it can be observed that the average time taken to create virtual sensor and virtual actuator combination is 1130.9 ms, while the time taken to provision a combination of these devices as bare metal is 593.6 ms. Here as well, the time taken for virtualization is nearly double the time taken for bare metal provisioning, with a factor of 1.9, which aligns with the results obtained above. Moreover, it can be seen that the time taken to provision the combination of the virtenio sensor and the robot is slightly greater, or nearly equal, to the sum of provisioning both the devices individually in most of the iterations, for both virtual device provisioning and bare metal device provisioning. Both these devices, i.e. the Virtenio sensor and the EV3 robot, utilize java-based libraries for running programs, which is why nearly 1 second is taken for their provisioning.

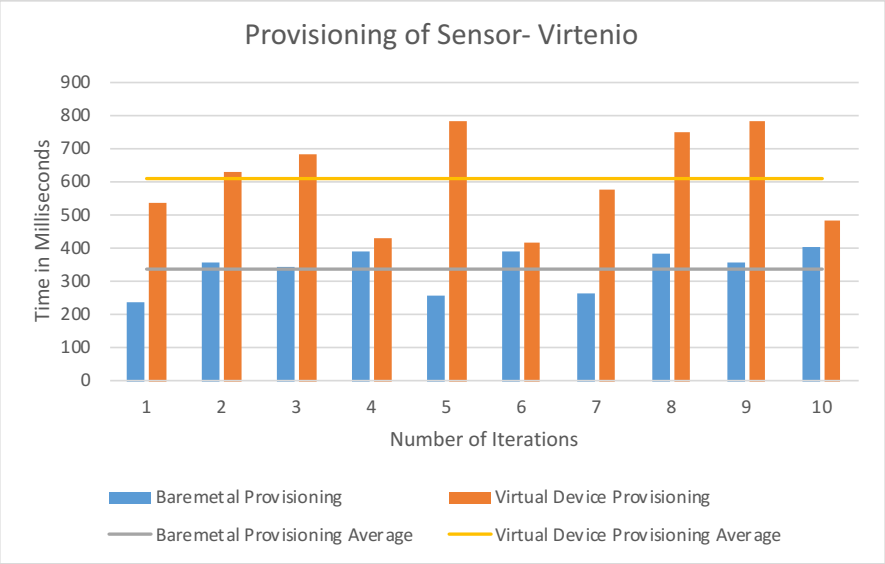


Figure 25. Bare Metal and Virtual Device Provisioning of Virtenio Preon32 Shuttle + VariSen Module

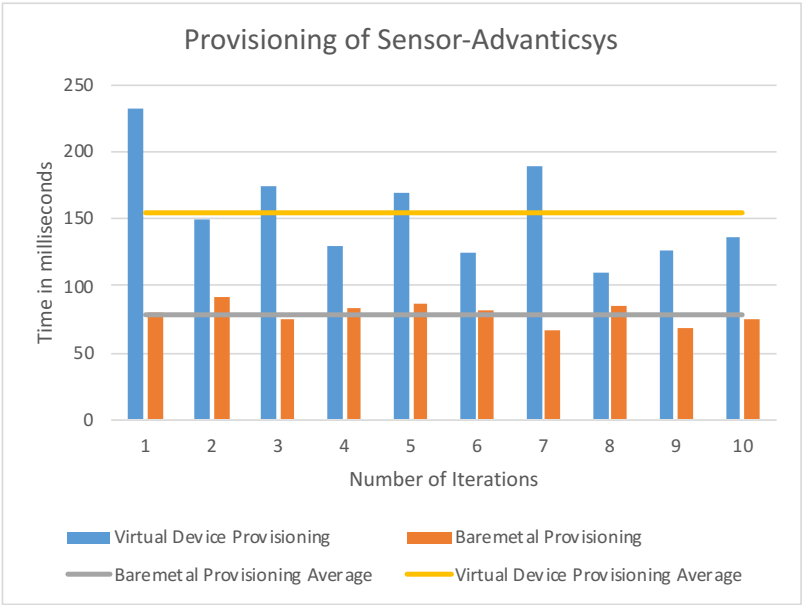


Figure 26. Bare Metal and Virtual Device Provisioning of Advanticsys CM5000 TelosB SkyMote

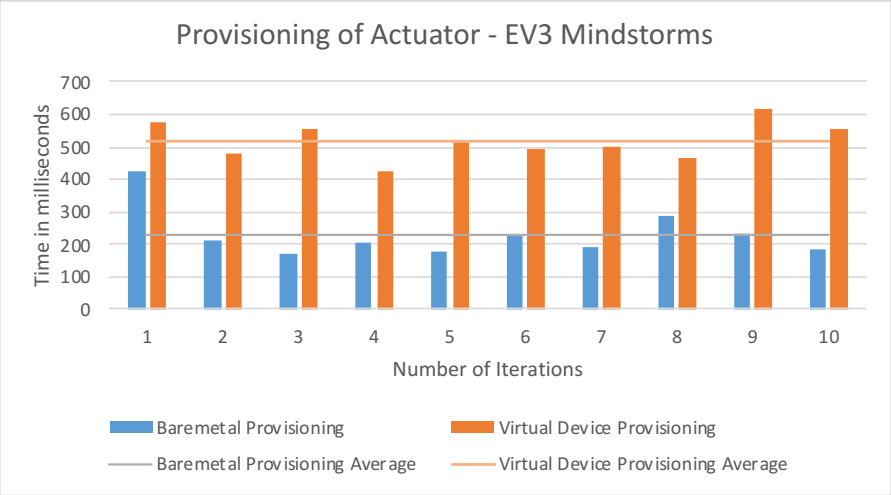


Figure 27. Bare Metal and Virtual Device Provisioning of LEGO EV3 Mindstorms Robot

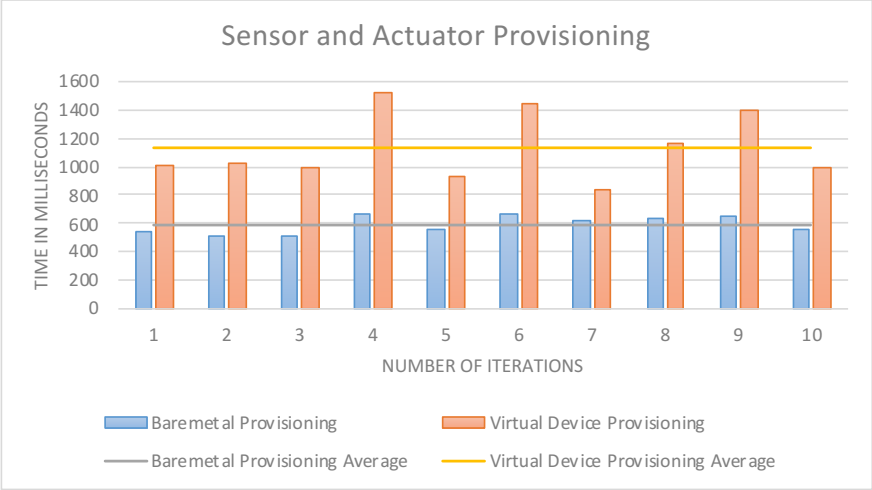


Figure 28. Bare Metal and Virtual Device Provisioning of Virtenio Sensor and EV3 Robot

### 5.3.3.2. IoT Device Provisioning Delay for Setup 2

Figure 29 shows the average delay, in milliseconds, for provisioning 2, 4, 8, 16, 32 devices. Each of these averages were taken over 10 iterations. Similarly, figure 30 shows the average delay, in seconds, for the provisioning of 10, 100, 200, 400, 1000 virtual devices, taken over 10 iterations. In order to provision this large number of devices, Contiki Cooja simulator was utilized for simulating the TelosB SkyMotes.

During the provisioning of 2 devices, the average device provisioning delay was found to be 317.1 ms. For 4 devices it was 777.7 ms, which is slightly greater than double the time taken for two devices, and so on. On observing the device provisioning delays for up to the 1000 devices, it was observed that the delays increased by nearly the same factor as the increase in the number of devices. As another instance, the device provisioning delay for 100 devices was 9.038 seconds, while that for 200 devices was 19.297 seconds, an increase by a factor of 2.1.

This indicates that the IoT device provisioning delay increased almost linearly with the increase in the number of devices to be provisioned. In order to test the scalability of the architecture, the number of devices were increased up to 1000 devices, and the linearity was confirmed by the nearly straight-line graph, as can be seen in the figures 29 and 30.

This is indeed expected in these results. The reason for this is that a majority of the device provisioning time, more than 80%, is spent in the orchestration of the devices, and the processing of the application's request and capabilities handling, as we will observe in the section 5.3.3.3. Much less part of the device provisioning time is actually spent in interacting with the IoT devices' proprietary interfaces. Therefore, the majority of the device provisioning time is not dependent on the type of device being provisioned from the infrastructure. In addition, as the number of devices in the request increase, the implemented prototype would take more time to go over each requested device one by one and accordingly handle the request. As shown in section 5.3.3.3., the orchestration delay also increases linearly when the number of requested devices increase. Thus, unless severe delays are suffered in the interaction with the proprietary interfaces of the IoT devices, the device provisioning would be more or less linear. Moreover, in our experimental setup, all the IoT devices are simulated TelosB SkyMotes in Contiki Cooja, on which virtual instances are provisioned. These TelosB SkyMotes are all of the same type. There can be several

possible cases, such as bare metal provisioning of all the requested devices, virtual provisioning of all devices running on the same type of real physical sensors, virtual provisioning of all devices running on different types of real physical sensors, or some devices being provisioned as bare metal and some being provisioned as virtual devices. However, the explanation given above would apply to all of these cases, and thus, it is expected that the results in all of them would be similar to the ones obtained in our experiment.

Furthermore, the linear scalability depicted by these graphs implies that the performance of the architecture remained consistent with the increase in the number of devices provisioned. The architecture was able to handle the provisioning of the devices without suffering any significant degradation or delays successfully up to 1000 devices.

However, this linear scalability might not prove beneficial for certain applications that require quicker provisioning of devices and have strict start times. More specifically, for provisioning 100 devices, the delay encountered is nearly 9 seconds, and for 1000 devices, it is nearly 1 minute 36 seconds. For instance, for certain real time and large scale applications, such as a ‘Monitoring of Cooling Systems’ in large factories or setups, where hundreds or thousands of temperature sensors are to be provisioned for ensuring that the equipment within that area remains in a cooled environment, provisioning delays as long as 1 minute and 36 seconds can cause significant damage to the equipment by the time the sensors get provisioned.

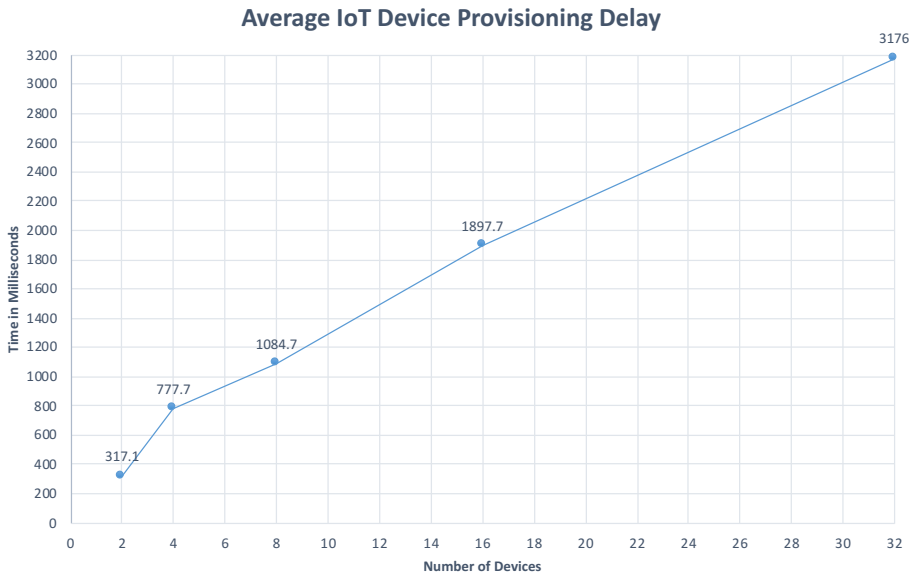


Figure 29. Average IoT Device Provisioning Delay, in milliseconds, for provisioning 2, 4, 8, 16, 32 devices

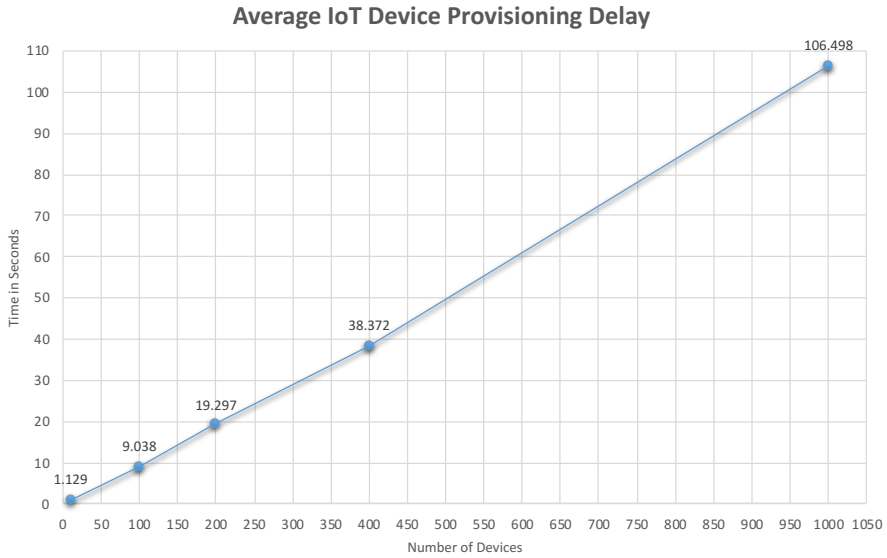


Figure 30. Average IoT Device Provisioning Delay, in seconds, for provisioning 10, 100, 200, 400, 1000 devices

### 5.3.3.3. Orchestration Delay for Setup 2

In order to calculate the orchestration delay, 2, 4, 8, 10, 16, 32, 100, 200, 400, 1000 devices were provisioned, and the time taken by the orchestration component to orchestrate the capabilities of the devices was measured. Figure 31 shows the delay, in milliseconds, experienced in orchestrating the services of 2, 4, 8, 16, and 32 devices, while figure 32 shows the orchestration

delay, in seconds, for 10, 100, 200, 400, and 1000 devices. Both these figures show the average orchestration delay for the different number of devices taken over 10 iterations. On comparing figures 29 and 31 it can be observed that in the case of 2 devices, the total device provisioning delay is 317.1 ms, while the orchestration delay is 254.61 ms, which is nearly 80.29% of the average IoT device provisioning delay. Similarly, the orchestration delay is nearly 77% of the average device provisioning delay for 4 devices; 84.7% of the average device provisioning delay for 8 devices, and so on. In fact, on observing all the delays, it is found that, on an average, the orchestration delay takes nearly 80.7% of the device provisioning delay for the different numbers of devices, up to 1000 devices. This denotes that a majority of the time (nearly 80.7%) is taken by the orchestration components during the provisioning of the IoT devices.

Moreover, the nearly straight graphs in the figures 31 and 32 show that the orchestration delay also increases in a linear manner as the number of devices increase. For example, the orchestration delay increases by a factor of nearly 2.3 as the number of devices double from 2 to 4. Similarly, the orchestration delay increases by a factor of nearly 2.1 as the number of devices double from 100 to 200. Therefore, the orchestration delay increases by nearly the same factor as the increase in the number of devices. This linear scalability shows that the performance of the architecture does not suffer significant degradation for up to 1000 devices, and that it is able to handle their orchestration successfully.

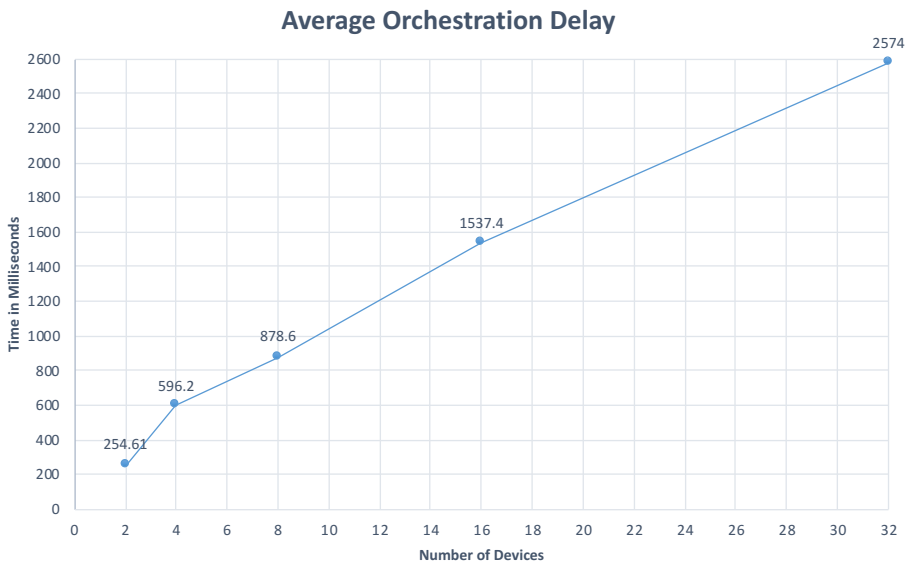


Figure 31. Average Orchestration Delay, in milliseconds, for orchestrating the services of 2, 4, 8, 16, 32 devices

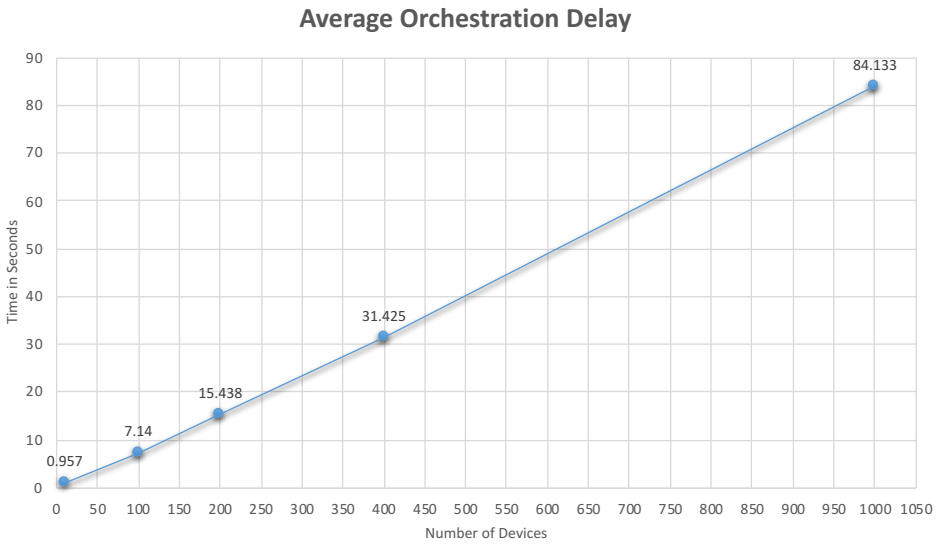


Figure 32. Average Orchestration Delay, in seconds, for orchestrating the services of 10, 100, 200, 400, 1000 devices

### 5.3.3.4. Sensor Threshold – Actuation Trigger Delay for Setup 3

Figure 33 shows the delay, in seconds, between detection of the threshold value of the temperature sensor, and corresponding starting of the actuator action. This test was repeated 10 times, with the minimum time taken being 2.971 seconds, and maximum being 3.37 seconds. The

average time taken over these 10 iterations was 3.203 seconds. In this case, the threshold was assumed to be 25ºC, and every time the sensor gave this output or larger, the command was sent automatically to the robot to start moving. The average delay calculated indicates that the robot starts moving nearly 3.203 seconds after the sensor output is found to exceed the threshold value. This value might not be acceptable for certain time sensitive applications which require quick action, and where even a second of delay can create a significant difference, such as smart patient-health monitoring applications, and so on. However, this delay might be acceptable in applications that can withstand few seconds of delay.

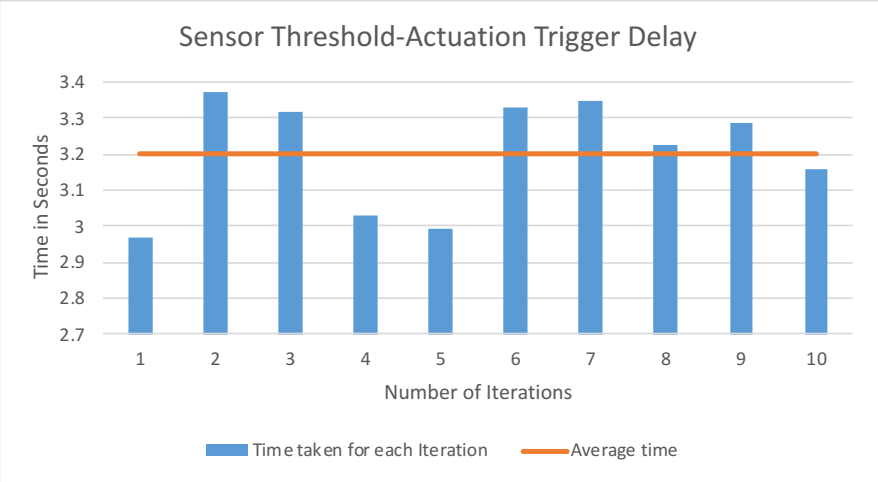


Figure 33. Sensor Threshold – Actuation Trigger Delay, in seconds, measured over 10 iterations

## 5.4. Conclusion

In this chapter, we began by providing an overview of the implemented prototype architecture, after which we gave a detailed description of the prototype architecture. Next, we described the performance evaluations. This included details of the performance metrics used, the experimental setup for performing these evaluations, followed by the results obtained and their discussion.

In the next chapter, we provide a conclusion to this thesis by summarizing it and discussing the future works yet to be done.

# Chapter 6

# Conclusion

In this chapter, we first provide a summary of the contribution of this thesis, and then highlight the possibility of the future research direction.

## 6.1. Contributions Summary

With the ever-increasing demand of IoT in daily life as well as in the industry, most IoT applications aim for cost and energy efficiency, scalability, and minimal latency in terms of resource provisioning, which is somewhat made possible through Cloud Computing. However, the conflicting properties of the cloud and IoT infrastructure, such as the heterogenous and resource constrained nature of the IoT devices, pose many challenges to the successful integration of Cloud computing and IoT. To bridge this gap, it is essential to decouple the IoT device services from the physical IoT devices through virtualization techniques, such as node level virtualization. This would allow the sharing of the capabilities of the IoT devices, thus improving cost efficiency, as well as more flexible and uniform access to the devices. Therefore, there is a need to design and implement an IoT IaaS that not only enables sharing the capabilities of IoT devices to improve costs, but also addresses other challenges that the integration of Cloud computing and IoT face, such as the heterogeneity of the devices, orchestration of the different IoT devices' services, bare metal provisioning of the devices, and publication and discovery of the capabilities of IoT devices.

In order to derive the requirements for the IoT IaaS, we presented a motivating scenario within a 'Smart Factory' environment. This scenario consisted of six applications that required sharing of the capabilities of the same physical devices, which included both sensors and actuators, as well

as bare metal access to the devices. In addition, some applications also required automatic triggering of actuator actions whenever some sensors obtained outputs beyond specified thresholds. This use case, thus, allowed us to derive several requirements for the IoT IaaS, which included node level virtualization as one of the primary requirements in order to enable the sharing of the physical resources. In addition, a publication and discovery mechanism for storing and querying the IoT devices' capabilities information, and an orchestration mechanism to combine the services of different devices, were also considered essential for the realization of the IoT IaaS. Two other critical requirements were also identified, which included having a mechanism to allow the bare metal provisioning of the physical IoT devices and enabling the applications to control and use actuators included in the infrastructure. We then proceeded to evaluate complete architectures for the IoT IaaS, and the models and frameworks that can aid the IoT IaaS, against these derived requirements. None of the works in the state-of-the-art could meet all the requirements.

Next, we proposed an architecture for the IoT IaaS that would meet all of our derived requirements. The architecture relied heavily on node level virtualization, since it enabled several virtual devices to be provisioned on top of a single physical device in order to increase resource utilization and meet the needs of the applications. Furthermore, the publication and discovery entities described within the architecture along with the database *Repository* provided a well-established mechanism to publish and discover the devices and their information as needed. In addition, the architecture also contained orchestration modules, such as the *Cloud Orchestrator* and *Capabilities Orchestrator,* that enabled the provisioning of a proper mechanism for the orchestration of the services of several IoT devices. Moreover, the architecture also contained components and interfaces that allowed the physical IoT devices to be provisioned as bare metal.

All of the devices present within the infrastructure included both sensing and actuation devices. Thus, a mechanism was also presented within the architecture for controlling and using actuators. Finally, the architecture also contained several RESTful interfaces, such as those for provisioning the devices as bare metal, for provisioning virtual sensors and actuators, and a high-level interface for exposing the proposed IoT IaaS to external applications. In the end, we proposed the procedures for the IoT IaaS architecture to depict its functioning.

This was followed by the implementation of a prototype for validating the IoT IaaS. A subset of the motivating 'Smart Factory' scenario was used along with the implementation of a subset of the proposed architecture. The subset of the motivating scenario implemented involved developing two applications, the 'Anti-Fire Systems' application and 'Monitoring of Cooling Systems' application which could utilize the interface for accessing the IoT IaaS.

Finally, three performance metrics were described along with the experimental setups. The results of each of the experiments were graphically depicted and analysed. The feasibility of the architecture was evaluated, as well as its scalability performance through extensive simulations. From the results it was evident that provisioning virtual devices on top of the physical devices took nearly double the time taken to provision the physical devices as bare metal. In addition, on extensive simulations of up to 1000 devices it was found that the IoT device provisioning delay increased almost linearly with the increase in the number of devices, i.e. showed linear scalability. Thus, the performance of the architecture did not suffer significant performance degradation up to 1000 devices. As the number of provisioned devices increased, this mechanism would not satisfy certain real time applications with stricter start times, since for 1000 devices the provisioning delay went up to 1 minute 36 seconds. However, it would still be an appropriate choice for non-real time applications or real-time applications with lenient start times. Furthermore, nearly 80.7% of the

device provisioning time was spent in the orchestration of the services of several devices. Finally, the experimentally calculated average delay experienced between the detection of the threshold values within the sensors' outputs and the corresponding action triggers were acceptable for applications that could withstand few seconds of delay.

## 6.2. Future Research Direction

In this work, while evaluating the scalability of the architecture, no particular algorithms were implemented to enhance the resource efficiency and performance. Future works can incorporate algorithms into the architecture to enable dynamic scaling and optimum utilization of the devices. Furthermore, our architecture did not involve network level virtualization, which can prove beneficial in future works and allow for efficient resource utilization within networks of IoT devices.

In addition, the proposed architecture for the IoT IaaS provided no mechanism for securing the IaaS, which is very critical nowadays. Furthermore, since the proposed architecture in this thesis supports virtualized as well as bare metal access to the IoT devices, security becomes all the more necessary. For instance, if the attackers gain bare metal access to the IoT device, they can directly corrupt IoT devices or render them completely useless, while if they gain virtualized access to the IoT devices, they can potentially corrupt the other virtual instances running on the same device and affect the working of the other applications. Thus, in order to secure the IoT IaaS one approach is to secure the interface which exposes the IoT IaaS to the applications, through techniques such as token based authentication. Moreover, encryption keys and SSL certificates can be incorporated into the IaaS in order to transmit and receive data to/from the IoT devices. Further research in this domain can definitely enhance the IoT IaaS.

Moreover, in our work, it was observed that as the number of devices to be provisioned increased, the time taken to provision them also increased linearly, which went up to nearly 1 minute and 36 seconds for 1000 devices. This might not be feasible for certain applications. Thus, there is a need for a mechanism that can reduce the factor by which the total IoT device provisioning time increases, i.e. as the number of devices increase by a certain factor, the total IoT device provisioning time increases by a reduced/lower factor, also called supra-linear scalability. Although supra-linear scalability is a rare scenario, it does hold significant research potential.

Another limitation of the IoT IaaS proposed in this thesis was that it provisioned only virtual and bare metal IoT devices to the applications but was unable to provision virtual IoT networks or bare metal IoT networks to the applications. However, certain applications may require the provisioning of virtual or bare metal IoT networks. For instance, in applications such as a 'Fire Direction Map' application, a network of IoT temperature and humidity sensors may be needed, which can allow these sensors to communicate with each other and, thus, enable the detection of the fire as well as its direction of propagation within an area. In such a case, the application would require the provisioning of either a bare metal or a virtual IoT network of these sensors. In order to make this happen, there can be two possible research directions. The first is the architectural direction, where specific layers, software modules, and interfaces can be added into the IoT IaaS. A possible challenge in this scenario will be the new interfaces, which will enable applications to request for the creation of virtual or bare metal IoT networks. The second is the algorithmic direction, which would involve the development of specific algorithms to enable efficient resource allocation. Finally, another possible future research direction can involve refining the functioning of the orchestration mechanism, proposed within the IoT IaaS architecture, through the addition of business process models in order to generate improved and optimized orchestration plans. Cloud based Business Process Modeling tools can enhance this possibility.

# References


[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," in *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347-2376, Fourthquarter 2015, doi: 10.1109/COMST.2015.2444095.

[2] Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., & Konwinski, A. et al. (2009). Retrieved 5 September 2020, from https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf

[3] What is IaaS? Infrastructure as a Service | Microsoft Azure. Retrieved 24 July 2020, from https://azure.microsoft.com/en-ca/overview/what-is-iaas/

[4] Botta, A., de Donato, W., Persico, V., & Pescapé, A. (2016). Integration of Cloud computing and Internet of Things: A survey. *Future Generation Computer Systems*, *56*, 684-700. doi: 10.1016/j.future.2015.09.021

[5] Khan, I., Belqasmi, F., Glitho, R., Crespi, N., Morrow, M., & Polakos, P. (2016). Wireless sensor network virtualization: a survey. *IEEE Communications Surveys and Tutorials*, *18*(1), 553–576.

[6] Madakam, S., Ramaswamy, R., & Tripathi, S. (2015). Internet of Things (IoT): A Literature Review. *Journal Of Computer And Communications*, *03*(05), 164-173. doi: 10.4236/jcc.2015.35021

[7] Silverio-Fernández, M., Renukappa, S., & Suresh, S. (2018). What is a smart device? - a conceptualisation within the paradigm of the internet of things. *Visualization In Engineering*, *6*(1). doi: 10.1186/s40327-018-0063-8

[8] R. Elhabyan, W. Shi and M. St-Hilaire, "Coverage protocols for wireless sensor networks: Review and future directions," in *Journal of Communications and Networks*, vol. 21, no. 1, pp. 45-60, Feb. 2019, doi: 10.1109/JCN.2019.000005.

[9] C. Nan, Y. Lee, F. Tila, S. Lee and D. H. Kim, "A Study of Actuator Network Middleware Based on ID for IoT System," *2015 8th International Conference on Grid and Distributed Computing (GDC)*, Jeju, 2015, pp. 17-19, doi: 10.1109/GDC.2015.12.

[10] Whitmore, A., Agarwal, A., & Da Xu, L. (2014). The Internet of Things—A survey of topics and trends. *Information Systems Frontiers*, *17*(2), 261-274. doi: 10.1007/s10796-014-9489-2

[11] Lin, J., Yu, W., Zhang, N., Yang, X., Zhang, H., & Zhao, W. (2017). A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. *IEEE Internet Of Things Journal*, *4*(5), 1125-1142. doi: 10.1109/jiot.2017.2683200

[12] S. Loveland, E. M. Dow, F. LeFevre, D. Beyer and P. F. Chan, "Leveraging virtualization to optimize high-availability system configurations", *IBM Syst. J.*, vol. 47, no. 4, pp. 591-604, 2008.

[13] Hypervisor. (2020). Retrieved 7 August 2020, from https://www.vmware.com/topics/glossary/content/hypervisor

[14] Gouda, K., Patro, A., Dwivedi, D., & Bhat, N. (2014). Virtualization Approaches in Cloud Computing. *International Journal Of Computer Trends And Technology*, *12*(4), 161-166. doi: 10.14445/22312803/ijctt-v12p132

[15] M. Durairaj M., P. Kannan (2014). A Study On Virtualization Techniques And Challenges In Cloud Computing. *International Journal of Scientific & Technology Research, 3*, 147-151.

[16] L. Yan, "Development and application of desktop virtualization technology," *2011 IEEE 3rd International Conference on Communication Software and Networks*, Xi'an, 2011, pp. 326-329, doi: 10.1109/ICCSN.2011.6013725.

[17] I. Khan, F. Z. Errounda, S. Yangui, R. Glitho and N. Crespi, "Getting Virtualized Wireless Sensor Networks' IaaS Ready for PaaS," *2015 International Conference on Distributed Computing in Sensor Systems*, Fortaleza, 2015, pp. 224-229, doi: 10.1109/DCOSS.2015.39.

[18] Rajaraman, V. (2014). Cloud computing. *Resonance*, *19*(3), 242-258. doi: 10.1007/s12045-014-0030-1

[19] Herbst NR, Kounev S, Reussner R (2013) Elasticity in cloud computing: What it is, and what it is not. In: 10th International Conference on Autonomic Computing (ICAC), San Jose, CA, USA, pp 23–27

[20] Zhang, Q., Cheng, L., & Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal Of Internet Services And Applications*, *1*(1), 7-18. doi: 10.1007/s13174-010-0007-6

[21] Moreno-Vozmediano, R., Montero, & Llorente, I. (2012). IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures. *Computer*, *45*(12), 65-72. doi: 10.1109/mc.2012.76

[22] C. G. Kominos, N. Seyvet and K. Vandikas, "Bare-metal, virtual machines and containers in OpenStack," *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, Paris, 2017, pp. 36-43, doi: 10.1109/ICIN.2017.7899247.

[23] Ullah, I., Ahmad, S., Mehmood, F., & Kim, D. (2019). Cloud Based IoT Network Virtualization for Supporting Dynamic Connectivity among Connected Devices. *Electronics*, *8*(7), 742. doi: 10.3390/electronics8070742

[24] Guerreiro, J., Rodrigues, L., & Correia, N. (2019). Resource Allocation Model for Sensor Clouds under the Sensing as a Service Paradigm. *Computers*, *8*(1), 18. doi: 10.3390/computers8010018

[25] Atzori, L., Bellido, J., Bolla, R., Genovese, G., Iera, A., & Jara, A. et al. (2019). SDN&NFV contribution to IoT objects virtualization. *Computer Networks*, *149*, 200-212. doi: 10.1016/j.comnet.2018.11.030

[26] M. Nazmul Alam and R. H. Glitho, "An Infrastructure as a Service for the Internet of Things," *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, Tokyo, 2018, pp. 1-7, doi: 10.1109/CloudNet.2018.8549493.

[27] I. Khan, F. Belqasmi, R. Glitho, N. Crespi, M. Morrow and P. Polakos, "Wireless sensor network virtualization: early architecture and research perspectives," in *IEEE Network*, vol. 29, no. 3, pp. 104-112, May-June 2015, doi: 10.1109/MNET.2015.7113233.

[28] A. Gupta and N. Mukherjee, "Can the Challenges of IOT be Overcome by Virtual Sensors," *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, Exeter, 2017, pp. 584-590, doi: 10.1109/iThings-GreenCom-CPSCom-SmartData.2017.92.

[29] A. Gupta and N. Mukherjee, "Implementation of virtual sensors for building a sensor-cloud environment," *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*, Bangalore, 2016, pp. 1-8, doi: 10.1109/COMSNETS.2016.7439978.

[30] Mattos, D., Velloso, P., & Duarte, O. (2019). An agile and effective network function virtualization infrastructure for the Internet of Things. *Journal Of Internet Services And Applications*, *10*(1). doi: 10.1186/s13174-019-0106-y

[31] A. K. Mandal, A. Cortesi, A. Sarkar and N. Chaki, "Things as a Service: Service model for IoT," *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, Helsinki, Finland, 2019, pp. 1364-1369, doi: 10.1109/INDIN41052.2019.8972241.

[32] K. P. S and V. P. M S, "Software Framework for Wireless Sensor Network Virtualization," *2019 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, Tirunelveli, India, 2019, pp. 1136-1143, doi: 10.1109/ICSSIT46314.2019.8987948.

[33] Retrieved 27 October 2020, from https://www.advanticsys.com/shop/mtmcm5000msp-p-14.html

[34] Ziegler, S. Preon32 - Innovative 2.4 GHz radio module. Retrieved 27 October 2020, from https://www.virtenio.com/en/portfolio-items/preon32/

[35] Ziegler, S. Preon32 Shuttle VariSen - Multi-sensor module for the Preon32 Shuttle. Retrieved 27 October 2020, from https://www.virtenio.com/en/portfolio-items/preon32-shuttle-varisen/

[36] LeJOS. Retrieved 27 October 2020, from https://en.wikipedia.org/wiki/LeJOS

[37] LEGO® MINDSTORMS® EV3 31313 | MINDSTORMS® | Buy online at the Official LEGO® Shop CA. Retrieved 27 October 2020, from https://www.lego.com/en-ca/product/lego-mindstorms-ev3-31313

[38] contiki-os/contiki. Retrieved 27 October 2020, from https://github.com/contiki-os/contiki

[39] Mehmood, T. (2017). COOJA Network Simulator: Exploring the Infinite Possible Ways to Compute the Performance Metrics of IOT Based Smart Devices to Understand the Working of IOT Based Compression & Routing Protocols. *ArXiv, abs/1712.08303*.

[40] Requests: HTTP for Humans™ — Requests 2.24.0 documentation. Retrieved 27 October 2020, from https://requests.readthedocs.io/en/master/

[41] MySQL :: MySQL 8.0 Reference Manual :: 1.2.1 What is MySQL?. Retrieved 27 October 2020, from https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html