An Architecture for an Infrastructure as a Service for the Internet of Things

Mohammad Nazmul Alam

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
Montréal, Québec, Canada

June, 2018

Abstract

Internet of things (IoT) refers to things such as sensors and actuators interacting with each other to reach common goals. It enables multiple applications in sectors ranging from agriculture to health. Nowadays, applications and IoT infrastructure are tightly coupled and this may lead to the deployment of redundant IoT infrastructures, thus, cost inefficiency.

Cloud computing can help in tackling the problem. It is a paradigm to quickly provision configured resources (computing, network, memory) on demand for cost efficiency. It has three layers, the infrastructure as a service (IaaS), the platform as a service (PaaS) and the software as a service (SaaS). Through the IaaS, configured hardware resources (CPU, storage, etc.) are provisioned on demand. However, designing and implementing an IoT IaaS architecture for the provisioning of IoT resource on demand remains very challenging. An example of a challenge is using an appropriate publishing and discovery mechanism suitable for IoT devices. Orchestrating a virtualized IoT device over several physical IoT devices is another challenge that needs to be addressed.

The main contribution of this thesis is twofold. First, a novel IoT IaaS architecture is proposed where IoT devices can be provisioned as a configured infrastructure resource on demand via node virtualization. Second, the architecture is prototyped and evaluated using real-life sensors that support node virtualization. Node level virtualization achieves resource efficiency in contrast to middleware solutions. The essential architectural features, such as publication, discovery, and orchestration are identified and proposed. Two sets of a high-level interface are also introduced. A low-level uniform interface is suggested to decouple the IoT devices from the applications by allowing the applications to access the heterogeneous devices in a uniform way. In addition, a cloud management interface is proposed to expose the IoT IaaS to the cloud consumers (for example - the PaaS, the application, etc.) and allow them to provision the IoT resources.

By allowing the capability sharing of the IoT devices using the node virtualization, the cost efficiency and energy efficiency are achieved in the proposed architecture. Addressing other challenges allowed the proposed architecture to expose the IoT devices to the IaaS in a more abstract manner. Thus allowing the application to provision the IoT resources on demand as well as handling the IoT device heterogeneity in the IaaS.

Acknowledgements

I would like to express my sincere thanks to Dr. Roch Glitho for supervising me. His guidance and direction made my research to stay on track and progress.

I thank Dr. E. Shihab and Dr. Y.-G. Gueheneuc for serving as members of my thesis committee. I also thank Dr. J. Rilling for serving as the chair at my thesis defense.

I also thank my colleagues at Concordia's lab for their help, cooperation, and encouragement. I would like to express my deepest appreciation to Vahid Maleeki Raee for his mental support during this journey. It is my honor to have him as my friend. Thanks to my friends who have shown me unceasing encouragement and support.

Finally, I would like to thank my wife, for her continuous support and encouragement during this journey.

Contents

List of Figures

List of Tables

Acronyms and abbreviations

6LoWPAN IPv6 over LoWPAN.

ANSI American National Standards Institute.

API Application Programming Interface.

ARP Address Resource Protocol.

BLE Bluetooth Low Energy.

BPEL Business Process Execution Language.

COAP Constrained Application Protocol.

CPU Central Processing Unit.

DC Direct Current.

DEA Droplet Execution Agent.

DUT Device Under Test.

ESB Enterprise Service Bus.

FIFO First In First Out.

FTP File Transfer Protocol.

GPS	Global Positioning System.
GUI	Graphical User Interface.
HTTP	HyperText Transfer Protocol.
HVAC	Heating Ventilation and Air Conditioning.
I/O	Input Output.
IaaS	Infrastructure as a Service.
ICMP	Internet Control Message Protocol.
IDE	Integrated Development Environment.
IoT	Internet of Things.
IP	Internet Protocol.
ISM	Industrial, Scientific and Medical radio bands.
JSON	Javascript Object Notation.
JVM	Java Virtual Machine.
LoWPAN	Lo-Powered Personal Area Network.
LTE	Long Term Evolution.
MAC	Media Access Control.
MQTT	Message Queue Telemetry Transport.
NIST	US National Institute of Standards and Technology.
PaaS	Platform as a Service.

PHY	Physical Layer.
QoS	Quality of Service.
REST	Representational State Transfer.
RFID	Radio Frequency Identification.
ROA	Resource Oriented Architecture.
RPi	Raspberry Pi.
SaaS	Software as a Service.
SLA	Service Level Agreement.
SMTP	Simple Mail Transfer Protocol.
SOA	Service Oriented Architecture.
SOAP	Simple Object Access Protocol.
SPI	Serial Programming Interface.
STL	Standard Template Library.
TCP	Transmission Control Protocol.
UART	Universal Asynchronous Receiver Transmitter.
UDP	User Datagram Protocol.
URI	Uniform Resource Identifier.
USB	Universal Serial Bus.
UUID	Universally Unique Identifier.

VM	Virtual Machine.
VMM	Virtual Machine Monitor.
WPAN	Wireless Personal Area Network.
WS	Web Services.
WSDL	Web Services Description Language.
XML	Extensible Markup Language.

Chapter 1

Introduction

1.1 Definition

In this section, we define the key terms associated with the thesis. The definition includes IoT, Cloud Computing, and IaaS. Then, the motivation and the problem statements are discussed followed by the summary of the thesis contributions Finally, we conclude this chapter by describing how rest of the thesis is organized.

1.1.1 Internet of Things

Internet of Things (IoT) refers to things such as wireless sensors, robots, Radio Frequency Identification (RFID) etc. able to interact and cooperate to reach common goals [1]. Generally, IoT devices are small, resource-constrained, battery-operated devices which are able to sense the environment and/or act on the environment. In this sense, IoT is a broad domain containing heterogeneous devices with different capabilities. A wireless sensor is a subset of IoT devices as it can sense the operating environment. On the other hand, a robot is also a subset of IoT devices as it is able to perform a predefined set of tasks on the environment (e,g - fire-fighting robots). There are several IoT devices in the market such as, Virtenio

Preon32, Advanticsys TelosB (SkyMote), Arduino Uno etc. Although Raspberry Pi (RPi) is a small *Personal Computer (PC)*[1], sometimes, it is also considered as IoT device as it can be configured to be an IoT device with additional peripherals.

1.1.2 Cloud Computing

Cloud Computing is a paradigm for swiftly provisioning a shared pool of configurable resources (network, storage, application, services) on-demand. It allows provisioning resources with minimal management effort and on a pay-per-use basis [2]. Since cloud computing allows us to easily access and use virtualized resources, we can adjust provisioned resources dynamically. This means that we can scale with ease which makes optimum resource utilization feasible [3]. It has three facades, Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). The lowest layer of cloud computing is the IaaS. The PaaS sits on top of IaaS and provides a rapid development environment to build and deploy SaaS applications.

1.1.3 IaaS

The capability provided to the consumer (e,g – the application, the PaaS etc.) is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software. The consumer does not manage or control the underlying cloud infrastructure but has control over some resources such as, storage, the deployed applications and possibly limited control over selective networking components (e.g., host firewalls) [2]. In other words, an IaaS provides the entire computing infrastructure as an on demand service, applying pay per use policy. Thus, an application can provision computing, network and other resources as per its requirements without worrying

[1]https://www.raspberrypi.org/about/

about how and where these applications will be deployed. Examples of some popular commercial IaaS are: the Amazon Elastic Compute Cloud (EC2), Microsoft Azure, DigitalOcen, IBM Cloud.

1.2 Motivation and Problem Statement

The cloud computing makes it easy and convenient for the applications to cope with the widespread and distributed nature of heterogeneous IoT device deployment. In early approaches, the applications were generally embedded within the IoT device itself. This yields high coupling between the applications and the IoT devices and that can lead to redundant deployment of the IoT infrastructure, incurring cost inefficiency. Early cloud-based solutions proposed several middle-ware based solutions to address the tight coupling issue. However, they did not explore the possibility of sharing the capabilities of the underlying IoT devices through node level virtualization. Thus the cost inefficiency problem still remained a challenge. In order to solve the coupling issue between the applications and the IoT devices, as well as the cost inefficiency, the IoT devices should be treated just like any other standard resources within the IaaS. This means that they should support virtualization. IoT device virtualization enables the execution of several concurrent applications on top of a same physical IoT device [4].There are already some IoT devices which are capable to be virtualized and commercially available(e,g – Virtenio, Raspberry Pi etc.). However, it is very challenging to design an IoT IaaS which includes both physical and virtual IoT devices. The main reason is the very high level of heterogeneity of IoT devices when it comes to their capabilities, how the capabilities are virtualized, and how the devices communicate with each other and communicate with applications. A very first challenge is the need of high level interfaces to access physical and virtual capabilities. The second challenge is how to publish and discover the capabilities. The third and last challenge is how to orchestrate the IoT devices.

1.3 Thesis Contribution

The thesis contributions are as follows:

- An experiment showing the advantages of node level virtualization over middleware solutions.
- An architecture for IoT IaaS.
- A high-level interface for uniformly accessing the heterogeneous IoTs in the IaaS.
- A mechanism for orchestrating different virtualized IoT in the IaaS.
- A high-level interface for accessing the IoT IaaS.
- A prototype implementation and performance evaluation.

1.4 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 discusses the key concepts related to our research domain in detail.

Chapter 3 introduces the motivating scenario and the set of requirements of the IoT IaaS is derived from the scenario. The state of the art is also evaluated against the requirements.

Chapter 4 presents the proposed architecture for an IoT IaaS. Architectural components and the proposed interfaces are discussed.

Chapter 5 describes the implementation architecture and technologies used for the proof-of-concept prototype. Then the performance measurements evaluating the architecture are presented.

Chapter 6 concludes the thesis by providing a summary of the overall contributions and identifying the future research directions.

Chapter 2

Background

This chapter presents the background concepts relevant to research domain of this thesis. The following concepts are explained: internet of things, virtualization, and cloud computing with a specific emphasis on the infrastructure as a service (IaaS). These concepts are introduced in the upcoming sections.

2.1 Internet Of Things

In this first section, the general definition of the internet of things (IoT) is given. Then it is followed by a brief description of the IoT communication standards, finally followed by the description language used in IoT.

2.1.1 General Defintion of IoT

Internet of Things (IoT) refers to things such as wireless sensors, robots, Radio Frequency Identification (RFID) etc. able to interact and cooperate to reach common goals [1]. It is a very vast, diverse and heterogeneous environment and thus has many challenges when it comes to interoperability within themselves. Generally, IoT devices are compact, battery operated, resource constraint device specialized for performing some specific task (e,g - sensing). However, recent advancement in technology allowed a new serious of miniature, battery operated IoT device to emerge in the market which can support running concurrent applications within the IoT device. For example, the most popular miniature PC platform, the Raspberry Pi (RPi), can also be configured with additional peripheral to act as an IoT device and is capable of running several applications concurrently in isolation. Moreover, it can be configured to run a container-based application as well. Based on the functionality, an IoT device can be classified into two classes, Sensor devices (e,g - Sensors) and Actuation devices (e,g – Robots).

Internet of Things (IoT) refers to things such as wireless sensors, robots, Radio Frequency Identification (RFID) etc. able to interact and cooperate to reach common goals [1]. It is a very vast, diverse and heterogenous environment and thus has many challenges when it comes to interoperability within themselves. Generally, IoT devices are compact, battery operated, resource constraint device specialized for performing some specifc task (e,g - sensing). However, recent advancement in technology allowed a new serious of miniature, battery operated IoT device to emerge in the market which can support running concurrent applications with in the IoT device. For example, the most popular minature PC platform, the Raspberry Pi (RPi), can also be configured with additional peripheral to act as an IoT device and is capable of running several applications concurrently in isolation. Moreover, it can be configured to run container based application as well. Based on the functionality, an IoT device can be classified into two classes, Sensor devices (e,g - Sensors) and Actuation devices (e,g - Robots).

2.1.1.1 Sensors

A Sensor is a device in a wireless network that is capable of performing some processing, gathering sensory information and communicating with other connected

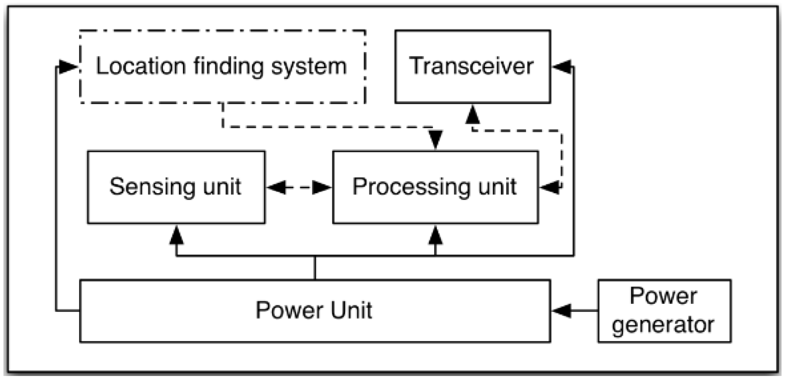


Figure 1: General architecture of a sensor node [4]

devices in the network[1]. Figure - 1 shows the general architecture of a Sensor device/node.

Some examples of Sensors are SkyMote (*TelosB*) by Advanticsys, *Preon32* by Virtenio, Raspberry Pi, Arduino etc.

2.1.1.2 Actuators

An actuator is a mechanism by which a control system acts upon an environment. The control system of the actuator is triggered generally by the other connected systems (e,g - Sensor, external events, hardware interrupts etc.). For example, firefighting robots are dispatched to a particular location of interest to control the fire. This "dispatch" command is generally initiated by a sensing system which detects the fire within that location. Once the control is triggered, the robots act on the fire automatically.

Some examples of Actuators are Lego *MindStrom Robots*, drones etc.

2.1.1.3 Similarities and Differences between Sensors and Actuators

Both Sensor and Actuators can be decomposed into three functional unit, namely Sensing/Actuation, Processing, and Communication. Sensing function sense the

[1]https://en.wikipedia.org/wiki/Sensor_node

environment the Sensor is acting then it processes the data using the Processing unit and finally, it distributes the sensed data to another system using Communication function. While an Actuator, upon receiving a control signal via Communication function, it processes the signal using Processing mechanism, then performs the action on the environment using Actuation function. Although the order of their workflow is different, the Processing and Communication in both Sensors and Actuators can be thought as a generic functionality. Thus the difference between a Sensor and an actuator, from a high level is the order of their workflow and the sensing/actuation functionality.

2.1.2 IoT Communication Standards

IoT communication is generally composed of two types of communication standards. One is the lower layer communication standard, which is essentially MAC/-PHY wireless standards. The other one is the higher layer communication standards, consisting of high-level protocols like COAP, REST, TCP, UDP, 6LoWPAN etc. We describe each of them in brief in the following subsections.

2.1.2.1 Lower Layer Communication Standards

Because IoT devices contains battery operated systems and resource constraint devices, generally the lower layer protocols are geared for energy efficiency. There are several standards which are used by IoT systems. A brief overview on them is described below.

2.1.2.1.1 IEEE 802.15.4: IEEE 802.15.4 is an IEEE standard which defines the physical layer and media access control (PHY/MAC) for low-rate wireless personal area networks (LR-WPANs). IEEE 802.15.4 is suitable for low data rate wireless connectivity among resource constraint devices that consume minimal

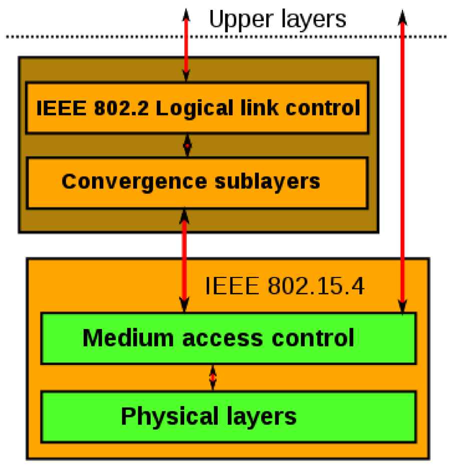


Figure 2: Protocol stack of IEEE 802.15.4

power and typically operate within 10 meters or less. The network topology can simply be a one-hop star, or, a self-configuring multi-hop network if more than 10 meter range is required. An 802.15.4 network can have a bandwidth from 20 kbps to 250 kbps, depending upon the operating ISM band [5]. IEEE 802.15.4 is the basis for the ZigBee specifications, which is essentially an extension over the 802.15.4. The protocol stack of IEEE 802.15.4 is shown in figure - 2.

2.1.2.1.2 Bluetooth Low Energy: The Bluetooth Low Energy (BLE), defined by the Bluetooth Special Interest Group (SIG), running in the same ISM band (2.4 GHz) as the classic Bluetooth, provides the same communication range with a reduced data rate (from 1-3 Mbits to 127 Kbits in BLE) while consuming minimal power (from 1 W to 0.01 - 0.5 W in BLE). BLE addresses the fact that the IoT device will be resource constraint by implementing simple protocol stack for BLE. A BLE powered device with a coin cell 1000 mAh battery, and 1-second advertising interval can last up to 5 years[2].

[2]https://www.aislelabs.com/reports/beacon-guide/

2.1.2.1.3 802.11.x: IEEE 802.11 (abgni) is a set of media access control (MAC) and physical layer (PHY) specifications for implementing wireless local area network (WLAN) computer communication. They operate in the 900 MHz, 2.4, 3.6, 5, and 60 GHz frequency bands. They are the world's most widely used wireless computer networking standards. Their throughput is very high and thus consumes much energy.

2.1.2.1.4 Radio Frequency Identification: Radio-frequency identification (RFID) uses electromagnetic fields to automatically identify and track tags attached to objects. The tags contain electronically-stored information. There are two types of tags, passive tags collect energy from a nearby RFID reader's interrogating radio waves. While, active tags is attached with a power source and may operate over more than 100 meters. It can operate in ISM band and as well as low frequency bands, and high frequency bands. However, the data rate is low and the power consumption is almost negligible. The RFID hardware is one of the most cheapest hardware in the market.

2.1.2.1.5 LoRa: LoRa is a patented digital wireless data communication IoT technology developed by Cycleo of Grenoble, France, and acquired by Semtech in 2012. LoRa uses sub-gigahertz ISM bands (169 - 915 MHz). It features low power operation (around 10 years of battery lifetime), low data rate (27 kbps - 50 kbps) and long communication range (2-5 km in urban areas and 15 km in clear line of sight). The networks topology is a star-of-stars topology, where the gateway nodes acts as a relay between end-devices and a central network server [6]. The main advantage of LoRa is that it allows the bypassing of mobile operator's network, even where other infrastructures are not available (e,g - rural/ underdeveloped/inhabited places).

2.1.2.1.6 Long Term Evolution: Long Term Evolution or LTE, is a communication standard found in telecommunication section. It leverages the W-CDMA or WCDMA (Wideband Code Division Multiple Access), which is an air interface standard found in 3G mobile telecommunications networks. It supports conventional cellular voice, text and MMS services, but can also carry data at high speeds, thus providing internet access. It is used in remote IoT device to leverage the available mobile network. However, it has one of the higher operating cost compare to other communication standards.

2.1.2.1.7 Z-Wave: Z-Wave is a wireless communications protocol developed by keeping home automation as primary goal. It is a mesh network using low-energy radio waves communicating in sub-gigahertz ISM band (915 for North America), to facilitate the communicate among appliances. It allows wireless control of residential appliances and other devices, such as lighting control, security systems, thermostats, windows, locks, swimming pools and garage door openers. It has a low data rate (typically 100 kbps) with low power consumption.

2.1.2.2 Higher Layer Communication Standards

There are several higher layer communication standards available to be used with IoT device. Some of them are described briefly below.

2.1.2.2.1 6LoWPAN The 6LoWPAN is an acronym of IPv6 over Lo-Powered Personal Area Network (LoWPAN). The idea behind 6LoWPAN is that, the Internet Protocol should be applied even to the smallest devices, allowing it to communicate through Internet Protocol [7]. There are some special characteristics of the LoWPANs, such as the use of small packet size, low bandwidth (20 - 250 kbps), a large number of devices, unreliable networks, longer sleep period to conserve energy etc. All of these characteristics are taken into account while designing

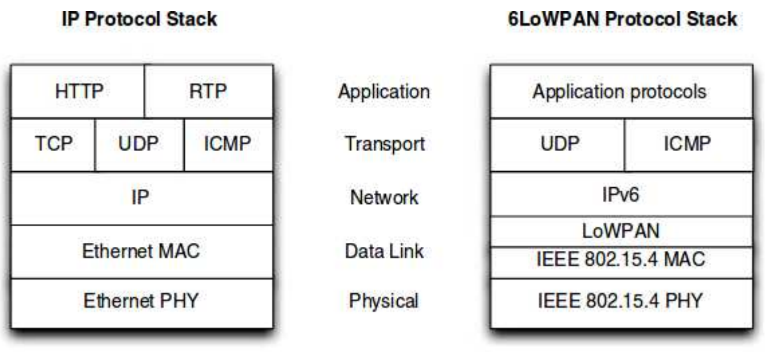


Figure 3: Classical IP Stack Vs. 6LowPAN Protocol Stack

the 6LowPAN [8], and thus it is one of the widely used protocol alongside of the 802.15.4 hardware. Moreover, it was made to be compatible with Internet Protocol (IP) so that it can leverage the existing widely deployed IP infrastructure without any additional effort. The 6LowPAN can also work over classical Bluetooth [9]. In figure - 3, the similarities between classic IP stack and 6LoWPAN protocol stack is shown.

2.1.2.2.2 Constrained Application Protocol The Constrained Application Protocol (CoAP) is a specialized web transfer protocol designed with a goal to be used in constrained nodes and constrained (e.g., low-power, lossy) networks. CoAP provides a request/response interaction model between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URIs and Internet media types. CoAP is designed to be used in constrained environments to allow the devices to use HTTP for integration with the Web [10]. Thus, the CoAP is designed to leverage existing web infrastructure available on top of the existing IP infrastructure. There are some key features of CoAP [10] such as, usage of UDP protocol with optional reliability with unicast and multicast support, support for web protocol (HTTP), asynchronous message exchanges, low processing overhead, support for URI, provision for security etc.

2.1.2.2.3 LoRaWAN LoRaWAN defines the communication protocol and system architecture for the LoRa powered network. This gives the IoT device to communicate with each other over a very long range utilizing very low power. The trade-off is that the data-rate is also very low. Devices in the network transmit data whenever they have something to send. LoRaWAN architecture dictates how the device joins a particular network, subscribes to a certain channel topic and how it can be incorporated with cloud applications [11].

2.1.2.2.4 Message Queue Telemetry Transport The Message Queue Telemetry Transport (MQTT) protocol is an application layer protocol designed for resource-constrained devices, running on top of TCP. Although HTTP also runs on top the TCP, the MQTT enjoys a less overhead than the HTTP. The reliability of messages in MQTT is taken care by three Quality of Service (QoS) levels (QoS 0, QoS 1, and QoS 2). While QoS 0 is the best effort delivery, QoS 1 or QoS 2 guarantees the reliable data transfer [12].

2.1.2.2.5 Micro Internet Protocol The Micro Internet Protocol or uIP is a software stack for connecting with standard TCP/IP stack. It was designed to be suitable for resource constrained system and thus only implements four of the basic protocol in the standard TCP/IP protocol suite (ARP, IP, ICMP, TCP). The code size and RAM requirements of uIP is an order of magnitude smaller than other generic TCP/IP stacks by leveraging the event-d programming model [13]. Application layer protocols such as HTTP, FTP or SMTP can be implemented as an application running on top of uIP.

2.1.3 Standard Description Language

There are a few description language available for IoT devices. They are described in the following text.

2.1.3.1 Sensor Model Language

Sensor Model Language (SensorML) is an Open Geospatial Consortium standard [14]. SensorML provides standard models and an XML encoding for describing sensors and measurement processes. It exposes the sensor as a web resource and provides the endpoint to execute remote functions on the sensors. Therefore, SensorML process models are functional models of a sensor system and related observation data processes [15]. As SensorML can describe work-flows, it can be used just like BPEL for implementing complex sensor system.

2.1.3.2 Sensor Markup Language

The Sensor Markup Language (SenML) is an open standard [16] for representing simple sensor measurements and device parameters using JSON. The standard defines several key attribute for a sensor device. One drawback of SenML is that it is target mainly for transmission of data, instead of describing the sensor itself (e,g - capabilities). SenML is lightweight and is designed targeting the limited capabilities of IoT devices, hence, the devices can easily encode measurements. Parsing SenML encoded data is very easy as it is implemented using JSON, thus making it efficient for the constrained devices [17].

2.2 Virtualization

In this section we provide some background on the key enabler technology for the cloud computing, that is the virtualization. In the following section we start by providing general definition, then move to traditional virtualization. We describe in brief the key concepts behind the traditional virtualization. Then in the following section we focus on the IoT virtualization followed by advantages of it to finish our brief description on the virtualization technology.

2.2.1 General Definition

Virtualization, is the use of an encapsulating software layer that surrounds or underlies an operating system and provides the same inputs, outputs, and behavior that would be expected from physical hardware [18]. The software that performs this is called a Hypervisor, or Virtual Machine Monitor (VMM). This abstraction means that an ideal VMM provides an environment to the software that appears equivalent to the host system, but is decoupled from the hardware state. The major advantage of virtualization is the efficient usage of hardware resources. Through virtualization multiple application running on same hardware is isolated from each other and have the perception of using the hardware exclusively. Thus increasing the overall resource utilization and cost efficiency. And because of these benefits it is one of the key enabler technology on which Cloud computing relies on. There are many types of virtualization available, such as system/node virtualization, network virtualization, database virtualization, storage virtualization etc. In this thesis we are interested in node level virtualization. Hence, our focus is only on system/node virtualization.

2.2.2 Traditional Virtualization

In traditional virtualization the typical resources that are virtualized, in order to provide infrastructure to the users, are computing (CPU), storage, network, memory. These virtualized resources are then put together as a virtual machine (VM). It can also be thought as a logical unit that allows time and resource sharing of host machine by partitioning them into dedicated execution environments [19]. So, typically, a host system (i,e - physical device) contains several VMs. Applications running within a VM has no knowledge of where the VM is placed within a data center. Full Virtualization, Para Virtualization and Hardware Assisted Virtualization are the three possible virtualization methods [20] used for traditional system virtualization. They are described briefly below -

2.2.2.1 Full Virtualization

In full virtualization a guest OS is fully decoupled from the underlying hardware by the virtualization layer. No modification to the guest OS is required in this type of virtualization and hence can be installed above the hypervisor directly. The hypervisor provides hardware resources to each guest OS [21]. However, whenever the guest OS calls a sensitive instruction, the hypervisor traps the instructions and return the proper result via emulation. Full virtualization provides best isolation and security for virtual machines and simplifies migration and portability as the same guest OS instance can run virtualized or on native hardware. Examples of full virtualization products are VMware's virtualization products and Microsoft Virtual Server [20].

2.2.2.2 Para-Virtualization

In Para-Virtualization, the guest OS is modified for the hypervisor. It refers to communication between the guest OS and the hypervisor to improve performance and efficiency [20]. In Para Virtualization the guestOS is modified in order to make hypercalls instead of containing sensitive instructions. Para-Virtualization is much easier to implement than full virtualization, however it has the worst compatibility and portability among the virtualization methods. The open source Xen project is an example of para-virtualization [20].

2.2.2.3 Hardware Assisted Virtualization

Hardware Assisted Virtualization enables efficient full virtualization by using help of hardware capabilities, primarily from the host processors. Privileged and sensitive calls are set to automatically trap to the hypervisor, removing the need for either binary translation or para virtualization. The guest state is stored in Virtual Machine Control Structures (also known as Pages). Processors with these hardware assist features such as Intel VT and AMD-V, can leverage the hardware assisted virtualization and bring best of both worlds, that is - guest OS portability, performance and reduced complexity of hypervisor [20].

2.2.3 IoT Virtualization

IoT virtualization differs from the traditional virtualization. In traditional virtualization, the host system is a general purpose computing system able to run several different application based on users need. In the context of IoT, IoT devices are specific computing device, generally geared towards performing specific functions (e,g - sensing or actuation). Hence, in IoT virtualization of the device means virtualization of its services or capabilities instead of the resources within the IoT device itself. Concretely, it is the sharing of the underlying device's capabilities by allowing execution of multiple concurrent application [22]. The key differences between traditional virtualization and IoT virtualization are listed below -

- The first difference is that a virtual machine (VM) allows the sharing of its resources (e.g., computing and storage) of the host machine, on the other hand, a virtual IoT (vIoT) allows sharing its capabilities (e.g. temperature, light, humidity, firefighting) by executing multiple application tasks concurrently. The key difference is that a VM aims at sharing the host machine resources, whereas a vIoT aims at sharing the capabilities of the host IoT device [22].

- The second difference is that multiple heterogeneous VMs (in terms of operating systems) can be simultaneously deployed on the same host. However, vIoTs are tightly coupled with their OS/middle-ware. For example, a sensor cannot support Contiki based vIoT, TinyOS-based vIOT and/or Lego *MindStorm* based vIoT at the same time. [22]

- The third difference is that in traditional virtualization the VMs are addressed by internet protocol (IP) addresses. This is due to the fact that, IP is dominant connectivity technology in data centers. However, in IoT domain, due to the nature of heterogeneous connectivity technologies, no single connectivity technology dominates. Thus, there is no standard for addressing a vIoT. The general norm is to assign a unique ID to the vIoT for addressing purpose [22].

- The fourth difference is that for a VM, there are no power/energy-related issues, whereas a vIoT inherits these issues from the constrained host IoT device. The always-on/always-available concept is not applicable to the IoT world [22].

- The fifth difference is that for VMs, there are already many open source and proprietary solutions (e.g., KVM and VMware) exists. However, very few such solutions exists in case of vIoT (e,g - JVM based *Preon32*) [22].

- The sixth difference is that for VMs, location is not issue and hence, it is possible to maximize the resource utilization to the fullest. Where for vIoT location is important and thus sometimes it may not be possible to maximize the resource utilization due to the conflicting location requirement from the application [22].

In the context of IoT there are two types of virtualization, Network virtualization and Node virtualization.

2.2.3.1 Network Virtualization

Virtual IoT network is formed by a subset of IoT nodes of an IoT network, with the subset dedicated to a certain task or an application at a given time [23]. With network virtualization, it is possible to create network slices which are owned by a particular application exclusively and thus provides network isolation. In this thesis, we do not consider network virtualization.

Typical architectures for network virtualization is shown in figure - 4(b) and (c).

2.2.3.2 Node virtualization

Node virtualization refers to the concurrent execution of tasks from multiple applications by the same IoT node [24]. It allows multiple applications to run in isolation concurrently in a single physical IoT device. Although, virtualization is common in classical computing node, it is not common in the context of IoT. In classical setting the virtualized node, also known as virtual machine (VM) is a general purpose computing resource that can be configured to perform several tasks. A variety of well known virtualization technique exists which provides the underlying resources from several vendor in an uniform way. However, in IoT setting, the physical device performs a specific sets of tasks and the methodology varies on vendor to vendor. This make it very difficult to virtualize IoT devices. Although there are some IoT device which can provide such virtualization (e,g - *Preon32*) out of box. In this thesis, we focuses on the node virtualization as the key technology to enable sharing of underlying physical IoT resources. General architecture for Node virtualization is shown in figure - 4(a). It is to be noted there are few works done in robot virtualization, [25] is one of such works.

2.2.3.3 Advantage and Disadvantages of Node Virtualization

There are several advantages and disadvantages of node virtualization compare to middleware solutions. Table - 1 lists these advantages and disadvantages.

Focus Point	**Node Virtualization**	**Middleware Solution**
Resource Utilization	Increased resource utilization	possibility of under utilizing the device by running a single task
Transparency from application's point of view	Achieves device transparency. The application are given to exclusive virtual IoT device that they can control or access	Achieves data transparency. The application is given access to data, which they can manipulate.
Contextual Information	Easy to attach meta data (i,e contextual information)	Some sort of processing is required, thus is more complex
Less number of Transmission	Virtual IoT device only transmits when the conditions set by the application is met	Device has to transmit at a fixed interval to allow the middleware to pickup the data, thus incurring higher transmission number
Application's requirement fulfillment	The application requirements are bounded by the underlying IoT device's capabilities	The requirements are bounded by the middleware's feature
Coupling with underlying infrastructure	High coupling	Loose coupling
Applicability	Not applicable to all IoT device	Applicable to all IoT devices

Table 1: Advantages and disadvantages of Node Virtualization

In comparing resource utilization, node virtualization achieves more than the middlewares. In node virtualization, multiple application can run on top of a single device, thus utilizing the IoT device efficiently. Where in the case of middleware, only a single task is running within an IoT device.

The node virtualization gives the application a virtual IoT device, which behaves just like the actual physical device. This gives the application a transparent device view, which it is able to control or access. On the other hand, middleware solution provides only data to the application, the application has no or very limited control over the underlying IoT device.

As the virtual IoT device runs within the actual physical IoT device, it has access to all the meta information related to the physical device. This makes it more convenient to add many contextual information (such as date, location, battery level etc.) to the actual data. This annotation is very useful in attaching context to the data. In case of middleware solution, there is some kind of processing required to annotate the data. This processing is rather complex (such as location information lookup, additional query for battery level etc.) comparing to the node virtualization case.

As we pointed out earlier, the virtual IoT device has the same behavior as the physical IoT device, thus it is possible for the application to express its requirement which will be directly satisfied by the virtual IoT device. For example, having different sample rate, different reporting time etc. can be met by the virtual IoT devices. This leads to efficient transmission of data, as the data is transmitted only when the application's requirements for it are met. On the other hand, to allow the middleware to collect data from the device, the device needs to report data at a fixed interval. And if the application at that given moment is not interested in such data, these transmissions yields inefficiency.

If the application asks for data, for example, every five (5) seconds. And the middleware is capturing data from the device, lets say, every thirty (30) seconds, then middleware cannot satisfy the application requirement. It has to either fail the request or provide staled data. It does not matter even if the device is able to fulfill the application requirement, the requirements are bounded by the middleware's features. On the other hand, node virtualization gives exactly what the application wants. And it is bounded by the device's capability.

For middleware solution, it is easy to add new devices as all it takes is to update the protocol converter, which captures the data from the new device and normalizes them before storing to the database. This provides a fairly loose coupling between the middleware and the IoT device or the infrastructure. However, when using node virtualization, due to the heterogeneity of the IoT devices, the vendor's proprietary control interface should be mapped to the IoT IaaS. This creates high level of coupling between the IoT device or infrastructure and the IoT IaaS.

For node virtualization, it is generally not applicable to all IoT devices, especially to those who operate on ultra low power range (e,g - running on coin cell battery and expected to run several years). However, as the middleware is mostly concerned with data, it is applicable to all types of IoT devices.

- Efficient resource utilization: Each of the applications running within the physical IoT device can perform different task and thus it has no different from. running several physical IoT devices. This allows the efficient usage of the IoT device.

- Cost efficiency: Because same physical IoT can be virtualized to provide several virtualized IoT device on top of it, the need for several IoT device within a defined environment is reduced, thus leading to cost efficiency.

- Transparency: The virtualized IoT devices are transparent to the IoT application deployed within the cloud and appear as the actual physical IoT device to the application.

- Contextual Information: it is natural to add contextual information to the data at the time of the data creation [26]. As virtualized applications are running within the actual IoT device, they have the access to all the information available to the actual physical IoT device. This gives an easy way to annotate data or attach contextual information to the data.

- Less number of Transmission: The virtualized IoT devices executes the applications as per the IoT applications requirements. Where in the middle-ware virtualization technique, the data is replicated to the IoT applications deployed in the cloud. In the latter way, the underlying device has to send data at every fixed interval, leading to possible redundant transmission. Where in the former way the virtualized application knows when to send the data. This leads to less number of transmission overall [26].

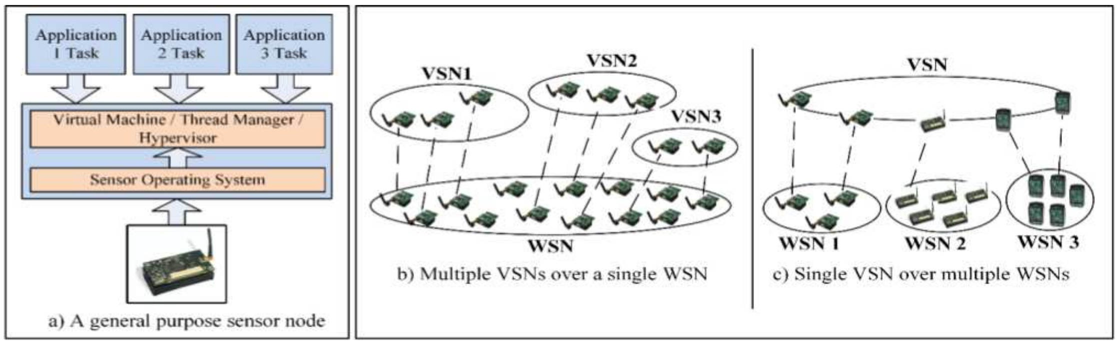

Application 1 Task
Application 2 Task
Application 3 Task
Virtual Machine / Thread Manager / Hypervisor
Sensor Operating System
a) A general purpose sensor node
VSN1
VSN2
VSN3
WSN
b) Multiple VSNs over a single WSN
VSN
WSN 1
WSN 2
WSN 3
c) Single VSN over multiple WSNs

Figure 4: General architecture used for IoT virtualization [4]

- Energy efficiency: Later, it will be shown that Node Level Virtualization in fact also achieves energy efficiency in contrast to running several physical IoT devices.

2.3 Cloud Computing

In this section, we present a general overview of Cloud Computing. We start with its definition followed by a specific focus on the Infrastructure as a Service (IaaS). The IaaS is discussed further in brief. Finally, the description is concluded giving the types of cloud and the advantages of using it.

2.3.1 Definition

Cloud computing has been defined in several ways. NIST (US National Institute of Standards and Technology) defines it as "model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [2]. Another way of thinking cloud computing as a "large pool of easily usable and accessible virtualized resources that can be dynamically reconfigured to adjust to a variable load (scale), allowing for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the infrastructure provider by means of customized SLAs" [3]. The definition provided by the NIST [2] covers all essential characteristics of Cloud Computing and hence widely accepted as the definition of Cloud Computing.

There are three layers in Cloud computing, namely SaaS, PaaS, and the IaaS.

Software as a Service (SaaS) is the highest layer in the cloud. In this layer, a software vendor can offer a hosted set of software (running on a platform and infrastructure) that the user does not have to own but rather pay for some element of utilization [27]. Examples of SaaS are Google Docs, Salesforce etc.

The PaaS is defined as an enabler for the service providers to develop and deploy their services onto the cloud without worrying about underlying infrastructure [2]. It also acts as an abstraction level on top of virtualized infrastructure, provisioning resources on demand during execution of running services [3]. Examples of PaaS are Microsoft Azure, Cloud Foundry etc.

Infrastructure as a Service (IaaS) is described in details in the following sections.

2.3.2 IaaS

In this section, we provide a definition of IaaS, then we discuss the layers within an IaaS briefly. Finally, we conclude the description on IaaS by providing some examples of IaaS.

2.3.2.1 Definition

The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications [2]. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

2.3.2.2 Layers within an IaaS

The general architecture for an IaaS can be shown in figure - 5 [28]. It shows the different layers within a typical classical IaaS. The IaaS layers are described in brief in the following sections.

2.3.2.2.1 Physical Layer: Although not shown in figure - 5, it is the lowest layer in an IaaS. It contains the actual physical devices. In a traditional IaaS, this layer consists the blade server racks, storage racks, network switch racks etc. within the data centers.

2.3.2.2.2 Virtual Machine Managers: The Virtual Machine Manager (VMM) is also widely known as the Hypervisor. Different hypervisors, installed in the physical devices, are contained within this layer (figure - 5(d)). The hypervisors provides vendor-specific simple substrate, such as create, delete, suspend to manipulate VMs in a single physical device. However, as the hypervisor differs, so does the syntax of their substrate and thus arise the need for an uniform interface to hide the vendor specific interface. Some example of hypervisors are VMware, Xen, KVM etc.

2.3.2.2.3 Virtual Infrastructure Management: The underlying infrastructure consists several VM managers in the lower layer. So, to manage the infrastructure, the virtual infrastructure management layer provides primitives to schedule and manage VMs across several host. As shown in figure - 5 (c), an IaaS might not have VI management layer and instead can provide the functionality from the cloud management layer directly. There are some proprietary and open source VI Manager available. For example, VMWare VSphere can only manage and provide virtual infrastructure made up with VMWare hypervisors. While, OpenNebula is capable of managing different hypervisors through Adapters (i,e - an uniform interface).

2.3.2.2.4 Cloud Management: Cloud management provides the mechanism to the user to create, control, and monitor virtualized resources in the IaaS. In order to do so, a cloud interface is required (shown as the orange box in figure - 5(b)). Additional to managing the virtualized resources, it can also some time provide the functionality required at the *Virtual Infrastructure Management* layer. Meaning, it can directly provide the primitives to schedule and manage VMs across several hosts, and thus provide the virtual infrastructure management capabilities.

2.3.2.2.5 Cloud Consumers: The cloud users situated in this layer. They use the cloud interface provided by the layer below and provision virtualized resource on demand. The typical IaaS users can be - individual users, the application itself, other IaaS providers, and other PaaS.

2.3.2.3 Examples of IaaS

Some examples of IaaS are Amazon Elastic Compute Cloud (EC2), Microsoft Azure, IBM Cloud, Rackspace, Google Compute Engine etc.

2.3.3 Types of Cloud

Based on who owns and uses the cloud, it can be classified as private cloud, public cloud, and hybrid cloud [29] [30]. These are described in brief in the following

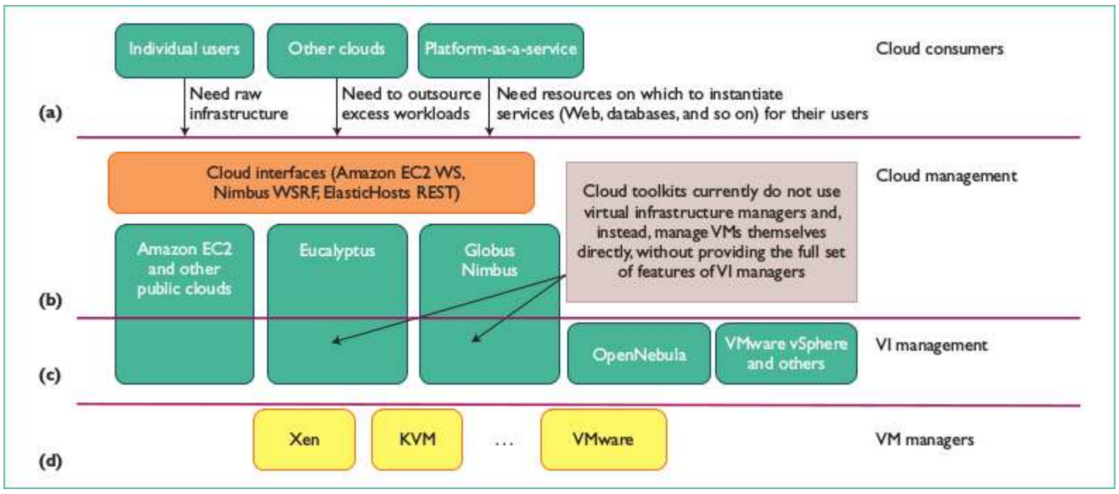

Individual users
Other clouds
Platform-as-a-service
Cloud consumers
(a)
Need raw infrastructure
Need to outsource excess workloads
Need resources on which to instantiate services (Web, databases, and so on) for their users
Cloud interfaces (Amazon EC2 WS, Nimbus WSRF, ElasticHosts REST)
Cloud management
Cloud toolkits currently do not use virtual infrastructure managers and, instead, manage VMs themselves directly, without providing the full set of features of VI managers
Amazon EC2 and other public clouds
Eucalyptus
Globus Nimbus
(b)
(c)
OpenNebula
VMware vSphere and others
VI management
Xen
KVM
...
VMware
VM managers
(d)

Figure 5: A typical architecture for IaaS [28]

sections.

2.3.3.1 Private Cloud

A private cloud is generally own and used by a specific organization. It is not open for public usage. It allows the employees with the organization to interact with the local data centers while having the same advantages of the cloud. This type of clouds provides performance, reliability and security [29].

2.3.3.2 Public Cloud

The clouds that is available for the general users as pay-per-use manner, typically expressed in hours, months, year, or a long contract. It is usually owned by big corporations such as Amazon, Google, or Microsoft. This type of clouds lacks some control over data, network and security settings, however has full control over the deployed application itself [29].

2.3.3.3 Hybrid Cloud

A hybrid cloud is combination of public, and private cloud, thus, combining the advantages of both of the world. It also allows cloud bursting to take place, which means a private cloud can burst-out to a public cloud when it requires more resources [30]. The main benefit of hybrid clouds is that it provides more flexibility than both public and private clouds [29].

2.3.4 Benefits of Cloud Computing

Cloud Computing offers several important benefits. They are:

- Scalability: Virtually unlimited scalability is possible because of the massive capacity offered by the cloud providers [31]. Services hosted on the cloud can be easily scaled which is very useful in the event of rapid service demand change. Typically the IaaS can support both vertical and horizontal scaling. For example, if a service deployed on the IaaS requires more memory, it can perform vertical scaling by providing additional memory to already deployed instance. However, if a service needs more memory due to excessive traffic load and thus it is better to load balance the traffic, horizontal scaling can be also done by deploying additional instances of the service.

- Elasticity: It refers to a system's capability of adapting to variable workload by provisioning and de-provisioning resources in an autonomic manner [32]. The IaaS, depending on the provider, can provide elasticity to a service. On the event on high work-load, it can automatically allocate more resource, in order to allow the service to execute as expected even under high work-load. On the other hand, once this sudden work-load is gone, the additional resource that was allocated can be de-provisioned to save cost.

- Reliability: Services running on the cloud should meet several desired requirements such as Quality of Service (QoS), availability, performance, fault tolerance, etc. These requirements are regulated under the framework of Service Level Agreement (SLA) between cloud service providers and customers. SLAs contain the details of the service as well as the penalty for violations [31].

- Multi-tenancy: Cloud providers can serve multiple customers by assigning and reassigning the virtualized and physical resources dynamically according to demand. It facilitates resource sharing resulting in optimum resource utilization and cost.

- On-demand self-service: Customers can provision cloud resources any time without human interaction with the cloud service providers [30]. This is enabled by exposing the IaaS to the Web in two folds. The first one is the exposing the IaaS as a programmable interface (also known as Application Programming Interface - API). The next one is to develop web interface utilizing the programmable web interface. A concrete example is the Amazon Web Services. It is backed by REST API endpoints (termed as AWS CLI), a programmable interface. Leveraging the programmable interface, a web GUI is also available.

- Pay-per-use Model: Customers are charged only for the amount of resources they consumed. This measurement parameter can vary based on the services offered. For instance, usage of a virtual machine (of a particular configuration) per hour, number of users consuming a service, etc. [29].

- Easy access: Customers can easily access provisioned resources over network through various types of devices.

2.4 Conclusion

In this chapter, we focused on the major technologies and concepts that are relevant to this thesis. The chapter was started by focusing on the Internet of Things (IoT).

A brief discussion on the IoT covering the definition, the communication standards, and the description language was given. Then we focused on the Virtualization covering the definition, the traditional virtualization, and the IoT virtualization. A brief overview of the cloud computing with its definition and the IaaS was given following the virtualization section. A general overview of the IaaS architecture was given. The types of cloud and its advantages described in brief before concluding the chapter.

In the next chapter, we present a motivating use case and derive the requirement from it. The state of the arts is then, evaluated against the derived requirement to see how well they fulfill the requirements.

Chapter 3

Use case and State of the art

In order to capture the requirements of an IoT IaaS, a motivating use case is first presented, then the requirements are derived from it. Finally, we evaluate and summarize the current state of the arts against the requirements and draw conclusion.

3.1 Use Case

Consider a “fire detection and notification” application (Anti-Fire) and a “smart heating ventilation and air conditioning” (Smart HVAC) application running in a smart home environment. The goal of the Anti-Fire application is to detect fire by sensing environment temperature and notify the inhabitants, whereas the goal of the Smart HVAC application is to provide a comfortable living atmosphere while maintaining the energy-consumption as low as possible. For the Anti-Fire application only one sensing capability is required, the temperature of the environment. On the other hand, for the Smart HVAC application, two sensing capabilities and two actuation capabilities are needed. By monitoring the environment's relative humidity and temperature, the Smart HVAC application can make a decision whether to start AC or to start heating, and thus optimize the energy savings.

The traditional way of deploying of the Anti-Fire application is to have one physical sensor deployed in the operating environment and the sensed data is pushed to the application. On the other hand, to deploy the Smart HVAC application, one physical sensor with two capabilities (temperature and humidity sensing) and one actuator with two capabilities (AC and Heating) would be required. In the traditional setting, thus we would require in total of two physical sensor and one actuator (considering it provides both heating and cooling capabilities). However, this is not an efficient approach because it will require a redundant deployment of temperature sensing capability. A shared sensing is a more efficient approach where the sensed data or capability is shared among the applications. This can be approached in two ways, one by using middle-ware solution within an IaaS. In this way, we essentially allow the sharing of sensed data among multiple applications. Most of the middle-ware based solution focuses on this "data- centric view of the IoT devices". On the other hand, using node level virtualization, the physical IoT device's sensing capabilities are shared among the applications. In the middle-ware based solution, the data is captured from a sensor and then replicated back to both of the application. There is some drawback of using middle-ware based solutions. First, they do not ensure efficient usage of the IoT device. Second, they fail to provide resources if the requirements can not be met by middle-ware but can be satisfied by the IoT device (e,g - two application having different sampling rate for a given sensor). In order to overcome the second issue, many middle-ware based solutions (e,g - [33] etc.) proposes data routing algorithms where a task within the IoT IaaS is responsible for dispatching the data to the application and thus can be viewed as a virtual sensor. However, if the underlying physical IoT device is deployed with certain parameters, the middle-ware based solution must rely on them and have to supply the staled data to the requesting application using the data routing algorithm. A concrete example is, if the physical IoT sensor has been programmed to send data every five (5) seconds to the middle-ware and the requirement of the application is to monitor the data every two (2) seconds, which is basically an arbitrary value less than the original sampling rate, the data routing algorithm has to provide the staled data to the application. The third issue with such solutions is the data normalization. As the middle-ware is tightly coupled with heterogeneous IoT devices, it has to normalize the data acquired from different vendor's IoT device. The fourth issue is that they will be energy inefficient. All of the middle-ware based solution relies on the fact that the underlying IoT device will be sending data at a fixed interval, even though, the data might be of no interest to the application. It will be filtered at the "virtual sensor" component within the IoT IaaS. Finally, even if one application does not require all of the sensing capabilities of the underlying IoT device, the IoT device must be programmed to send all the sensed capabilities to the middle-ware. This is because, even if one capability is not programmed to report to middle-ware, then there will be no way to active it later without performing sensor reprogramming, which can lead to downtime of currently deployed applications.

In the light of the above discussion, it is clear that IoT device level virtualization is indeed more efficient in terms of resource utilization [4]. This thesis focuses on the latter.

3.2 Requirements

For the Anti-Fire, the cloud IoT application requires an IoT device in the smart home environment to sense the temperature of the environment and report it back to the cloud IoT application. And for the smart HVAC cloud application, it requires an IoT device capable of sensing the temperature, and humidity as well as control the temperature of the environment by heating or cooling. An efficient approach for provisioning the IoT application in the physical IoT device is to have two concurrent application deployed within the physical IoT device. This is because both applications have a subset of overlapping capabilities requirements and thus it is more efficient to use node level virtualization to create two virtual IoT device on top of the physical IoT device.

Hence, our first requirement is the sharing of sensing capabilities via node virtualization due to its efficiency in resource utilization.

Now considering that we have node level virtualization to create virtual IoT devices, we still need to know if there is a physical IoT device within the IoT IaaS which can fulfill the cloud applications requirements. In order to solve this issue, the information regarding the physical IoT device must be available to the IoT IaaS. Publication is a popular mechanism to allow such information to be published and stored in a database (commonly termed as a repository). Another mechanism, the discovery, is required to match the supplied requirements against the supported capabilities and parameters to find out the physical IoT device which can fulfill the cloud IoT application's requirements and virtualize it.

So, our second requirement is the need of mechanisms for the publication and discovery of the capabilities offered by IoT devices.

Now for the two application, the smart HVAC application may be difficult to deploy as a single IoT device. This is because it consists both sensing and actuation task and the underlying IoT device may be of one type. For example, within the IoT IaaS, there may be a sensing device and an AC, and a heater. In order to provision such application, all of the IoT devices must be virtualized and orchestrated to make such application possible to deploy over the IoT IaaS.

Based on the above discussion, our the third requirement is an orchestration mechanism.

In order to address the heterogeneous devices at the physical layer, a common technique is to use adapters. The Adapter is a component which maps the proprietary interface with a common unified interface and thus solves the issue of handling heterogeneous. However, we still need to define a high-level interface for the Adapter component, such that it can support the primitives (create, delete) and maps that on to the proprietary interface of the vendor hardware.

Finally, in order to access and provision a virtual IoT device, the cloud user must be provided with a high-level interface. This interface is required to expose the IoT IaaS to the users (e,g - application, the PaaS). This interface must be designed so that it can support the creation, deletion and orchestration mechanism transparently within the IoT IaaS.

Finally, the fourth and last requirement is the need for two sets of high-level interfaces. One set of high-level interfaces are required for accessing and managing the heterogeneous IoT nodes in a uniform manner and another set of a high-level interface is required for interacting with the IoT IaaS users (e.g. PaaS).

It is to be noted that, although security is a major concern in IoT domain, this thesis does not cover the security aspect. However, there are several layers of security available ranging from symmetric key cryptography (PHY/MAC) to asymmetric key cryptography (application protocols) based on key exchange protocols [34].

3.3 State of the Art

In the subsequent sections, we first evaluate the current state of arts and draw summary from it focusing on full-fledged IoT IaaS that were proposed in the state of art. Then, we evaluate and summarize the proposed IoT frameworks in the current state of art. It is to be noted that, some of the requirements were not covered by any of the proposed IoT IaaS so far. Hence, these requirements are discussed in a section of its own.

3.3.1 State of the Art of the Proposed IoT IaaS

There are a few full-fledged IoT IaaS, that was proposed in the state of the art. The IoT IaaS are the SenIaaS [35], the Cloud4Sens [36], and the Early Architecture for WSN Virtualization [22]. We evaluate each of the full-fledged IoT IaaS focusing on the derived requirement in the following sections.

3.3.1.1 SenIaaS

In [35], which is an extension to [37] and [33], the authors presented an IoT IaaS architecture which relies on virtualization technologies according to their claim. The novelty of the proposed architecture is the presentation of a software component which exposes the remote sensor as a native resource within a VM. The VM is deployed within the IaaS, just like traditional IaaS. Hence, the architecture proposes an IoT IaaS extending upon the traditional IaaS. In order to extend the IaaS, the architecture proposes a modified *Compute Node* architecture that includes additional component to allow the exposure of the remote sensor as a software component within the VM. Also, they claimed that they use a novel routing algorithm to route the remote sensor data from the sensor to appropriate software component to achieve efficiency. As several *Compute Node* can request the data from different remote sensor, thus the algorithm routes the data to the appropriate remote sensor accordingly. In the proposed architecture, the remote sensor device is sending the sensed data to a component named *Listener Broker*. The main functionality of the *Listener Broker* is to route the sensed data to appropriate *Listener* component. Each of the *Listener* component is receives data from a remote sensor. Hence the mapping between the *Listener* and the remote sensor is 1:1. However, each remote sensor can provide data to several *Listener* components. Thus making it a 1:m mapping with remote sensor and *Listener*. Each of the *Compute Node* has its exclusive virtualized *Listener* component. Hence, the author identified the *Listener* component as primary point of virtualization (PPoV), as it is the logical representation of remote physical sensor within the IaaS. Within a classical VM the remote sensor representation (i,e - the *Listener* component) is exposed as a local resource by the means of a software component. This software component is identified as point of virtualization (PoV) by the authors. There is a 1:1 mapping with PPoV and PoV by the means of a component named *Jumper*. The *Jumper* essentially routes the data from the *Listener* component to the software component within the deployed VM. The application deployed within the VM can use the exposed software component within the VM to read the remote sensor just like local resource within the VM. The SenIaaS uses POSIX I/O standard interfaces for communicating with the remote sensors, which author claimed as virtual sensors. The authors uses POSIX read(), write() API to access the software component within the VM which represents the PPoV within the VM. A read() operation on the software component within the VM triggers a read operation on the *Listener* component which is bridged with the software component via the *Jumper*. The read operation on the *Listener* allows it to push the data to software component with the help of *Jumper*.

In the following sub sections, we evaluate the SenIaaS IoT IaaS in contrast to the derived requirements.

3.3.1.1.1 Sharing Sensing Capabilities via Node Virtualization Although the authors claimed that they have proposed their architecture utilizes node level virtualization, we argue that, it is still a middle-ware based solution. From the previous discussion of SenIaaS, it is clear that, the proposed architecture uses a middle-ware software component to present a *"virtual sensor"* within the IaaS. The *Listener*, the *Jumper* and the software component within the VM contributes all together to provide a notion of *"virtual sensor"*. It is clear that the remote sensors are not virtualized at all, rather the data received from them are duplicated and routed to the appropriate VM with the help of the aforementioned software components. The proposed architecture does not use node level virtualization. Thus, it does not yields cost efficiency at the IaaS level as well as it does not meet our first requirement.

3.3.1.1.2 Publication and Discovery of IoT Capabilities The proposed SenIaaS do not tackle the issue of publication and discovery. Also, it does not describe what and how the description language are used. So, we conclude that, it does not met our second requirement.

3.3.1.1.3 Interface for Accessing and Managing the Heterogeneous IoT Nodes in a Uniform Manner The proposed architecture utilizes POSIX I/O standard API for accessing the *"virtual sensors"*. However, POSIX I/O is a pure data acquisition interface which does not cover the control part. Moreover, the inherent meaning of the data interface depends on the application itself as no standard is enforced on the content of the POSIX data API. Finally, the authors do not describe how the primitives such as create, delete virtual sensor, will work on the proposed architecture. Hence, we conclude that it does not provide the interface for accessing and/or managing the underlying physical IoT devices. Finally, in SenIaaS no cloud access interface is proposed.

It is clear that it does not describe how the uniform interface is provided by the SenIaaS architecture.

3.3.1.2 Cloud4Sens

In [36], the authors proposed a cloud architecture, the Cloud4Sens, with a goal to leverage both the data centric view of the IoT device and as well as the device centric view of the device. By the data centric view, the authors meant the middleware based solution where instead of virtualizing the underlying device itself, the data produced by it is replicated and distributed. In this view, the application do not have any knowledge of how the data was originally produced or by whom. The application generally does not have any control over the underlying IoT infrastructure. On the other hand, it also tries to leverage the device centric view of the IoT device. In this view, the authors meant the usage of virtualized sensor. The advantage of this view is the application can exert exclusive control over the virtualized IoT device and have full control over it, just like using any other physical IoT device. The proposed architecture contains a software component, the *Adapter*, which captures the data from remote sensing infrastructure. It is a common technique to use the *Adapter* to provide uniform interface for accessing heterogeneous devices. However, in this case, it acts as a protocol converter. The Adapter component receives the data in different format from different sensing infrastructure and normalize them to a common format. This normalized data is then stored in a database, so that the data can be later transfer to the application, if requested by the application. The proposed architecture uses Sensor Web Enablement (SWE) abstraction layer to expose the virtualized infrastructure. The SWE abstraction layer (SAL) is compliant with the OGC-SWE standard, and includes a set of XML-based languages and Web service interface specifications to facilitate the discovery, exchanging, and processing of sensor observations. The SWE standard is based on WSDL and thus follows a SOA architectural style. The SWE abstraction layer (SAL) provides a consistent interface for accessing the data from the underlying sensing infrastructure to the IaaS or the PaaS. It is used in conjunction with XMPP to provide an interface to the application, and/or the user. XMPP is a protocol suitable for event dissemination and presence detection. The Cloud4Sens is able to act as a middle-ware based solution (or the data centric view of IoT device) by using a database to store the data and distribute it later. By using the *Adapter*, the author claims that their architecture uses the virtualization techniques (i,e - node level virtualization) to virtualize the whole sensing infrastructure to the application. This way, the author claims that the Cloud4Sens supports the device centric view.

In the following sections, we evaluate the Cloud4Sens against the derived requirements.

3.3.1.2.1 Sharing Sensing Capabilities via Node Virtualization In the device centric view of the Cloud4Sens, the authors proposed a way to virtualize the whole IoT infrastructure using Sensor Web Enablement (SWE) abstraction layer. This SWE abstraction layer (SAL) is a software component based solution within the IaaS.

Thus, this proposed work does not meet our first requirement, as it does not use node level virtualization.

3.3.1.2.2 Publication and Discovery of IoT Capabilities In the proposed architecture the SAL relies on XML-based language and WSDL to expose the functionality of the underlying IoT infrastructure. It follows a SOA architectural style. While this is acceptable for the proposed architecture as they are performing the publication and discovery from the middle-ware. But, this methodology is not efficient for resource constrained devices. Moreover, the author does not describe any concrete discovery mechanism. Hence, although the proposed architecture do contains intention in the right direction, it fails to fulfill our second requirement.

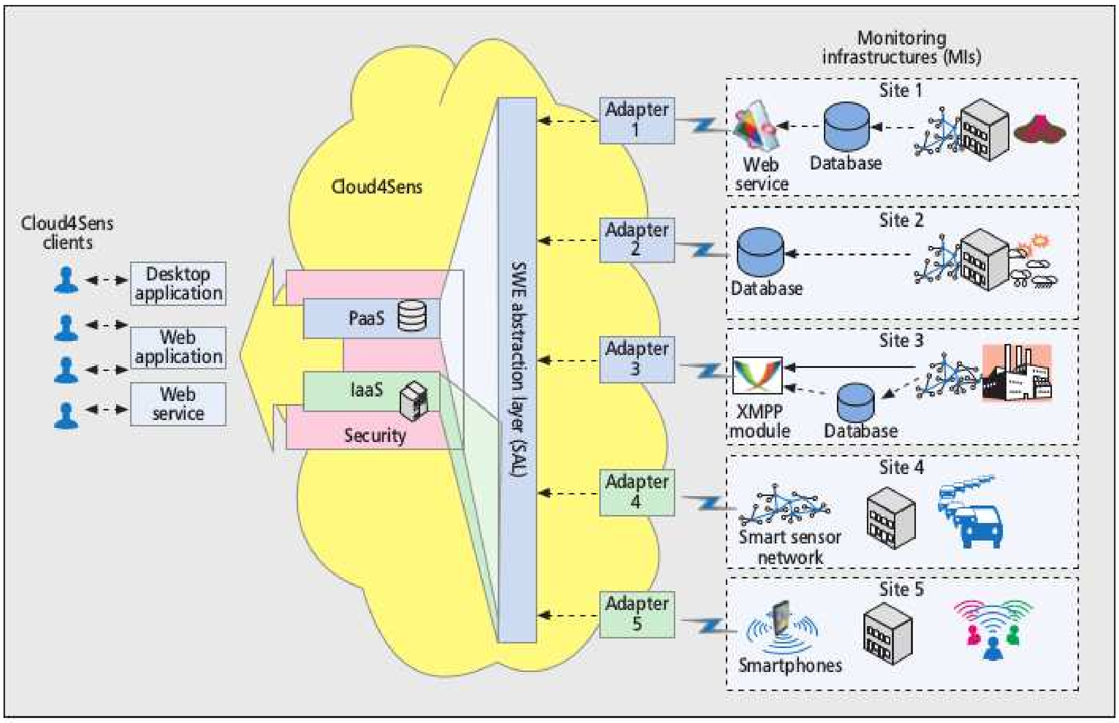


Figure 6: An architecture for providing virtualized IoT infrastructure

3.3.1.2.3 Interface for Accessing and Managing the Heterogeneous IoT Nodes in a Uniform Manner Although the Cloud4Sens uses XMPP for event handling and presence management, it is not clear how the virtualized IoT devices are managed or accessed from the user or the application. The proposed architecture does not tackle this issue at all.

Hence, it does not met our fourth requirement as well.

From the above discussion it is clear that the proposed Cloud4Sens do not met any of our derived requirements completely.

3.3.2 An Early Architecture for WSN Virtualization

The proposed architecture in [4] provides a means to provide virtualized WSN to the application. The architecture follows the same architectural layering as in the traditional IaaS, as pointed out by [28]. The proposed architecture is consists of four layers - the *Physical Layer*, the *Virtual Sensor Layer*, the *Virtual Sensor Access Layer*, and the *Overlay* Layer. The author categorized the underlying IoT devices based on three roles, namely, *Type A* sensors, *Type B* sensors, and the *Gateway To Overlay (GTO)* node. The *Type A* sensors are resource constrained IoT device and not capable of performing the function of *GTO*. The *Type A* devices are also not capable of performing node level virtualization. The *Type B* sensors are more capable sensors which can perform the *GTO* translation by themselves. The *GTO* is a function which provides a translation mechanism, so that the less capable IoT device can be exposed to the application overlay. This function is only available on a capable IoT device (i,e - *Type B* devices) and used by less capable IoT devices (i,e - *Type A* devices). The *Type A*, and the *Type B* devices reside in the *Physical Layer*. Various information regarding the physical devices are stored in the repository. So, that it can be matched later with the application's requirements. The authors express that somehow, the out of band communication (OOB) with the underlying physical sensor and the repository, allow the sensor devices to publish these information to the repository. The author uses SenML combined with JSON as the description language. The proposed architecture use node level virtualization on the capable IoT device. The virtualized IoT devices are logically presented in the *Virtual Sensor Layer*. The *Virtual Sensor Access Layer* only contains the *Sensor Agent* component. It exposes the virtualized and non virtualized IoT device situated within the *Physical Layer* and the *Virtual Sensor Layer* to the upper layer, the *Overlay Layer*. The main goal of this *Sensor Agent* is to map a set of uniform data interface D_i to the underlying proprietary interface PD_i and map another set of uniform control interface C_i to the underlying proprietary control interface PC_i. Hence, the sensor agent is responsible for exposing the underlying devices to the application using two sets of interfaces. Finally, the *Overlay Layer* consists of the independent application overlay. This layer is able to serve the requesting application a dedicated application overlay for accessing the underlying IoT device. The application can provision a new overlay network and thus achieve virtualized WSN. The WSN virtualization is created with the *virtual sensor*s from the *Virtual Sensor Layer* via the *Sensor Agent*.

In the following subsequent section, we evaluate the architecture against the requirements.

3.3.2.0.1 Sharing Sensing Capabilities via Node Virtualization [4] is one of the few examples that meet our first requirement. In this proposed architecture, the node level virtualization was achieved by reprogramming the wireless sensor network and thus reconfigure them to run concurrent applications.

3.3.2.0.2 Publication and Discovery of IoT Capabilities In the proposed architecture, the authors used OOB communication for facilitating the publishing of information regarding the physical device. Although this is very close to meeting our requirement, the authors did not go into details how the publication will work within the IoT IaaS and assumes that the publication of IoT devices will be already completed when they are deployed in the IoT IaaS. Hence, we conclude that it does not meet with our second requirement completely.

The author uses SenML combined with JSON by [4], which meets our goal for using lightweight description language for IoT devices. Hence, it did meet the requirement for using description language suitable for IoT device.

3.3.2.0.3 Interface for Accessing and Managing the Heterogeneous IoT Nodes in a Uniform Manner In the proposed architecture, two sets of uniform interface is proposed for hiding the underlying heterogeneous nature of IoT devices. One set is for accessing the data and another set is for managing the device itself. Although this work is the most well defined and complete in terms of defining a uniform interface for the physical IoT device compared to other works, it does not define the primitives [28], that is, the minimal set of operations required for creating, and deleting virtual IoT nodes. Moreover, how the virtual IoT device is created is not defined in the proposed architecture. Hence, we conclude that, it also did not meet our fourth requirement.

3.3.3 Summary of the State of the Art of the Proposed IoT IaaS

Table - 2 shows the summary of meeting the requirements against the proposed architecture. A "✓" means that the corresponding requirement was meet by the proposed architecture. A "*" means that the proposed architecture did not meet the requirement at all.

Requirements	State of the Arts					
	SenIaaS			Cloud4Sens	WSN Virtualization Architecture	
	[35]	[37]	[33]	[36]	[22]	[4]
Node Level Virtualization	*	*	*	*	✓	✓
Publishing Mechanism	*	*	*	*	*	*
Discovery Mechanism	*	*	*	*	*	✓
Description Language	*	*	*	*	*	✓
Orchestration Mechanism	*	*	*	*	*	*
Minimum Uniform Interface	*	*	*	*	*	*
Cloud Access Interface	*	*	*	*	*	*

Table 2: Current State of the arts of the proposed IoT IaaS fulfilling the derived requirements

3.3.4 State of the Art of the Framework that can be used in IoT IaaS

Many works on IoT focuses on the framework which provides a specific solution to a problem. Some of these frameworks are designed with IoT IaaS in focus and aligned with our requirements. We evaluated two such frameworks. One is the "Framework for increasing dependability of IoT device using virtualization" proposed in [38]. And the other framework focuses solely on how the publication and discovery of the IoT devices can be facilitated in an "A RESTful Framework for Web of Things" proposed in [39].

We evaluate each of the IoT frameworks focusing on the derived requirement in the following sections. It is to be noted that the frameworks are not evaluated against the node level virtualization as they are proposed as a framework and not a full-fledged IoT IaaS. The framework may be able to use within an IoT IaaS.

3.3.4.1 Framework for Increasing Dependability using Virtualization

In [38], the author proposes a framework for the sensor to increase the dependability of the IoT application. First, it is assumed by the framework that the dependencies of the IoT application are known at before deploying the IoT application. This way, it is possible for the framework to map the dependencies of the application to a redundancy model. A redundancy model dictates how the redundancy of underlying IoT device is handled. Taking all of these into account, the proposed framework uses the redundancy model and replaces a failed *Virtual Service* with another suitable *Virtual Service*. The novelty of the work is that it can sustain up to a certain level of failure of IoT device within the IaaS efficiently by utilizing virtualization. It is assumed by the framework that the capabilities of IoT device will be somehow present within a database. The redundancy model knows which are the compatible underlying IoT device and hence can switch from a *Virtual Service* to another one if the current IoT node fails. The overall mechanism is controlled by the *Virtual Service Manager* within the framework. The framework can sustain 40% physical node failure with more than 90% recoverability. However, after 40% the framework cannot successfully recover and hence the recoverability goes down as the shortage of underlying operational physical IoT device is slowly approached.

In the upcoming sections, we evaluate the framework against the requirements.

3.3.4.1.1 Publication and Discovery of IoT capabilities In the framework, it is assumed that the capabilities of IoT device will be somehow present within a database. And the redundancy model somehow knows which are the underlying physical devices that can replace each other by fulfilling the application's requirement. Hence, the framework does not focus on the publication and discovery mechanism. Thus, we conclude that the framework does not meet the second requirement.

3.3.4.1.2 Interface for Accessing and Managing the Heterogeneous IoT Nodes in a Uniform Manner The framework increases dependability of IoT applicable by switching to new virtualized node when one node fails. But it does not describe how this switch is made or how the new virtualized nodes are created or accessed. Hence, we conclude this does not met our fourth requirement as well.

3.3.4.2 A RESTful Framework for Web of Things

In [39] the authors focus on the issue of exposing IoT node as a web service. The work focuses on the exposing of a *Virtual Sensor* properties using URI resources, and mapping them in a RESTful framework. This involves modeling of a *Virtual Sensor* as a connected graph of properties. This graph contains all the unique resource identifier (URI) relative to the actual virtual sensor itself. This gives a very good way to model a virtual sensor in RESTful paradigm. For discovering a resource using URI, the framework leverages regex, a well-known pattern matching technique, to match a set of URI or sub URI. Hence, the novelty of this work is it applies a systematic way with the help of the regex and the RESTful paradigm to access the underlying *Virtual Sensor*. Several separate tools were developed by the authors (Javascript and java libraries) to access the *Virtual Sensor*. These tools provides the application programming interface (API) to the application. The APIs provides a way to access the *Virtual Sensor* properties in a transparent and easy way from the application by leveraging the framework. The work assumes the *Virtual Sensor* is a middle-ware component within IaaS.

3.3.4.2.1 Publication and Discovery of IoT capabilities The modeling of the resources exposed by the virtual sensor using the RESTful paradigm is well done. However, instead of addressing how the publication and discovery of such information will take place, the author resorts to developing separate libraries in Java and Javascript programming language. This leads to "do-it-yourself" RESTful approach with no provision for interoperability [40], as the mechanism for publishing and discovery not defined but the resource endpoint is well defined. Hence, we conclude it does not fully met our second requirement.

3.3.5 Summary of the State of the Art of the Frameworks that can be used in IoT IaaS

Table - 3 shows the summary of meeting the requirements against the proposed frameworks. A "✓" means that the corresponding requirement was met by the proposed framework. A "*" means that the framework failed to met the requirement at all.

Requirements	State of the Arts	
	Framework For Increasing Dependability using Virtualization	A RESTful Framework for Web of Things
	[38]	[40]
Publishing Mechanism	*	*
Discovery Mechanism	*	*
Description Language	*	*
Orchestration Mechanism	*	*
Minimum Uniform Interface	*	*
Cloud Access Interface	*	*

Table 3: Current State of the arts of the proposed framework against the derived requirements

3.4 On the issue of "Interface for accessing IoT IaaS from end user"

To our best of knowledge, there is almost no literature focuses on the high-level interface for exposing the IoT IaaS to the end user (e,g - application and PaaS). This is partly because most of the proposed IoT IaaS focuses on how the data should be disseminated to the application. And thus automatically assumes that the underlying infrastructure will somehow be already deployed for the application. However, in reality, there should be a cloud interface as pointed in the general architecture of a classical IaaS (figure - 5. Which can be used by different cloud users to provision the infrastructure on demand. Moreover, the interface should be flexible enough to allow the user to define virtual IoT device spanning over several physical IoT device transparently. For example, if the user's application requirements can only be met by combining services from two or more physical IoT device, they will be virtualized and orchestrated to complete the virtual IoT device. While, if they can be met by single physical IoT device, only it will be virtualized and no orchestration will be required. However, from the application's point of view, there will be no difference between the physical and virtual IoT device, and it will be as transparent as like accessing a physical IoT device with the given capabilities.

We emphasize the fact that, there is a gap in the literature focusing on this particular requirement.

3.5 On the issue of "Orchestration"

There is significant work done for the event-driven orchestration in classical computation. But the challenge becomes hard to tackle with the existing model-driven orchestration techniques due to the fact that IoT devices are resource constrained. Most of them (e,g - [41] and [42]) leverage technologies such as simple object access protocol (SOAP), web services description language (WSDL), enterprise service bus (ESB) which are not lightweight. And thus, not practical for resource-constrained IoT device.

Moreover, the existing business process aware orchestration mechanisms are not geared toward supporting IoT applications as pinpointed in [43]. The popular orchestration mechanism used in traditional clouds are Software Containerization, Reverse Local Proxy and Resource Offering [44]. Software Containerization is similar to container technology. In this methodology, all of the orchestration component are bundled within a single package and then deployed on the cloud. However, as pointed out earlier in the motivating use case, it may be the case that the application is not possible to deploy in a single IoT device, hence, this technique is not suitable for IoT IaaS. In the Reverse Local Proxy, exposes an endpoint so the interested party can connect to that endpoint. And that endpoint may be reverse proxied to actual service. To make reverse local proxy to work, the endpoint must be available to all server. Given that information, it is very complex and hard to incorporate reverse local proxy in an IoT context as the capabilities are not exposed as listening service, instead they are serving service. Finally, the resource offering technique is used for scaling the VM in traditional IaaS. This is not applicable for IoT IaaS. Generally speaking, none of the Orchestration patterns like Software Containerization, Reverse Local Proxy and Resource Offering has been designed with key functionality like IoT event management/handling in mind, thus they fails to fulfill the requirement.

Hence, to our best of knowledge, this is one of the under-looked research topics in the domain of IoT.

3.6 Conclusion

In this section, we first presented a motivating use case. Using that use-case we derived the requirements for the IoT IaaS. We then reviewed the current state of art against these derived requirements. None of the state of art was able to fulfill all of the requirements. To our surprise, we were unable to find literature which focuses on the issue of orchestration and cloud interfaces. Finally, we presented two summary tables showing which of the requirements were fulfilled by which state of the art.

In the next chapter, we present our proposed architecture, describe the associated components, and their functionality in details.

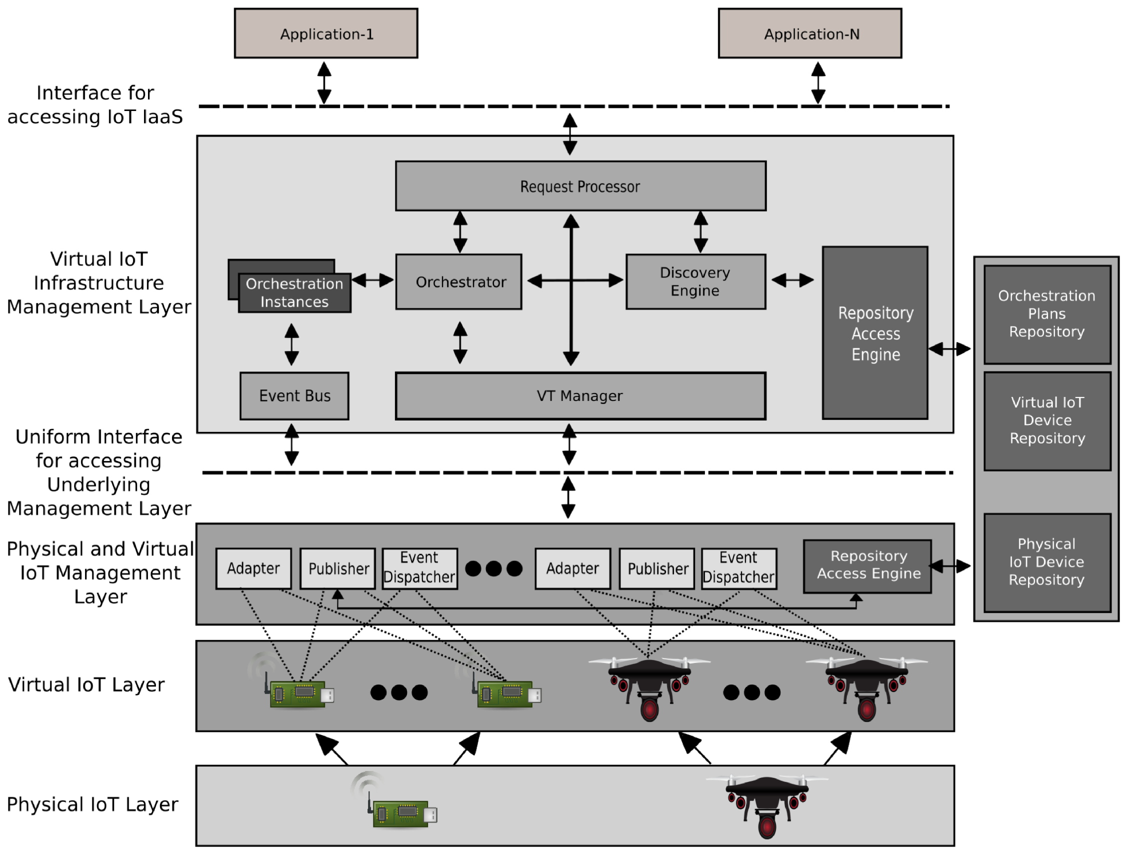
Chapter 4

An IaaS Architecture for the IoT

In this chapter, we focus on our proposed IoT IaaS architecture. First, we provide a general description of our proposed architecture. We then focus on a layer by layer taking a bottom-up approach. First, we cover the Physical IoT Layer, then slowing moving up we finish by covering the *Repository.* The two sets of the interface are also discussed along the way. The lower layer uniform interface and the cloud access interface are discussed according to the order they appear in the architecture diagram. The interaction between various components within the IoT IaaS is shown in brief focusing the motivating use-case presented in chapter - 3. We conclude the chapter by showing how the proposed architecture meets the requirements.

4.1 General Overview

The proposed architecture is shown in figure - 7. In the proposed architecture, the *Physical IoT Layer* is the lowest layer. It consists of all the physical IoT devices. On top of this layer is the *Virtual IoT Layer.* This layer contains the logical representation of the virtualized IoT devices. The *Physical and Virtual IoT Management Layer* provides the necessary abstraction for accessing and managing



Application-1
Application-N
Interface for accessing IoT IaaS
Request Processor
Virtual IoT Infrastructure Management Layer
Orchestration Instances
Orchestrator
Discovery Engine
Repository Access Engine
Orchestration Plans Repository
Event Bus
VT Manager
Virtual IoT Device Repository
Uniform Interface for accessing Underlying Management Layer
Physical and Virtual IoT Management Layer
Adapter
Publisher
Event Dispatcher
Adapter
Publisher
Event Dispatcher
Repository Access Engine
Physical IoT Device Repository
Virtual IoT Layer
Physical IoT Layer

Figure 7: Proposed IoT IaaS Architecture

underlying heterogeneous devices in a uniform way. The uniform interface sits in between the *Physical and Virtual IoT Management Layer* and the *Virtual IoT Infrastructure Management Layer* and shown as a horizontal dotted line in the architecture diagram. The *Virtual IoT Infrastructure Management Layer* controls and manages the virtual IoT infrastructure using the uniform interface. It also exposes a set of interface for accessing the IoT IaaS. On the highest layer, the IaaS users are shown, in this case only the application is shown. However, it can be also a PaaS. On the right side of the architecture, several repositories are shown. These are used for publication and discovery of the IoT device.

In the next subsections, we describe each of the layers and the components within along with its intended functionality.

4.2 Physical IoT Layer

This layer contains the all physical IoT devices within the IoT IaaS. The IoT device may have sensing and/or actuation capabilities. If real-time application provisioning is required without virtualization, this layer provides direct access to non-virtualized IoT devices. In real life scenario, the physical devices are distributed throughout several sets of defined geographic areas. And perform sensing and/or actuation on that predefined environment.

4.3 Virtual IoT Layer

In this layer, all of the virtual IoT devices reside. This is more of a logical layer. This layer provides a transparent view of the IoT device to the application as each of the application has the exclusive control over the virtual IoT device. The applications can access the virtual IoT device residing in this layer which was provisioned by or for them. It is to be noted once virtualized there is no difference between the virtual IoT device and the underlying physical IoT device from the application's point of view.

4.4 Physical and Virtual IoT Management Layer

This layer contains four component, *Adapter*, *Publisher*, *Event Dispatcher*, and a *Repository Access Engine*. Each of the components is described in detail below -

4.4.1 Adapter

This component maps the uniform interface for accessing and managing the underlying physical and/or virtual device to the proprietary interface of the physical IoT device, and vice verse. For example, if a primitive (e,g - create) is called to virtualize a physical device, the Adapter will translate that command to appropriate proprietary function and invoke them. Once the physical device is virtualized, it will then return the result in a uniform manner.

4.4.2 Publisher

This component is responsible for publishing the Physical and/or Virtual IoT device's meta information to a repository. The functionality of this component can be divided into two parts. One functionality is to normalize the properties exposed by different vendor IoT device to a common property. Another functionality is to decouple the location of the repository from the underlying IoT device using the *Repository Access Engine.*

In order to capture the meta information regarding the physical and virtual IoT device, a modeling of the information is required. We propose two terms for modeling such information. One we call the *Global Contextual information* and other is called *Local Contextual Information.* These are discussed briefly in the following subsections.

4.4.2.1 Global Contextual Information

Some information within the physical IoT device remains same for all of the deployed virtual IoT device on top of it. For example, the location, the battery level, the hardware address of the device, remains the for all of the virtual IoT devices which sits on top of the physical device. There is some other information which is exclusive to the physical device only. For example, support for the virtualization capability, the number of maximum virtual IoT device that can be deployed on top of it and the number of currently deployed virtual IoT device on top of it. Hence, the Global Contextual Information can be attached with either read or read/write permission.

Table - 4 lists a non-exhaustive list of Global Contextual Informations.

Information Type	Read Only	Read Write
Shared Among Virtual IoT Devices	Location, Battery Level, Hardware Address, Manufacturer, Type	
Exclusive To Physical IoT Devices	Capabilities (e,g-Temperature), Maximum No of Virtual IoT (vIoT) device supported	Virtualization Capabilities, Number of currently deployed vIoTs

Table 4: Non exhaustive list of Global Contextual Information

4.4.2.2 Local Contextual Information

As discussed in the previous section, that some information regarding physical IoT device is shared across all the virtual IoT devices running on top of it. Similarly, there is some information which is exclusive to individual virtual IoT devices running on same physical IoT device. We termed this information as the Local Contextual Information. For example, sample rate, data-mode, data type, and type of the capabilities are different for different virtual IoTs within the same physical IoT device. Just like the Global Contextual Information, the Local Contextual Information is also associated with read and/or write permission with it.

As discussed in the previous section, that some information regarding physical IoT device is shared across all the virtual IoT devices running on top of it. Similarly, there are some information which are exclusive to individual virtual IoT devices running on same physical IoT device. We termed these information as Local Contextual Information. For example, sample rate, data mode, data type, type of the capabilities are different for different virtual IoTs within the same physical IoT device. Just like Global Contextual Information, Local Contextual Information are also associated with read and/or write permission with it. Table - 5 lists a non-exhaustive list of Local Contextual Informations.

Information Type	Read Only	Read Write
Shared Among Virtual IoT Devices	Location, Battery Level, Hardware Address, Manufacturer, Type	
Exclusive To Virtual IoT Devices	Deployed Capabilities (e,g-Temperature),	Sampling Rate, Data Mode, Data Type, Events

Table 5: Non exhaustive list of Local Contextual Information

4.4.3 Event Dispatcher

The *Event Dispatcher* component is responsible for dispatching the event to the *Event Bus* component within the *Virtual IoT Infrastructure Management Layer*. This component fires the pre-defined event whenever the conditions are met within a virtual of physical IoT device and pushes the event to the *Event Bus* for further processing. As there will be a routing decision to make, it is much more efficient to leave such decision to the higher layer, in this case, the *Virtual IoT Management Layer*, due to the fact that the global view of the overall IoT network may not be available in a single node.

4.4.4 Repository Access Engine

The *Repository Access Engine* component provides the access to appropriate repositories to the *Publisher* component. This decouples the knowledge of knowing appropriate repositories from the underlying physical IoT device. The *Repository Access Engine* publishes the *Global Contextual Information* to the *Physical IoT Device Repository* and the *Local Contextual Information* to the *Virtual IoT Device Repository.*

4.5 Uniform Interface For Accessing Underlying IoT Devices

There is a need for some uniform primitives, such as create, delete etc., to access, and manage (e,g - virtualize) the underlying virtual and/or physical IoT device. The obvious primitives are create, delete and update [28]. As we are not considering IoT device migration hence, we do not cover the update primitive for the uniform interface. Next, in order to identify other primitives we followed a systematic approach to identify the primitives. The overall systematic approach is given below [45]-

1. identify the dataset
2. split the dataset into resources
3. for each resources -
   (a) Name the resource using a URI
   (b) identify the subset of the uniform interface exposed by the resource
   (c) Design the representation of the resources as received as a response or sent as a request to the IoT device
   (d) Finally, by exploring how the new service behave and what happens on successful execution, if require define new events and/or interfaces.

In the next subsections, we apply the process described above. It is to be noted that we used JSON as our description language.

4.5.1 Identifying the Dataset

As the goal is to perform operation on the underlying IoT devices, the dataset consists all the underlying physical and virtual IoT device. As, we can either perform a Control operation on the IoT devices (e,g - virtualize) or a Data operation on the IoT device (e,g - read the current measurements); hence, the dataset consist of two types of operations - Control and Data Plane operations.

4.5.2 Split into Resources

Next we want to split the dataset into resources. The overall identified resources are of two type, one is the physical IoT devices and another is the virtual IoT devices.

4.5.3 Identify Operations and URIs

In this section first we identify the URI for the resources. Then we identify the required operation on the control plane and data plane respectively. These are discussed in the following sections.

Identifying the URI

First, we define a base URI with a placeholder *<BASE_URI>* to denote the base URI of every other URI we are going to identify. This placeholder can be replaced with any host-name, for example, iot-iaas-internal.com is one such replacement. Moving next, We identify any physical IoT device or any virtual IoT device with an Universally Unique Identifier (UUID). Hence, our URI for any physical IoT or virtual IoT device looks like below -

<BASE_URI>/<UUID>

Identifying the Operation on Control Plane The control plane is mostly associated with the underlying Physical IoT device. This is because, the basic primitives, such as create, delete operates on the physical IoT device and either virtualize or removes a previously virtualized IoT device (e,g - *create-viot, delete-viot.* Apart from the create and delete, there is need an interface for publishing the capabilities (e,g - *publish-capabilities*) . Finally, we need an interface for querying the IoT device (e,g - *context*) and another interface for letting the IoT device push it's state to the upper layer or the repository (e,g - *notify-update*). Therefore, the identified operations are - *create-viot, delete-viot, publish-capabilities, context, notify-update.*

Identify Operations on Data Plane

The data plane is mostly associated with virtual IoT device. Hence, the support for getting the measurement data is required (e,g - *data*), similarly for sensing class IoT devices, an interface for notifying the event is needed (e,g - *notify-event*), while, for the actuation class IoT devices, an interface for handling an event is required (e,g - *handle-event*). The interface to *data* measurement data is mostly to support pull style query of virtual IoT device. Thus the overall identified operations in the data plane are - *data, notify-event, handle-event.* A brief summary of the Control and Data Plane operations with examples are given in table - 6

Plane	**Operation**	**Explanation**	**Focus Point**	**Example Values**
Control	create-viot	Create virtual IoT with given parameters	Method	Post
			Parameters	{service-type, location, sampling-rate, data-mode}
			Success	200 OK <*UUID*>
			Failure	422 Unprocessable Entity on Syntax Error
	delete-viot	Delete the virtual IoT identified by the supplied <*UUID*>	Method	Post
			Parameters	{service-type, location, sampling-rate, data-mode}
			Success	200 OK <*UUID*>
			Failure	422 Unprocessable Entity on Syntax Error
	publish-capabilities	List the capabilities and properties of a given physical IoT device <*UUID*>	Method	Post
			Parameters	{service-type, location, sampling-rate, data-mode}
			Success	200 OK <*UUID*>

(continued)

Plane	Operation	Explanation	Focus Point	Example Values
			Failure	422 Unprocessable Entity on Syntax Error
	context	Returns the global context associated with the physical sensor	Method	Post
			Parameters	{service-type, location, sampling-rate, data-mode}
			Success	200 OK <*UUID*>
			Failure	422 Unprocessable Entity on Syntax Error
	notify-update	Notifies the upper layer regarding self state change	Method	Post
			Parameters	{service-type, location, sampling-rate, data-mode}
			Success	200 OK <*UUID*>
			Failure	422 Unprocessable Entity on Syntax Error
Data	get	Gets the data from the virtual IoT device	Method	GET
			Parameters	
			Success	200 OK
			Failure	Error message/code (e,g – 404 Not Found on giving uuid that does not exist, 403 Forbidden on inadequate permission, 422 Unprocessable Entity on wrong parameter values)
	notify-event	Notify an event	Method	Post
			Parameters	{callback-url, event, data}
			Success	200 OK
			Failure	Error message/code (e,g – 404 Not Found on giving uuid that does not exist, 403 Forbidden on inadequate permission, 422 Unprocessable Entity on wrong parameter values)
	handle-event	Handles an event	Method	Post
			Parameters	{event-data}
			Success	200 OK
			Failure	Error message/code (e,g – 404 Not Found on giving uuid that does not exist, 403 Forbidden on inadequate permission, 422 Unprocessable Entity on wrong parameter values)

Table 6: Summary of Uniform Interface for Accessing and Managing IoT devices

4.6 Virtual IoT Infrastructure Management Layer

This layer manages the underlying Virtual IoT Infrastructure and provides an external interface (i,e - high-level interface) to the IoT IaaS users (e,g - the PaaS). There are several key components in this layer. These components are *Virtual Things Manager* (VT Manager), *Event bus*, *Publisher*, *Repository Access Engine*, *Orchestrator*, *Orchestration Instances*, *Request Processor*, and *Discover Engine*. All of these components are discussed in details in the following sections.

4.6.1 Virtual Thing Manager

This component is the primary component responsible for managing the underlying IoTs, through the uniform interface. For example, to create a virtual IoT device, this component will send "create-viot" primitive with appropriate parameters to the designated physical IoT device. Once the virtual IoT device has been created, it will store the information of the virtual IoT device in the *Virtual IoT Repository* using the *Repository Access Engine*.

4.6.2 Event Bus

The *Event Bus* works in conjunction with the *Event Dispatcher* situated in the lower layer through the uniform interface. The primary task of the *Event Bus* is to route the incoming event to all of the registered event handlers for that event. It has the global view of the underlying IoT IaaS network and thus knows how to route the event to appropriate event handler. There are two places where the event will be routed. One is another underlying IoT device and another one is the orchestration instances. The orchestration instances can route the event back to the application if the application itself is a registered event handler for the received event in the *Event Bus*.

4.6.3 Publisher

The *Publisher* component publishes the new or updated state of the virtual IoT device to the *Virtual IoT Device Repository*. It is used by the *VT Manager* which provides the new updated local contextual information to the *Publisher* component to publish it to the *Virtual IoT Device Repository*.

4.6.4 Orchestrator

In case, if an orchestration is required to fulfill the requirements of the application, then the *Orchestrator* will orchestrate the virtual IoT devices in a composite way such that it will be a collection of underlying virtual IoT device. For, this to orchestration to be done, an orchestration plan is required which gives the steps to be needed in order fulfill the given application requirements. The orchestration plans are resided in the *Orchestration Repository* and is accessed by the *Orchestrator* via *Repository Access Engine*. The *Orchestrator* then uses the *VT Manager* to provision the underlying virtual infrastructure to the request application or the user.

4.6.5 Orchestration Instances

Once the *Orchestrator* is finished with provisioning the underlying virtual IoT infrastructure, an *Orchestration Instance* is created. The general interaction of the orchestration is shown in figure - 8. From a high level view, there are mainly three types of task which is associated with an orchestration instances. These are the *Sensing Task*, the *Actuation Task*, and the *Application Task*. The *Sensing Task* generates events and push the event to the *Orchestration Instance* via *Event Bus*. On the other hand, the *Actuation Task* consumes the generated events as control signal to initiate the desired actuation on the operating environment. Finally, the

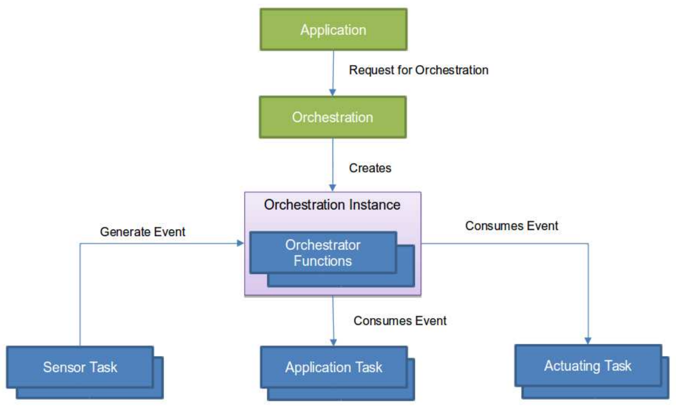


Figure 8: General Orchestration Interaction

Application Task can also consume the generated events in order to notify the application user, if the user intervention or attention is required.

4.6.6 Request Processor

This component provides two functionality. First, it acts like an Adapter which maps the IoT IaaS access interface to the appropriate internal interface. Second, it routes the request based on the application's requirements. If an application requirements do not need orchestration, it will direct the *VT Manager* to provide the virtual IoT infrastructure. If the application requires orchestration, it will direct the *Orchestrator* to provide the virtual IoT infrastructure.

4.7 Interface for accessing IoT IaaS

As there is very less work focused covering this interface, we again take the systematic approach to identify the required interface for accessing the IoT IaaS. First, we will provide a short introduction of the model we are using for representing a IoT device from the point of view of the *Virtual IoT Infrastructure Management Layer*. Then, just like we derived the uniform interface, we will identify the overall dataset, then split the dataset into resources and finally identify the operation and representation for each of the resources. At the end, we will summaries the interfaces together.

4.7.1 Modeling of Sensor Representation

We use Universally Unique Identifier (UUID) to uniquely denote a physical IoT device or a virtual IoT device. More concretely, we use *UUID* to uniquely distinguish a resource, whether it is a composite IoT device, a virtual IoT device or a physical IoT device. As a composite IoT device can contain one or more IoT devices thus, essentially, the composite IoT device's UUID is associated with all of the IoT device's *UUID* which made up the composite IoT device. If this relationship is represented as graph, it will be a N-ary tree with depth of one (1).

Next, each of the event are associated with its source and the destination of the event handler can be either in the same IoT device, or in a different IoT device, or the application itself. Hence, the events can be represented again as a N-ary tree with single depth. Similarly, actions can be represented as N-ary tree with single depth. Thus the overall representation look like figure - 9.

4.7.2 Identify the Dataset

From the high-level, the user wants to provision, access, and control a thing. A thing can be made up of sensors, actuators, or a combination of both. Hence, the overall dataset is all the underlying things.

4.7.3 Split the Dataset into resources

As per the previous discussion, it is evident that two immediate resources which can be identified are the Sensor and the Actuator. Hence, we need a way to

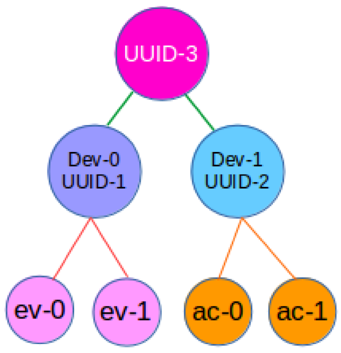


Figure 9: Modeling of the IoT device from IoT IaaS point of view

provision (i,e - create), delete, and list them. Considering the previous discussion for the Uniform Interface to Access IoT devices, we define another placeholder URI, <*IOT_IAAS*>, as the base URI. It can be replaced with any host-name. For example, iotiaas.com is a good candidate to be used as a replacement URI for the <*IOT_IAAS*>. Next, we define *things* as the sub URI which is associated with the things. Hence, the canonical URI for accessing the things within the IoT IaaS would be <*IOT_IAAS*>*/things*. Next goal is to support the three basic primitives, that is, to create a thing, to delete a previously created a thing and to list a thing. These operations can be satisfied by the HTTP method semantic POST, DELETE and GET respectively. Once the things are created there is a need for accessing the resources exposed by that single thing. As all the things are identified by <*UUID*>, we can easily map the URI for a single thing based on that <*UUID*>. Hence, our final resource URI for things is -

1. URI for things :

   <*IOT_IAAS*>*/things*

2. URI for accessing resources exposed by a particular thing :

   <*IOT_IAAS*>*/things/*<*UUID*>

A particular thing exposes more implicit resources, as it has several properties. For example, A thing can have events, event handlers, and actions. Thus there is a need to add, delete, and view them in order to both define them at the time of provisioning and/or manage them later. The add, delete, and view operation can be satisfied by the HTTP POST, DELETE, and GET method respectively. Hence, the set of implicit resources for a particular thing are listed below.

3. URI for events in things :

   <IOT_IAAS>/things/<UUID>/events

4. URI for event-handlers in things :

   <IOT_IAAS>/things/<UUID>/events/<event-index>/callbacks

5. URI for actions in things :

   <IOT_IAAS>/things/<UUID>/actions

In the next subsection we will define the representation associated with the URI.

4.7.4 Representation

In order to define a thing, we first need to specify how many capabilities are required. Along with that, we also need to define the parameters and what are the criteria (i,e - requirement) that has to be fulfilled for each of the capabilities within the thing. The parameters of a capabilities is dependent on the *"type"* of the thing (either - sensing or actuation). Table - 7 and Table - 8 shows a non-exhaustive parameter list for sensing and actuation type capabilities respectively.

Name	Type	Example
index	integer	0, 1, ...
service-type	string	humidity, temperature
data-mode	string	push, pull
data-format	string	degree Celsius, relative humidity
sampling-rate	integer	500, 1000, 60000
event	array of objects	

Table 7: Example Key values for Sensing type services

Name	Type	Example
index	integer	0, 1, ...
service-type	string	cooling, heating, firefighting
actions	array of objects	

Table 8: Example Key values for Actuation type services

Similarly, the keys for matching criteria can be from table - 9.

Name	Type	Example
location	string / latitude-longitude	“tselab”, “montreal old port”
battery	string	“good” (assuming good means more than 80% capacity remaining)
vendor	string	“raspberry pi”
required-peripheral	array of strings	“wifi”, “bluetooth” etc

Table 9: Example Key values for defining criteria

None of the table - 7, 8, and 9 are exhaustive listing.

As shown in table - 7, the “event” has a type of array objects as a value. This means that one can define several events as objects and provide them as an array of objects as the value corresponding to the “event” key. Table - 10 shows the possible keys to construct the event objects. Similarly, some of the possible key values for the “callbacks” in the actuation type of service are shown in the table - 11.

Name	Type	Example
index	integer	0, 1, ...
name	string	“highTempEvent01”

function-name	string	"threshold"
function-data	string	"$data >= 27"
callbacks	array of objects	"{"index" : 0, "target" : 1, "uri": "dev-$1/action-$0"}"

Table 10: Example Key values for defining events

Name	**Type**	**Example**
index	integer	0, 1, ...
name	string	"AC01"
endpoint	string	"/turn-on-ac"

Table 11: Example Key values for defining actions

The final missing piece is the association description between event and callbacks. To define the association, we again use a placeholder. This is because, at the time of provisioning the real *UUID* is not known, and it can only be known after the provision is completed. At the run-time, this placeholder value is replaced with actual *UUID* by the *Event Bus*, or the *Orchestrator* to resolve the final version of the callback. For example, the placeholder $dev-1/action-$0 will be replaced by the *UUID* of the service with an "index" value of 1, then within that service, the action with an "index" value of 0 will be invoked.

4.7.5 Summary

The table - 12 shows a non exhaustive list of the interfaces that are used to access the IoT IaaS.

Operation	**Parameters**	**Explanation**	**HTTP Method**	**URI**
create	index	Unique index of the sensor or actuator	POST	*<IOT_IAAS>/things/*
	type	Type specifying the resource category		
	params	Defines the parameter related to the resource category		
	criteria	Criteria to be used for selecting underlying resource		

delete	uuid	Universally Unique Identifier (UUID) to denote which thing to delete	DELETE	<IOT_IAAS>/things/
list all	N/A	N/A	GET	<IOT_IAAS>/things/
list single	N/A	N/A	GET	<IOT_IAAS>/things/<UUID>
list events	N/A	N/A	GET	<IOT_IAAS>/things/<UUID> /events
add event	index	A unique index of the event	POST	<IOT_IAAS>/things/<UUID> /events
	dev	source device		
	name	human readable name		
	function-type	Type of the event trigger function		
	function-data	Additional data associated with trigger function		
	callback	Callback to be notified on trigger		
delete event	index	A unique index associate with the event	DELETE	<IOT_IAAS>/things/<UUID> /events
	dev	Device index where the event resieds		
list callbacks	N/A	N/A	GET	<IOT_IAAS>/things/<UUID> /events/<dev-idx>/<event-idx> /callbacks
add callback	N/A	N/A	POST	<IOT_IAAS>/things/<UUID> /events/<dev-idx>/<event-idx> /callbacks
delete callback	N/A	N/A	DELETE	<IOT_IAAS>/things/<UUID> /events/<dev-idx>/<event-idx> /callbacks/<callback-order>
list actions	N/A	N/A	GET	<IOT_IAAS>/things/<UUID> /actions
add action	index	A unique index of the action	POST	<IOT_IAAS>/things/<UUID> /actions
	dev	source device		
	name	human readable name		
	endpoint	trigger endpoint to activate the action		
	endpoint-data	Additional data associated with trigger action		
delete action	index	A unique index associate with the action	DELETE	<IOT_IAAS>/things/<UUID> /actions
	dev	Device index where the action resieds		

Table 12: Non exhaustive list of interface for accessing IoT IaaS

4.8 Repository

There are overall three repositories required, one for storing the orchestration plan, another one for the virtual IoT device information (e,g - local contextual information) and finally, the last one for storing physical IoT device information (i,e - global contextual information). These three are described in the following section.

4.8.1 Orchestration Plan Repository

This repository stores the orchestration plan and provide the plan if requested via the *Repository Access Engine*. An orchestration plan generally contains the steps to orchestrate the resources based on the requirements.

4.8.2 Virtual IoT Device Repository

The *Virtual IoT Device Repository* contains the information regarding the provisioned virtual IoT device. This is generally a tuple of tuples where *UUID* is the unique value within the tuples. The tuple contains the key-value attribute pair tuples with permission and thus act as the local contextual information. On the other hand, once provisioned the first entry is provided by the *VT Manager*. While, later on the Virtual IoT device itself can update the information base on its state.

4.8.3 Physical IoT Device Repository

The *Physical IoT Device Repository* contains the information regarding the underlying physical IoT device. This is also a tuple of tuples where *UUID* is the unique value within the tuples. Just like *Virtual IoT Device Repository*, it contains the key-value attribute pair tuples with permission and thus act as the global contextual information. Generally when an IoT device's state is changed, it will notify the change to the *VT Manager* or it can publish the updated data to the repository by itself.

4.9 Interaction Between IoT IaaS components

In order to give a mental picture of how the different components within the IaaS interact together, we use the motivating use-case as the scenario on which the interaction between the different components is shown. Considering the motivating use-case, we focus on the subset of the use-case where we assume the smart HVAC application requires orchestration. We first discuss how the overall provisioning will work and how the different component will work together to orchestrate the required virtual IoT device. Then we discuss how the virtual IoT devices are created within the IaaS, finally we show how the orchestration instance will work once the virtual IoT device in operation. These are discussed in the subsequent sections below. In the end we summarize the section by focusing on some of the details that were left out of the interaction figure due to figure space and complexity.

4.9.1 Overall Interaction between IaaS components : Smart HVAC

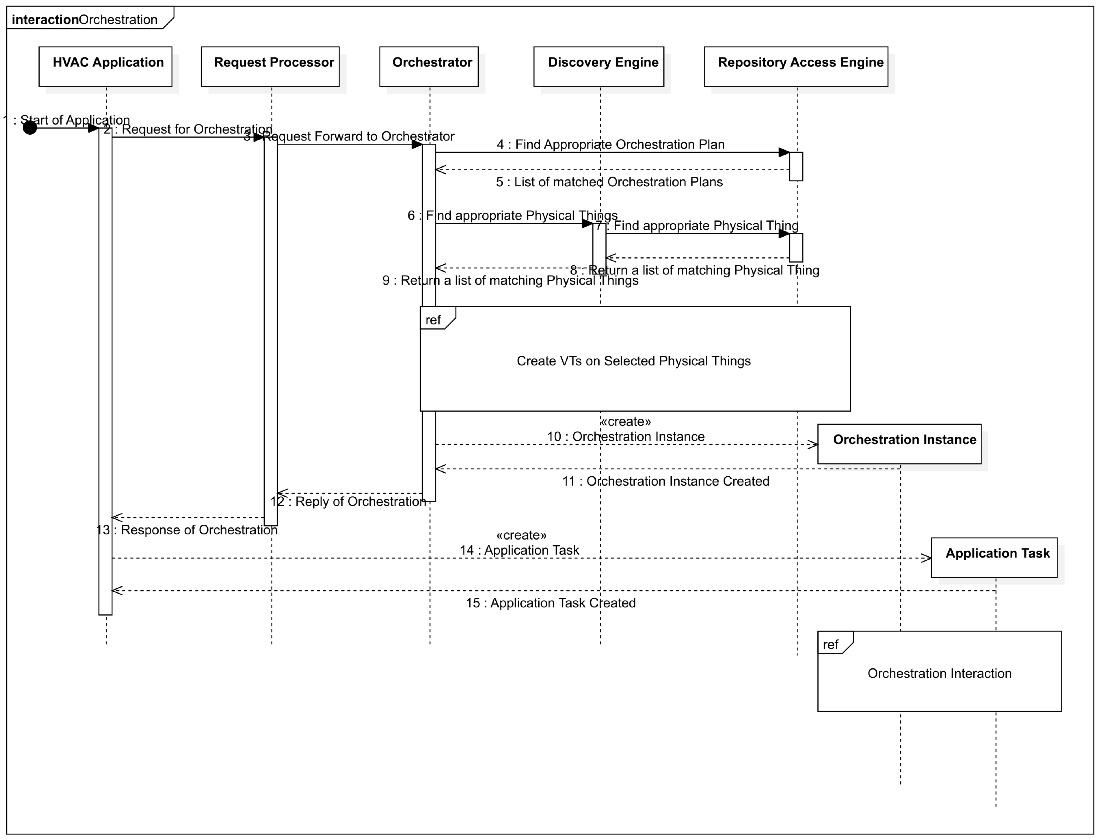
Figure - 10 shows the overall interaction between different components when the smart HVAC application wants to provision a virtual IoT device for measure temperature, humidity and also wanted to control the temperature via actuation service. In the following sections we show the layer to layer interaction. It is to be noted that, the sequence diagram does not cover all the possibilities due to reducing the complexity of diagram. This is specially true for negative results, the sequence diagram assumes that all of the operation execution will be successful.

4.9.1.1 Application Requesting to the IoT IaaS

The HVAC application request for a virtual IoT device having two sensing capabilities (temperature and humidity) and one actuation capability (cooling). We defined the application is also wanting to know when the temperature gets higher than a certain threshold (e,g - 27-degree Celsius) and the relative humidity gets above another threshold (e,g - 50% relative humidity). In that case, the application wants the AC to be turned on. The application also wants to get notified once this conditions are met. The heating is excluded for the sake of reducing diagram complexity, however, it is trivial and follows the same path as the requested actuation capability. This request is send to the *Interface for accessing the IoT IaaS*.

4.9.1.2 Within the Virtual IoT Infrastructure Management Layer

The received request from the application is forwarded to the *Request Processor*. This is the entry/exit point for most of the requests/responses. The *Request Processor* determines that this requires orchestration, and it forwards the request to the *Orchestrator*. The *Orchestrator* upon receiving the request, uses the *Repository Access Engine* to find a suitable orchestration plan for the orchestration to take place. Once it finds such plan, it then uses the *Discovery Engine* to find a set of suitable underlying physical IoT devices which can meet the requested requirement from the application. Then the *Orchestrator* executes the orchestration plan using the *VT Manager* and gets the list of virtualized IoT devices. This virtualization of underlying physical IoT device is not shown in figure - 10, and it is covered in the later section. Finally, the *Orchestrator* creates an *Orchestration Instance*. Once the *Orchestration Instance* is created, the *Orchestrator* returns the success



interactionOrchestration
HVAC Application
Request Processor
Orchestrator
Discovery Engine
Repository Access Engine
1 : Start of Application
2 : Request for Orchestration
3 : Request Forward to Orchestrator
4 : Find Appropriate Orchestration Plan
5 : List of matched Orchestration Plans
6 : Find appropriate Physical Things
7 : Find appropriate Physical Thing
8 : Return a list of matching Physical Thing
9 : Return a list of matching Physical Things
ref
Create VTs on Selected Physical Things
«create»
10 : Orchestration Instance
Orchestration Instance
11 : Orchestration Instance Created
12 : Reply of Orchestration
13 : Response of Orchestration
«create»
14 : Application Task
Application Task
15 : Application Task Created
ref
Orchestration Interaction

Figure 10: Interaction Between Application and IoT IaaS

result to the *Request Processor.* The *Request Processor* then forward the result to the *Application.* As the returned result will contain all the necessary information regarding orchestration, the HVAC application then starts an application task to handle the event for the *Orchestration Instance* to communicate with the HVAC application.

4.9.1.3 Among Virtual IoT Infrastructure Management, Physical IoT Device, and Virtual IoT Device Layer

Figure - 11 shows the interaction between the *VT Manager* and the components in the underlying layers. Once the *VT Manager* receives the request for virtualization, it uses the *Uniform Interface for accessing the IoT device* to invoke the primitive *"create-viot"* on the *Hypervisor* of the underlying physical IoT device.

These physical IoT devices are resided within the *Physical IoT Device Layer.* Once, the virtual IoT device is created it contains has the unique *UUID* as the identifier associated with it. The *VT Manager* virtualizes all of the required IoT devices. These virtualized IoT devices, in this specific case, the *Virtual Temperature Sensor*, the *Virtual Humidity Sensor*, and the *Virtual AC Actuator*, are logically belongs to the *Virtual IoT Device Layer.* After all the virtual IoT devices are created, the *VT Manager* uses the *Repository Access Engine* to save the information regarding the newly created virtual IoT devices in the *Virtual IoT Device Repository.*

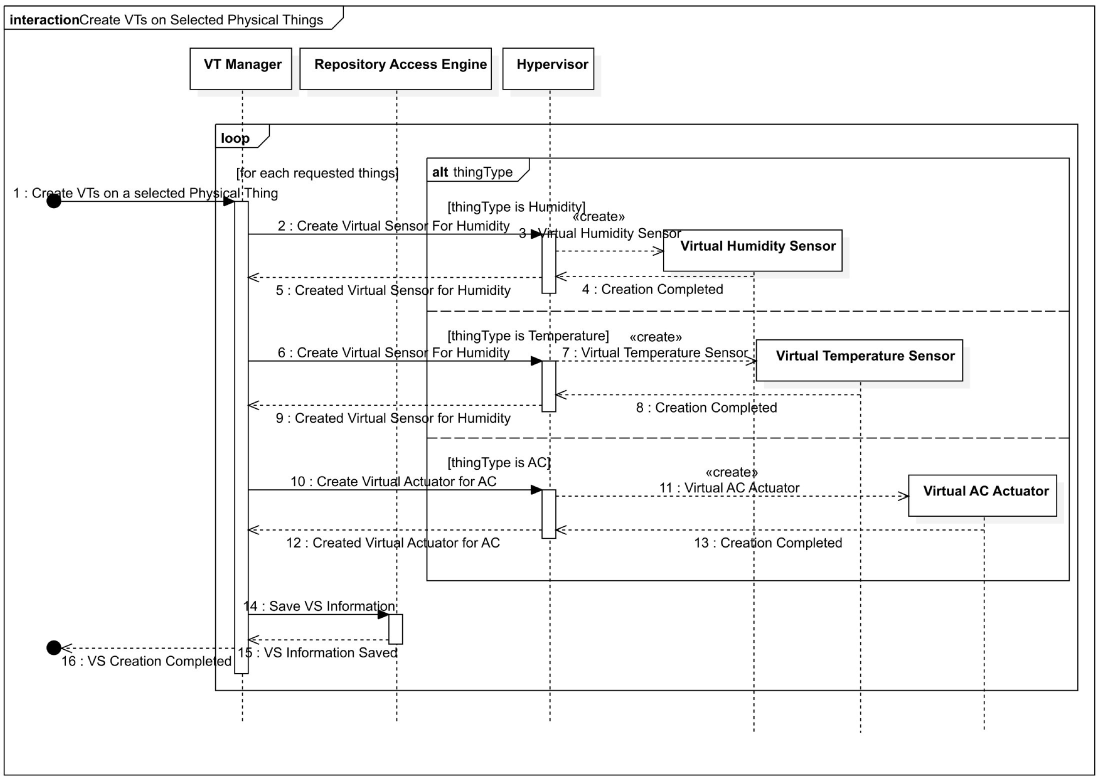

interactionCreate VTs on Selected Physical Things
VT Manager
Repository Access Engine
Hypervisor
loop
[for each requested things]
alt thingType
1 : Create VTs on a selected Physical Thing
[thingType is Humidity]
«create»
2 : Create Virtual Sensor For Humidity
3 : Virtual Humidity Sensor
Virtual Humidity Sensor
5 : Created Virtual Sensor for Humidity
4 : Creation Completed
[thingType is Temperature]
«create»
6 : Create Virtual Sensor For Humidity
7 : Virtual Temperature Sensor
Virtual Temperature Sensor
8 : Creation Completed
9 : Created Virtual Sensor for Humidity
[thingType is AC]
«create»
10 : Create Virtual Actuator for AC
11 : Virtual AC Actuator
Virtual AC Actuator
12 : Created Virtual Actuator for AC
13 : Creation Completed
14 : Save VS Information
15 : VS Information Saved
16 : VS Creation Completed

Figure 11: Interaction with VT Manager and Lower Layer components

4.9.1.4 Between Orchestration Instance and the Application

Once the provisioning is completed, the application is ready to use the virtual IoT device. Figure - 12 shows the interaction between the *Orchestration Instance* and the application. The *Virtual Temperature Sensor* notifies the *Orchestration*

Instance via the *Event Bus* (not shown in the figure) once the predefined threshold is exceeded. Similarly, the *Virtual Humidity Sensor* notifies the *Orchestration Instance* once the predefined threshold for humidity is exceeded. Once both conditions are met, the *Orchestration Instance* will dispatch the event to *Virtual AC Actuator*'s endpoint and to the *Application Task.* Upon receiving the control signal, the *Virtual AC Actuator* will start cooling the environment while the *Application Task*, which is associate with the application itself will receive the notification that such an event has occurred.

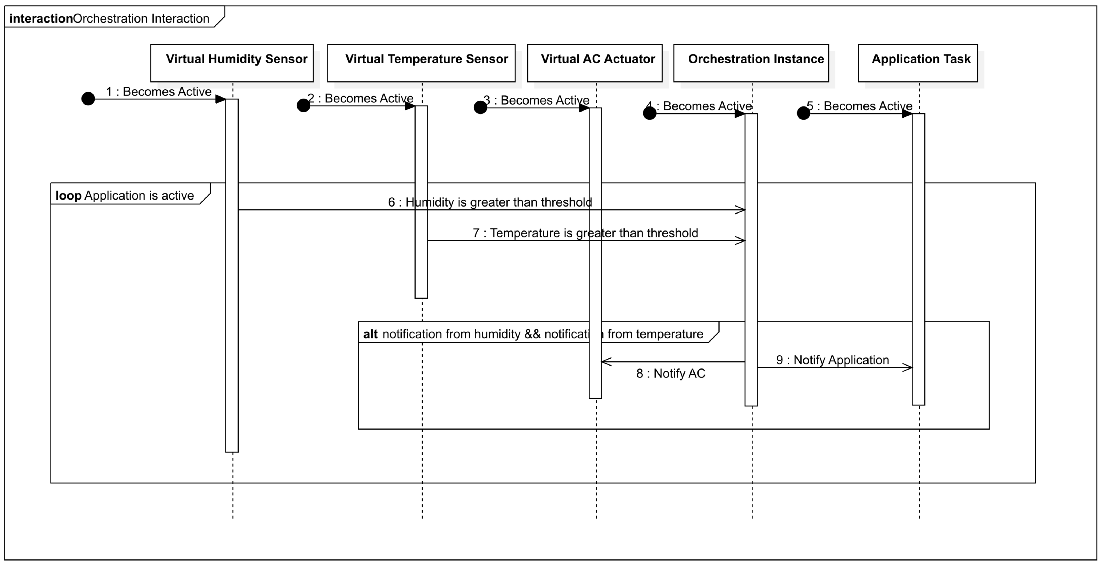

interactionOrchestration Interaction
Virtual Humidity Sensor
Virtual Temperature Sensor
Virtual AC Actuator
Orchestration Instance
Application Task
1 : Becomes Active
2 : Becomes Active
3 : Becomes Active
4 : Becomes Active
5 : Becomes Active
loop Application is active
6 : Humidity is greater than threshold
7 : Temperature is greater than threshold
alt notification from humidity && notification from temperature
9 : Notify Application
8 : Notify AC

Figure 12: Interaction between application and Orchestration Instance at runtime

4.9.1.5 Summary of Interaction between components and the HVAC application

As mentioned earlier, the figure did not include all of the possible cases, such as failed cases, to reduce diagram complexity. Moreover, certain components were left out as their interaction is well defined by other components. For example, the *Orchestration Repository*, the *Virtual IoT Device Repository*, and the *Physical IoT Device Repository* were left out of the figure because the *Repository Access Engine* abstracts their interaction with other components. This was done to reduce the both size and interaction complexity of the figure. The *Event Bus* was left out of the figure for the same reason.

The discussion in the previous sections on the interaction between the components within the different layers of the IoT IaaS gives a good mental image of how the IoT IaaS works when an application is requesting provisioning, as well as when the application is executing. Hence, covering both manage and access operation from the point of view of the users of the IoT IaaS.

4.10 Evaluation of proposed architecture against the requirements

The requirements derived from the motivating use-case must need to be satisfied by the proposed architecture. In fact, the proposed architecture satisfied them well.

The proposed architecture relies on the node level virtualization for efficient resource utilization. Thus, the first requirement is met by the proposed architecture

The architecture contains the appropriate components (*Publisher*) to publish the information to the repository. Moreover, we provide a loosely coupled modeling of the properties of the IoT device, which gives us the ability to store the information in a generic manner. Next, the *Discovery* component allows the architecture to find the published information from the repository.

Hence, the proposed architecture incorporates the publishing and discovery mechanism which is suitable for IoT device. This fulfills the second requirement.

To support the orchestration, the *Orchestrator*, the *Orchestration Instance*, and the *Orchestrator Repository* are provided with the architecture. A model of how the orchestration should work is also explained.

The means to orchestrate complex services are provided with the architecture, therefore, the third requirement is met by the proposed architecture.

Finally, a brief discussion of how the underlying IoT device should be accessed uniformly through the *Adapter* component and what will be the substrate of the uniform interface was provided. Another short discussion on how the cloud users can access the IoT IaaS transparently using a high-level interface was provided. A systematic way was used to derive the interface. Basic parameters and methods were explained.

Finally, the two set of interfaces were provided and a brief discussion of them was made. This meets the fourth and final requirement that was derived from the motivating use-case.

4.11 Conclusion

In this chapter, we presented our proposed architecture, explained the component's functionality in each layer. We also systematically approached the derivation of the high-level interfaces. In the end, we justified how the proposed architecture was able to meet the previously derived requirements from the motivating use-case.

In the next chapter, we will discuss regarding an implemented prototype of the proposed architecture, along with an extension for accessing the prototype from the PaaS and a SaaS application. We will discuss regarding the result obtained and draw the conclusion from it.

Chapter 5

Validation of the Architecture

In this chapter, we start by presenting an experiment on showing how node-level virtualization achieves energy efficiency. We describe the experimental setup, the methodology and the result in brief in the subsequent sections. Then we moved over to the prototype architecture by providing a brief description of it. We describe in details on the prototype architecture in later sections. Finally, we discuss results of various experiments and analyze them accordingly. We conclude the chapter by summarizing it.

5.1 Experiment: Energy Efficiency by Node Virtualization

In this section, we describe a setup for conducting an experiment to show that how node level virtualization achieves energy efficiency. Following the experimental setup, the methodology is described. In next sections, the results are shown and analyzed. Finally, a summary is drawn to conclude the discussion on the experiment.

5.1.1 Experimentation Setup

The experiment was performed using the Advanticsys SkyMote (*TelosB*). It is a very popular and resource constraint device capable of performing - temperature, humidity, and light intensity sensing. The experimental setup is shown in figure - 13. The device under test (DUT) is the *TelosB* with node-level virtualization support. Hence, it is possible to measure the energy consumption when the device is virtualized with several virtual devices on top and when the device is not virtualized and acting as a physical sensor only. The INA219 is a current shunt and power monitor with an I^2C interface. The current and power measurement peripheral is from Texas Instruments (TI). The DUT is powered using a 3.3V

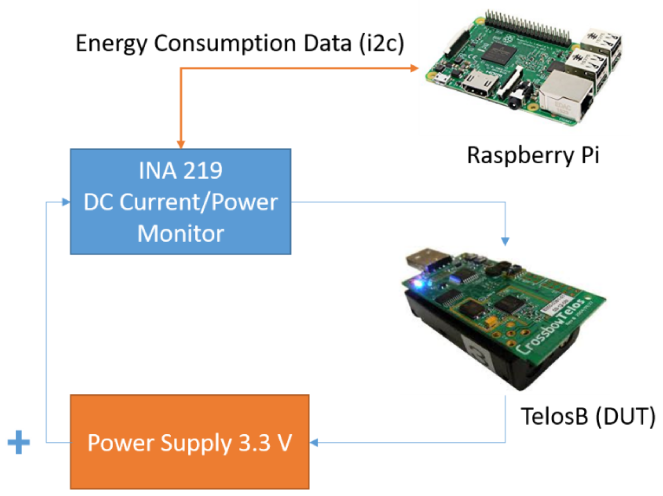


Figure 13: Energy Consumption Measurement Setup

power source, and the INA219 is put into series in between them. As standard PC does not expose any I^2C bus, we used a raspberry pi (RPi) small PC to interface the INA219 with it. A small data collection was written on the raspberry pi to collect the data. The raspberry pi was powered separately from an external power source. INA219 was configured with the software written in the raspberry pi with a current detection sensitivity of 0.1mA. This setup gives the maximum accuracy possible with the INA219 within the operating voltage (3.3V).

5.1.2 Experimentation Methodology

We first focus on the scenario which was taken into consideration for each of the DUT. Next, we focus on the process of collecting the data. These are described in the following sections.

5.1.2.1 Experiment Scenario

We defined an aggressive task which is sending (i,e - pushing) the data to its gateway every second. The justification for not selecting more aggressive task (i,e - sending multiple data in every second) is, doing so, the accuracy of the captured reading is reduced. This is because, then in order to avoid hardware bottleneck of the raspberry pi, we have to collect a low amount of samples. Again the justification of not selecting more relaxed task (i,e - sending data in between larger time intervals) is that they somehow draw the similar conclusion as the aggressive task. The only difference is that the convergence is slow. We first ran four physical devices with the task we mentioned and collected the data for 10 minutes with 50 samples/seconds. Then we ran with four virtual devices with the same task deployed on a single device. The justification of selecting 50 sample/seconds is that we found this number by trial and error which does not bottleneck a 3^{rd} generation of raspberry pi, which is the latest, fastest among the raspberry pi at the time of writing this thesis. With 50 sample/seconds, we can measure the power consumption pretty accurately. This was tested using a multimeter to determine if the sample rate chosen was good enough or not. Running the task for 10 minutes gave us the insight to draw a conclusion, as long run did not change anything. The number four for a task is selected based on an earlier experiment which revealed that *TelosB* can handle a maximum of four virtual devices before running out of memory.

5.1.2.2 Data Collection

When the first task is running the cumulative data is stored in the raspberry pi. Using discrete power consumption equation we determine the discrete power consumption on each second for the first scenario. The result is the cumulative power consumption of four physical devices. In the second scenario, the single data of the physical device is stored in the raspberry pi. Again using the discrete power consumption equation we determine power consumption for each second for the second scenario. Hence, the result is the power consumption of the physical device with four deployed virtual device on top of it. The general equation for power is -

$$P = V \times I$$

And the discrete power consumption equation is given by -

$$E = \sum (P \times S)$$

where S is the interval in seconds. Thus giving the units to be milliwatts-seconds (mW-s) as the power (P) measured in milliwatts. Finally, we plotted the energy consumption in Y-axis and duration in X-axis and analyze the power consumption per seconds.

5.1.3 Results and Analysis

Figure - 14 shows the result we obtained for the scenario discussed in the earlier subsection. The blue line denotes the cumulative energy consumption by four physical devices. While the orange line denotes the energy consumed by a single device hosting four virtual devices concurrently. The maximum power consumed by the four devices cumulatively were 56.63308 mWatt-Seconds, which was 165%

higher than the single device hosting four virtual devices (34.29838259 mWatt-Seconds). On the other hand, the lowest power consumption for the four devices cumulative were 34.91550037 mWatt-Seconds, which was 467% higher than the virtualized DUT (7.467722927 mWaat-Seconds). The average consumption was 42.42325955 mWatt-Seconds with a standard deviation of 2.868032853 for the case of four DUTs. Where, for the virtualized DUT, the average was 15.89496789 with a standard deviation of 7.230499442.

The results were as expected. It is because some inherent power consumption is required to power up all the peripherals within the IoT device. Therefore, the four physical devices would require higher power to keep its peripherals running even without carrying out any useful task. On the other hand, the single device with four virtual devices would require less power to keep its peripherals active. The idle mode CPU current draw for the *TelosB* was 0.8 - 2 mA, which was cumulated for four physical devices, raising the current draw to about 10 mA in combined. This is true for other peripherals as well. For example, the scenario was actively using the temperature sensor, hence some power was needed to drive the temperature sensor itself. As expected, and also supported by the *TelosB* datasheet[1], the major current draw occurs in transmitting the sensed data to the gateway (18-23 mA). Single DUT which was not virtualized was sending data one time per second, where the virtualized DUT was sending data four times per second. However, the transmission power consumption was the same as four non virtualized devices were sending the data. Hence, cumulatively they were equal for both test cases. It is to be noted that, there was no network optimization technique applied which would further improve the energy efficiency in the virtualized DUT. Network coding and aggregation can provide substantial increase in energy efficiency.

In the case of the virtualized DUT, the standard deviation was higher than

[1]http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf

the first case (i,e - running the scenario in four physical DUTs). Our explanation to this high standard deviation is that the temperature sensor was used more and thus, the virtualized DUT stayed the minimal time in the completely idle mode. This was justified by the fact that reducing the number of virtual devices hosted on the DUT reduces the standard deviation accordingly.

Similar tests were performed using Virtenio *Preon32* and the results were similar.

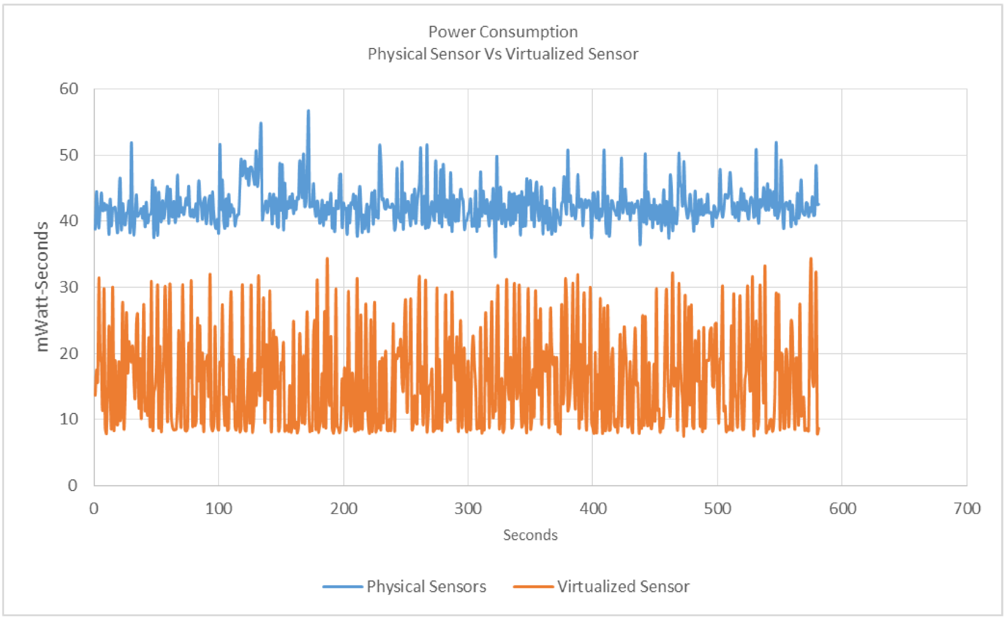


Figure 14: Result of the experiment of node-level virtualization energy efficiency

5.2 Prototype Architecture Overview

In this section we first present the implemented scenario, then a high level description of how the prototype works, we then conclude the section by giving a brief description on the software and hardware used for implementing the prototype.

5.2.1 Implemented Scenario

The implemented scenario is a subset of our motivating use-case. We implemented the Anti-Fire and the Smart HVAC applications from the motivating use-case. For Smart HVAC application the implementation considered only the sensing capabilities (i,e - temperature and humidity). The actuation capabilities were not covered in the implementation. Therefore, the scenario contains two applications, an Anti-Fire application, and a Smart HVAC application, deployed as a SaaS application on top of a PaaS that utilizes the virtualized IoT infrastructure provided by the prototyped IoT IaaS. We used two different sensor vendor, one is Advanticsys (*TelosB*) and another one is Virtenio (*Preon32*). The goal of the Anti-Fire is to detect fire and notify the user as fast as possible. As the Anti-Fire application has a higher priority than the Smart HVAC application, the requirements are more aggressive than the Smart HVAC application. For this reason, we assumed that Anti-Fire application requires a faster sensing rate (a sample-rate of 1 sample/second) while the Smart HVAC application has a more relaxed sensing requirements. For Smart HVAC, the requirement is to notify the application whenever the temperature and humidity of the environment exceeds a predefined threshold. In this case, it is 27 degree Celsius and 50% relative humidity respectively. The underlying IoT infrastructure should notify the application if these predefined thresholds are exceeded.

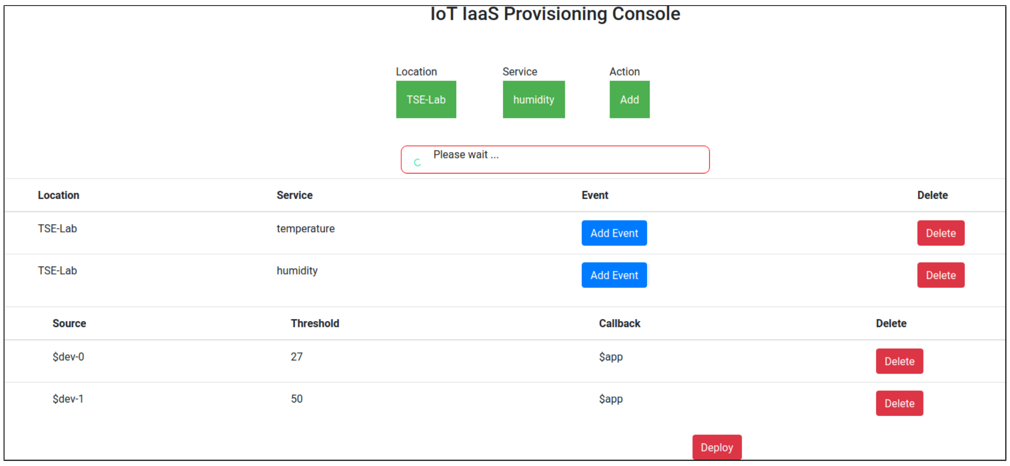
the discussion of the implemented scenario, the Anti-Fire application can be deployed without any orchestration, while the Smart HVAC application can be deployed through orchestration.

5.2.2 Description of Implemented Prototype

The application programming interface (API) for accessing the IoT IaaS from the user is a set of REST API. This facilitates a programming interface suitable for the machine to machine communication. We developed a Graphical User Interface (GUI) which utilizes this programmable interface and simplifies the operation through the GUI. Figure - 15 shows the GUI for requesting the provisioning of the underlying IoT device. The applications are deployed on top of an open source PaaS, namely the Cloud Foundry. It utilizes traditional IaaS to provide infrastructure for the application (e,g - VMs). An extension of the PaaS captures the request from the GUI and forwards it to the prototype IoT IaaS. IoT IaaS then proceeds with the virtualization of underlying IoT infrastructure. Upon completion, the IoT IaaS returns the results with appropriate URIs to access the underlying virtualized resources, that is the virtual temperature IoT device, virtual humidity IoT device, and the composite IoT device.

In terms of matching the application requirements with the published IoT device Global Contextual Information, we used only predefined location. And this location is exposed to the GUI using a REST API. It is to be noted that, this API was added for the convenience of developing the GUI and does not fall under the API for communicating with the IoT IaaS.

We used a total of four devices in four combinations from two vendors (*TelosB* and *Preon32*) to validate the prototype. First, we used two *TelosB* only. Then we used two *Preon32* only, moving next we mixed one *TelosB* with one *Preon32* and finally, we limit the capabilities of the devices to a single exclusive capability to validate orchestration in different cases. We also validated the prototype by providing not satisfiable requirements. For example, providing non-exclusive single capabilities (i,e - either only temperature or only humidity), asking for a virtual IoT device that has two different capabilities (i,e - temperature and humidity).


IoT IaaS Provisioning Console

Location	Service	Event	Delete
TSE-Lab	temperature	Add Event	Delete
TSE-Lab	humidity	Add Event	Delete

Source	Threshold	Callback	Delete
$dev-0	27	$app	Delete
$dev-1	50	$app	Delete

Figure 15: GUI for easily provisioning the underlying IoT device

5.2.3 Softwares and Hardwares Used

In this section, we describe the software and hardware used for implementing the prototype IoT IaaS.

5.2.3.1 Cloud Foundry

Cloud Foundry is an open source PaaS. The Cloud Foundry Runtime runs applications in packages called "droplets" in DEAs (Droplet Execution Agents). DEAs are managed by the Cloud Controller and monitored by the Health Manager, while Routers manage application traffic, do load balancing, and combine logs. In turn, DEAs call on service broker nodes, which communicate over a message bus. The Cloud Controller has access to a blob store and a database of application metadata and service credentials [46]. One of the key features of the Cloud Foundry is that it allows the installation into an off-premise site without the need for an existing commercial IaaS provider. Hence, we deployed the Cloud Foundry in a local machine.

5.2.3.2 Restbed

Restbed[2] is a framework for writing RESTful applications in C++ programming language. It is based on C++11 and uses STL exclusively. It does not have any other dependencies and very fast in routing and execution. At the time of writing the thesis, it is an open source project, contains several features and fully compliant with HTTP 1.0/1.1+. It supports all the HTTP methods. As we developed our prototype IoT IaaS in C++ programming language, we used restbed to expose the functionality of the IoT IaaS to the Cloud Foundry, (i,e - the PaaS).

5.2.3.3 JSON for Modern C++

JSON for Modern C++[3] is an open source library for C++11 to generate and parse JSON data representation as a first class data type within C++ application. We used it for parsing and generating JSON to and from IoT device and the provision GUI.

5.2.3.4 Advanticsys TelosB (SkyMote)

The *TelosB* , showed in figure - 16, is a constrained IoT device with minimal processing power and memory. It utilizes IEEE 802.15.4 2.4Ghz wireless MAC/PHY as the wireless communication mechanism. It has total three capabilities off the shelf - temperature sensing, humidity sensing, and light intensity sensing. We only used the temperature and humidity sensing for our prototype implementation. The capabilities of the *TelosB* can be expanded by adding daughter boards on top of it. It also has three led for status operation and two user switches (reset and user programmable). It has a USB to UART SPI (Serial Programming Interface) to program the device. The programming language of *TelosB* is C like, but not

[2]https://github.com/Corvusoft/restbed
[3]https://github.com/nlohmann/json

ANSI-C. For battery operation, *TelosB* has power terminal for attaching battery case, it requires 3.3V DC to operate.

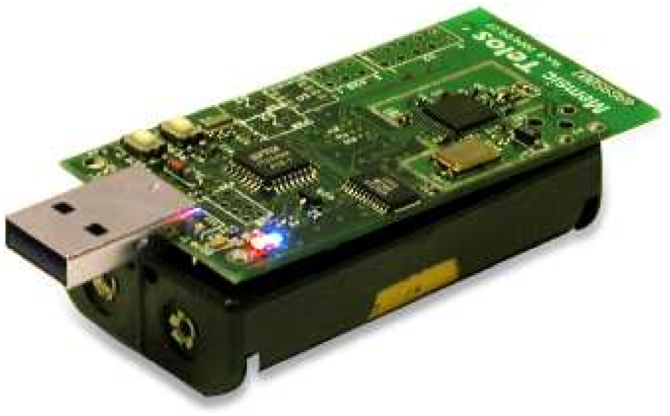


Figure 16: TelosB (©Advanticsys^{TM})

5.2.3.5 Virtenio Preon32

The *Preon32* , showed in figure - 17, is much capable than *TelosB* but battery operated resource-constrained device. It also uses IEEE 802.15.4 2.4 GHz MAC/PHY for its wireless communication. It does not contain any capabilities off the shelf. However, an expansion (i,e - daughter) board is given with the motherboard to be added for gaining sensing capabilities. The expansion kit contains six capabilities off the shelf - temperature, humidity, light intensity, magnetometer, accelerometer and gyroscope sensing capabilities. However, for the prototype implementation, we only used temperature and humidity sensing capabilities. It also has a USB to UART SPI (Serial Programming Interface) to program the device. The programming language is Java and it has a modified JVM inside the device. The libraries are completely different and do not support most of the Java standard libraries. The modified Java Runtime's API is provided as a documentation. For battery operation, the *Preon32* has an enclosing connector and requires 9V DC to operate.

Figure 17: Preon32 Components (©Virtenio^{TM})

5.2.4 Programming Language and IDE Used

For the application and the GUI, we used PHP as the server-side programming language, HTML5, CSS3 and Javascript as the client side markup/scripting language.

For developing the prototype, C++11 programming language standard was used. Eclipse-CDT was used as the editor and GNU-GCC tool-chain was used as the compiler and linker.

5.3 Prototype Architecture

The prototype architecture is shown in figure - 18. In the prototype architecture, the Anti-Fire, and the Smart HVAC application is deployed as SaaS applications on top of the Cloud Foundry. The Cloud Foundry is backed by the traditional IaaS. An OpenStack instance deployed over a VirtualBox instance was used to provide the traditional IaaS provisioning capabilities. The PaaS relied on the provided traditional IaaS and provisioned separate containers for the Anti-Fire and the Smart HVAC application. It is to be noted that at the time of writing this thesis, even though the overall architecture of the Cloud Foundry remained the same, it has made switch to the container based deployment of an application.

An extension for the Cloud Foundry was developed so that it can also provision the IoT devices as resources using the IoT IaaS.

In the following subsections, the details of the IoT IaaS and the covered functionality within the prototype implementation will be discussed. The deployed applications are discussed in brief after that. As the detailed interaction was covered in chapter - 4, hence, we will summarize the validation of the prototype in short.

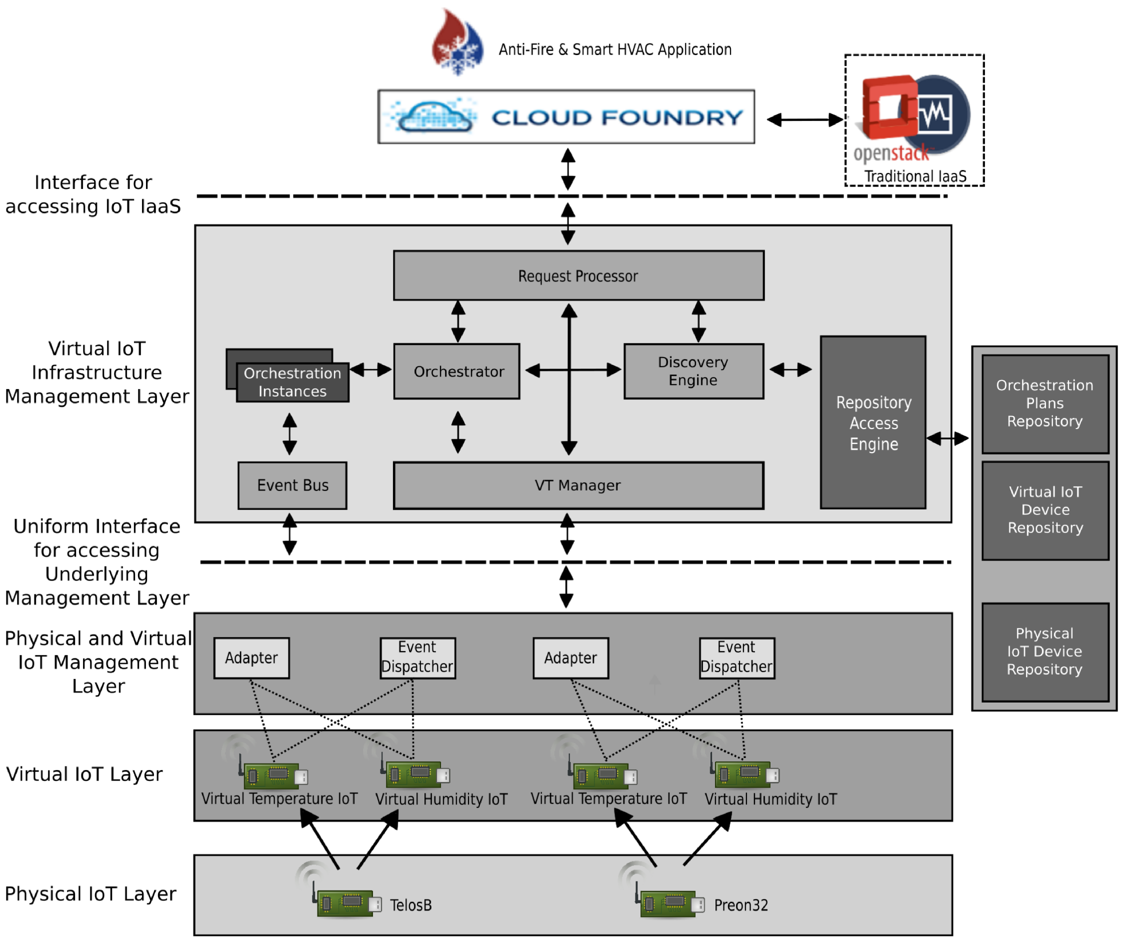


Figure 18: Prototype Architecture of IoT IaaS

5.3.1 Prototyped IoT IaaS

In this section, we will take the bottom up approach and cover from the *Physical IoT Device Layer* upto the *Interface For Accessing IoT IaaS*. The two sets of interfaces are also covered. We will briefly describe to what extent the functionality was covered, how we implemented it, and what was excluded. These are described below in subsequent subsections.

5.3.1.1 Physical IoT Device Layer

For our prototype implementation, we excluded the IoT device with actuation capabilities. We only focused on the IoT devices consisting of sensing capabilities. In our prototype implementation, this layer consisted only of two vendor's devices, *TelosB* and *Preon32* .

5.3.1.2 Virtual IoT Device Layer

As in the prototype implementation we only considered two types of sensing capabilities, namely, temperature sensing and humidity sensing, this layer consists the Virtual Temperature IoT Device and the Virtual Humidity IoT Device.

5.3.1.3 Physical and Virtual IoT Management Layer

In our prototype scenario, we assumed that the *Global Contextual Information* of the device was already published in the *Physical IoT Device Repository.* This is because, the underlying IoT devices did not have any GPS module, and even so, GPS module would not work in a lab environment, without a clear view of the sky. Hence, although the IoT IaaS can detect the devices automatically and probe them, the devices themselves do not publish the location information to the repository. As discussed earlier, the location information is used as the criteria for the Anti-Fire and Smart HVAC application. For the above reasoning, the *Repository Access*

Engine and the *Publisher* were excluded from the implementation as well.

The *Adapter* and the *Event Dispatcher* was implemented.

5.3.1.4 Uniform Interface For Accessing Underlying IoT Devices

For this interface, there are two types of operation - Control and Data plane operations.

In the Control Plane - *"create-viot"* and *"delete-viot"* were implemented. The *"publish-capabilities"*, *"context"*, and *"notify-update"* were excluded. As the *Global Contextual Information* is already published within the *Physical IoT Device Repository*, we excluded the interfaces that supported the automation of the publication of such information.

In the Data Plane - *"data"* and *"notify-event"* were implemented. The The *"handle-event"* was left out as our assumption did not consider the IoT devices with actuation capabilities.

5.3.1.5 Virtual IoT Infrastructure Management Layer

We implemented a simple *Request Processor*. It takes the request from the application via the *Interface For Accessing the IoT IaaS*. The *Request Processor* is then either forward the request to *Orchestrator* or the *VT Manager*. The *Orchestrator* was implemented using a simple queue. The sequence within the orchestration plan dictates how the orchestration will be carried out. The sequence is executed in a FIFO (First In First Out) order. While executing the sequence, if any of the previous operations fails, the whole orchestration fails. The *Discovery Engine* provides the list of suitable physical IoT device as a list of *UUID*s. It uses the *Repository Access Engine* to access the *Physical IoT Device Repository*. The *VT Manager* is one of the complex and large components within this layer. It sends the primitives to the underlying IoT device and virtualizes them with given parameters. Once virtualized, the IoT IaaS layer stores the *UUID* of the virtual IoT device in the *Virtual IoT Device Repository.* The events and the callbacks are also kept in a database within the IaaS. This simulates the behavior of having a global view of the underlying IoT device network. The *Event Bus* uses this information to route the event to all of the registered event handlers. All of the repositories were implemented as an in-memory database.

5.3.1.6 Interface For Accessing The IoT IaaS

The cloud access interfaces that are required for the scenario were implemented others were excluded. Hence the covered interfaces were - *create, delete, list single, add event, delete event, add callback*, and *delete callback* were implemented. Rest were excluded.

5.3.1.7 The Anti-Fire Application

The Anti-Fire application deployed on top of the PaaS has a requirement of 1 sample per seconds (i,e - a sample rate of 1000 ms). It does not define any predefined threshold and it uses pull style data acquisition. That is the Anti-Fire application requests the data every seconds, even though the underlying virtual IoT device can send the data. It then plots a running graph of the data to help user visualize the data. The requirement for the Anti-Fire application is the temperature capability and the location. The location was a predefined value and we set it to “TSELab”.

5.3.1.8 The Smart HVAC Application

The Smart HVAC application uses events and notifications and has more relaxed requirements for sampling rate. It defines a threshold for the temperature at 27 degree Celsius and the relative humidity at 50%. Hence, it requires two capabilities. The location is chosen as the “TSE-Lab”. The application task contained the event handler. This way the application can show the temperature of humidity data once the threshold is crossed. The application task was developed using the HTTP event delivery mechanism [47].

5.3.1.9 Summary

The prototype applications were able to provision both simple virtual IoT devices and composite virtual IoT device. The underlying virtualized IoT devices were made possible through the usage of *Node Level Virtualization.* The provisioning of composite virtual IoT device validates the *Orchestration* mechanism. Two different vendor devices were used to validate the *Uniform Interface For Accessing Underlying IoT Devices.* We simulating the publication of information to the repository by pushing a predefined location value whenever the IoT device was connected to the IoT IaaS. The discovery mechanism was able to pick the published information automatically and use it accordingly. This validates the *Publication and Discovery* mechanism for matching the requirements against the properties of the underlying IoT device. Finally, The Cloud Foundry was extended to use the *Interface for accessing IoT IaaS* to manage, and access IoT devices. Thus, completing the interface validation. Therefore, we conclude that the prototype implementation validated the proposed IoT IaaS architecture.

5.4 Performance Measurement

In this section we first describe the performance metric, then the setup for evaluating the performance metric, finally, we conclude the section by presenting the result obtained and analyzing them.

5.4.1 Performance Metric

Two performance metric was selected for measurement.

One is the end to end provisioning delay. Our goal was to observe the end to end provisioning delay when the underlying physical IoT device slowly approaches its capacity. This means the number of virtual IoT devices provision on top a single physical IoT device was slowly increased and the end to end provisioning delay was observed. This performance metric would give us the impact of node level virtualization on the underlying IoT device.

The second performance metric selected was the end to end orchestration delay. This metric will help us to decide if the orchestration delay is suitable for different types of deployment. Our goal was to investigate the real-time deployment and the non-real time deployment and figure out if the end to end delay is acceptable or not.

5.4.2 Experiment Setup

To evaluate the first performance metric, we first virtualize each of the *TelosB* and *Preon32* , and slowly increased the number of guest virtual IoT device running on top of them, to a maximum of four. The number four (4) was chosen through experiment, which led to conclude that it is the highest number of virtual IoT device that a *TelosB* can handle without running out of memory. We repeated the test ten (10) times to reduce the fluctuation of the delay and took an average of it. The ten (10) test number is arbitrarily taken and increasing the test run did not give us any different insights. We also ran the non-virtualized *TelosB* and *Preon32* and compare the results with the virtualized one.

The second performance metric was evaluated by issuing orchestration over different physical IoT device. In this case, we limited the capabilities of the underlying IoT to single exclusive capability in two different physical IoT device. Then the smart HVAC application can only be orchestrated over this two physical device. We ran the test twenty (20) iterations. The twenty was chosen arbitrarily as increasing the number (for example - 25, 30) or reducing the number a little bit (for example - 15, 10) did not give us new insights on the obtained results.

5.4.3 Results and Analysis

For the first performance metric evaluation, the results are shown in figure - 19. As expected, the overhead for creating virtual IoT devices was significant. In case of *TelosB* for instance, the highest and lowest overhead (compared to the case where physical sensors are used) is 252.5% (69.8ms vs 19.8ms) and 152.02% (49.9ms vs 19.8ms) respectively. For the *Preon32* , it is approximately 496.5% (156.3ms vs 26.2ms) and 417.9% (135.7ms vs 26.2ms).

Although *Preon32* is more capable than *TelosB* , our suspicion is that the JVM overhead within the *Preon32* contributes to overall delay. This is because the JVM manages the virtualized IoT device within the device. Hence, context switching may induce additional overhead. For *TelosB* , as it uses C language, it is more "close to the metal" hence less overhead.

One key note to be taken from the results is that the although the virtualization overhead is high percentile wise, they are not high as an absolute number. And it occurs only once (at the time of virtualization). Hence, it may be well worth to trade this overhead for efficiency (both cost and energy), as once deployed there is no difference between a physical IoT device and a virtual IoT device from the application's point of view.

The result of the second experiment is shown in figure - 20. The bar represents the delay when the Smart HVAC application is orchestrated over two virtual sensors running in different physical sensors. The experienced delay for overall

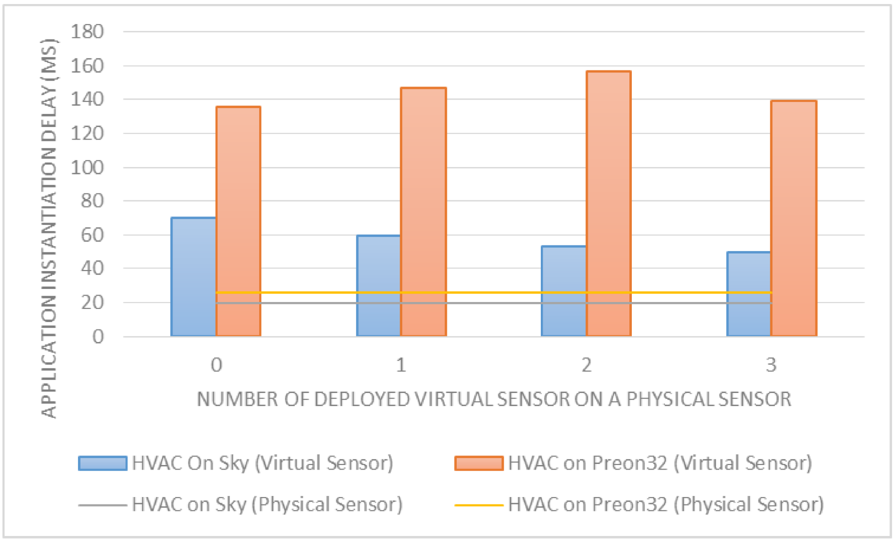


Figure 19: Application Instantiation Delay

orchestration was approximately 1 second (max: 1214ms, min: 1127ms, average: 1166.5ms), which is not very promising for real-time application instantiation with tighter provisioning deadline. However, this delay might be acceptable for those applications where the application start time is not critical.

5.5 Conclusion

In this chapter, we showed how the node level virtualization can achieve energy efficiency. After that, the prototyped architecture was discussed in brief along with the software and hardware used to develop it. Then the detailed prototype architecture and the applications were discussed to summarize the prototype validation. Finally, the performance measurement was made, the result was shown, and analyzed.

We conclude our thesis in the next chapter by focusing on the summary of the thesis and the future work to be done.

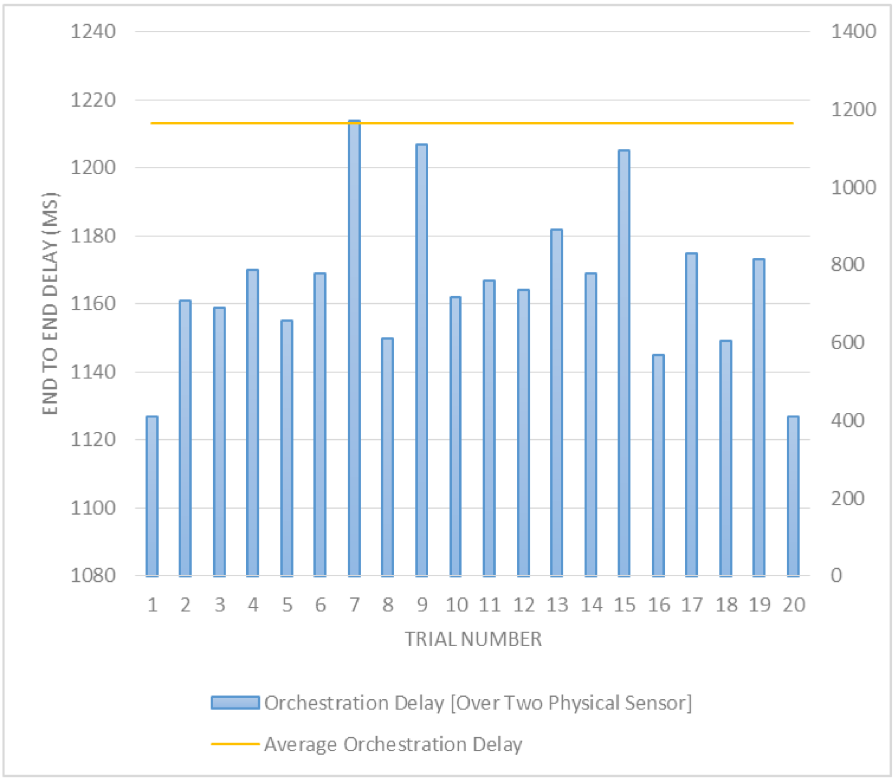


Figure 20: Delay for end to end orchestration

Chapter 6

Conclusion

In this chapter, we will first highlight the contributions of this thesis and then focus on the possible future research direction.

6.1 Contributions Summary

Internet of Things brings up a vast amount of applications in all sorts of different domains. However, provisioning the IoT devices through the cloud to achieve cost efficiency is a challenge. In the early days of IoT, the cloud was used to manage the wide spear deployment of the IoT devices and IoT device was facing high coupling with the application deployed on it. Then many solutions emerged to solve coupling issue with the IoT device and the applications. But the cost efficiency challenge was not addressed. This problem is hard to tackle due to several issues imposed by the inherent nature of the IoT devices. Due to the heterogeneous nature of the IoT environment, it becomes a challenge to provide a means to share the underlying capabilities of the IoT devices with several applications. An IoT IaaS that can provide capabilities sharing can achieve cost efficiency at the IoT device level.

In order to understand the requirements, we presented a motivating use-case in a smart home environment. The use-case had two applications with overlapping capability requirements. We determined several requirements for an IoT IaaS in order to allow the sharing of capabilities with several applications. Node level virtualization is one such key requirements. However, along with it, additional requirements were identified to realize and share the capabilities in a uniform way. The publishing and discovery mechanism, along with the orchestration mechanism plays an essential role in realizing the IoT IaaS. Finally, we identified that in order to handle the heterogeneity at the physical IoT device level and to allow the users to provision and access the IoT devices through the IoT IaaS, two sets of a high-level interface is required. Based on the requirements we evaluated the existing full blown IoT IaaS architecture and the frameworks that can be used within an IoT IaaS. None of the state-of-the-art was able to fulfill all the requirements.

We then proceeded to propose an IoT IaaS architecture that can fulfill the derived requirements. The proposed architecture leveraged node level virtualization as a feature within the IoT device to provide virtual IoT devices. The heterogeneity issue was solved by providing the Adapter components in the Virtual and Physical IoT device layer in the proposed architecture and by introducing the uniform interface for accessing the IoT device. This provided a uniform data and control plane to the Virtual IoT Infrastructure management layer in the proposed architecture, regardless of the underlying IoT device's proprietary interface. Publishing and discovery were normalized through modeling, stored in a database and matched against requirements. We showed an orchestration modeling that focuses on how orchestration on an IoT IaaS should be facilitated. Finally, a mean to expose the functionality of the IoT IaaS was exposed to the user. For all the interface, we used REST paradigm and used IoT friendly description model. In the end, we show how the different components within the IoT IaaS interact with each other both at provision time and at runtime.

We first showed how node level virtualization achieves energy efficiency by conduction an experiment. We described the experimental settings, methodology, results and analysis. Then, a prototype for validating the IoT IaaS was implemented using the subset of the motivating use-case. A subset of the proposed architecture was implemented and a subset of the two applications (Anti-Fire and the Smart HVAC) was developed and deployed as a SaaS on top of the Cloud Foundry, a PaaS. An extension of the PaaS was made to communicate to the IoT IaaS by leveraging the interface for accessing the IoT IaaS.

Finally, two performance metric was defined, and experimental setup was defined. The results from the experiment was shown and analyzed. From the result and analysis, it was apparent that even the virtualization overhead seems high relative to physical device deployment, it will be worth trade-off if the device is provisioned for a longer time as the absolute value is not very high. For the end to end orchestration, it was apparent that the orchestration mechanism proposed would not be able to satisfy the hard provisioning deadline of less than one second for real-time applications. However, it is still a good choice for non real-time applications.

6.2 Future Research Direction

In this work, we did not include network-level virtualization. In future work, network-level virtualization can be incorporate to provide both node and network virtualization for applications. Some of the key aspects of an IaaS were omitted in the proposed IoT IaaS. Security is one of such. A automated asymmetric public/private key exchange protocol and mechanism can be incorporate to secure the IoT devices. A user authentication mechanism can also be attached to the security mechanism to allow end to end (i,e - application to IoT device) security. A image service for underlying IoT device can be introduce to allow reprogramming of the IoT device, this can allow the provisioning of real-time applications and switching back to virtualized provisioning when the real-time application are no longer needed. Virtual IoT device migration is still an underlooked research domain. The feasibility and requirement can be investigate further to reach to a decision about its impact to the IoT applications.

References

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[2] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[3] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[4] Imran Khan, Fatna Belqasmi, Roch Glitho, Noel Crespi, Monique Morrow, and Paul Polakos. Wireless sensor network virtualization: early architecture and research perspectives. *IEEE Network*, 29(3):104–112, 2015.

[5] Jianliang Zheng and Myung J Lee. A comprehensive performance study of ieee 802.15. 4. *Sensor network operations*, 4:218–237, 2006.

[6] Ferran Adelantado, Xavier Vilajosana, Pere Tuset-Peiro, Borja Martinez, Joan Melia-Segui, and Thomas Watteyne. Understanding the limits of lorawan. *IEEE Communications Magazine*, 55(9):34–40, 2017.

[7] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 78–82. ACM, 2007.

[8] Nandakishore Kushalnagar, Gabriel Montenegro, and Christian Schumacher. Ipv6 over low-power wireless personal area networks (6lowpans): overview, assumptions, problem statement, and goals. Technical report, 2007.

[9] J Nieminen, T Savolainen, M Isomaki, B Patil, Z Shelby, and C Gomez. Ipv6 over bluetooth (r) low energy. Technical report, 2015.

[10] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (coap). 2014.

[11] N Sornin, M Luis, T Eirich, T Kramp, and O Hersent. Lora specification 1.0. *LoRa™ Alliance, San Ramon*, 2015.

[12] Dinesh Thangavel, Xiaoping Ma, Alvin Valera, Hwee-Xian Tan, and Colin Keng-Yan Tan. Performance evaluation of mqtt and coap via a common middleware. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*, pages 1–6. IEEE, 2014.

[13] Adam Dunkels. uip-a free small tcp/ip stack. *The uIP*, 1, 2002.

[14] Mike Botts and Alexandre Robin. Opengis sensor model language (sensorml) implementation specification. *OpenGIS Implementation Specification OGC*, 7(000), 2007.

[15] Nengcheng Chen, Chuli Hu, Yao Chen, Chao Wang, and Jianya Gong. Using sensorml to construct a geoprocessing e-science workflow model under a sensor web environment. *Computers & Geosciences*, 47:119–129, 2012.

[16] Cullen Jennings, Jari Arkko, and Zach Shelby. Media types for sensor markup language (senml). 2012.

[17] Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. An iot gateway centric architecture to provide novel m2m services. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 514–519. IEEE, 2014.

[18] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, security threats, and solutions. *ACM Computing Surveys (CSUR)*, 45(2):17, 2013.

[19] Violeta Medina and Juan Manuel García. A survey of migration mechanisms of virtual machines. *ACM Computing Surveys (CSUR)*, 46(3):30, 2014.

[20] Jack Lo. Vmware and cpu virtualization technology. *World Wide Web electronic publication*, 2005.

[21] Ashiq Khan, Alf Zugenmaier, Dan Jurca, and Wolfgang Kellerer. Network virtualization: a hypervisor for the internet? *IEEE Communications Magazine*, 50(1), 2012.

[22] Imran Khan, Fatima Zahra Errounda, Sami Yangui, Roch Glitho, and Noël Crespi. Getting virtualized wireless sensor networks' iaas ready for paas. In *Distributed Computing in Sensor Systems (DCOSS), 2015 International Conference on*, pages 224–229. IEEE, 2015.

[23] Anura P Jayasumana, Qi Han, and Tissa H Illangasekare. Virtual sensor networks-a resource efficient approach for concurrent applications. In *Information Technology, 2007. ITNG'07. Fourth International Conference on*, pages 111–115. IEEE, 2007.

[24] Imran Khan, Fatna Belqasmi, Roch Glitho, and Noel Crespi. A multi-layer architecture for wireless sensor network virtualization. pages 1–4, 2013.

[25] Carla Mouradian, Fatima Zahra Errounda, Fatna Belqasmi, and Roch Glitho. An infrastructure for robotic applications as cloud computing services. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 377–382. IEEE, 2014.

[26] Salvatore Distefano, Giovanni Merlino, and Antonio Puliafito. Device-centric sensing: an alternative to data-centric approaches. *IEEE Systems Journal*, 11(1):231–241, 2017.

[27] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[28] Borja Sotomayor, Rubén S Montero, Ignacio M Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet computing*, 13(5), 2009.

[29] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[30] Ilango Sriram and Ali Khajeh-Hosseini. Research agenda in cloud technologies. *arXiv preprint arXiv:1001.3259*, 2010.

[31] Rajkumar Buyya, James Broberg, and Andrzej M Goscinski. *Cloud computing: Principles and paradigms*, volume 87. John Wiley & Sons, 2010.

[32] Nikolas Roman Herbst, Samuel Kounev, and Ralf H Reussner. Elasticity in cloud computing: What it is, and what it is not. In *ICAC*, volume 13, pages 23–27, 2013.

[33] Sunanda Bose, Nandini Mukherjee, and Sujoy Mistry. Environment monitoring in smart cities using virtual sensors. In *Future Internet of Things and Cloud (FiCloud), 2016 IEEE 4th International Conference on*, pages 399–404. IEEE, 2016.

[34] Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. Security for the internet of things: a survey of existing protocols and open research issues. *IEEE Communications Surveys & Tutorials*, 17(3):1294–1312, 2015.

[35] Sunanda Bose and Nandini Mukherjee. Sensiaas: A sensor-cloud infrastructure with sensor virtualization. In *Cyber Security and Cloud Computing (CSCloud), 2016 IEEE 3rd International Conference on*, pages 232–239. IEEE, 2016.

[36] Maria Fazio and Antonio Puliafito. Cloud4sens: a cloud-based architecture for sensor controlling and monitoring. *IEEE Communications Magazine*, 53(3):41–47, 2015.

[37] Sunanda Bose, Atrayee Gupta, Sriyanjana Adhikary, and Nandini Mukherjee. Towards a sensor-cloud infrastructure with sensor virtualization. In *Proceedings of the Second Workshop on Mobile Sensing, Computing and Communication*, pages 25–30. ACM, 2015.

[38] Kashif Sana Dar, Amir Taherkordi, and Frank Eliassen. Enhancing dependability of cloud-based iot services through virtualization. In *Internet-of-Things Design and Implementation (IoTDI), 2016 IEEE First International Conference on*, pages 106–116. IEEE, 2016.

[39] Federica Paganelli, Stefano Turchi, and Dino Giuli. A web of things framework for restful applications and its experimentation in a smart city. *IEEE Systems Journal*, 10(4):1412–1423, 2016.

[40] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[41] Matjaz B Juric. Wsdl and bpel extensions for event driven architecture. *Information and Software Technology*, 52(10):1023–1043, 2010.

[42] Wei Chen, Jun Wei, Guoquan Wu, and Xiaoqiang Qiao. Developing a concurrent service orchestration engine based on event-driven architecture. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 675–690. Springer, 2008.

[43] Kashif Dar, Amir Taherkordi, Harun Baraki, Frank Eliassen, and Kurt Geihs. A resource oriented integration architecture for the internet of things: A business process perspective. *Pervasive and Mobile Computing*, 20:145–159, 2015.

[44] Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. Patterns for software orchestration on the cloud. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*, page 17. The Hillside Group, 2015.

[45] Fatna Belqasmi, Roch Glitho, and Chunyan Fu. Restful web services for service provisioning in next-generation networks: a survey. *IEEE Communications Magazine*, 49(12), 2011.

[46] David Bernstein. Cloud foundry aims to become the openstack of paas. *IEEE Cloud Computing*, 1(2):57–60, 2014.

[47] Elio Damaggio, Martin Thomson, and Brian Raymor. Generic event delivery using http push. 2016.