UNIVERSITY OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEGREE PROGRAMME IN WIRELESS COMMUNICATIONS ENGINEERING

MASTER'S THESIS

CONTAINER-BASED MICROSERVICE ARCHITECTURE FOR LOCAL IOT SERVICES

Author	Johirul Islam
Supervisor	Dr. Erkki Harjula
Second Examiner	Prof. Mika Ylianttila
Technical Advisor	Tanesh Kumar

May 2019

Islam J. (2019) Container-based Microservice Architecture for Local IoT services. University of Oulu, Faculty of Information Technology and Electrical Engineering, Degree Programme in Wireless Communications Engineering. Master's Thesis, 68 p.

ABSTRACT

Edge services are needed to save networking and computational resources on higher tiers, enable operation during network problems, and to help limiting private data propagation to higher tiers if the function needing it can be handled locally. MEC at access network level provides most of these features but cannot help when access network is down. Local services, in addition, help alleviating the MEC load and limit the data propagation even more, on local level. This thesis focuses on the local IoT service provisioning. Local service provisioning is subject to several requirements, related to resource/energy-efficiency, performance and reliability.

This thesis introduces a novel way to design and implement a Docker container-based micro-service system for gadget-free future IoT (Internet of Things) network. It introduces a use case scenario and proposes few possible required micro-services as of solution to the scenario. Some of these services deployed on different virtual platforms along with software components that can process sensor data providing storage capacity to make decisions based on their algorithm and business logic while few other services deployed with gateway components to connect rest of the devices to the system of solution. It also includes a state-of-the-art study for design, implementation, and evaluation as a Proof-of-Concept (PoC) based on container-based microservices with Docker. The used IoT devices are Raspberry Pi embedded computers along with an Ubuntu machine with a rich set of features and interfaces, capable of running virtualized services.

This thesis evaluates the solution based on practical implementation. In addition, the thesis also discusses the benefits and drawbacks of the system with respect to the empirical solution. The output of the thesis shows that the virtualized microservices could be efficiently utilized at the local and resource constrained IoT using Dockers. This validates that the approach taken in this thesis is feasible for providing such services and functionalities to the micro and nanoservice architecture. Finally, this thesis proposes numerous improvements for future iterations.

Keywords: Virtualization, Containerization, Docker, IoT, Orchestration, Microservices.

TABLE OF CONTENTS

FOREWORD

This thesis has accomplished at Center for Wireless Communications-Networks and Systems (CWC-NS) research group under Center for Wireless Communications in the University of Oulu, Finland. It has been funded by the Academy of Finland project for Industrial Edge and MEC-AI research project collaborated by the Centre for Ubiquitous Computing, Centre for Wireless Communications; and Research Unit of Mathematical Sciences headed by professors Jukka Riekki, Mika Ylianttila; and Mikko Sillanpää. I would like to pay special thanks to my supervisors Dr. Erkki Harjula and Prof. Mika Ylianttila for their direct nonstop support throughout the entire period of my master's thesis journey. I am very grateful to my technical supervisors Tanesh Kumar, Ahsan Manzoor who were doctoral students at CWC-NS research group in the University of Oulu. I also would like to pay my gratitude to Pekka Karhula, Research Scientist at VTT Technical Research Centre of Finland Ltd, Finland for his partial contribution to this PoC implementation.

I am very much grateful to Center for Wireless Communications as well as the University of Oulu, which provided an academic platform and learning environment to think and grab actual things behind the scenario as out of the box that enhance my skillset. I would like to thank all of my family members for their love and support especially my younger brother Mahidul Islam. I would like to pay thanks to my friends especially Md. Ziaul Hoque, Md Shofiqul Islam and Md Delowar Hossain for their company and fun throughout my master's journey. Finally, I am also grateful to Finnish education authority to provide me the opportunity to pursue my master's degree.

Oulu, May, 27 2019

Johirul Islam

LIST OF ABBREVIATIONS AND SYMBOLS

AI	Artificial Intelligence
ATT	Attribute control
API	Application Programming Interface
BLE	Bluetooth Low Energy
CC	Cloud Computing
CCTV	Close Circuit TV
CLI	Command Line Interface
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
CSI	Channel State Information
DTLS	Datagram Transport Layer Security
EaaS / XaaS	Everything as a Service
EC	Edge Computing
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GND	Ground
GPIO	General Purpose Input Output
GUI	Graphical User Interface
HCI	Host Controller Interface
HTTP	Hyper Text Transfer Protocol
I/O	Input / Output
IaaS	Infrastructure as a Service
ICT	Information and Communication Technology
IETF	Internet Engineering Task Force
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IoE / IoX	Internet of Everything
IoT	Internet of Things
IP	Internet Protocol
IR	Infrared Radiation
JSON	JavaScript Object Notation
L2CAP	Logical Link Control and Adaptation Protocol
LED	Light Emitting Diode
LXC	LinuX Containers
M2M	Machine-to-Machine
MC	Mist Computing
MEC	Mobile Edge Computing, Multi-access Edge Cloud
MITM	Man-In-The-Middle
NAT	Network Address Translation
NFV	Network Function Virtualization
npm	node package manager
OOP	Object Oriented Programming
OS	Operating System
OUT	Output
PaaS	Platform as a Service
PIR	Passive Infrared Radiation

PoC	Proof of Concept
RAM	Random Access Memory
REST	Representation State Transfer
RFC	Request for Comments
RFID	Radio Frequency Identification
RPi	Raspberry Pi
SaaS	Software as a Service
SBC	Single Board Computer
SDLC	Software Development Life Cycle
SMS	Short Message Service
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UI	User Interface
UID	Unique Identifier
UML	Unified Modelling Language
URI	Universal Resource Identifier
URL	Universal Resource Locator
VCC	Common Collector Voltage
VM	Virtual Machine
VPN	Virtual Private Network
WSN	Wireless Sensor Network
YAML	Yet Another Markup Language
Class	Blueprint of an object in OOP paradigm
Client	A device that will served by server machine
DevOps	Developers + Operations
Host	A computer works as server that handles and process requests
kernel	Core computer program that control everything of an OS
NoOps	Common features in microservices
Port	The window that makes communication between server and client
B	Byte
kB	kilobytes
MB	Megabytes

1 INTRODUCTION

The current digital world is mostly surrounded by gadgets. Users get various services using these smart devices. Gadgets, such as smartphones, tablets, PDAs among others are already been heavily used in various aspect in daily life. For example, gadgets play a key role in applications such as banking, healthcare, and logistics. With the evolution of recent enabling technologies such as IoT, edge/fog computing, virtualization and blockchain together with high-speed 5G networks are driving this digitalization to the next level. The new digital vision would assume gadget-free services, i.e. users can able to access all required services which were previously available through gadgets, now can be accessed without gadgets [14]. The nearby surroundings will be smart enough to recognize the particular context and offer digital services accordingly. Various services and computations will be embedded in smart spaces and services are given to the valid users whenever required and disappeared when not required. These smart and gadget-free services are also referred to as “Service Bubble” which are delivered to gadget-free users [23]. The development of such smart services needs various technological capabilities such as smart and low power sensor and communication technologies, printed electronics; and cloud computing. Service composition and decomposition require by various software, hardware resources for computations and processing as the services should not be interrupted with the mobility of users having no hand-carry gadgets [45]. Thus, this vision of current gadget-centric world towards future user-centric world require various advanced technological developments.

1.1 Purpose and Objective

In this time, the digital services reside on the cloud or the edge are accessible with the help of an active internet connection through several computational resources. These services may disrupt and face security threats all along the communication path between a remote server and a client.

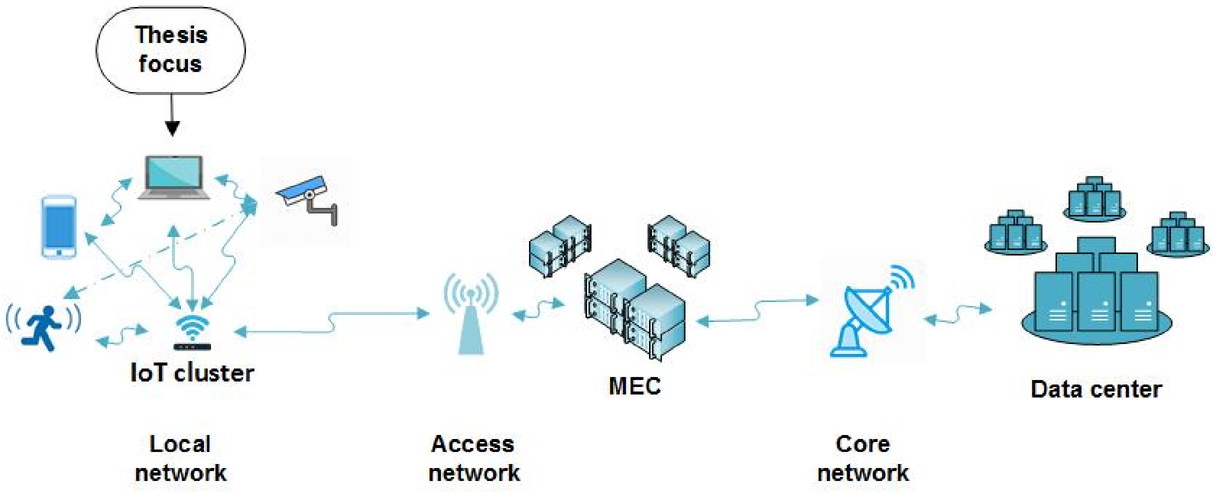


Figure 1. Future IoT network architecture [23].

All these obstacles in between a remote server and a client are continuously pushing us to move services from remote to local subsystem. Currently, there is no such model that could help us to move services and resolve the above problem. This model should be able to deploy these digital services into the local IoT environment so that it reduces the computational complexity and hence accessible in “offline” mode too. Here, this thesis proposes a novel way and shows how to move and deploy services into the local system.

1.2 Scope

A virtualized system of architecture is required for on-demand service composition based on the available hardware and software resources at the current location of a user. Hence, there is a need for a novel decentralized service model that suited for gadget-free devices in local IoT network. Harjula et al. [23] proposed a nanoEdge conceptual model that addresses this problem. This model will generate on-demand services with the collaboration of network, data processing, storage; and security services. This thesis is the implementation part of that work. This PoC exemplifies and provides a reliable and smart solution for future distributed gadget-free service for local IoT components with a real-world use case scenario.

Rest of this thesis is design as follows – Chapter 2 presents comprehensive related background technologies related to the fields of the internet of things and wireless sensor networks. Chapter 3 provides detailed information about the design and implementation of Docker-based micro-services in IoT. Chapter 4 contains a detailed description of the evaluation results and comparison of the implemented technologies with similar other IoT technologies. Finally, the implementation of this thesis is discussed in chapter 5 with possible future improvements.

2 RELATED WORK

This chapter will cover the theoretical part for a better understanding along with the definitions approaching in this thesis. Then this thesis will study about the chosen technology step by step and move towards the design and implementation of Docker-based micro-service for IoT with several protocol stacks.

2.1 Internet of Things

Kevin Ashton [1], co-founder of the Auto-ID Center at MIT, introduced first the term Internet of Things (IoT) for a presentation he made to Procter & Gamble (P&G). He used this in RFID technologies in 1999 to grab the attention of P&G's senior management. A couple of years have been passed after the first appearance of IoT and being major popular research and industrial topics in the area of Information and Communication Technology (ICT). The IoT term appears so frequently but not well defined in ICT and raise the doubt of what would be real elements of the terminology [1]. IoT is a system without requiring human interaction of interrelated computing devices that are provided with Unique Identifiers (UIDs) with the ability to transfer data over a network [2]. In general, an IoT system often consists of three basic collective components such as device, gateway; and cloud [69]. A device includes hardware and software to interact with the rest of the world whereas a gateway enables devices to connect the internet to reach cloud services. A cloud is a list of resources to process client request from a device through a gateway. Figure 2 represents the traditional architecture of an IoT system.

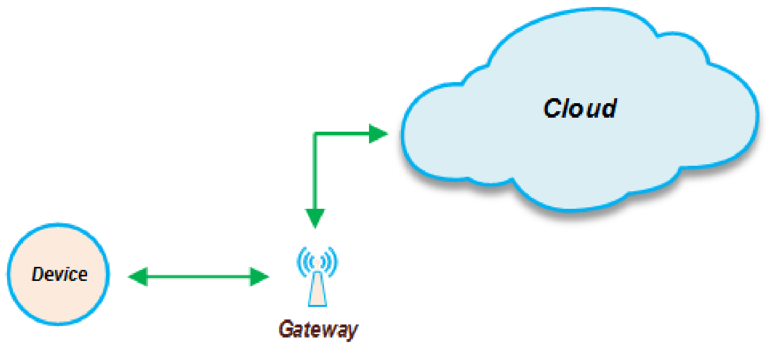


Figure 2. IoT traditional architecture.

Machine-to-Machine (M2M) communication enables ubiquitous connectivity among autonomous devices without human interaction or with minimal human interaction to support data transfer among sensors and actuators [65]. It is a major concern to IoT infrastructure. One of the key challenges in M2M communication is to provide an effective way for multiple access in the network and to minimize network overload. In M2M communication, a network domain comprises of a core network, access network, M2M service capabilities, M2M applications. M2M devices connect to network domain either directly or indirectly through a gateway. Nowadays, centralized cloud is not enough to describe the IoT architecture. Hence, Edge Computing (EC), simple form of a cloud with limited computing capacity, has been introduced to increase the computing capacity and reduce the energy consumption in M2M communication [65]. Therefore, modern IoT may consist of cloud, edge; and end-devices. Figure 3 shows the modern architecture of an IoT system.

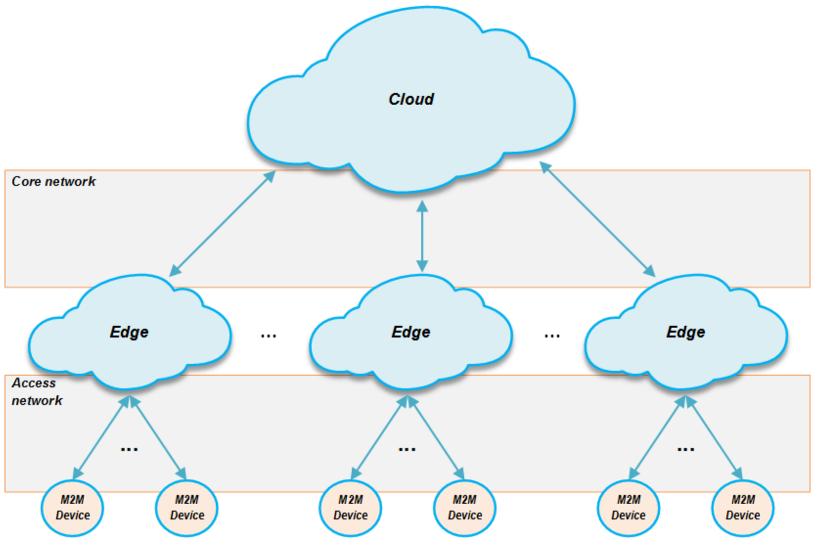


Figure 3. IoT modern architecture.

IoT covers a variety of protocols, domains, and applications. Following subsections describe the existing technologies that are used in current IoT systems.

2.1.1 IoT Protocols

The main objective of IoT is the integration of billions of smart electronic objects so that they can communicate and exchange data among them. The key requirements of IoT include low latency, high bandwidth, privacy and security. To satisfy these requirements, IoT requires a set of rules or protocols by dividing the entire communication system into different multiple layers, each having their own challenges and requirements. Researchers and developers have come up with new IoT communication protocols in order to make successful autonomous connectivity among IoT devices. These protocols are enabling the empirical communication of IoT devices. Figure 4 shows the layer-wise heterogenous recent IoT protocol standard.

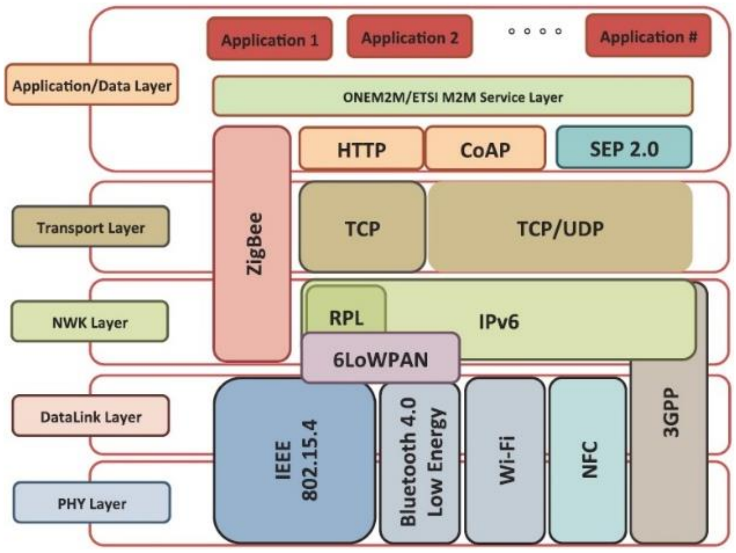


Figure 4. Heterogeneous IoT standard protocols [41].

All these protocols have designed for different purposes that are mostly related to the role of the specific layer they operate on. On the application layer, MQTT and CoAP are the most widely used protocols for IoT communications, particularly for low power devices. These protocols have a small payload that enabling faster communications by using less power to enhance and prolong battery lifetime.

2.1.1.1 CoAP

HTTP is a resource-consuming internet protocol due to high overhead for e.g. synchronous operation, text-based format; and header structure. Chan and Kunz et al. [39] proposed CoAP that allows the simplest communication protocol, especially for electronic devices. IETF defined this by the open standard RFC 7252, that runs on top of UDP by default but can also be implemented over TCP, DTLS or SMS. This protocol is designed for low-power devices such as sensors and actuators. CoAP is affordable with resource constraints when network bandwidth is critical. Figure 5 represents the network stack offered by CoAP. [42]

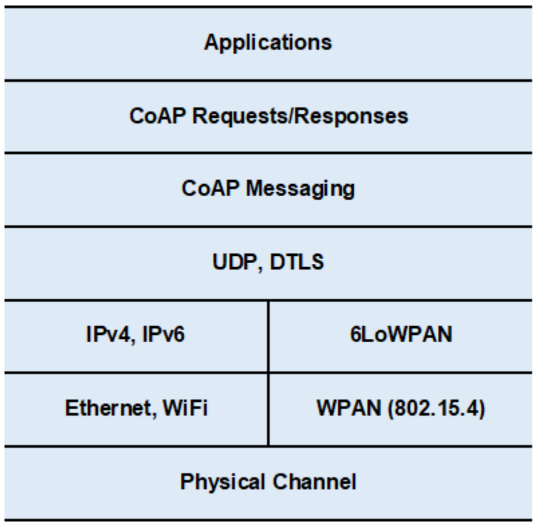


Figure 5. Network protocol stack with CoAP [42].

CoAP enables web integration with URIs and MIME types like HTTP. However, CoAP clients can talk to HTTP servers and vice versa with an interface CoAP and HTTP. Unlike HTTP, CoAP has its own response codes. This protocol is fit into a single datagram with a single frame at the IEEE 802.15.4 layer or Ethernet. It can avoid fragmentation at underlying layers, especially at the link layer. Figure 6 represents the CoAP subsystem architecture. [42]

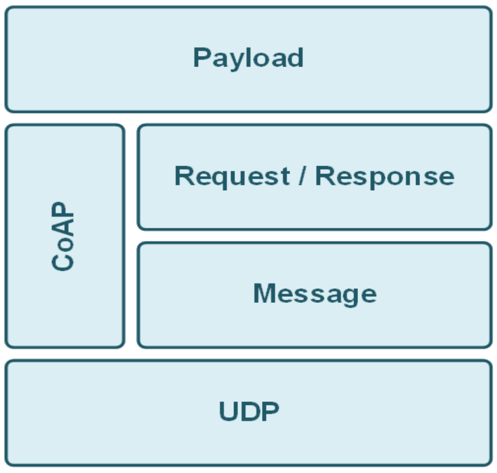


Figure 6. CoAP subsystem architecture.

By default, CoAP uses 5683 and 61616 port for server and client respectively. Exchange messages are 16-bit with 2-bit message type as conformable or CON (00 or 0), nonconformable or NON (01 or 1), acknowledged or ACK (10 or 2) and retransmitted (or reset) or RST (11 or 3). CoAP request and response are composed with a 3-bit class (c) and a 5-bit detail (d) in ***c.dd*** format. 0, 2, 4 or 5 value is used for a class for requests, success responses, client error responses and server error responses respectively. Depending on the class value, a detail holds a request or response code. CoAP code 0.00 implies an empty message whereas 2.01, 2.02, 2.04, 2.05, 4.04, 4.05 indicates created, deleted, updated, reset content, not found and not allowed respectively. The message ID is in 16-bit where a token is in 0 to 8 bits [43]. All modern programming languages like Java, C++, Python and many others have support for CoAP [44]. CoAP has a similarity with HTTP in request and repose. If for example, a client wants to access a *url* with GET method allowing *param1* and *param2* parameters then it could be –

coap://host:port/url?param1=value1&param2=value2

2.1.1.2 MQTT

Message Queue Telemetry Transport or MQTT is a publish/subscribe messaging protocol that operates on the application layer which is extremely simple and lightweight for M2M communications. This is designed for constrained devices concerning issues like low-bandwidth and high-latency. The design principles of MQTT turn the protocol more precisely where the bandwidth and the battery power are at a premium. The typical MQTT architecture is depicted in Figure 7.

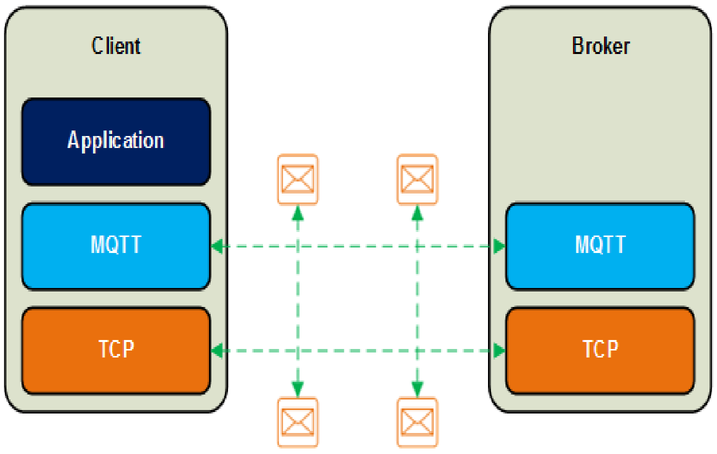


Figure 7. MQTT subsystem architecture [63].

In general, the MQTT protocol consists of a server (broker) and a huge number of clients (up to 10k devices). Each client connects to the broker with its own unique identifier (client ID). A broker manages client's connections and exchanges messages among them. In addition, a database persistence often requires in order to keep messages for future reference. A broker is responsible for distributing messages to all clients reside into a network when those were temporarily disconnected and returned back into online again. This process of distribution is called the retain message.

A client must send a keep-alive message periodically to the broker to keep the connection alive when it is in idle state (not pulling or pushing messages) for a long time, otherwise, the broker terminates the connection after a timeout [63]. Basically, a connection timeout is calculated by the following equation –

$$CT = 1.5 * KAT \tag{1}$$

Where, CT is the connection timeout and KAT is the keep alive time [63]. According to the equation, the connection time is proportional to the keep alive time.

2.1.2 M2M Protocols

Higher-level application-layer protocols are not needed in all communication scenarios. Thus, few low-level standard rules or protocols are required that enable us to exchange of information from one machine to another. These should be secure and reliable for low power devices and prolong the battery life. This section focuses on M2M protocols operating on lower layers.

2.1.2.1 6LoWPAN

IEEE802.15.4 is a standard for low power Wireless Personal Area Network (WPAN) proposed by 802.15 working group. It operates in 16 channels at 250kbps (over 2.4GHz band), in 10 channels at 40kbps (over the 915MHz band) and in one channel at 20kbps (over the 868MHz band) with 16- bit or 64-bit addressing mode. This protocol resolves most key challenges of the WSN related to security, auto networking and low power consumption [70]. Figure 8 shows the Internet Protocol version 6 (IPv6) over Low power Wireless Personal Area Networks (6LoWPAN) architecture.

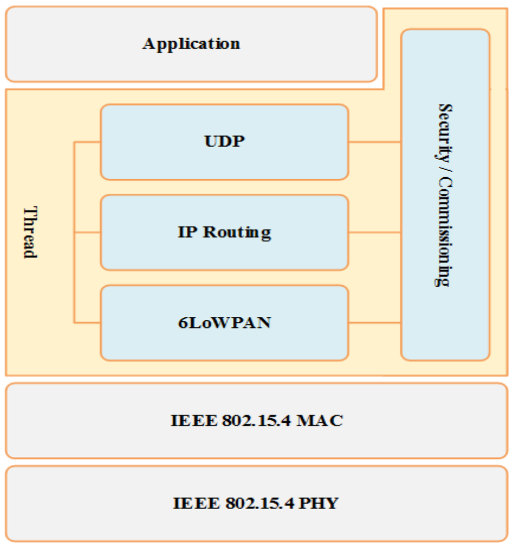


Figure 8. 6LoWPAN protocol architecture [77].

The 6LoWPAN developed by the Internet Engineering Task Force (IETF) is in favor of WSN challenges. It enables the efficient use of IPv6 at low-power device through an adaptation layer. [70]

2.1.2.2 BLE

Bluetooth Low Energy (BLE) is a protocol used for low powered devices that designed on top of physical layer and data link layer. The classic Bluetooth then these devices generally face fast battery drain with the fact of frequent loss of connectivity among those devices. As a result, these devices require frequent pairing and repairing to share data. BLE is an emerging low power wireless technology that is developed for short-range network. It provides a single-hop communication which enables its usability to consumer electronics and security applications. A typical BLE protocol is shown in Figure 9.

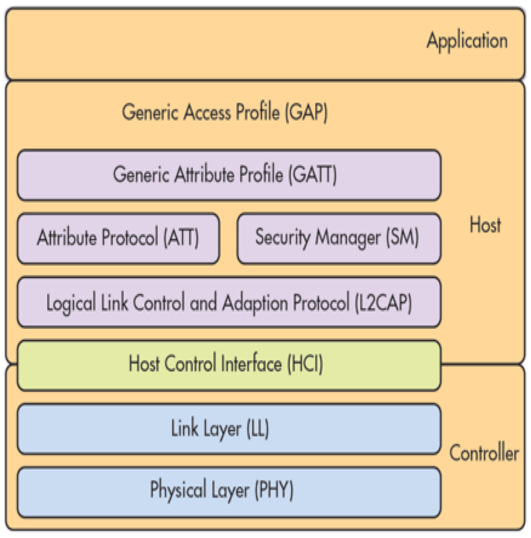

Application
Generic Access Profile (GAP)
Generic Attribute Profile (GATT)
Attribute Protocol (ATT)
Security Manager (SM)
Host
Logical Link Control and Adaption Protocol (L2CAP)
Host Control Interface (HCI)
Link Layer (LL)
Controller
Physical Layer (PHY)

Figure 9. Typical BLE protocol stack [71].

A typical BLE stack consists of a controller and a host. Controller part usually forms with the physical and link layer while the host part forms with a Logical Link Control and Adaptation Protocol (L2CAP), an Attribute control (ATT), a Generic Attribute Profile (GATT), a Security Manager (SM) and Generic Access Profile (GAP). Host Controller Interface (HCI) is standardized with the host and the controller part. [70]

2.2 Cloud Computing

Cloud Computing (CC) is a centralized model that enables global access to configurable resources and services through applications, networks and storage devices. These resources are dynamically configured and accessible by any internet connected node, anywhere in the world. It allows the optimum resource utilization to adjust to an automatic variable load [3]. Figure 10 shows how a user get access to the cloud.

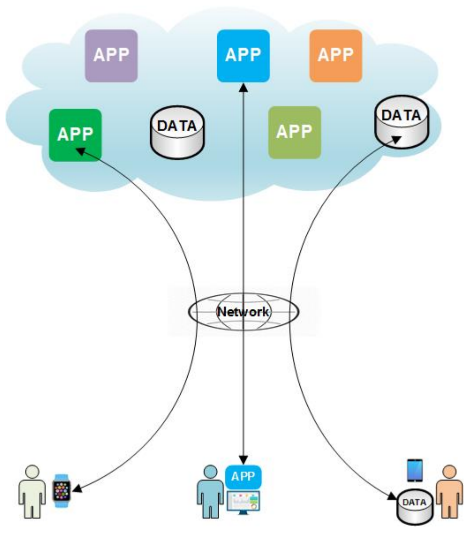


Figure 10. Data processing at the cloud [80].

Service providers can easily deploy their services without having the base setup which is provided by CC platform [5] – [7]. On demand scalability, storage capacity; and security are very regular benefits of cloud computing. In addition to these benefits, control of cloud services in the form of Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) turns into Everything as a Service (EaaS or XaaS) which fulfil the requirements of any organization. However, the centralized cloud faces a few key challenges even though it has a high computational performance at the server. High latency is one of the biggest challenges due to the very high distance between the client device and the server station. This is controversial and becomes very problematic in exchanging data from a server when a client has low-speed internet connection especially for online gaming [8]. Privacy and the public services provided by CC are vulnerable in the fact of security [9] – [13]. Securing the confidential data of end-users and organizations is a major concern that propagates with a public network to the data center [14] – [16]. These are the crucial driving factors to move services towards Edge Computing which pushes cloud services to the edges of a network [17] – [19].

2.2.1 Web Protocols

Typically, a web page is composed of many contents such as text, images, videos, scripts, and more. These contents may come from either a single server or several many distributed servers.

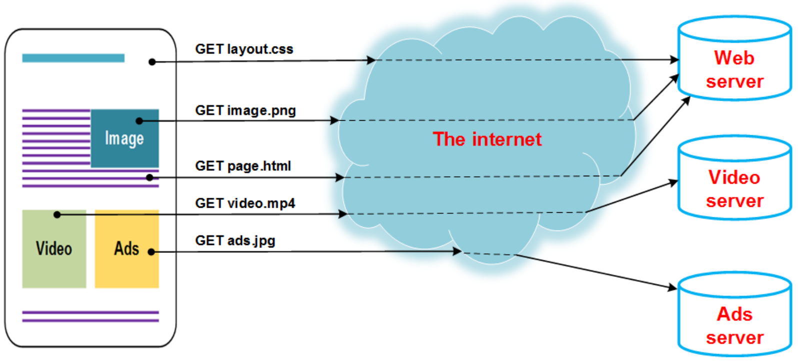


Figure 11. Client-Server communication with HTTP [40].

Clients and servers exchange messages (as opposed to a stream of data) by a web application. The client message is called a request whereas the server message as an answer is called response. [40]

2.2.1.1 HTTP

Hypertext Transfer Protocol is a standard web protocol to specify how content will be exchanged between a server and a client. It allows a web client to fetch web resources, such as HTML documents. Theoretically, it uses the Transmission Control Protocol (TCP) over Internet Protocol or IP. The contents of a server can be sent to the client as HTML, XML, JSON format. Part of a document can be updated by the client request. Figure 12 presents a typical HTTP protocol stack. [40]

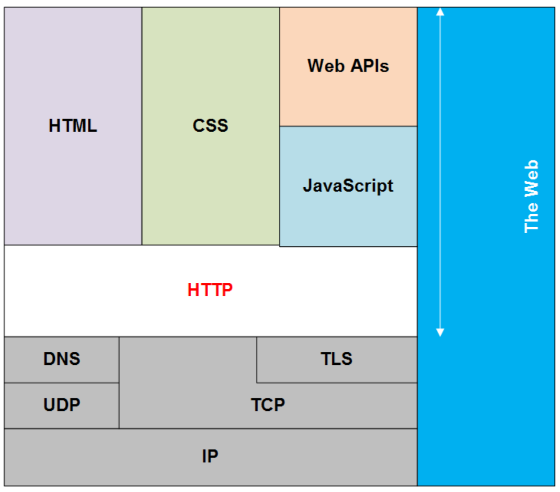


Figure 12. HTTP protocol stack [40].

A client can access a web resource with a standard method defined for HTTP such as GET, POST, PUT, DELETE; and many others. Typically, port 80 is allocated for HTTP by default.

HTTP GET request *url* with parameters *param1* and *param2* can be accessed through a browser as –

http://host:port/url?param1=value1&param2=value2

The server sends a status attached to the HTTP header along with the message body in response to a client. These statuses have a standard format for example – 1xx (informational), 2xx (success), 3xx (redirection), 4xx (client error) and 5xx (server error). Appendix 2 shows the HTTP status codes.

The HTTP is developed by Tim Berners-Lee over 1989-1991. It has no version since it was invented and later introduced with HTTP/0.9. The first version of HTTP (0.9) has no support for other types of documents (as an attachment) and has no headers (status and codes). After that, HTTP/1.0 has been introduced in 1991-1995 that enables us to add documents along the text-based messages with Content-Type. It also informs, the browser about the headers (status and codes) in response to the client request. From 1995, HTTP/1.1 has been started developing to support caching, chunk response with numerous languages support along with encoding mechanism. In 2016, HTTP/2 has been released with secured cookie. [78]

2.2.1.2 HTTPS

The text-based HTTP protocol has a bunch of drawbacks in privacy and security, especially for the online transaction in banking and e-commerce trading. The Man-In-The-Middle (MITM) attack allows an intruder to steal data passing through the communication channel [72]. Figure 13 indicating the problem space in HTTP and alternative solution to the problem.

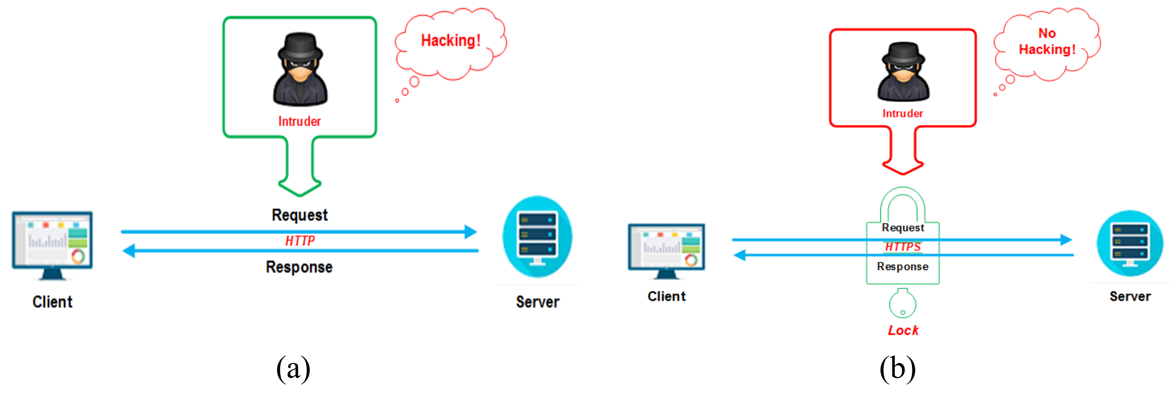


Figure 13. Intruder in (a) HTTP (b) HTTPS.

An HTTP connection basically secured with a certificate provided by Transport Layer Security (TLS) formerly known as Secure Sockets Layer (SSL). By default, port 443 is reserved for secure HTTP or HTTPS. The whole communication channel is vulnerable if this certificate is not trustworthy and the original certificate will be replaced at the server-end by the attacker. [72]

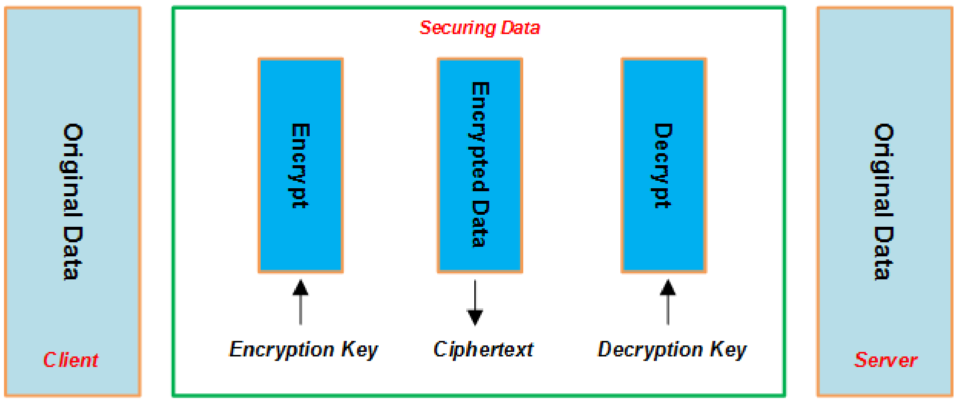


Figure 14. Encryption and decryption process at communication channel over HTTPS [73].

The attack will be successful if a client neglects the browser's warning without a double check of the certificate. This will be minimized with the encryption and decryption algorithm. Both of a client and a server first encrypt the original data by an encryption key before sending those and decrypt the encrypted data after they received before processing. In encryption process, a human unreadable cipher text is generated with the original data and an encryption key. On the other hand, a decryption key is used with the encrypted data (the ciphertext) to get back the original data at the decryption process. [73]

2.2.1.3 *Web Sockets*

Typical HTTP communication cycle includes a client request and a server response. In this communication, a client will not get any update information from the server without making a new request to the server. Websocket is a protocol similar to HTTP that allows persistent connectivity under the hood of the web browser. Figure 15 represents how websocket enables continuous connectivity in between a server and clients.

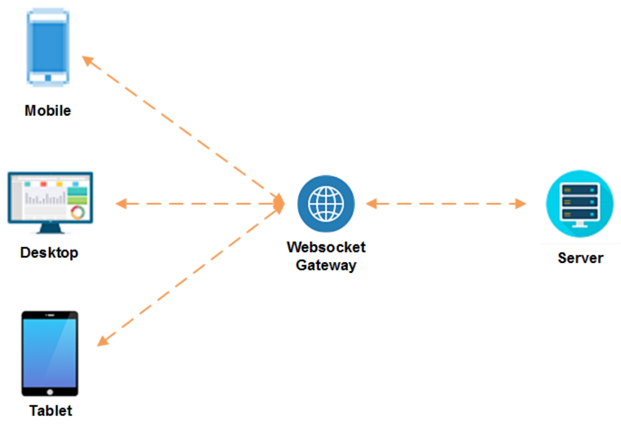


Figure 15. Websocket in communication.

Websocket consists of an opening handshake and a basic message framing and works over TCP. It provides a mechanism for browser-based applications that require two-way communication with servers to exchange data rather opening multiple HTTP connections [74]. It works like a full-duplex phone call where the stream of information is constantly flowing to a web browser. It is the foundation for the most real-time web application. If there is a change in any device that connected through websocket then other devices connected to the gateway will be updated automatically. For example, if Desktop has a new state then server, Mobile and Tablet will be updated automatically. Websockets are mostly used in chat and messaging applications, many real-time systems like stock exchange applications, and so on.

2.2.2 APIs

Application Programming Interface (API) enables us to store and retrieve data to or from database irrespective of the access application. It allows us to use a feature that is written by someone else around the world. For instance, a weather API can help us to know about the environmental things like the temperature, humidity, etc. API endpoint or url is required to get access to a piece of information from a server. The information is gathered as key-value paired which is parsed with JSON or XML or any other suitable format. Figure 16 shows a typical communication based on API.

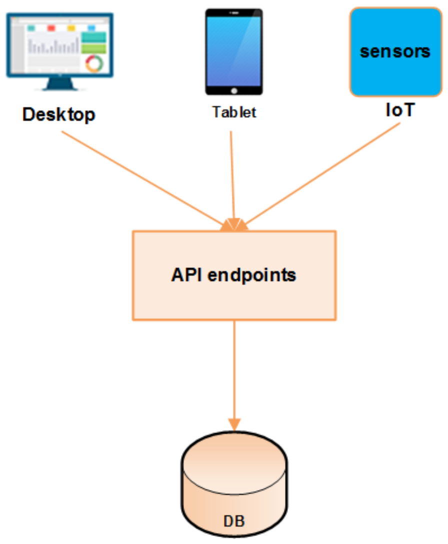


Figure 16. Typical communication with an API [81].

Data in a remote datacentre can be accessed from many devices such as a web browser, a mobile application; etc. All these access devices may not have the same application to the access data. In such cases, APIs can help us to represent data for different view application.

2.3 Edge, Fog and Mist Computing

Data is continuously being generated at the edges of networks particularly in IoT. On the other hand, the cloud is generally too far from end-to-end devices meanwhile bandwidth is limited in the communication system. Moreover, the services will be affected when a lot of requests is being processed at the remote cloud server beyond the cyber security threats. Considering these factors, cloud computing is not always efficient model for data processing especially when the data is generating at the edge of the network [19]. Figure 17 shows the possible problem of space for cloud computing.

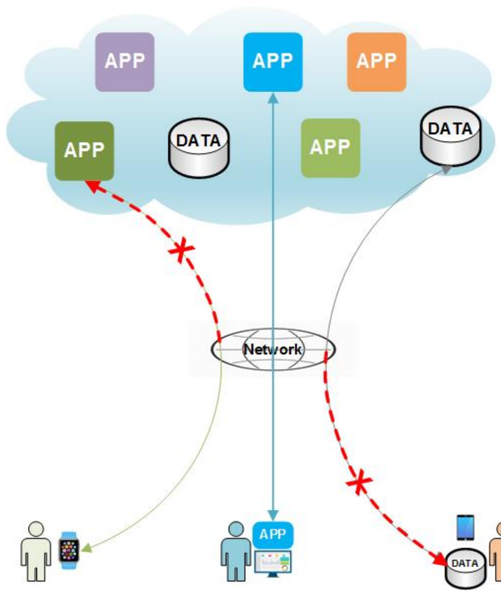


Figure 17. Service disruption between the cloud and the end-to-end device.

Therefore, it would be better and more efficient to move remote cloud services those requiring low latency, high reliability, context information, reduction of data towards the edges of networks. The next subsections will present the most common alternative models as a solution to the cloud computing in favor of IoT.

2.3.1 Edge Computing

Edge Computing (EC) introduces a new layer of services between the cloud and the end-to-end devices at the edge of network that producing data directly. It optimizes the performance of services resides in the cloud by improving the speed of data processing at the edge of the network near to end-device.

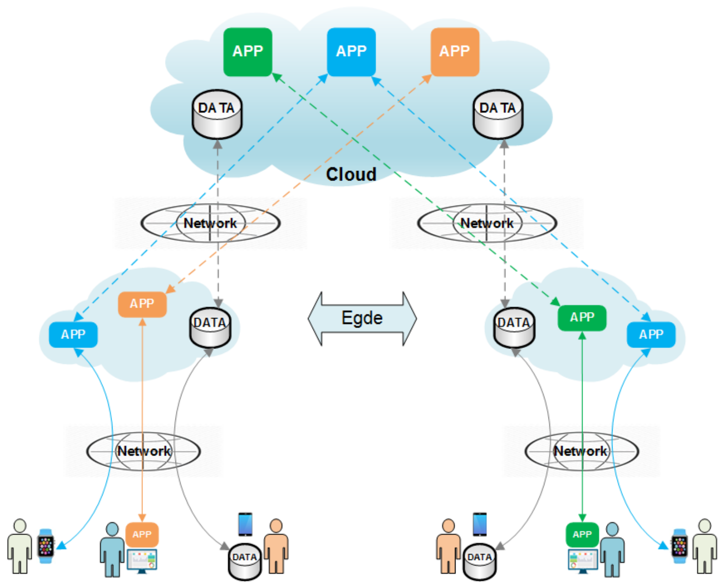


Figure 18. Data processing at the edge of network.

It greatly improves the latency between IoT nodes and servers by reducing the physical distance. It helps us to resolve security and privacy by limiting the scope of propagation of private data along the public networks [20].

2.3.2 Fog Computing

Edge is not enough to speed up the data processing at the end-devices especially for industry. Fog computing is a term closely related to Edge computing and has a stronger focus on enterprise services especially in the form of SaaS or PaaS [21].

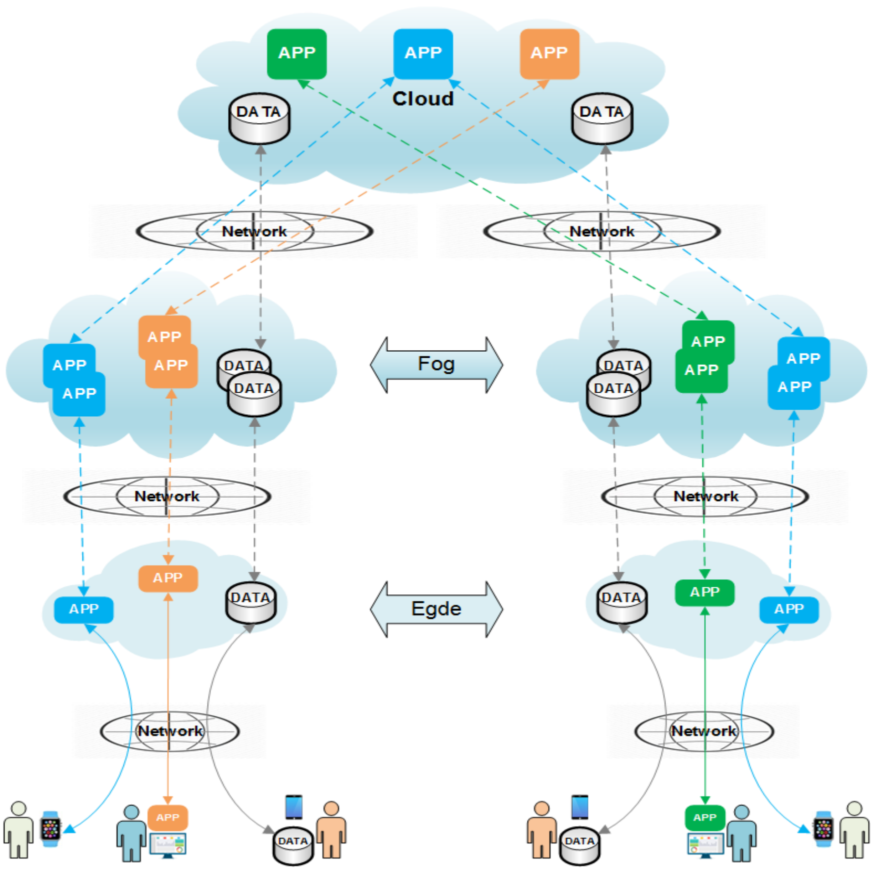


Figure 19. Data processing at the fog layer.

Fog computing is a term proposed by Cisco [76] which enhance the computational capacity of the edge of network. Basically, it improves the performance of edge and gives resiliency to the unstable networks [22]. In practice, fog is the virtualized solution to cloud while edge is the infrastructural solution to cloud computing. So, in essence, fog computing is a standard for IoT while edge is the backbone concept [76]. Decentralized fog allows a reparative structure in the edge concept so that enterprises can perform better compared to cloud.

2.3.3 Mist Computing

Ultra-low powered devices require more sophisticated services in the local IoT network. These low powered devices require virtualized service provisioning that taking place at IoT infrastructures often called as Mist Computing (MC).

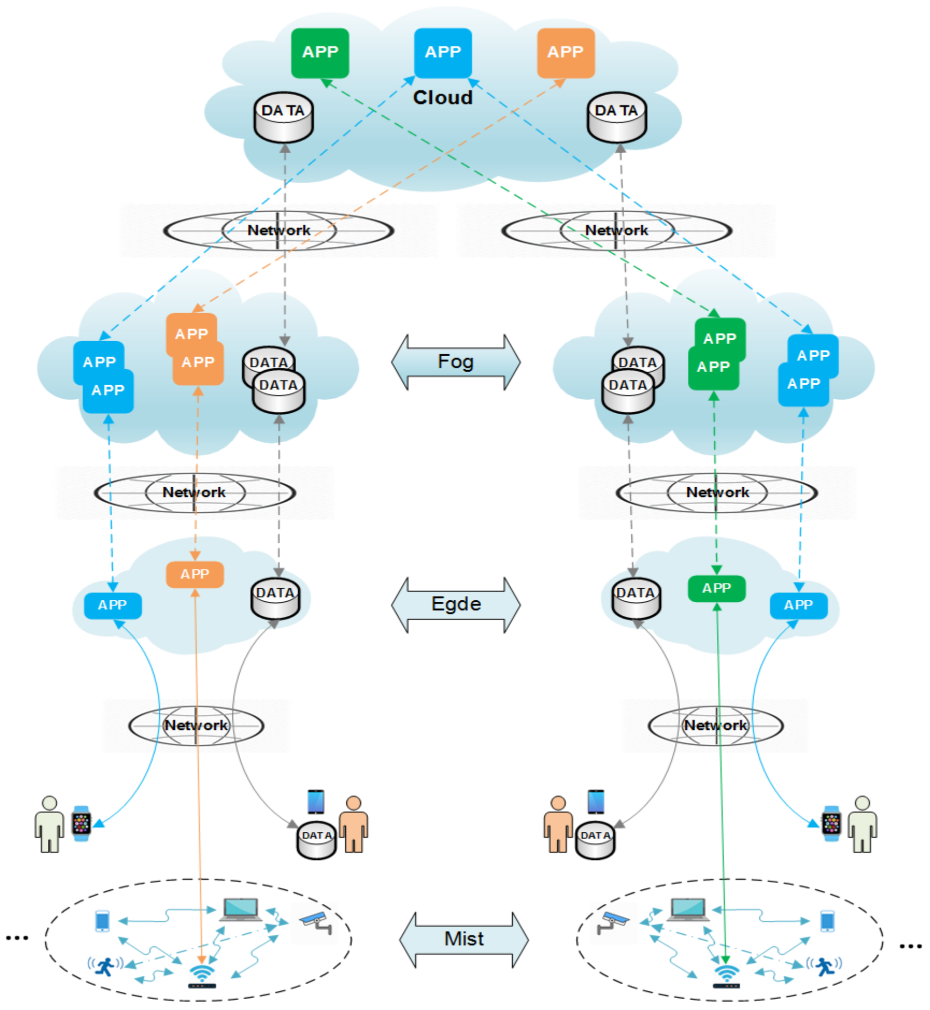


Figure 20. Data processing at the mist layer.

It basically introduces another new layer in between edge computing and end-to-end devices at the local position. It improves the performance of edge computing having the services that only required to end-to-end devices. [23]

2.4 Virtualization

Virtualization is the core technology behind cloud computing and offers the ability to control complex systems [4]. It allows us separating the operating system from the underlying hardware [24]. It creates a virtual version of a resource such as a server, storage device, network or even an operating system. The framework of this technology divides the resource into one or more execution environments [25]. It also includes resource optimization, dynamic load balancing, and application isolation. Typical classification of virtualization shown in Figure 21.

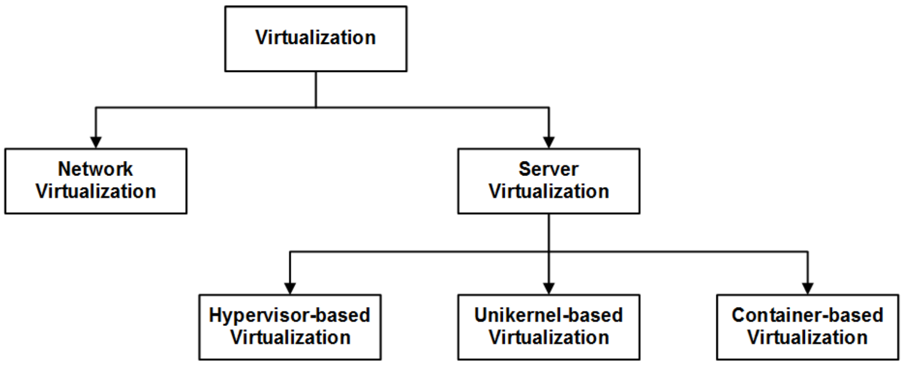

Virtualization
Network Virtualization
Server Virtualization
Hypervisor-based Virtualization
Unikernel-based Virtualization
Container-based Virtualization

Figure 21. General taxonomy of virtualization.

Network virtualization allows to create logical network where server virtualization enables virtualization of physical hardware devices. Following subsections present server and network virtualization.

2.4.1 Network Virtualization

Network Virtualization (NV) is the process of separating the network from the underlying network hardware. It can create logical, virtual networks that are decoupled from the network devices. It ensures better integration to network and support virtual environments increasingly like virtual private network or VPN [26]. Network Functions Virtualization (NFV) virtualizes the typical network elements, such as routing by switches, firewalls by using Network Address Translation (NAT); and all other core network components [59][60].

2.4.2 Server Virtualization

Server virtualization enables the conversion of a physical server into several multiple virtual machines that can be deployed on any arbitrary hardware. Each newly created virtual server acts like a unique physical device. Full virtualization, para virtualization, host OS virtualization, and containerization are common approaches of server virtualization. [26]

2.4.2.1 Hypervisor-based Virtualization

The hypervisor is computer software or hardware that creates a new partition and introduces guest OS on top of the physical hardware and host operating system. Figure 22 represents the architecture of hypervisor-based virtualization on top of physical hardware and the host OS.

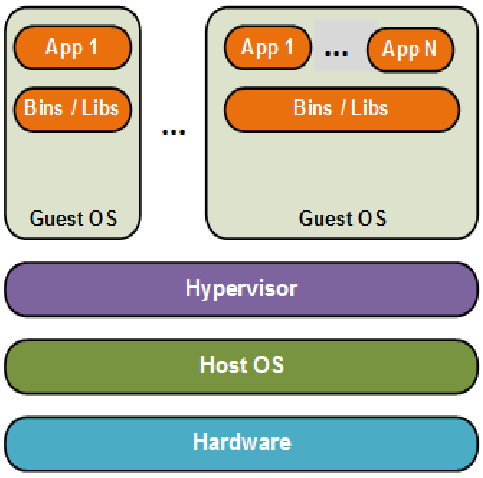


Figure 22. Typical Hypervisor-based virtualization [34].

Hypervisor requires a management software like *Oracle VirtualBox* and *VMware* to create and manage a new virtual system. The hypervisor has various limitations. The biggest drawback of this mechanism is that, it reserves resources like CPU and RAM even if they are not in use. This may cause for a negative impact on host OS with performance degradation. For an instance, A system has 16 GB of RAM and 3 guests OS is serving on it as Windows, Linux; and Mac OS where 8 GB, 4 GB; and 4 GB are allocated respectively. In this case, 8 GB that allocated for Windows OS will not sharable with neither Linux and nor Mac OS. This is also true for Linux and Mac system. It requires an image for guest OS like iso file used for a real OS which is another drawback of hypervisor-based virtualization. Start-up booting time and integration are other two important parameters that cause and lead us to use containerization solution to virtualization technique.

2.4.2.2 Unikernel-based Virtualization

The traditional hypervisor-based system populates a full guest OS inside a host operating system and allocates physical and logical resources for that guest OS. Considering these pitfalls with the issue when the system does not require modification after deployment, *unikernel* can be an effective solution [34]. Unikernel is single kernel-based virtualization and is a single-purpose appliance which is used at compile time. Figure 23 gives an architectural overview of a *unikernel* system.

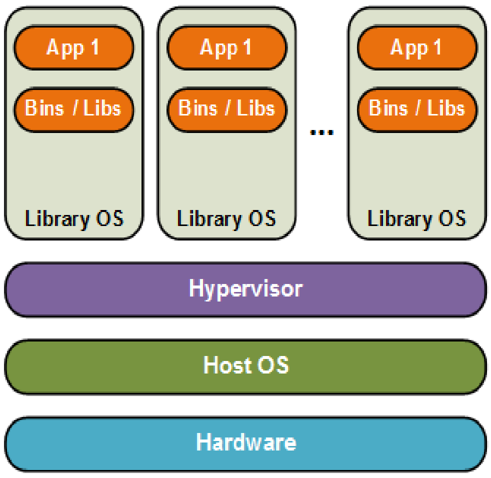


Figure 23. Unikernel-based virtualization [34].

Security and *unikernels* are tightly coupled. Moreover, a *unikernel* provides high security by suppressing unnecessary components from the applications and hence reducing the scope of attacking surface. The scope of attacking surface of a *unikernel* system is strictly confined to the embedded application within the system. In a *unikernel* system, everything is directly compiled into the application layer separately. As a result, a hacker may able to break a single *unikernel* system but not spontaneously to all.

2.4.2.3 Container-based Virtualization

Containerization offers the most effective lightweight virtualization which allows sharing physical resources like CPU and RAM from the underlying hardware and software [34]. It wraps all requirements into a software package like an executable file to execute the instructions given the system using container engine. Figure 24 shows a building block of a container in containerization technology.

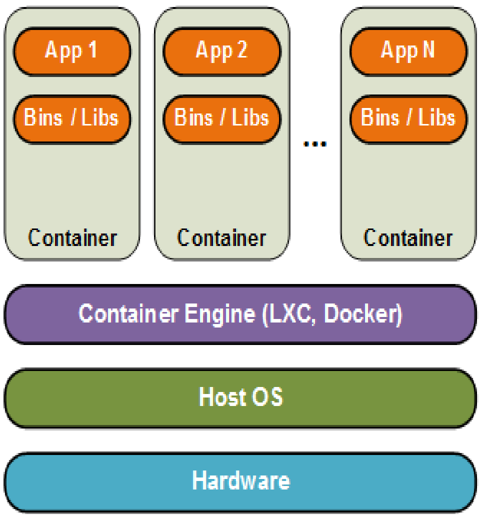


Figure 24. Container-based virtualization [34].

Container technologies provide various benefits for example fast construction, instantiation, and initialization of virtualized instances compared to hypervisors. In addition, systems on IoT edge can be benefited from the small virtual images of containers. Following subsections describes further details of lightweight technology.

Docker and container orchestration are major concern of this thesis which very related to virtualization. Those will be described into separate section rather describing here.

2.4.3 Docker

Hypervisor-based system solutions became problematic in resource allocation, start-up time and adaptability. Container-based Docker ecosystem offers a better solution to develop and deploy applications by using containers just requiring Docker on the host machine. It is an open source platform to build, ship, and run any app, anywhere [35]. Docker containers allow a developer to package up an application with all requirements, such as libraries and other dependencies to build software correctly that workable in all other machines and ship that as a single package. The application will run on any other machine having Docker regardless of the customized settings. Anyone can contribute to Docker and extend it to meet their own requirements if they need additional features.

2.4.3.1 Docker Image and Docker Container

A Docker image is an inert and immutable file created from *Dockerfile* that's essentially a snapshot or blueprint or template of a Docker container. *Dockerfile* holds all necessary commands or instructions to run a software knowing the dependencies of that software using Docker engine. Every Docker image requires a special image which is called the base image of that new image which is the first line at *Dockerfille*. Docker engine starts execution from the base image and creates an intermediate image and saves the final one by dropping immediate images. A Docker image is created with the *build* command and produces a Docker container when started with *run* command. A Docker image can be stored in a Docker registry or repository system like *dockerhub* for future use and reference. [37]

A container is a runnable instance of an image. Image and container are very similar to class and object in the Object-Oriented Programming (*OOP*) paradigm. To use a programming metaphor, if an image is a class, then a container is an instance of that class which is a runtime object. An image is a blueprint of a container whereas a container is running instance of that image. Containers are lightweight and portable encapsulations of an environment where they could run and deploy applications.

Mouat et al. [37] stated - An image is a template that can be turned into a container by running the image. Docker engine takes the image, adds a read-write filesystem on top of the system and initializes various settings on the environment including network details, container name, ID and other resources to turn an image into a container. In a Docker engine, a container can be either in running or stopped or even not instantiated or exited state. It can also restart and will retain its necessary settings if there is a change in the filesystem.

2.4.4 Container Orchestration

Services must be answerable to the fact of - fault-tolerant, on-demand scalability, resource optimality, can update or rollback without any downtime, able to discover other services automatically and can communicate with them. However, container orchestration provides us the tool known as container orchestrator to deploy and manage any application on the system [36]. Figure 25 shows the typical container orchestration proposed by Amazon Web Services (AWS).

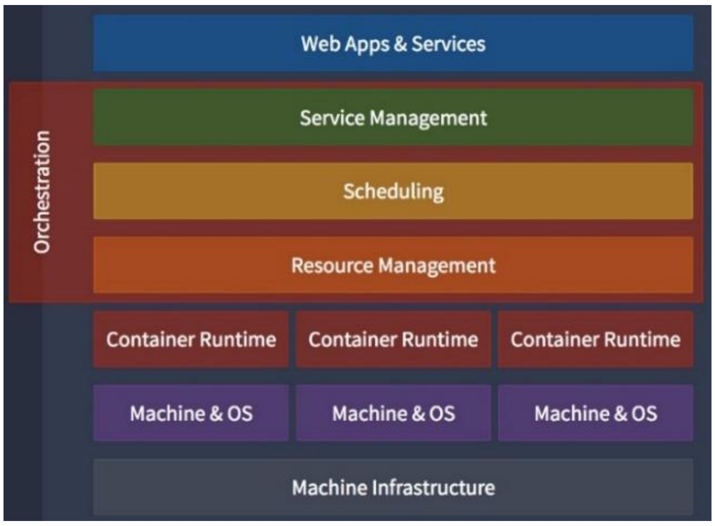


Figure 25. Container orchestration [61].

There are many container orchestrators available such as Swarm, Kubernetes, Marathon; and ECS. This thesis focuses on Swarm and Kubernetes.

2.4.4.1 Docker Swarm

Docker Swarm is a container management tool for orchestrating Docker containers. Docker Swarm has a leading node with some other nodes. The leading node is Swarm manager where other nodes referred to as workers. Swarm managers can execute commands or authorize other machines to join the swarm as workers. Figure 26 represents the general architecture of a Docker Swarm system.

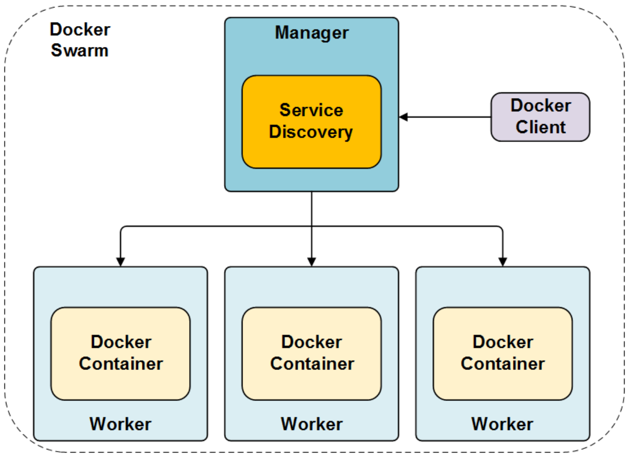


Figure 26. Docker Swarm.

A manager has the ability to talk with all the workers and a worker can talk with the manager but not with other worker nodes. A machine can be turned into a swarm manager simply by enabling the swarm mode on that host machine. From then on, Docker commands can be run and executed on the swarm that is in all machines, rather than just on the operating manager machine. [38]

2.4.4.2 Kubernetes

Kubernetes is an underlying technology used for container orchestration. In harmony, it reduces the operational burden and allows many containers to work together. Kubelets helps Kubernetes to interacts with Docker engine. Kubelets works under the hood of Kubernetes and can consolidate with Docker engine to organize the scheduling and execution of containers. Figure 27 portrays an architectural overview of Kubernetes system.

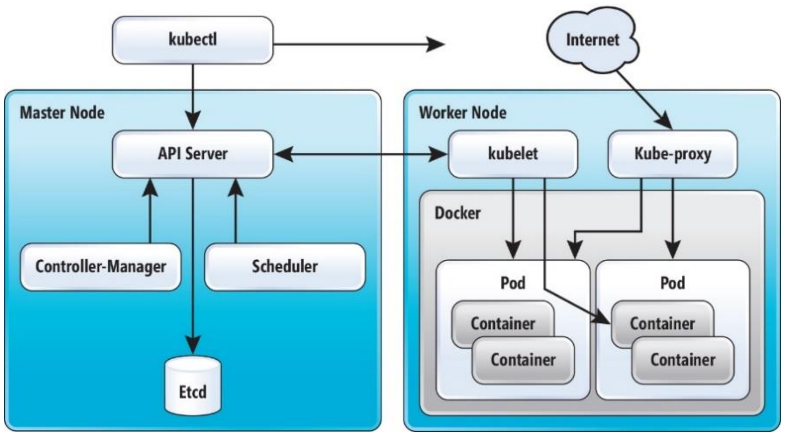


Figure 27. Kubernetes system architecture [62].

Kubernetes handles service-discovery, network policies, and load balancing as well. Both Swarm and Kubernetes are great tools for container management where the Graphical User Interface (GUI) adds extra benefit to Kubernetes with Command Line Interface (CLI). [36]

2.5 Application Design

Software Development Life Cycle (SDLC) is usually composed of numerous steps from planning to maintenance and follows a strategic rule during the entire development life cycle. Most SDLC includes – studies over requirements, design, implementation, testing, deployment, maintenance by scaling and many other steps. Personnel (either a developer or a tester) in the software industry is often facing various problem from testing to maintenance [33]. Figure 28 shows the possible stage of a problem that faces in SDLC.

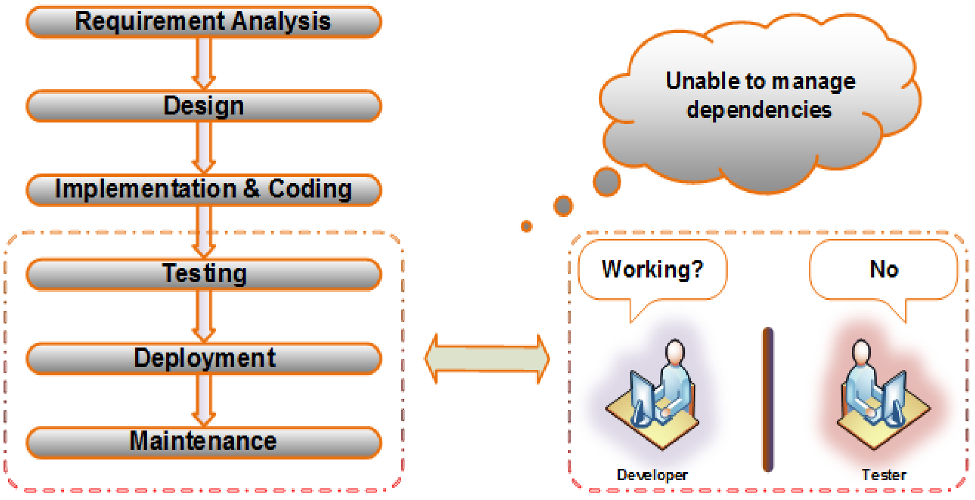


Figure 28. Problem space in SDLC.

A developer developed software that may workable just in his/her own machine. Then s/he sent that for next phase i.e testing. The software just is broken or crashed when a tester (or DevOps or other developers) run that on their own machine. There could have several reasons that cause failure. This failure is reasonable for either

- A developer may not concern with the task(s) that given to his college(s). As a result, (s)he may not aware of the related dependencies so that the application is workable to others.
- The dependencies may be defined in terms of
  - Hardware (e.g. Memory)
  - Software (e.g. OS and version)

Software personnel needs a dynamic novel automated solution that can resolve the above problems. Virtualization has been introduced in software industry to overcome these problems.

2.5.1 Design Patterns

Monolith and microservice [66] are two popular design patterns have been introduced by software developers. In this section, the architectural design and practical scope in real the world will be discussed.

2.5.1.1 Monolith Application

When the entire functionality of the application is packaged together as a single unit or application then the application is called monolith application. It hooks up all the services of into a single unit as a large-scale application. For example, an online shopping website will typically consist of product, cart, payment, and other features. All possible features in a large-scale monolith application are implemented and packaged together as a single application. [28]

The monolith application contains some advantages including modularity (all codes are in a single place), IDE-friendly, easy sharing, simplified testing; as well as easy deployment. Besides that, it has the following disadvantages consisting of limited agility, discontinuous delivery, sticky to technology stack; and technical debt. [27]

2.5.1.2 Microservice Application

Microservices is a software architectural design pattern that decomposes and decouples all the possible features into several small single pieces of applications with a limited scope of functionality. Balalaie et al. [29] shows a migration process from a large-scale monolith application to multiple smaller services. In their architecture, each smaller service deployed on its own archive with a possible decomposition of dependencies and then builds a single application. For example, each of online shopping would have services for a product, cart, payment, and other features.

Kumar et al. [30] shows the use and scope of microservices in his blog post. Domain-driven design (decentralized data management), single responsibility principle, independently deployable, upgradable, scalable and replaceable, potentially heterogeneous; and light-weightiness are the key characteristics and benefits of microservices. NoOps is called common additional features that are introduced in microservices having service discovery, service replication, service monitoring; and service resiliency.

2.5.2 Mobile Agent

There is a need for software agent in any Artificial Intelligence (AI) system which is capable of training from the fact of the environment by observing it [46]. It enables system components to get and set automatic and intelligent actions on behalf of the system owners. During the execution of a task, a mobile software agent or mobile agent (in short) can relocate itself by moving between the two system devices.

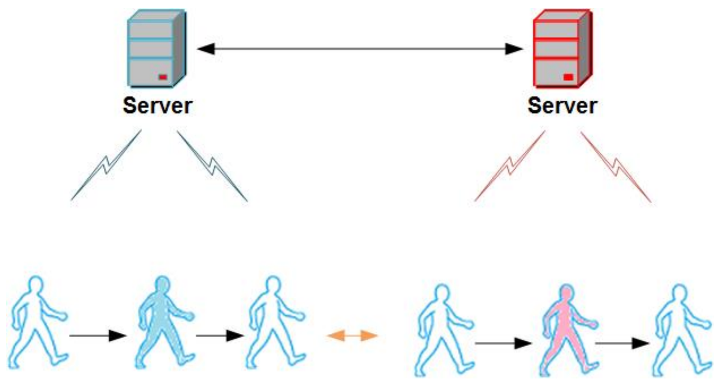


Figure 29. Software mobile agent follows user.

IoT devices get and set smart behavior (such as composition, aggregation; and relocation of services) by implementing mobile functionality into the system domain. An autonomous mobile agent can be beneficial to the IoT system since it interacts and responds on behalf of the system [47][48].

2.5.3 Actor Model

A modern Operating System (OS) creates a process as a request to run a program. In particular, a process is basically a program that is currently executing with one or more local variable and a counterpoint. Every process should have at least one thread. A process can also have multiple threads. Multiple threading or multi-threading is a popular approach used in concurrent computing. Resource overhead is the most common drawback that causes for a thread. It includes memory overhead since it uses local variables in a program. On the other hand, CPU overhead is another inherent problem introduced in the scheduling of threads by the operating system. Actor model helps us to resolve this, especially in concurrent computing. [67]

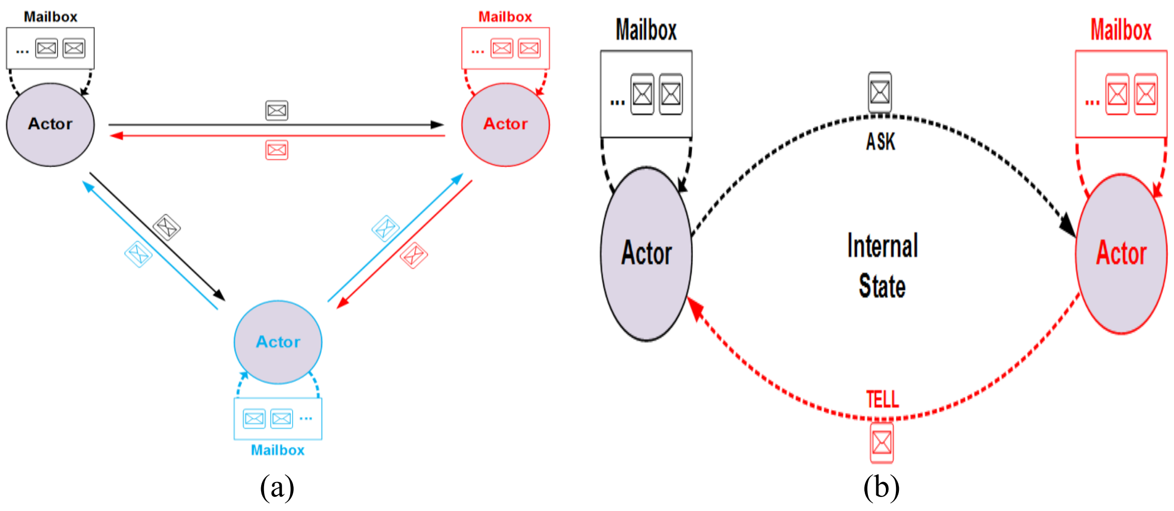


Figure 30. Actor model (a) message passing (b) internal state.

Actor models are defined as the compositions of several actors that do not have shared state. These actors communicate by means of exchanging messages among them. Distributed microservices can take the advantages of the actor model in IoT Edge computing [31]. Haubenwaller and Vandikas have utilized this model in their service composition. They stated - data may process at the local system by IoT devices rather sending to a central location for processing [32]. The tasks can be split into smaller tasks and deployed IoT devices efficiently. In practice, this model is conceptually very close to microservice architecture since the actors are message-driven having single responsibility to each of them. Therefore, it can be applied to microservices systems in between microservices.

3 DESIGN AND IMPLEMENTATION

This thesis is focused on the implementation using distributed services in favor of the future gadget free decentralized local IoT edge network. A demo environment has been installed to demonstrate the operating of this thesis before the actual work. Following subsections describe the action plans taken for IoT distributed services.

3.1 Application Scenario

In order to define the requirements of the system, a use case scenario has defined. Following section describes the use case scenario with the service model that launches for the completion of the use case.

3.1.1 Use Case

This thesis takes the use case scenario defined in Harjula et al. [23] as the basis for the implementation. In the use case scenario, Alice organizes a team with Bob, Carl, and David. Then she creates a meeting event and sends a request to her team members. Each participant needs to be verified and authenticated with their identity by the system or the participants before entering the meeting room since highly confidential topics will be discussed in the meeting. Now she needs a secured system that deployed locally which will lead a participant from the entrance to the meeting room. In this secured system, it can have 3 major services for a successful meeting as authentication, guidance; and meeting contents. Gadget-free devices such as smart ring could be used for identification [45].

3.1.2 Example Scenario

At first, Alice activates the presence detection by a motion sensor which will detect movement at the corridor. User authentication will be activated when movement detected by the movement sensor. Multimodal user authentication, used in the implementation, consists of three separate and independent authentication functions, including WiFi Channel State Information (CSI), Bluetooth Low Energy (BLE); and Video Surveillance authentication. Every 3 authentications can be applied to the participants.

WiFi CSI is the combined effect of fading, scaring and power decay along with distance and describes the signal strength and its propagation in between the transmitter and the receiver. It could be used in the authentication. A user can be identified with the help of channel estimation and by matching the average walking speed of a user with the user’s profile. Bob visited this place before and his profile has saved into this CSI log. So, the system can identify Bob delegation to the meeting.

Many tracking and security system requires continuous communication with low powered devices. Bluetooth Low Energy (BLE) is one of the most emerging technologies which is used in wireless Personal Area Network (PAN). Bluetooth Special Interest Group (Bluetooth SIG) introduces novel use of BLE in positioning and tracking with proximity sensing and BLE based electronic key in security. The second authentication of the system is BLE ID-based authentication. Carl never visited the location before. A BLE device is provided to authenticate him by the system itself by the host Alice. WiFi CSI takes time into the authentication process. Hence, the system starts WiFi CSI and BLE simultaneously when movement is detected on the corridor.

Video Surveillance Camera or Closed-Circuit Television (CCTV) is mostly used in many security systems for identification of an object. The popularity of this manual recognition is being increasing day by day from the last few decays. David also never visited the location before and has no BLE. So, he could be authenticated by neither WiFi CSI nor BLE. Therefore, he could be identified merely by Alice with a surveillance camera placed at the corridor.

Once a participant is recognized by the system then the system then guides a participant to the meeting room. A mobile agent could introduce into the system to guide the participant [46] – [48].

When a person is detected in front of the meeting room the system again identifies the first participant. The system must hide the personal stuff and download and present the meeting content by the storage service as soon as the first participant arrives at the meeting room. The real picture for the aforementioned scenario is reflected in Figure 31.

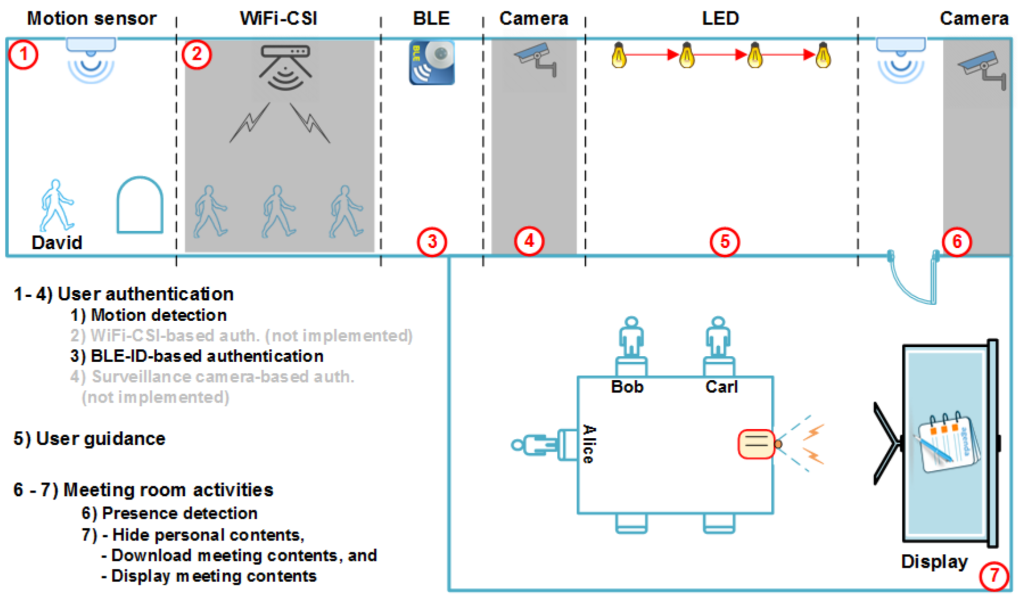

Motion sensor
WiFi-CSI
BLE
Camera
LED
Camera
1
2
3
4
5
6
7
David
Bob
Carl
Alice
Display
1 - 4) User authentication
1) Motion detection
2) WiFi-CSI-based auth. (not implemented)
3) BLE-ID-based authentication
4) Surveillance camera-based auth. (not implemented)
5) User guidance
6 - 7) Meeting room activities
6) Presence detection
7) - Hide personal contents,
- Download meeting contents, and
- Display meeting contents

Figure 31. Real picture of the PoC scenario [23].

3.2 Requirements

In the above scenario, the IoT network is formed with many constrained devices along with various sensors and actuators. Since the above scenario is assumed for the local network, thus the interoperability and the resource consumption got the maximum priority over other challenges for the IoT network. The constrained devices used at the local layer brings the limitation to the storage, data processing; and computations. Therefore, the main challenge is to provide an efficient way to utilize local resources without compromising the performance of the services. These services must be efficient in resource consumption, authentication, quick deployable and initiable. Moreover, scalability and security are also two fundamental requirements for local IoT network among others, but these are not focus for this thesis.

In this context, container-based lightweight virtualization of services may provide the maximum efficiency at the resource consumption. This architecture should be deployable and re-deployable based on the requirement. Docker containers could be useful for the demo scenario. Either Docker Swarm or Kubernetes could be used as container management tools but here Docker Swarm will be used for the rest of the implementation.

3.3 Design

In this thesis, the following services have been defined to implement the PoC for the nanoEdge concept, defined in [23]: 1) presence detection service placed at the entrance point to detect and authenticate arriving users, 2) a controller service to keep a status log when the system is processing a request to lead a participant from entrance to the meeting room, 3) BLE scanning service installed into entrance and in front of meeting room to authenticate a user. Two other authentication mechanisms such as WiFi-CSI, Surveillance Camera-based authentication has been left out of this PoC, 4) API service as part of authentication where BLE IDs belong to specific participants, 5) LED guidance which enlightens the way to meeting from entering point and 6) meeting room service which includes presence detection service to detect a person in front of the meeting room using a PIR motion sensor, and also hide personal content then presents the meeting contents after downloading those from the storage for the first attendant.

This thesis has defined the service model for this demo project. The service model of this architecture demonstrates how entire system will work for the scenario. Then it has also defined the required services to satisfy the project requirements.

3.3.1 Service Composition, Modification and Termination

At first, Alice follows few steps to define and start the required services [23]. These steps include – 1) create the meeting event with an online (e.g. cloud) service management tool connected to the nanoEdge API gateway, 2) choose a meeting service template, 3) selects an available meeting room from a calendar view, 4) start defining the service based on those.

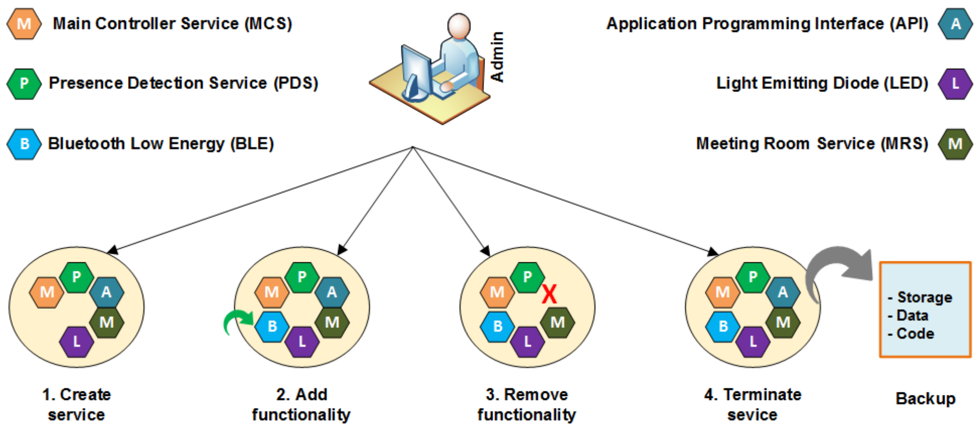

M Main Controller Service (MCS)
P Presence Detection Service (PDS)
B Bluetooth Low Energy (BLE)
Admin
Application Programming Interface (API) A
Light Emitting Diode (LED) L
Meeting Room Service (MRS) M
1. Create service
2. Add functionality
3. Remove functionality
4. Terminate sevice
- Storage
- Data
- Code
Backup

Figure 32. Service creation, modification and termination [23].

In this model, a service administrator or admin can create a service, add and remove functionality, and terminate services. Five services for a system has been created (deployed) initially as for the requirement of the system by the system admin. Next, the system admin realizes to add BLE service as a functionality to the authentication process. Then he realizes to put down the API service from this architecture. Finally, the system can have an automatic backup process for storage, data, code and so on before termination of service.

3.3.2 Service Design

The conceptual actor model has been applied to manage concurrent request from client devices. According to this model, an actor can define some general rules so that the components of a system could asynchronously behave and interact with each other. Generally, it processes a request with a message as a parameter from a mailbox having a queue of multiple messages. In this thesis, it has tried to design the example scenario as maximum as possible decoupling of services from each other which is presented in Figure 33 in favor of microservices.

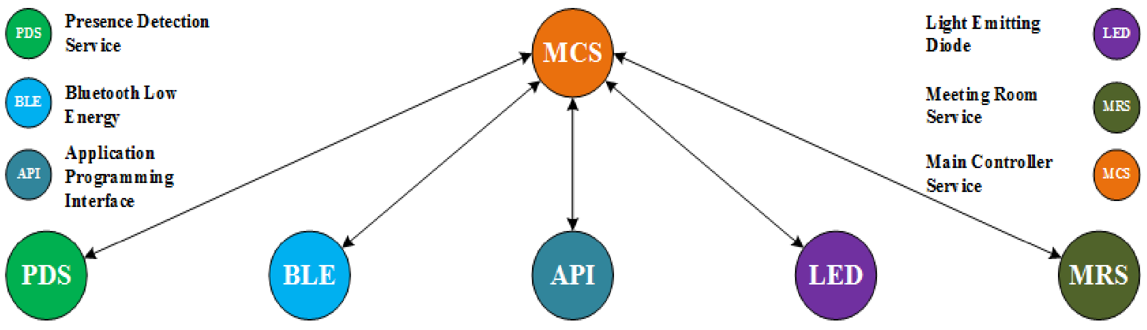


Figure 33. Design of the PoC scenario.

In the implementation, it processes the request with a message at every service to activate the next level of service to process. When a participant enters the corridor, presence detection service (PDS) will be activated automatically and sent a request to the main controller service (MCS) to activate BLE scanning. BLE sends encoded list of BLE device in response to the request of main controller service. API service further let informs the main controller service whether a BLE devices is authenticated or not in response to it. Then LED service will enlighten the way to meeting room like a mobile agent. When a participant is in front of the meeting room then meeting room service (MRS) will be activated and authenticates the participant again and do numerous tasks as described above in this section.

3.4 Implementation

This thesis implemented all the required 6 services as stated in the previous section. Every service can talk to each other with CoAP messaging protocol and can exchange information with GET request. Following sections in this chapter will describe implementation details with associated appropriate subsections.

3.4.1 Services

All the required services have their own loyalty in minimal scope. Here, this thesis described all of them one by one into the following subsections.

3.4.1.1 Main Controller Service

A special service is required that cares about the Machine-to-Machine (M2M) communication and routes the incoming request to appropriate resource. Main controller service introduces the CoAP server for M2M communication. Figure 34 shows the overall request and response along with their method that happens in this project scenario.

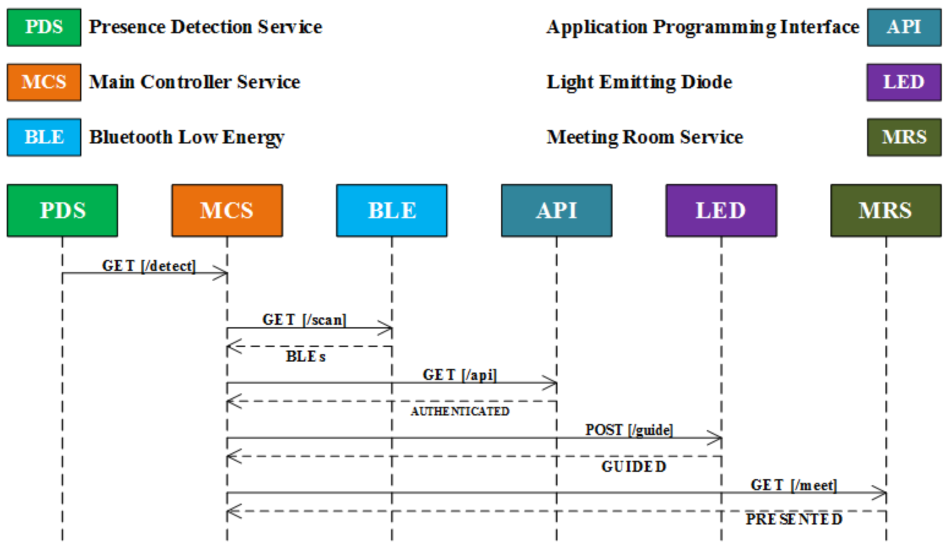


Figure 34. Messaging among of services in demo scenario.

There are 4 resources and all of them is handled by a GET method besides a POST method. This service records current processing status about every request that is in progress. Table 1 shows list of resources along with Uniform Resource Locator (URL) and logged a successful status after processing the request. The initial status is defined as INIT.

Table 1. Available CoAP resources in main controller service

URL	**Method**	**Resource**	**Status**
/detect	GET	DetectionResource	DETECTED
/scan	GET	ScanningResource	SCANNED
/api	GET	APIResource	AUTHENTICATED
/guide	POST	GuideResource	GUIDED
/meet	GET	MeetingResource	PRESENTED

Main controller service takes the control when it gets a request from Presence Detection Service (PDS) and starts processing the request to handle. Then it automatically sends a request to BLE scanning server to activate and start scanning. The main controller does this internally with *popen* [54] and transfer control to the BLE scanning service. BLE scanner further transfers the control back to the main controller with scanned encoded BLE devices as a response. MCS then sends the encoded BLE devices to API service to verify them by letting control to it. API returns the control back to the main controller with the appropriate status of the BLE authentication. The main controller sends a request to LED guidance with control to enlighten the meeting room path with the mobile agent. The main controller gets back the control from LED guidance when the authenticated user finally reaches in front of the meeting room. The main controller finally transfers the control to the Meeting Room Service (MRS) and hence do its tasks and then returns the control to the main controller service. MCS keeps track of successful status as shown in Table 1Table 1. It also keeps track of fail status and store in log prepending NOT to each successful status. For example, if a user is not authenticated with BLE key, then the main controller service will store the keep log the status as NOT AUTHENTICATED. All the device to device communication done with CoAP protocol.

3.4.1.2 Presence Detection Service

Presence Detection Service (PDS) is the very first service that detects presence of a warm body in the entrance. This uses PIR motion sensor as a hardware component. This component uses Infrared Radiation technology to detect a warm body passing through it. Infrared Radiation (IR) has a wavelength from about 800 nm to 1 mm greater than the red end of the visible light spectrum but less than microwaves and is emitted particularly by heated objects. Passive Infrared Radiation (PIR) motion sensor made of IR sensitive two material slots.

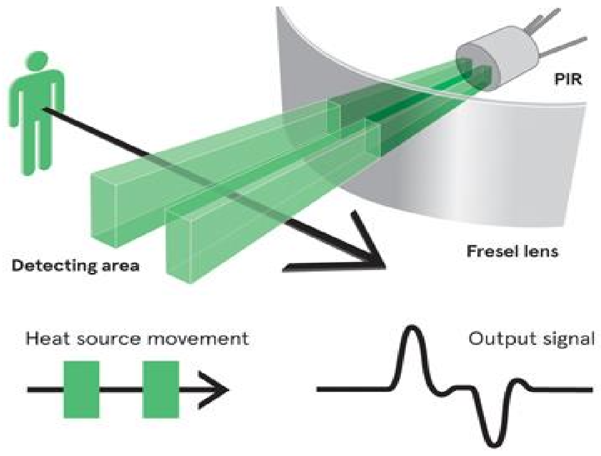
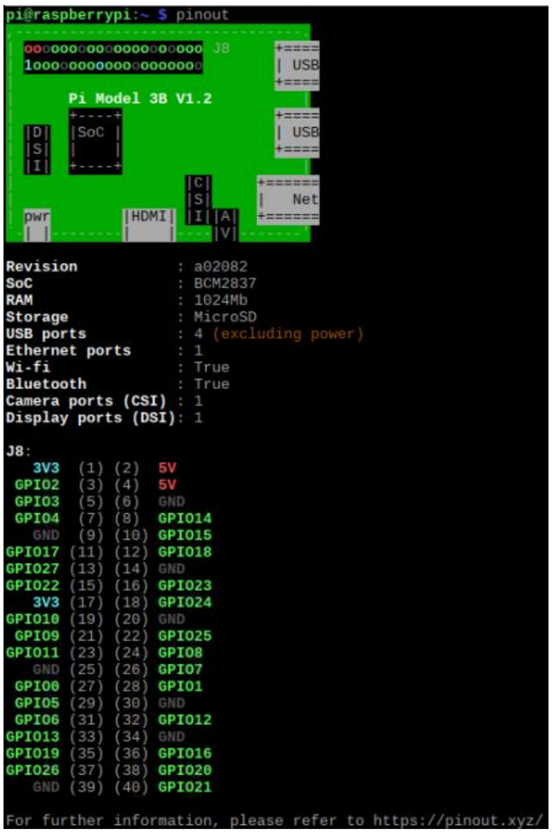


Figure 35. Presence detection with PIR motion sensor [51].

When a warm body like a human pass by the detecting area of PIR movement detection sensor, it first intercepts one half (material slot) of the PIR sensor, which causes a positive differential change between the two halves and hence rises voltage up into electrical circuit. It goes reverse back when the warm body leaves the sensing area whereby the sensor generates a negative differential change and hence again voltage goes down to zero levels into the electrical circuit. These change in voltage deference in the circuit make pulses are what is detected.

In this implementation, RPi3 and a PIR motion detection sensor as a physical apparatus has been used for presence detection. There are 3 pins in PIR sensor denoted as GND (ground), VCC (Common Collector Voltage); and OUT (Output of the sensor reading). It is required to access RPi's pin for General Purpose Input Output (GPIO) to read the output from PIR as motion detected in PIR. GPIO has 3 mode which includes BOARD, WiringPi; and BCM. GPIO pin config is not same in practice and hence need to set mode first to access the pin in RPi as GPIO such as GPIO4 is pin 4 in BCM mode while BOARD and WiringPi show it pin 7. There are a couple of ways to check RPi's pin config, but ***pinout*** [52] is most popular Command Line Interface (CLI) tool beside RPi's pinout [53] that described in detail. Figure 36 shows the pinout of BCM2837 RPi3 that used for presence detection service.



```
pi@raspberrypi:~ $ pinout
,--------------------------------.
| oooooooooooooooooooo J8     +====
| 1ooooooooooooooooooo        | USB
|                             +====
|      Pi Model 3B V1.2          |
|      +----+                 +====
| |D|  |SoC |                 | USB
| |S|  |    |                 +====
| |I|  +----+                    |
|                   |C|     +======
|                   |S|     |   Net
| pwr        |HDMI| |I||A|  +======
`-| |--------|    |----|V|-------'

Revision           : a02082
SoC                : BCM2837
RAM                : 1024Mb
Storage            : MicroSD
USB ports          : 4 (excluding power)
Ethernet ports     : 1
Wi-fi              : True
Bluetooth          : True
Camera ports (CSI) : 1
Display ports (DSI): 1

J8:
   3V3  (1) (2)  5V
 GPIO2  (3) (4)  5V
 GPIO3  (5) (6)  GND
 GPIO4  (7) (8)  GPIO14
   GND  (9) (10) GPIO15
GPIO17 (11) (12) GPIO18
GPIO27 (13) (14) GND
GPIO22 (15) (16) GPIO23
   3V3 (17) (18) GPIO24
GPIO10 (19) (20) GND
 GPIO9 (21) (22) GPIO25
GPIO11 (23) (24) GPIO8
   GND (25) (26) GPIO7
 GPIO0 (27) (28) GPIO1
 GPIO5 (29) (30) GND
 GPIO6 (31) (32) GPIO12
GPIO13 (33) (34) GND
GPIO19 (35) (36) GPIO16
GPIO26 (37) (38) GPIO20
   GND (39) (40) GPIO21

For further information, please refer to https://pinout.xyz/
```

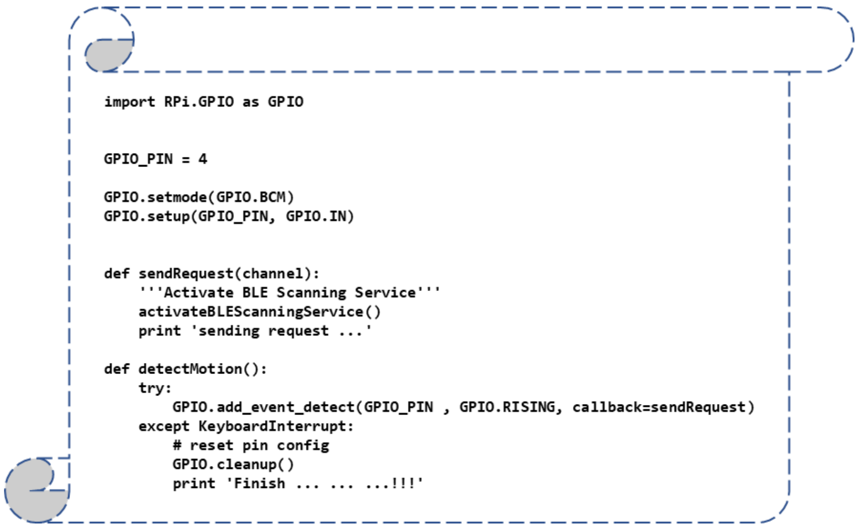
Figure 36. Pin configuration in Raspberry Pi 3 model B.

Connection configuration has been accomplished with RPi's pinout and BCM is used as pin mode to access RPi's pin for GPIO which is shown in Table 2. Here RPi's pin 1 and 7 provides 3.3 V to PIR and GPIO4 is used as input to RPi 3 as an output from PIR respectively.

Table 2. Connection configuration between PIR and Raspberry Pi 3 (RPi)

PIR pin	**RPi pin**
GND	9 (GND)
VCC	1 (3V3)
OUT	7 (GPIO4)

Finally, an event handler and a callback function help us when an object is detected around the PIR motion sensor. The event handler function checks if there is a voltage rise on the RPi GPIO4 where the callback function sent a CoAP request to main controller service to activate BLE scan service for scanning BLE devices along the corridor. Figure 37 shows code snippet that used to handle an event when there is a motion in PIR motion sensor.



```
import RPi.GPIO as GPIO


GPIO_PIN = 4

GPIO.setmode(GPIO.BCM)
GPIO.setup(GPIO_PIN, GPIO.IN)


def sendRequest(channel):
    '''Activate BLE Scanning Service'''
    activateBLEScanningService()
    print 'sending request ...'

def detectMotion():
    try:
        GPIO.add_event_detect(GPIO_PIN , GPIO.RISING, callback=sendRequest)
    except KeyboardInterrupt:
        # reset pin config
        GPIO.cleanup()
        print 'Finish ... ... ...!!!'
```

Figure 37. Code for motion event in PIR detection area.

CoAP client code to activate BLE scanning is written in *Twisted* framework which is very similar to Hyper Text Transfer Protocol in (HTTP) (REpresentation State Transfer) REST architecture. The code segment clearly demonstrates CoAP request function named *activateBLEScanningService()* written for BLE scanning in *sendRequest(channel)* callback function. The callback function call when there is a voltage up in GPIO_PIN into the GPIO event. Figure 38 illustrates that a GET request will send from presence service to main controller service.

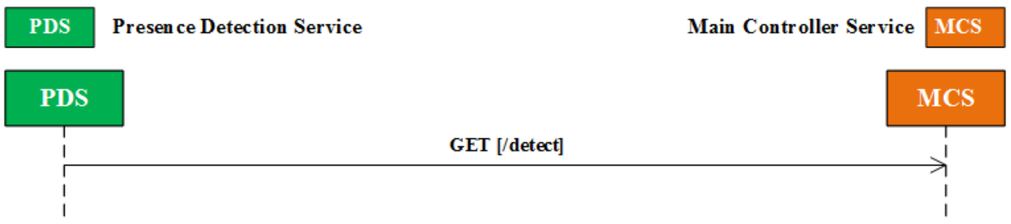


Figure 38. Request from PDS to MCS.

PDS aware of motion and periodically checks motion PIR sensor's effective area. If there is a change in sensor data, it gets control and then it sends a request to MCS to activate BLE service with a message status. Initially, PDS sets this INIT which will update to DETECTED and transfer control to MCS. The status record will be reset to INIT after sending the request to MCS. The main controller also keeps the status before processing BLE service for future reference. The entire procedure illustrated in the following Figure 39. The entire processing is handled in PDS.

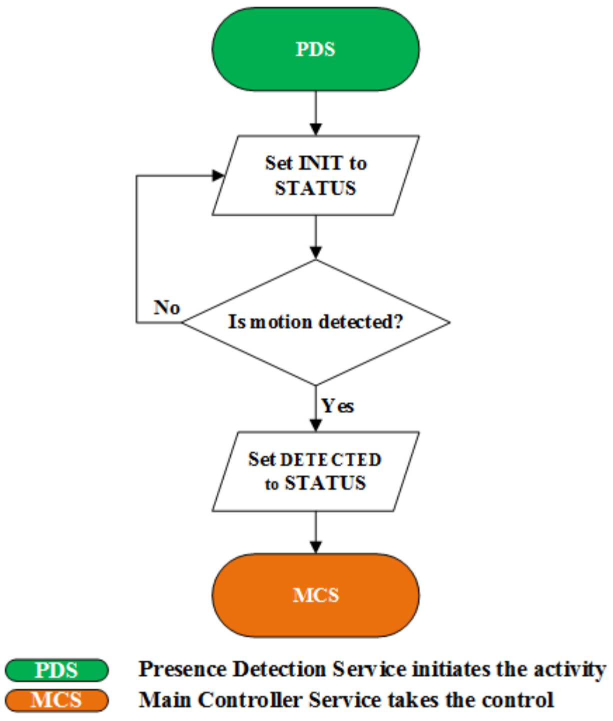

PDS
Set INIT to STATUS
Is motion detected?
No
Yes
Set DETECTED to STATUS
MCS
PDS Presence Detection Service initiates the activity
MCS Main Controller Service takes the control

Figure 39. Workflow for a request originated from PDS to MCS.

3.4.1.3 BLE Authentication Service

Bluetooth Low Energy (BLE) service gets control from the main controller to scan available BLE by advertising packet. Figure 40 depicts the connectivity between the main controller service and BLE authentication service.

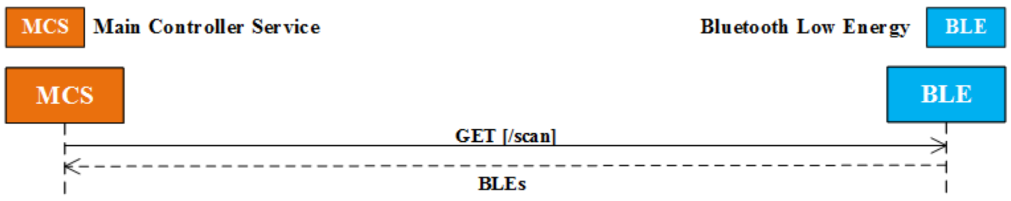
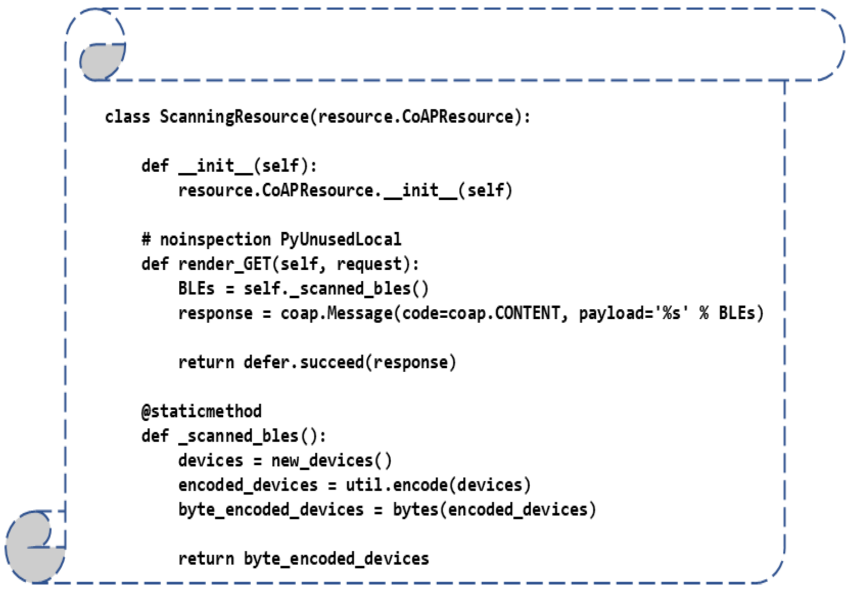


Figure 40. Communication between MCS and BLE service.

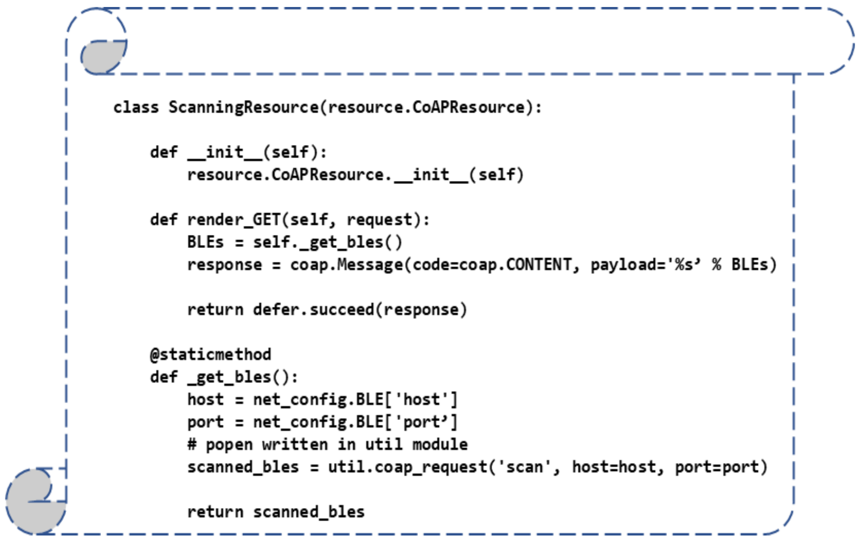
This service starts scanning usually scan for 3 seconds by default and encrypts all BLE devices after successful completion of encryption by using a modern algorithm. It requires network administration capacity to enable BLE scanning as root user [55]. Figure 41 depicts the connectivity between the main controller service and BLE authentication service.



```
class ScanningResource(resource.CoAPResource):

    def __init__(self):
        resource.CoAPResource.__init__(self)

    # noinspection PyUnusedLocal
    def render_GET(self, request):
        BLEs = self._scanned_bles()
        response = coap.Message(code=coap.CONTENT, payload='%s' % BLEs)

        return defer.succeed(response)

    @staticmethod
    def _scanned_bles():
        devices = new_devices()
        encoded_devices = util.encode(devices)
        byte_encoded_devices = bytes(encoded_devices)

        return byte_encoded_devices
```

Figure 41. BLE service server code segment.

The main controller sends this request to BLE scanning service with *popen* and it gets back the control as a response from BLE scanning service. In the main controller service, the *coap* subprocess is responsible for this internal communication between MCS and BLE authentication services. Here, MCS or main controller service and BLE authentication service can exchange the information with the help of CoAP messaging protocol.



```
class ScanningResource(resource.CoAPResource):

    def __init__(self):
        resource.CoAPResource.__init__(self)

    def render_GET(self, request):
        BLEs = self._get_bles()
        response = coap.Message(code=coap.CONTENT, payload='%s' % BLEs)

        return defer.succeed(response)

    @staticmethod
    def _get_bles():
        host = net_config.BLE['host']
        port = net_config.BLE['port']
        # popen written in util module
        scanned_bles = util.coap_request('scan', host=host, port=port)

        return scanned_bles
```

Figure 42. BLE client code written in MCS.

BLE service gets control when it gets a request from main controller service i.e MCS. It encrypts BLE device ID if it found any devices at the entrance. MCS keeps either DETECTED or NOT DETECTED based on the response of BLE service.

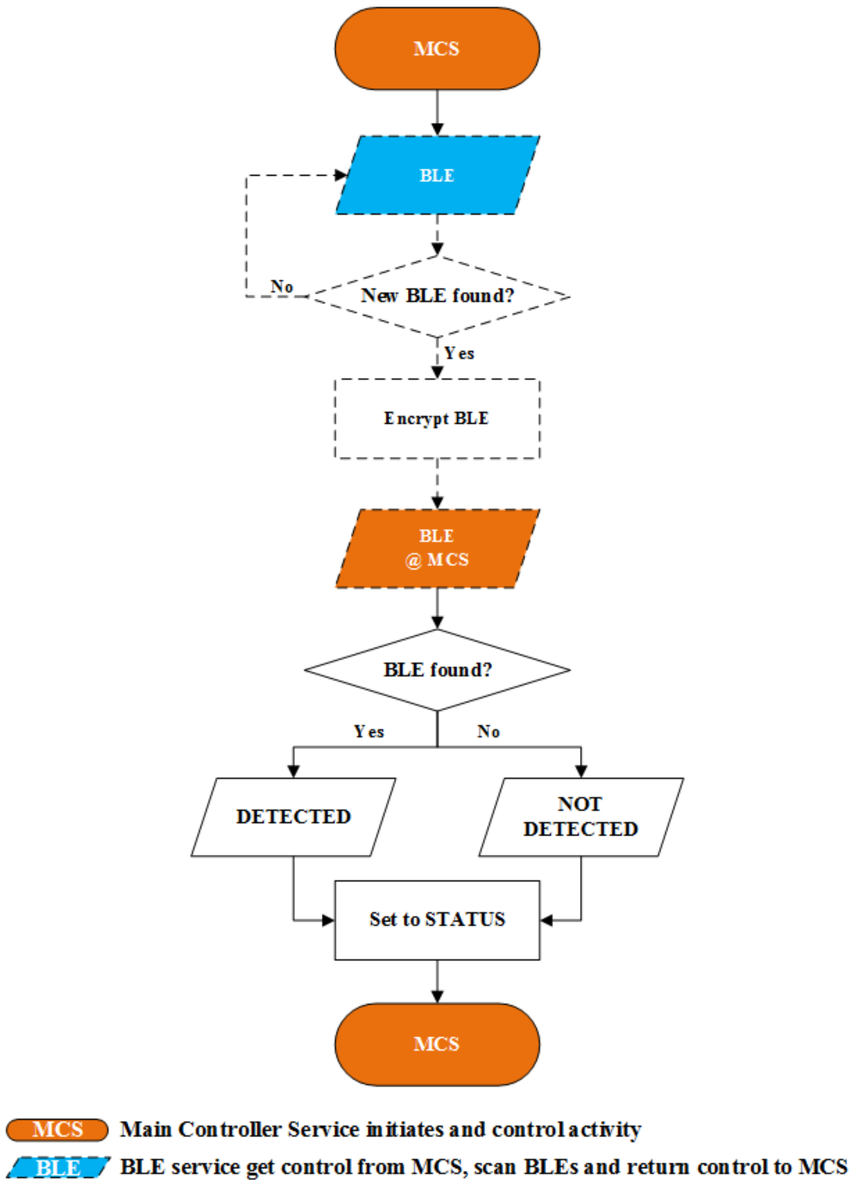

MCS
BLE
New BLE found?
No
Yes
Encrypt BLE
BLE @ MCS
BLE found?
Yes
No
DETECTED
NOT DETECTED
Set to STATUS
MCS
MCS Main Controller Service initiates and control activity
BLE BLE service get control from MCS, scan BLEs and return control to MCS

Figure 43. Status record in MCS & BLE.

Solid lines indicate that actions were taken by main controller service whereas dashes are for BLE service.

3.4.1.4 API Service

In the demo project, Application Programming Interface (API) service has been implemented as another microservice so that the meeting events stuff can be kept for future record. It gets control from main controller service or MCS having list of BLE devices to check delegates or guests of a meeting event. Figure 44 denotes the exchange of information between main controller service and the API.

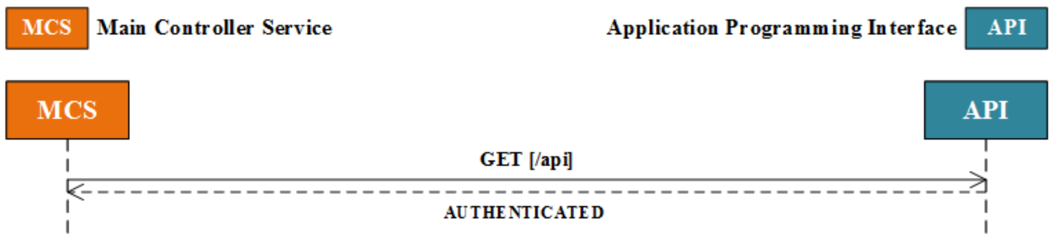


Figure 44. Messaging between MCS and API microservices in PoC.

In all Object-Oriented Programming (OOP), classes are the blueprint of objects consisting of properties and methods [56]. API service has few classes having an appropriate logical relationship. Appendix 1 represents the relationship among the entities that made for this implementation whereas the following Figure 45 shows UML class diagrams of the API service.

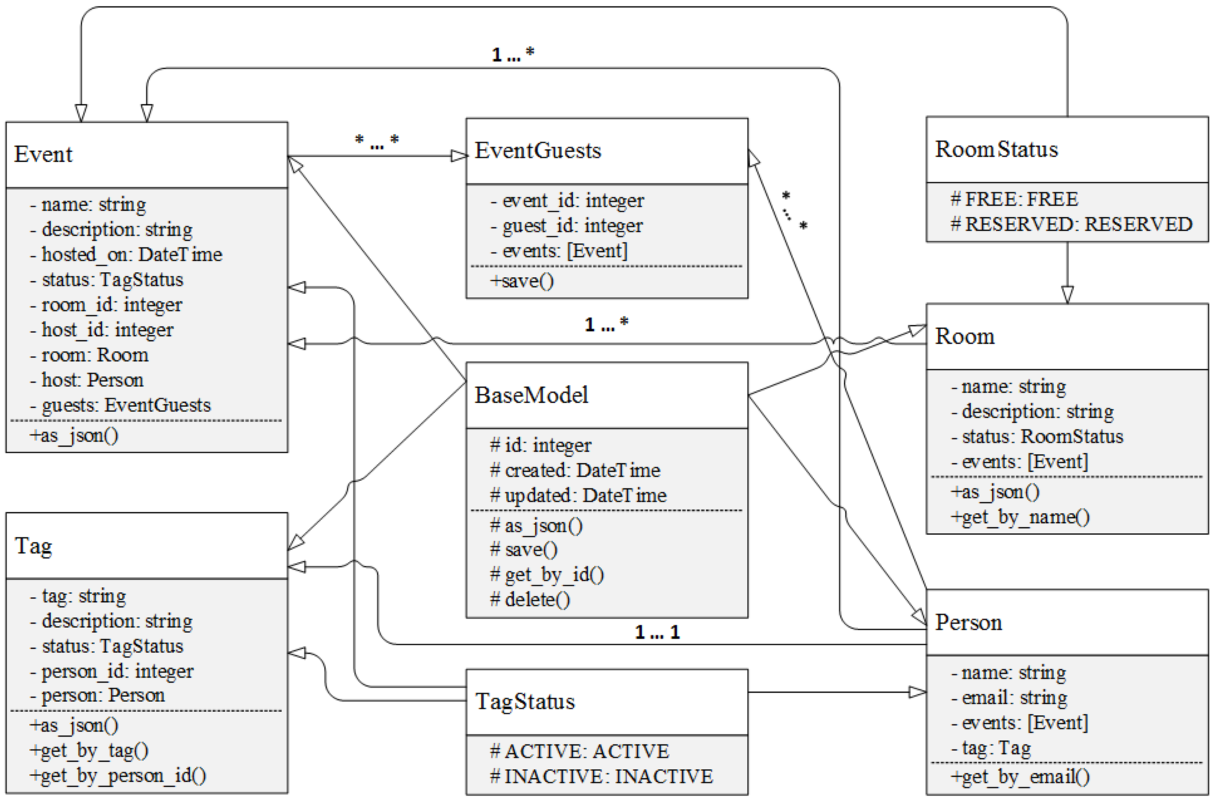


Figure 45. UML class diagram for API service.

Constrained Application Protocol (CoAP) was used to make a communication in the main controller service and the API service. CoAP itself is on top of user datagram protocol or UDP. The API works as a server machine where the main controller service or MCS acts as a client that request resources for BLE authentication. Following Table 3 denotes the available resources into API service.

Table 3. API resources available in API service

URL	Method	Resource	Purpose
/api/persons	GET, POST	PersonsResource	Check list, create new
/api/persons/{id}	GET, PUT	PersonResource	Check & update
/api/rooms	GET, POST	RoomsResource	Check list, create new
/api/rooms/{id}	GET, PUT	RoomResource	Check & update
/api/tags	GET, POST	TagsResource	Check list, create new
/api/tags/{id}	GET, PUT	TagResource	Check & update
/api/events	GET, POST	EventsResource	Check list, create new
/api/events/{id}	GET, PUT	EventResource	Check & update

The API service is completely made of *python* with *flask* framework. The incoming request from MCS is received at API with contained application protocol. The request is further processes with *popen* that uses *curl* to access TagResource uniform resource locator or URL. These URLs are in REST architecture. Figure 46 shows what status will be stored at main controller when the request starts from MCS that sends to API service which finally stops at MCS.

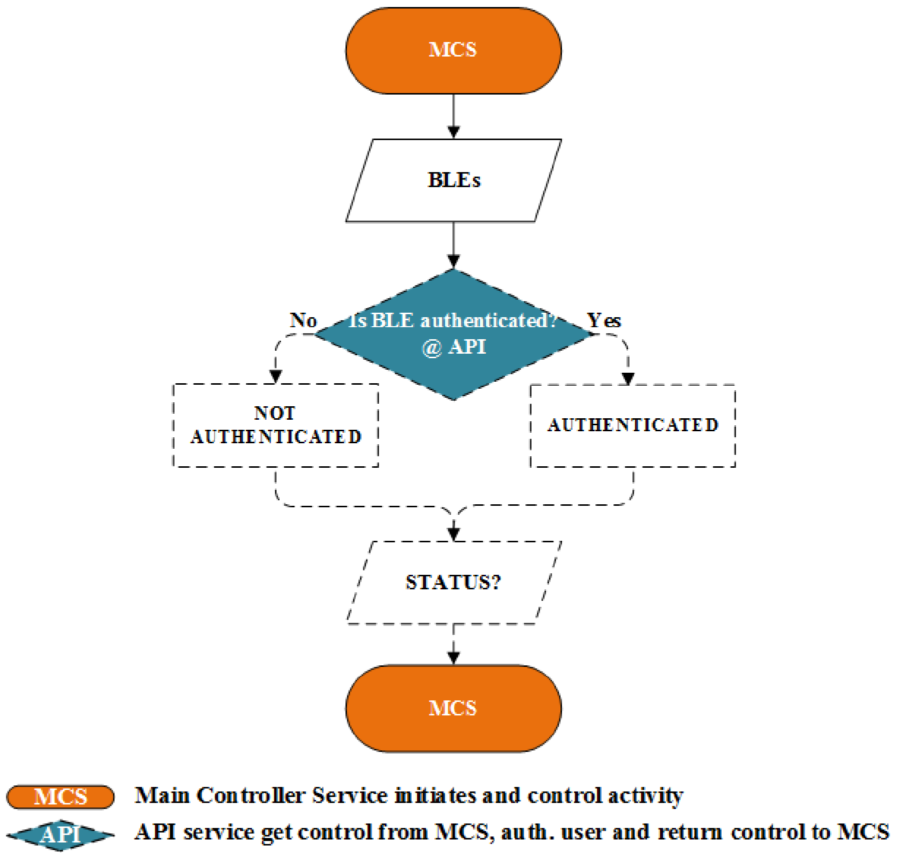

MCS
BLEs
No
Is BLE authenticated? @ API
Yes
NOT AUTHENTICATED
AUTHENTICATED
STATUS?
MCS
MCS
Main Controller Service initiates and control activity
API
API service get control from MCS, auth. user and return control to MCS

Figure 46. MCS log status as a response from API.

Every solid line into the above flow chart is the request that occurs from MCS while the rest of the dash lines are processed at API for responding to the request of main controller service.

3.4.1.5 LED Guidance Service

Light Emitting Diode (LED) is used for guiding a guest from entrance corridor to the meeting room. The LED has been used as a component and placed many locations along the entire path from source i.e entrance to destination i.e meeting room. LED guidance service will be activated by the main controller service or MCS. Figure 47 represents the communication state with request and response.

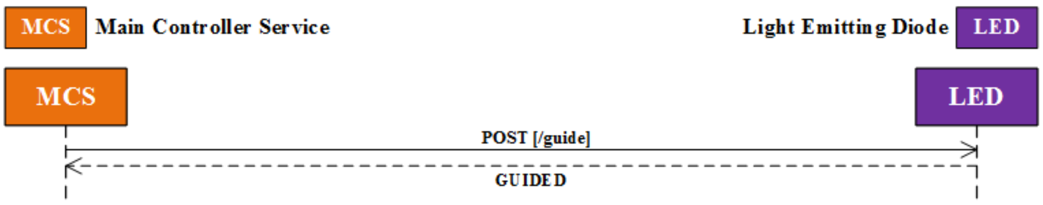


Figure 47. Request vs response in between MCS and LED guidance.

Mobile agent algorithm [46] – [48] has been implemented to the LED guidance service. MCS keeps a status record for MCS – LED communication. It involves, 1) MCS sends a CoAP GET request to LED, 2) set STATUS = NOT GUIDED, 3) blink LEDs until a guest or participant does not reach to meeting room, 4) stop LED if guest reach and sends payload message to MCS as response; and 5) update STATUS = GUIDED. Figure 48 shows how status changes in this case.

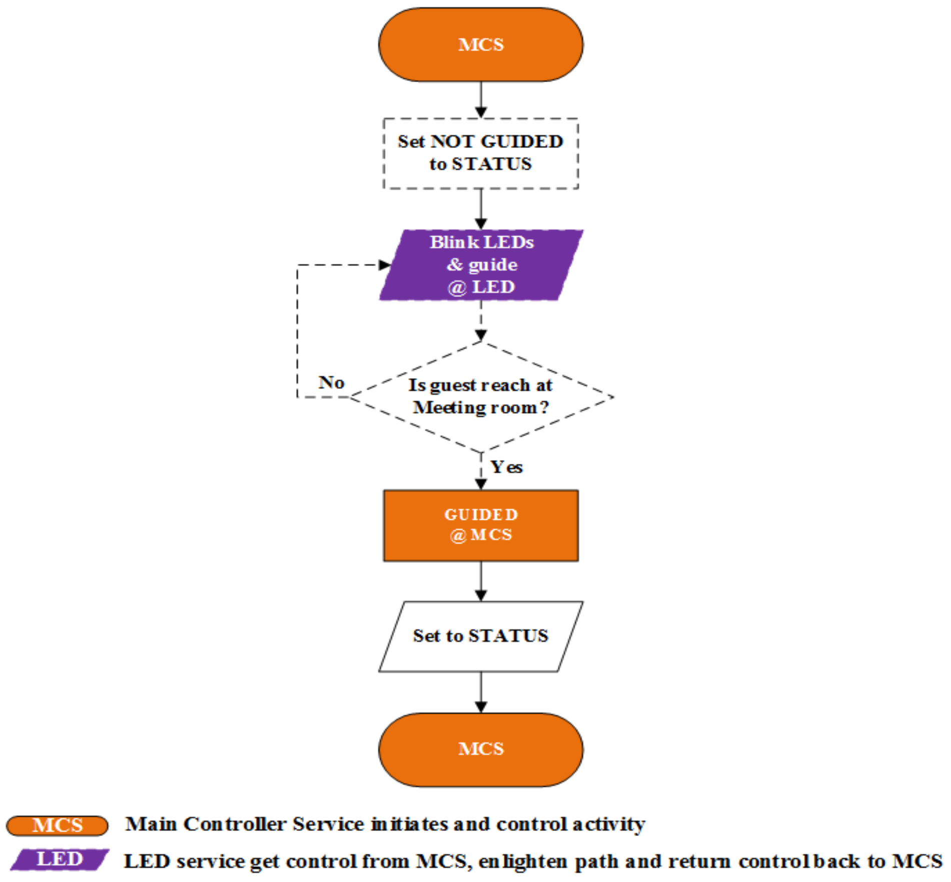

MCS
Set NOT GUIDED to STATUS
Blink LEDs & guide @ LED
No
Is guest reach at Meeting room?
Yes
GUIDED @ MCS
Set to STATUS
MCS
MCS
Main Controller Service initiates and control activity
LED
LED service get control from MCS, enlighten path and return control back to MCS

Figure 48. Status at MCS - LED communication.

Here solid lines refer to the actions taking place at the main controller service while dash lines indicate the steps performed at LED guidance.

3.4.1.6 Meeting Room Service

Meeting Room Service or MRS is used for authentication, hide personal contents, download and presents meeting contents. This service is activated automatically when the MCS – LED process has been over. Figure 49 shows the request and response between MCS and MRS.

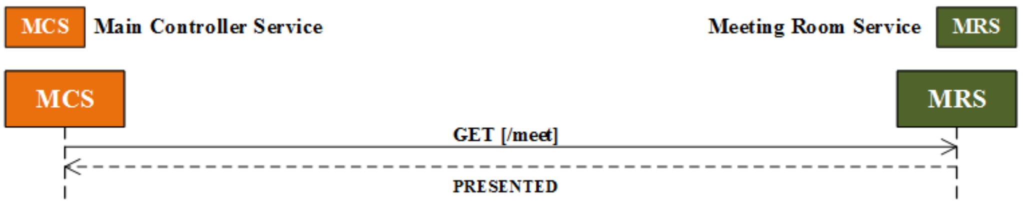


Figure 49. MCS – MRS communication state.

MCS gets control from prior state and upgrades status as NOT PRESENTED primarily at the. After that, it sends a request to finish further processes i.e authentication, content related subprocesses. Authentication is done by authentication service described at the earlier. Status is updated as soon it hides private contents and presents meeting document after downloading from the server. Figure 50 represents the workflow that carried out at MCS – MRS.

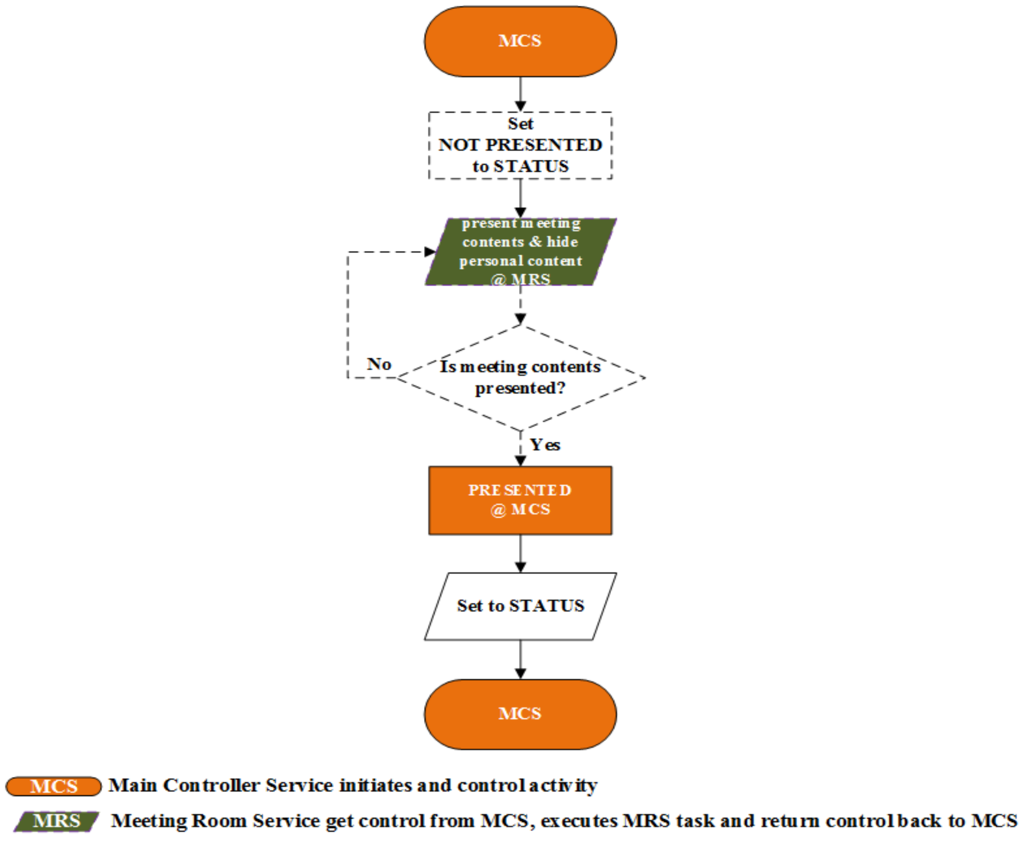

MCS
Set
NOT PRESENTED
to STATUS
present meeting contents & hide personal content @ MRS
No
Is meeting contents presented?
Yes
PRESENTED @ MCS
Set to STATUS
MCS
MCS
Main Controller Service initiates and control activity
MRS
Meeting Room Service get control from MCS, executes MRS task and return control back to MCS

Figure 50. MCS – MRS workflow.

Solid lines and dashes are presented for work action at MCS and MRS services.

3.4.2 Service Interaction

An example use case scenario has been set at the very beginning of the implementation as a proof of concept to contribute to the decentralized IoT edge nanoservice architecture for future gadget-free computing. Practical use case scenario of the PoC implementation is presented in the following Figure 51.

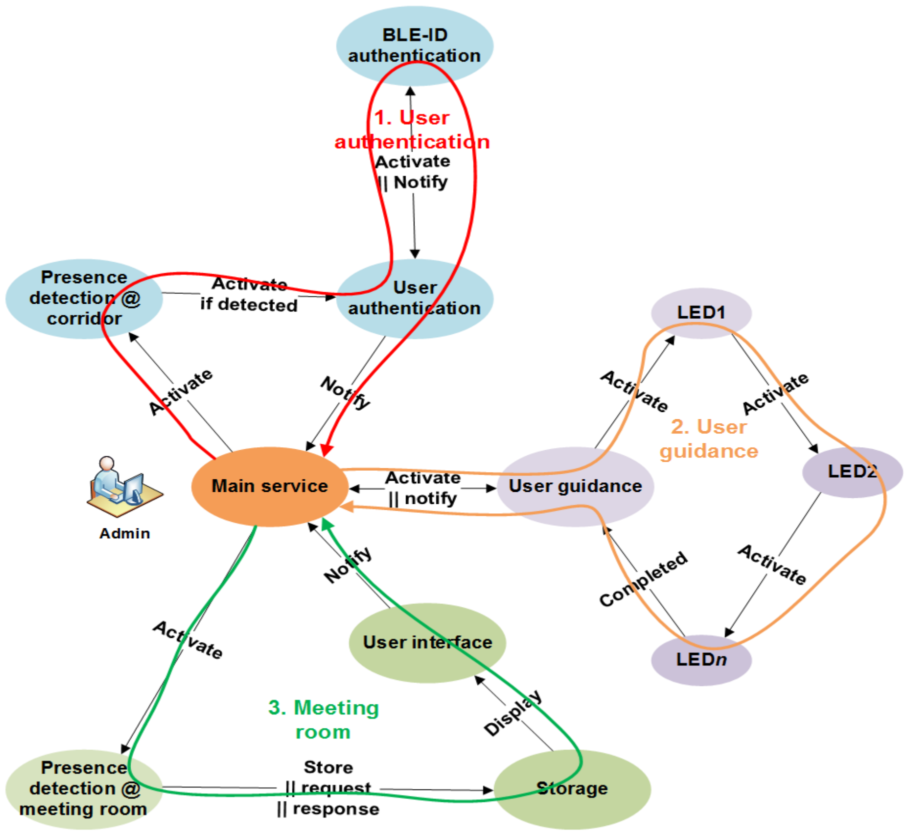


Figure 51. Example scenario for proof of concept [23].

Presence detection service will be activated automatically when a participant is detected by PIR motion sensor that put on the entrance. PDS will create an actor that set a command message to start the automatic authentication by BLE and this message to MCS. After that, MCS will create another actor that put a command message to BLE to scan Bluetooth devices around the location of the scanner and get the list of scanned devices from BLE. MCS then creates another actor that sends these to API to recognize a participant with a BLE-ID. When a participant is authorized by the system then MCS will create a new actor that put a command message to LED to guide the participant to the meeting room. Lastly, MCS will create an actor with a command message to MRS to display meeting content at the monitor when a person is detected for the first time at the meeting room. The whole scenario is presented below.

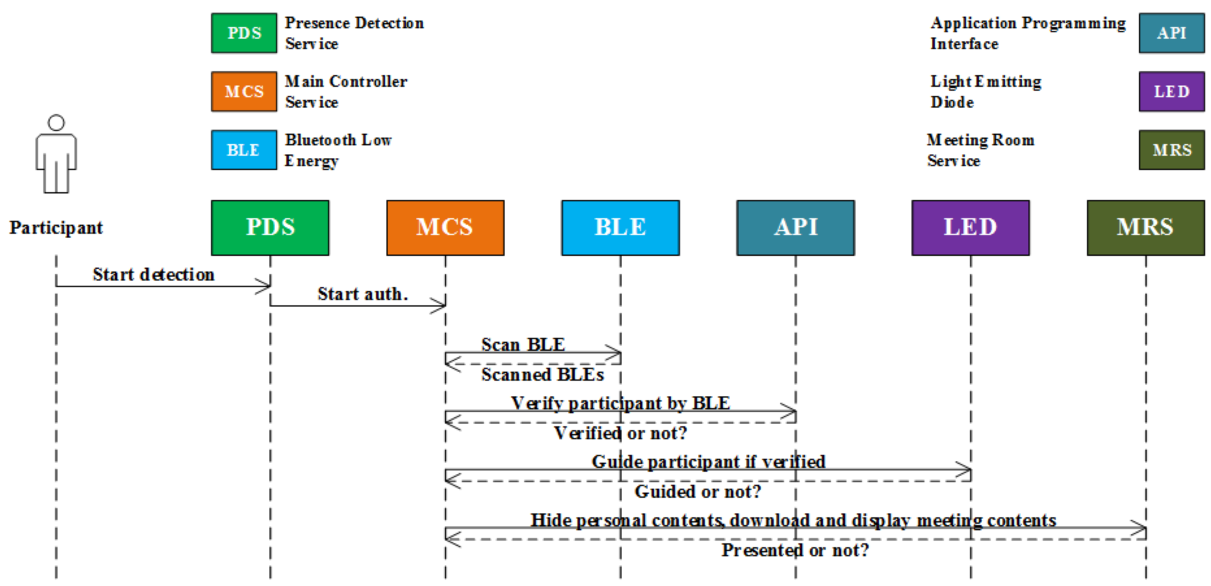

PDS Presence Detection Service
MCS Main Controller Service
BLE Bluetooth Low Energy
Application Programming Interface API
Light Emitting Diode LED
Meeting Room Service MRS
Participant
PDS
MCS
BLE
API
LED
MRS
Start detection
Start auth.
Scan BLE
Scanned BLEs
Verify participant by BLE
Verified or not?
Guide participant if verified
Guided or not?
Hide personal contents, download and display meeting contents
Presented or not?

Figure 52. Internal service interaction.

3.4.3 Deployment and Container Orchestration

A new container image is constructed for each required microservice that developed for this demo scenario. Every container image is based on an appropriate base image that satisfies the requirements of a service which is called a base image of that container. The summary of the final deployment is concluded in Table 6.

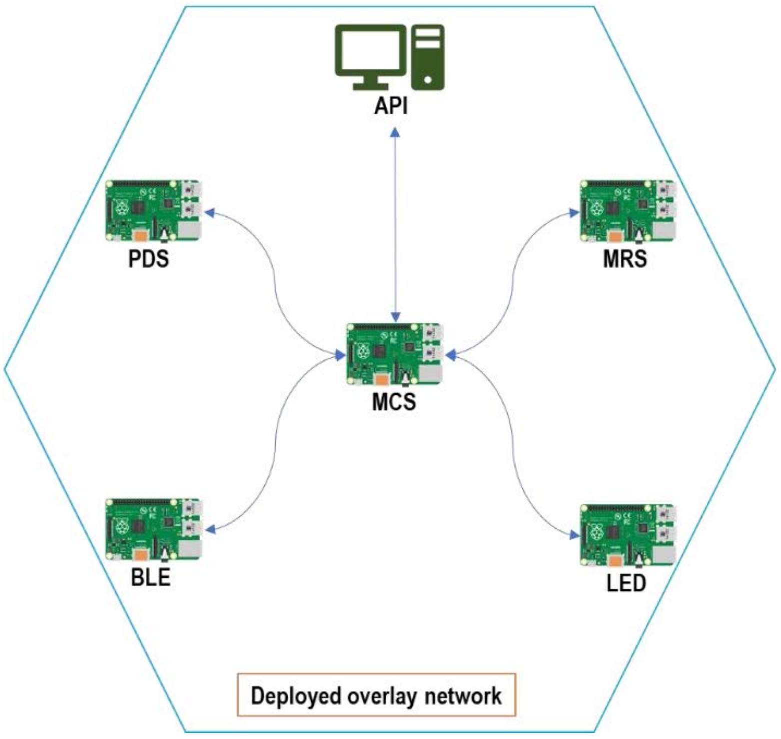


Figure 53. Container orchestration for the PoC.

In the implementation, each service is deployed in a separate Docker host. In this system, services are isolated and mostly decoupled. This architecture is known as an overlay network. This network is best when containers are running on different Docker hosts, or when multiple applications work together using swarm services [68]. Container orchestration has been done with Docker Swarm and Kubernetes. The PoC implementation required 6 services. These services were orchestrated by the following procedure:

1. **Deploying Services**: Deploying each service into the associated host.
    - PDS, MCS, BLE, LED; and MRS on 5 RPis.
    - Deploy API on Linux machine.
2. **VM Creation**: Create 7 different VMs where
    - 6 of them for services as worker nodes; and
    - 1 for manager node.
3. **Attaching VMs**: Attach each service to a separate VM.
4. **Joining Nodes**: join each worker node to the manager node.

4 EVALUATIONS

During the prototyping work, some observations were made in favor of the implementation for the demo scenario, although the focus of this paper is on deployment for local IoT edge services. Some common observations were made based on the implementations and its functionality with the resource consumption with few performance metrics. These studies are fit for the empirical design based on the available resources that have shown a significant performance. Following sections describe parametric case studies that taken at simulations.

4.1 Evaluation Setup

Six microservices were developed for the demo scenario which is demonstrated in section 4.1.1. REST API deployed into ubuntu 18.04 and LED guidance deployed into Raspberry 2 Model B. Rest of the four micro-services deployed into Raspberry Pi 3 Model B+. Recall section 4.1.2 again where a rough technical specification discussed for the implementation of the demo project. Each micro-service uses CoAP protocol for M2M communication. Here every micro-service primarily required a client script also beside the server script and deployed separately except presence detection service or PDS. Both client and server script were written in Python 2. In the demo scenario, the client and server were handled with subprocess pythons' module which introduces complicacy and not recommended [64]. Node-based CoAP CLI is used to process client request into the dependant server as discussed in Sola's article [42] requiring node installation additional rather having Python-based client container. In addition, this process significantly reduces the number of containers which save memory.

Container orchestration has been implemented into Oracle VirtualBox with version 5.2. In the microservice deployment, Docker Swarm and Kubernetes is used. Docker swarm and Kubernetes are two platforms for orchestrating containers. However, Docker swarm is completely based on command line interface or CLI while Kubernetes came up with CLI as well as GUI. CLI based application has great performance over GUI though GUI gets better user experience over CLI. Docker Swarm is integrated into the Docker ecosystem with its own API. In practice, Kubernetes is ahead of Swarm in an average. Largest open source community.

4.1.1 Hardware Specifications

This thesis has installed a physical hardware setup for the entire system to fulfil Alice's requirements. Physical setup requires a Passive Infrared Radio (PIR) sensor for motion detection or presence detection service. In this system, authentication service includes BLE beacon to authenticate a participant. This thesis will focus on only BLE based authentication for this PoC, instead of multimodal authentication mechanism proposed by Harjula et al. [23]. LED was needed for mobile agents and require UI devices like monitor for authentication; and projector for screening the meeting content. A host machine is required to deploy API for authentication and storage services and a WiFi access point (router) to deploy all services under the same network. Design section of this chapter illustrates possible microservices and implementation section describes the technical requirements to ended up a successful project for the demo case. Table 4 shows service oriented physical equipment to design a fully functional scenario.

Table 4. Tentative physical requirements for services in PoC implementation

Services	**Requirements**
Presence Detection Service (PDS)	PIR sensor, RPi3
Main Controller Service (MCS)	RPi3
Bluetooth Low Energy (BLE)	BLE, RPi3
Application Programming Interface (API)	Linux machine
Light Emitting Diode (LED)	Arduino shield, LED, RPi3
Meeting Room Service (MRS)	BLE, RPi3, UI (e.g monitor)

4.1.2 Software Specifications

The implementation also figures out all the required technologies subject to the PoC implementation. Table 5 shows service oriented technically specified terminologies used to design a fully functional PoC implementation.

Table 5. Service oriented technical specifications for PoC implementation

Services	**Requirements**
Presence Detection Service (PDS)	CoAP, Python
Main Controller Service (MCS)	CoAP, Python, NodeJs
Bluetooth Low Energy (BLE)	CoAP, Python
Application Programming Interface (API)	Python, Flask, MySQL, CoAP
Light Emitting Diode (LED)	C++, Python, CoAP
Meeting Room Service (MRS)	Python, CoAP

To establish machine to machine communication, it has used the Constrained Application Protocol (CoAP). The protocol itself is on top of a well-known framework. Thanks to *txThings* [50] and *CoAPthon* [49] developers' team for their great contribution. LED guidance service implemented with *CoAPthon* where rest of the services are implemented with *txThings*. Here presence detection service is fully designed as CoAP client whereas the main controller service is a mixer of server and client request simultaneously. Rest of the microservices is developed for handling client request as separated service-oriented server.

4.2 Evaluation Results

The implementation has an evaluation setup for the experiment during the deployment. The evaluation is carried mainly for measuring the source consumption, the performance of a microservice. Following sections describe the optioned evaluation results.

4.2.1 Resource Consumption

It is important to keep track of storage consumption of the microservices since they are deployed locally on IoT edge. Table 6 shows the resource consumption of deployed microservices at the demo scenario implementation. Here, *Dockerfile* and Docker-compose indicate the actual reading. Although every device allocates at least 4 kB for a file if that is not empty. On the other hand, database (DB) persistence is an on-demand service that deployed with API service.

Table 6. Resource consumption

Component / service	Space on disk							
	Installation	Base Image		Container Size		Dockerfile	Docker-compose	Source code
		Name	Size (MB)	Shared (MB)	Run			
Docker	95 480 kB							
Kubernetes	195 998 kB							
DB		mysql	0 MB	485.5 MB	7 B			
API		python:3-onbuild	690.5 MB	88.18 MB	414 kB	1 213 B	2 546 B	223 kB
MCS		arm32v7/python:2.7.15-jessie	557.3 MB	42.31 MB	231 kB	418 B	321 B	73.8 kB
PDS		arm32v7/python:2.7.15-jessie	557.3 MB	42.34 MB	235 kB	613 B	309 B	34.3 kB
BLE		arm32v7/python:2.7.15-jessie	557.3 MB	52.49 MB	860 kB	791 B	514 B	62.5 kB
LED		resin/rpi-raspbian:jessie	128.2 MB	276.5 MB	148.3 MB	476 B	189 B	2.76 MB
MRS		arm32v7/python:2.7.15-jessie	557.3 MB	54.94 MB	1245 kB	975 B	719 B	67.6 kB

The demo application has been developed on top of Docker ecosystem and container orchestration has been done in Docker swarm as well as Kubernetes. Docker ecosystem includes Docker and Docker-compose to build and run Docker images into Docker daemon inside a host operating system and consumed about 96 MB. The version of Docker was 18.03.1-ce (for Ubuntu) as well as 18.09.0 (for RPi) and Docker-compose was 1.23.1. On the other hand, the entire Kubernetes ecosystem (includes kubeadm, kubectl, kubelet; and kubernetes-cni) consumed about 196 MB having version 1.13.1.

4.2.2 Performance

This implementation has performed the evaluation of the model after the successful equipment setup including hardware and software. It also observed overall start-up time to initialize a service. All microservices in the demo scenario has a great performance at boot up stage. The overall experimental outcome was taken from the deployed microservices. In the implementation, six microservices were used for the demo scenario. Figure 54 presents the overall deployment time of during the service deployment.

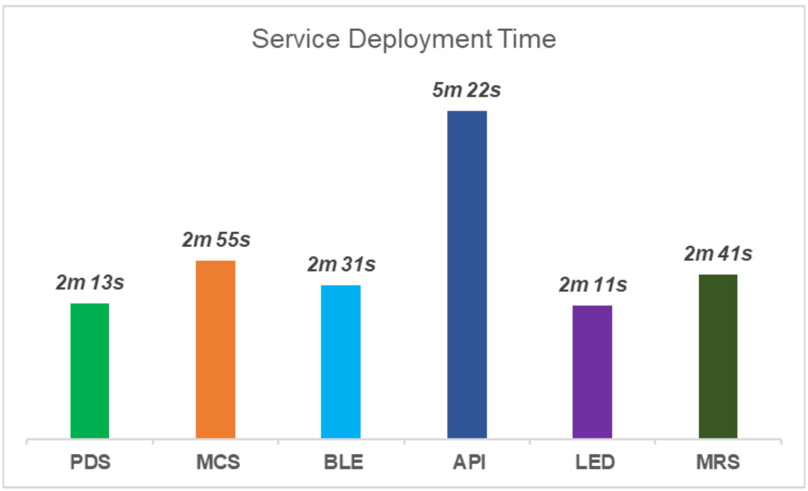


Figure 54. Service deployment time.

According to Figure 54, Presence Detection Service (PDS), Main Controller Service (MCS), BLE service (BLE), API service (API), LED service (LED) and Meeting Room Service (MRS) took 2m 13s, 2m 55s, 2 m 31s, 5m 22s, 2m 11s, and 2m 41s respectively at the deployment process. The deployment time is mostly depending on the available resources where a service is being deployed and the available bandwidth of the internet connection. During building a Docker image, a Docker daemon will first download the dependencies regarding the specification. In general, a deployment process takes less amount of time if there is already have a base image on a Docker daemon. In the implementation, the Docker daemon has no base image. For this reason, it took much time to deploy a service on a Docker engine. The effect on service deployment time for an existing image would evaluate in future.

The service initiation time was also observed after a successful deployment. These deployed services take roughly 11-15 seconds on an average when they are ready to use after a successful deployment.

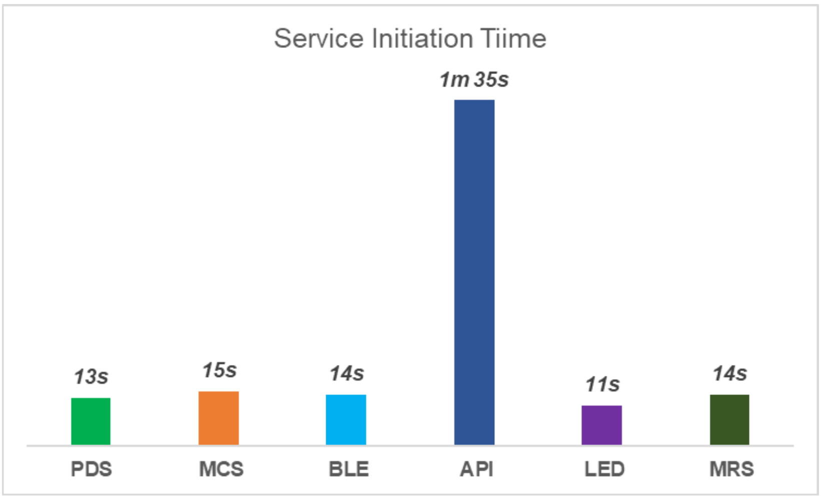


Figure 55. Service initiation time.

According to Figure 55, six services such as Presence Detection Service (PDS), Main Controller Service (MCS), BLE service (BLE), API service (API), LED service (LED) and Meeting Room Service (MRS), took roughly 13s, 15s, 14s, 1m 35s, 11s, and 14s respectively for service initiation. Here, API took maximum service initiation time among 6 services. API is dependent on the database (DB) component which takes around 1 minute extra to start working. Therefore, API is waiting for DB and this takes around 1.5 minutes to work if DB is not in service or not in a downstate.

4.2.3 Authentication Accuracy

This PoC has implemented with BLE-based authentication. This authentication process gives 100% accuracy on the average network condition. It took 3 seconds on an average to authenticate a person. The summary of the experiment is shown in Table 7.

Table 7. Averaging the authentication time

Serial Number	**Authentication Time (x_i)**	**Average ($\sum x_i / N$)**
1	4s	3s
2	3s	
3	2s	

BLE devices send packets to others to let them know about its existence known as packet advertising. Technically, it works in the range of 300 feet or 100 meters but practically it is quite challenging to work over 10 meters or 20 meters [82].

In the implementation, authentication is done in two separate processes: 1) BLE scanning which takes place in between MCS and BLE services; and 2) authentication with API which takes place in between MCS and API services. In the experiment, the total time to authenticate a person is depends on the quality of the packet advertising, physical distance between the BLE-based authentication server and the BLE key; and the overall internal processes of the MCS-BLE and the MCS-API. Any of these can affect the authentication process and hence increase the total authentication time.

5 DISCUSSION AND FUTURE WORK

The IoT network comprises of several resources such as sensors, actuators; and other constrained devices. The key challenges for such an IoT network include resource and energy consumptions, low latency, high bandwidth, reliability, scalability, interoperability, security, and privacy among others. The gadget-free vision assumes secure access to ubiquitous user-friendly digital services without using explicit gadgets. IoT networks are considered to be vital for developing such gadget-free and smart systems. The major requirements include inter-operable service provisioning and availability of un-interrupted secure services during the transition of a user in smart spaces. The other requirements are authentication of valid users through the capabilities available in the nearby smart surroundings and provide low latency aware secure services. Thus, the vital challenge is how to utilize local resources efficiently in order to fulfill these requirements. The main problem is the limitation of hardware resources at the local layer which brings a limitation to the storage, processing, and computations. Hence, because of restricted resources, there may cause various issues to the local processes in terms of, for example, may not be secure, unavailability of the service or might take longer delays executing processes. In this context, a microservices architecture is required for such context-aware inter-active smart environment that utilizes local resources but can provide various virtualized services at the local layer. This architecture should be deployable and re-deployable based on the requirement. In this thesis, an implementation carried out in favor of this architecture. This chapter highlights the outcome of this thesis with its target point. Finally, a goal is set for future improvements for this thesis.

5.1 Achieving the Thesis Goal

The target of this thesis was to design a local system to improve service provisioning at the local IoT network by utilizing local resources and surroundings. In chapter 2, the existing models and related work has been reviewed first before going into in-depth implementation for prototyping. It has discussed various modern terminology in the field of communication technology such as Cloud, Edge, Fog, and latest Mist computing. The IoT protocols, application design patterns and other correlated technologies described in related work help us to achieve the thesis goal. Then, several IoT protocols have been introduced for the implementation of the work. The entire IoT protocols have been broken down into five several layers. These five layers include application, transport, network, datalink, and the physical layer. Here, it has discussed numerous advanced protocols used in the application layer. Container-based lightweight virtualization brings the opportunity to use local resources efficiently. Container orchestration tools such as Docker Swarm and Kubernetes has been demonstrated with mobile agent and actor model.

In chapter 3, this thesis set a use case scenario to satisfy the implementation. In the thesis flowchart, UML, activity; and sequence diagrams has been used for a successful demonstration. The big challenge in the implementation was to choose an appropriate protocol for the applications required for this demo scenario. since it uses several constrained devices such as RPis and others, Constrained Application Protocol or CoAP at the application layer, got the best fit for the demo project. Microservices design approach provided maximum flexibility to the development. This approach helps us at the design of services by decoupling them on a logical level. Each service deployed as a single application in microservice paradigm into separate container. Interoperability has been reached by the conceptual actor model whereas the mobile agent helped us in the mobility of smart services. In the implementation, it has used

IoT devices like PIR motion detection sensor and surveillance camera for the demo project. These devices were embedded with the Raspberry Pi, Single Board Computers (SBCs). The overlay network design enabled us to access these devices with their respected IP address. During the implementation, it has made a few scripts for those devices to configure and redefine features along with the devices.

In chapter 4, evaluation of the work has been done for the implementation. At first, the hardware and the software setup for the evaluation of the implementation has discussed. The entire hardware system was on Unix OS i.e RPi (the Raspbian Jessie OS) and Ubuntu (Linux). For simplicity, a few shell scripts have been written to enhance the implementation with proper documentation. Those shell scripts make the deployment process easy enough to understand as well decorated documentations. The implementation has accomplished carefully with Python code whereas the server and the network virtualization were done with the lightweight Docker-based containerization technology.

Scalability and security are two important measurement factors to evaluate an IoT network. Here, this implementation has observed the maximum network size and the security of the demo scenario though these were not the main goal of this study, but these were important for the observation. In this demo scenario, it has deployed the required services through an access point having an IP of C class, hence, it can have 254 hosts at maximum. On the other hand, it has a great benefit in terms of security without global access to hackers since all the microservices has deployed into local servers.

5.2 Future Work

In the implementation, a user was verified with only BLE instead of approaching the full multimodal authentication process which is combination of WiFi CSI, BLE and surveillance camera, as defined in the concept article [23]. Face detection using surveillance camera is an advanced and a popular approach that recently applied in modern artificial intelligence system to identify a person by the system accurately. It could introduce the face detection algorithm into the demo scenario at the authentication service to identify a user.

The PoC is carried out with official regular base images which consume massive disk spaces compared to alpine-based images. All these regular images have three basic options such as Jessie, Wheezy or Slim. These regular images have extra configuration besides the essential OS libraries [83]. On the other hand, alpine images contain minimum required dependencies along with the OS libraries and take small hardware resource without extra configurations and settings [79]. Therefore, alpine-based image could be introduced to the demo project to reduce resource consumption.

The evaluations have been carried out for 6 - 7 persons in a simple environment setup. However, it could be evaluated within a complex environment by increasing the number of participants. It would be more interesting for future work to study with a full simulation about the scalability, performance, energy efficiency of this PoC implementation. Some other optimization matrices including latency, bandwidth, throughput, cost, network stability may also require special attention to build a sustainable IoT infrastructure.

6 CONCLUSION

Huge amounts of data are generated constantly at the IoT networks. Cloud services are deployed to the edges of networks for data processing. In present, edge services may not always be efficient for data processing generated at the local IoT network. Low power IoT devices sometimes may not able to communicate with servers due to network problems and hence the data will be lost forever, or a service execution will fail. To resolve this issue, at least some parts of services need to move from edge to the local IoT cluster. In particular case, moving services from a network to another is not an easy process.

The primary target of this thesis was to implement a nanoEdge model that specified for local IoT service provisioning to minimize the deployment complexity. In this thesis, a PoC has been implemented for the earlier proposed model. The implementation has presented how local IoT services could can be deployed with Docker containerization technology with a demo project. This PoC shows how microservices for IoT devices can be moved from the cloud and edge ecosystem to the proximity to the local IoT networks and hence microservices can be virtualized for service provisioning. Since the model is deployed into a local IoT network, hence it can avoid many security threats and can work on “offline” without having an active internet connection. The overall performance shows the feasibility of the thesis work which is suitable for local IoT system and may contribute towards IoT infrastructure for future gadget-free 6G technology as well.

7 REFERENCES

[1] L. Atzori, A. Iera and G. Morabito, "Understanding the Internet of Things: definition, potentials, and societal role of a fast evolving paradigm", Ad Hoc Networks, vol. 56, pp. 122-140, 2017. Available: 10.1016/j.adhoc.2016.12.004.

[2] "What is internet of things (IoT)? - Definition from WhatIs.com", IoT Agenda, 2019. [Online]. Available: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT. [Accessed: 25- Feb- 2019].

[3] L. Vaquero, L. Rodero-Merino, J. Caceres and M. Lindner, "A break in the clouds", ACM SIGCOMM Computer Communication Review, vol. 39, no. 1, p. 50, 2008. Available: 10.1145/1496091.1496100.

[4] "Introduction to Cloud Computing", YouTube, 2019. [Online]. Available: https://www.youtube.com/watch?v=QYzJl0Zrc4M&list=PLF360ED1082F6F2A5&index=17. [Accessed: 26- Feb- 2019].

[5] M. Monshizadeh, V. Khatri and A. Gurtov, "NFV security considerations for cloud-based mobile virtual network operators", 2016 24th International Conference on Software, Telecommunications and Computer Networks (SoftCOM), 2016. Available: 10.1109/softcom.2016.7772161.

[6] I. Khan, F. Belqasmi, R. Glitho, N. Crespi, M. Morrow and P. Polakos, "Wireless sensor network virtualization: A survey", IEEE Communications Surveys & Tutorials, vol. 18, no. 1, pp. 553-576, 2016. Available: 10.1109/comst.2015.2412971.

[7] P. Lubomski, A. Kalinowski and H. Krawczyk, "Multi-level Virtualization and Its Impact on System Performance in Cloud Computing", Computer Networks, pp. 247-259, 2016. Available: 10.1007/978-3-319-39207-3_22.

[8] S. Choy, B. Wong, G. Simon, and C. Rosenberg, “The brewing storm in cloud gaming: A measurement study on cloud to end-user latency”, in Proceedings of the 11th Annual Workshop on Network and Systems Support for Games, ser. NetGames ’12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 2:1–2:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2501560.2501563.

[9] A. Sadeghi, C. Wachsmann and M. Waidner, "Security and privacy challenges in industrial internet of things", Proceedings of the 52nd Annual Design Automation Conference on - DAC '15, 2015. Available: 10.1145/2744769.2747942.

[10] "Office of Personnel Management data breach", En.wikipedia.org, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Office_of_Personnel_Management_data_breach. [Accessed: 26- Feb- 2019].

[11] P. Kasinathan, C. Pastrone, M. Spirito and M. Vinkovits, "Denial-of-Service detection in 6LoWPAN based Internet of Things", 2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), 2013. Available: 10.1109/wimob.2013.6673419.

[12] "When 'Smart Homes' Get Hacked: I Haunted A Complete Stranger's House Via The Internet", Forbes.com, 2019. [Online]. Available: http://www.forbes.com/sites/kashmirhill/2013/07/26/smart-homes-hack/. [Accessed: 26- Feb- 2019].

[13] T. Kumar, M. Liyanage, I. Ahmad, A. Braeken, and M. Ylianttila, “User privacy, identity and trust in 5G,” A Comprehensive Guide to 5G Security, pp. 267–279, 2018.

[14] T. Kumar, M. Liyanage, A. Braeken, I. Ahmad and M. Ylianttila, "From gadget to gadget-free hyperconnected world: Conceptual analysis of user privacy challenges", 2017 European Conference on Networks and Communications (EuCNC), 2017. Available: 10.1109/eucnc.2017.7980650.

[15] M. Liyanage, J. Salo, A. Braeken, T. Kumar, S. Seneviratne and M. Ylianttila, "5G Privacy: Scenarios and Solutions", 2018 IEEE 5G World Forum (5GWF), 2018. Available: 10.1109/5gwf.2018.8516981.

[16] V. Ramani, T. Kumar, A. Bracken, M. Liyanage and M. Ylianttila, "Secure and Efficient Data Accessibility in Blockchain Based Healthcare Systems", 2018 IEEE Global Communications Conference (GLOBECOM), 2018. Available: 10.1109/glocom.2018.8647221.

[17] "Mobile edge computing a key technology towards 5G", https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf, 2015, eTSI White Paper No. 11[Accessed 26 February 2019].

[18] P. Garcia Lopez et al., "Edge-centric Computing", ACM SIGCOMM Computer Communication Review, vol. 45, no. 5, pp. 37-42, 2015. Available: 10.1145/2831347.2831354.

[19] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, "Edge Computing: Vision and Challenges", IEEE Internet of Things Journal, vol. 3, no. 5, pp. 637-646, 2016. Available: 10.1109/jiot.2016.2579198.

[20] F. Bonomi, R. Milito, P. Natarajan and J. Zhu, "Fog Computing: A Platform for Internet of Things and Analytics", Big Data and Internet of Things: A Roadmap for Smart Environments, pp. 169-186, 2014. Available: 10.1007/978-3-319-05029-4_7 [Accessed 26 February 2019].

[21] F. Bonomi, R. Milito, J. Zhu and S. Addepalli, "Fog computing and its role in the internet of things", Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12, 2012. Available: 10.1145/2342509.2342513.

[22] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila and T. Taleb, "Survey on Multi-Access Edge Computing for Internet of Things Realization", IEEE Communications Surveys & Tutorials, vol. 20, no. 4, pp. 2961-2991, 2018. Available: 10.1109/comst.2018.2849509.

[23] E. Harjula and M. Ylianttila, "Decentralized IoT Edge Nanoservice Architecturefor Future Gadget-Free Computing".

[24] "Introduction to Virtualization", YouTube, 2019. [Online]. Available: https://www.youtube.com/watch?v=zLJbP6vBk2M&list=PL3EFBFBCE1249ABC0&index=11. [Accessed: 26- Feb- 2019].

[25] R. Morabito, "Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation", IEEE Access, vol. 5, pp. 8835-8850, 2017. Available: 10.1109/access.2017.2704444.

[26] "InfoClipz: Server virtualization", YouTube, 2019. [Online]. Available: https://www.youtube.com/watch?v=L_vRWJiOy40. [Accessed: 26- Feb- 2019].

[27] "Microservices Pattern: Monolithic Architecture pattern", microservices.io, 2019. [Online]. Available: https://microservices.io/patterns/monolithic.html. [Accessed: 26- Feb- 2019].

[28] B. Hübner, E. Walhorn and D. Dinkler, "A monolithic approach to fluid–structure interaction using space–time finite elements", Computer Methods in Applied Mechanics

and Engineering, vol. 193, no. 23-26, pp. 2087-2104, 2004. Available: 10.1016/j.cma.2004.01.024.

[29] A. Balalaie, A. Heydarnoori and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture", IEEE Software, vol. 33, no. 3, pp. 42-52, 2016. Available: 10.1109/ms.2016.64.

[30] "Microservices Architecture: What, When, and How - DZone Microservices", dzone.com, 2019. [Online]. Available: https://dzone.com/articles/microservices-architecture-what-when-how. [Accessed: 26- Feb- 2019].

[31] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence" in IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence. New York, NY, USA: ACM, 1973, pp. 235–245. [Online]. Available: https://dl.acm.org/citation.cfm?id=1624804.

[32] A. Haubenwaller and K. Vandikas, "Computations on the Edge in the Internet of Things", Procedia Computer Science, vol. 52, pp. 29-34, 2015. Available: 10.1016/j.procs.2015.05.011.

[33] "Docker Beginner Tutorial 1 - What is DOCKER (step by step) | Docker Introduction | Docker basics", YouTube, 2019. [Online]. Available: https://www.youtube.com/watch?v=wi-MGFhrad0&list=PLhW3qG5bs-L99pQsZ74f-LC-tOEsBp2rK. [Accessed: 26- Feb- 2019].

[34] R. Morabito, V. Cozzolino, A. Ding, N. Beijar and J. Ott, "Consolidate IoT Edge Computing with Lightweight Virtualization", IEEE Network, vol. 32, no. 1, pp. 102-111, 2018. Available: 10.1109/mnet.2018.1700175.

[35] "What Is Docker & Docker Container | A Deep Dive Into Docker | Edureka", Edureka, 2019. [Online]. Available: https://www.edureka.co/blog/what-is-docker-container. [Accessed: 26- Feb- 2019].

[36] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes", IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, 2014. Available: 10.1109/mcc.2014.51.

[37] Mouat, A. and Bednark, R. (2019). What is the difference between a Docker image and a container?. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/23735149/what-is-the-difference-between-a-docker-image-and-a-container/26885610#26885610 [Accessed: 26- Feb- 2019].

[38] "Get Started, Part 4: Swarms", Docker Documentation, 2019. [Online]. Available: https://docs.docker.com/get-started/part4/. [Accessed: 26- Feb- 2019].

[39] Y. Chen and T. Kunz, "Performance evaluation of IoT protocols under a constrained wireless access network", 2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT), 2016. Available: 10.1109/mownet.2016.7496622.

[40] "An overview of HTTP", MDN Web Docs, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. [Accessed: 26- Feb- 2019].

[41] B. Ramachandran, "Internet of Things – Unraveling technology demands & developments", Anything Connected, 2019. [Online]. Available: https://connectedtechnbiz.wordpress.com/2014/10/17/iot-unraveling-technology-demands-developments/. [Accessed: 26- Feb- 2019].

[42] R. Sola, "CoAP: Get started with IoT protocols - Open Source For You", Open Source For You, 2019. [Online]. Available: https://opensourceforu.com/2016/09/coap-get-started-with-iot-protocols/. [Accessed: 26- Feb- 2019].

[43] Z. Shelby, K. Hartke and C. Bormann, "The Constrained Application Protocol (CoAP)", 2014. Available: 10.17487/rfc7252.

[44] "CoAP — Constrained Application Protocol | Implementations", Coap.technology, 2019. [Online]. Available: http://coap.technology/impls.html. [Accessed: 26- Feb- 2019].

[45] T. Kumar, P. Porambage, I. Ahmad, M. Liyanage, E. Harjula and M. Ylianttila, "Securing Gadget-Free Digital Services", Computer, vol. 51, no. 11, pp. 66-77, 2018. Available: 10.1109/mc.2018.2876017.

[46] T. Leppänen, J. Riekki, M. Liu, E. Harjula and T. Ojala, "Mobile Agents-Based Smart Objects for the Internet of Things", Internet of Things, pp. 29-48, 2014. Available: 10.1007/978-3-319-00491-4_2.

[47] T. Leppanen, A. Heikkinen, A. Karhu, E. Harjula, J. Riekki and T. Koskela, "Augmented Reality Web Applications with Mobile Agents in the Internet of Things", 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies, 2014. Available: 10.1109/ngmast.2014.24.

[48] T. Leppanen, I. Milara, Jilin Yang, J. Kataja and J. Riekki, "Enabling user-centered interactions in the Internet of Things", 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2016. Available: 10.1109/smc.2016.7844457.

[49] G. Tanganelli, C. Vallati and E. Mingozzi, "CoAPthon: Easy development of CoAP-based IoT applications with Python", 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), 2015. Available: 10.1109/wf-iot.2015.7389028.

[50] M. Wasilak, "mwasilak/txThings", GitHub, 2019. [Online]. Available: https://github.com/mwasilak/txThings. [Accessed: 26- Feb- 2019].

[51] "Avnet: Quality Electronic Components & Services", Avnet.com, 2019. [Online]. Available: https://www.avnet.com/wps/portal/abacus/resources/engineers-insight/article/adapting-pir-sensor-technology-to-new-applications/. [Accessed: 26- Feb- 2019].

[52] "8. Command-line Tools — Gpiozero 1.5.0 Documentation", Gpiozero.readthedocs.io, 2019. [Online]. Available: https://gpiozero.readthedocs.io/en/stable/cli_tools.html. [Accessed: 26- Feb- 2019].

[53] "Raspberry Pi GPIO Pinout", Pinout.xyz, 2019. [Online]. Available: https://pinout.xyz/. [Accessed: 26- Feb- 2019].

[54] "17.1. subprocess — Subprocess management — Python 2.7.16rc1 documentation", Docs.python.org, 2019. [Online]. Available: https://docs.python.org/2/library/subprocess.html. [Accessed: 26- Feb- 2019].

[55] "IanHarvey/bluepy", GitHub, 2019. [Online]. Available: https://github.com/IanHarvey/bluepy/blob/master/docs/scanner.rst. [Accessed: 26- Feb- 2019].

[56] "Python Classes", W3schools.com, 2019. [Online]. Available: https://www.w3schools.com/python/python_classes.asp. [Accessed: 26- Feb- 2019].

[57] "Interfacing hardware with the Raspberry Pi", Rs-online.com, 2019. [Online]. Available: https://www.rs-online.com/designspark/interfacing-hardware-with-the-raspberry-pi. [Accessed: 26- Feb- 2019].

[58] "Raspberry Pi GPIO Tutorial", Pi My Life Up, 2019. [Online]. Available: https://pimylifeup.com/raspberry-pi-gpio/. [Accessed: 26- Feb- 2019].

[59] "draft-natarajan-nfvrg-containers-for-nfv-03 - An Analysis of Lightweight Virtualization Technologies for NFV", Tools.ietf.org, 2019. [Online]. Available:

https://tools.ietf.org/html/draft-natarajan-nfvrg-containers-for-nfv-03. [Accessed: 28- Mar- 2019].

[60] L. Vaquero and L. Rodero-Merino, "Finding your Way in the Fog", ACM SIGCOMM Computer Communication Review, vol. 44, no. 5, pp. 27-32, 2014. Available: 10.1145/2677046.2677052.

[61] A. Padmanabhan, "Container Orchestration", Devopedia.org, 2019. [Online]. Available: https://devopedia.org/container-orchestration. [Accessed: 28- Mar- 2019].

[62] M. Otey, "Microsoft Taps Google's Kubernetes for Windows Container Orchestration -- Redmondmag.com", Redmondmag, 2019. [Online]. Available: https://redmondmag.com/articles/2017/08/01/container-orchestration-with-kubernetes.aspx. [Accessed: 28- Mar- 2019].

[63] "3.2: Architecture: Clients and Broker", Embedded101, 2019. [Online]. Available: http://www.embedded101.com/Develop-M2M-IoT-Devices-Ebook/DevelopM2MIoTDevicesContent/articleid/219?dnnprintmode=true&mid=948&SkinSrc=[G]Skins%2F_default%2FNo+Skin&ContainerSrc=[G]Containers%2F_default%2FNo+Container. [Accessed: 28- Mar- 2019].

[64] "Running Bash commands in Python", Stack Overflow, 2019. [Online]. Available: https://stackoverflow.com/questions/4256107/running-bash-commands-in-python/51950538#51950538. [Accessed: 29- Mar- 2019].

[65] M. Hasan, E. Hossain and D. Niyato, "Random access for machine-to-machine communication in LTE-advanced networks: issues and approaches", IEEE Communications Magazine, vol. 51, no. 6, pp. 86-93, 2013. Available: 10.1109/mcom.2013.6525600.

[66] J. Lumetta, "Monolith vs microservices: which architecture is right for your team?", freeCodeCamp.org, 2019. [Online]. Available: https://medium.freecodecamp.org/monolith-vs-microservices-which-architecture-is-right-for-your-team-bb840319d531. [Accessed: 30- Apr- 2019].

[67] P. Hyde, Java thread programming. Indianapolis, Ind.: Sams, 1999.

[68] "Networking overview", Docker Documentation, 2019. [Online]. Available: https://docs.docker.com/network/. [Accessed: 30- Apr- 2019].

[69] "Overview of Internet of Things | Solutions | Google Cloud", Google Cloud, 2019. [Online]. Available: https://cloud.google.com/solutions/iot-overview. [Accessed: 11- May- 2019].

[70] R. Tabish, A. Ben Mnaouer, F. Touati and A. Ghaleb, "A comparative analysis of BLE and 6LoWPAN for U-HealthCare applications", 2013 7th IEEE GCC Conference and Exhibition (GCC), 2013. Available: 10.1109/ieeegcc.2013.6705791.

[71] S. Gupta and R. Kumar, "BLE v4.2: Creating Faster, More Secure, Power-Efficient Designs—Part 1", Electronic Design, 2019. [Online]. Available: https://www.electronicdesign.com/communications/ble-v42-creating-faster-more-secure-power-efficient-designs-part-1. [Accessed: 11- May- 2019].

[72] F. Callegati, W. Cerroni and M. Ramilli, "Man-in-the-Middle Attack to the HTTPS Protocol", IEEE Security & Privacy Magazine, vol. 7, no. 1, pp. 78-81, 2009. Available: 10.1109/msp.2009.12.

[73] "Understanding HTTPS Protocol", Codeproject.com, 2019. [Online]. Available: https://www.codeproject.com/Articles/995655/Understanding-HTTPS-Protocol. [Accessed: 12- May- 2019].

[74] I. Fette and A. Melnikov, "The WebSocket Protocol", 2011. Available: 10.17487/rfc6455.

[75] P. Walkar, "What's the difference between edge computing and cloud computing?", quora, 2019. [Online]. Available: https://www.quora.com/Whats-the-difference-between-edge-computing-and-cloud-computing. [Accessed: 12- May- 2019].

[76] Linthicum, D. (2019). Edge computing vs. fog computing: Definitions and enterprise uses. [online] Cisco. Available at: https://www.cisco.com/c/en/us/solutions/enterprise-networks/edge-computing.html [Accessed 19 May 2019].

[77] Os.mbed.com. (2019). Thread - Reference | Mbed OS 5 Documentation. [online] Available at: https://os.mbed.com/docs/mbed-os/v5.12/reference/thread-tech.html [Accessed 26 May 2019].

[78] MDN Web Docs. (2019). Evolution of HTTP. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP [Accessed 26 May 2019].

[79] Stamat, D. (2019). Microcontainers – Tiny, Portable Docker Containers. [online] blog.iron.io. Available at: https://blog.iron.io/microcontainers-tiny-portable-containers/ [Accessed 27 May 2019].

[80] Nigania, J. (2019). Know The New Cloud Called Edge In 2K19. [online] Houseofbots.com. Available at: https://www.houseofbots.com/news-detail/11402-1-know-the-new-cloud-called-edge-in-2k19 [Accessed 27 May 2019].

[81] Javainuse.com. (2019). [online] Available at: https://www.javainuse.com/spring/ang7-hello [Accessed 27 May 2019].

[82] Learn.adafruit.com. (2019). Bluetooth & BTLE | All the Internet of Things - Episode One: Transports | Adafruit Learning System. [online] Available at: https://learn.adafruit.com/alltheiot-transports/bluetooth-btle [Accessed 27 May 2019].

[83] Janetakis, N. (2019). Alpine Based Docker Images Make a Difference in Real World Apps. [online] via @codeship. Available at: https://blog.codeship.com/alpine-based-docker-images-make-difference-real-world-apps/ [Accessed 27 May 2019].

8 APPENDICES

Appendix 1 Database structure for API service

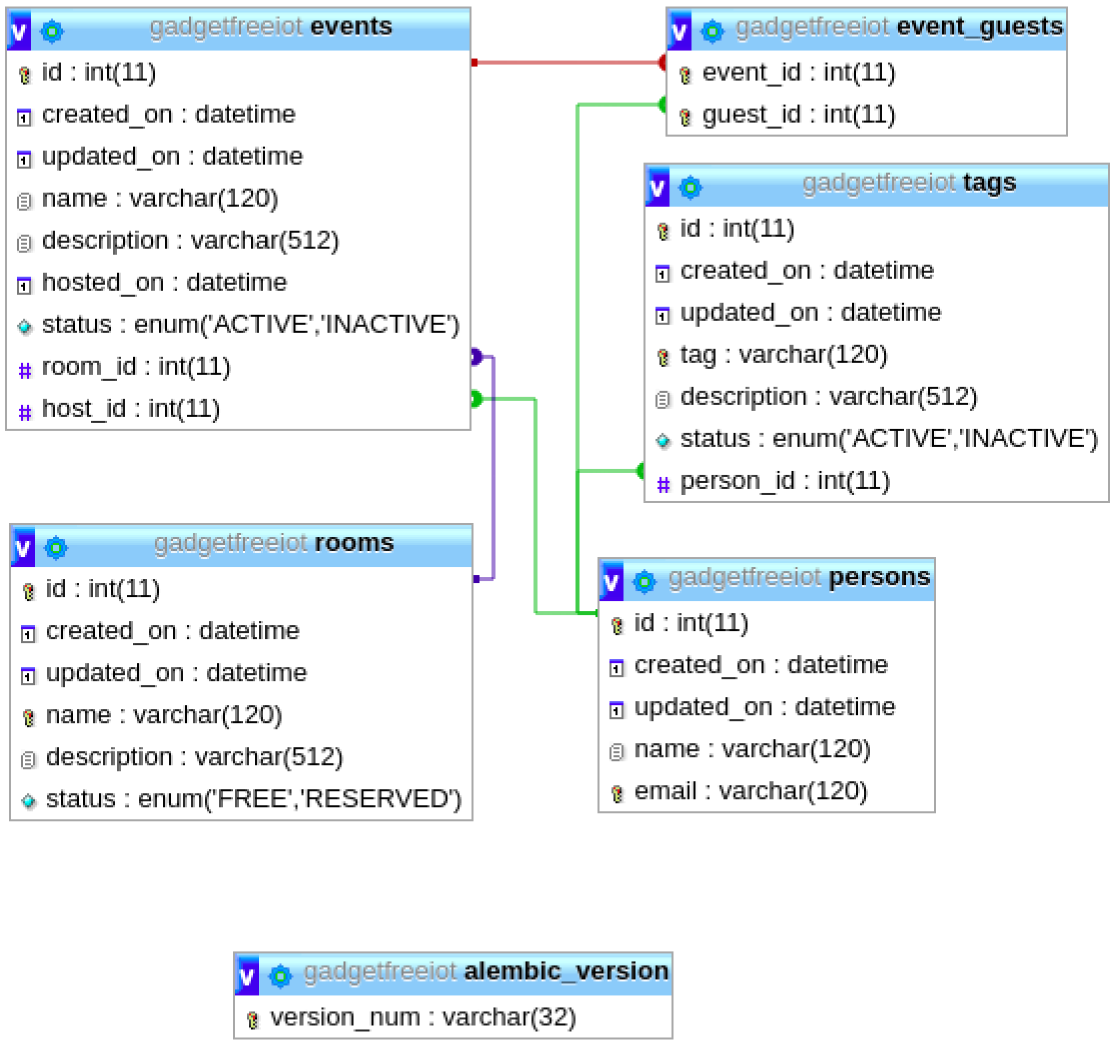
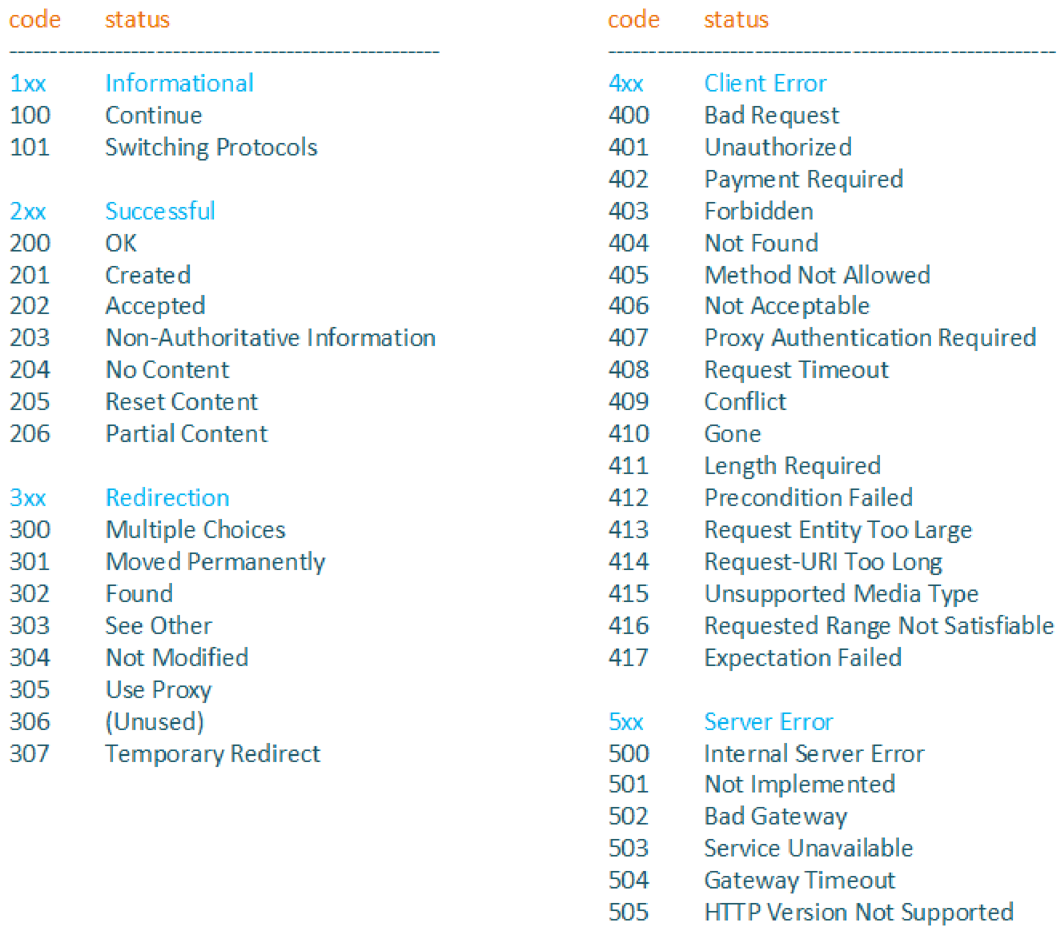


Figure 56. Database structure for API used in the demo application.

Appendix 2 HTTP status codes

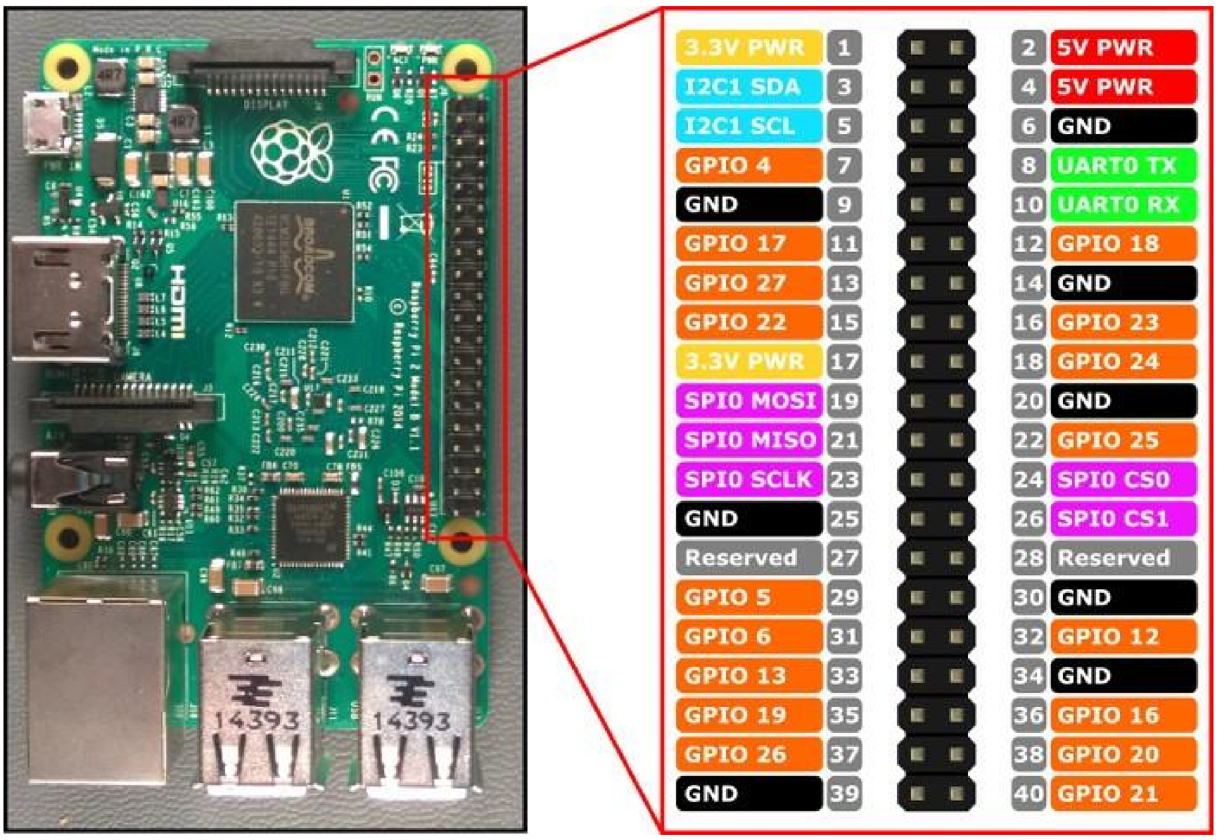


code	status
1xx	Informational
100	Continue
101	Switching Protocols
2xx	Successful
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
3xx	Redirection
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
306	(Unused)
307	Temporary Redirect

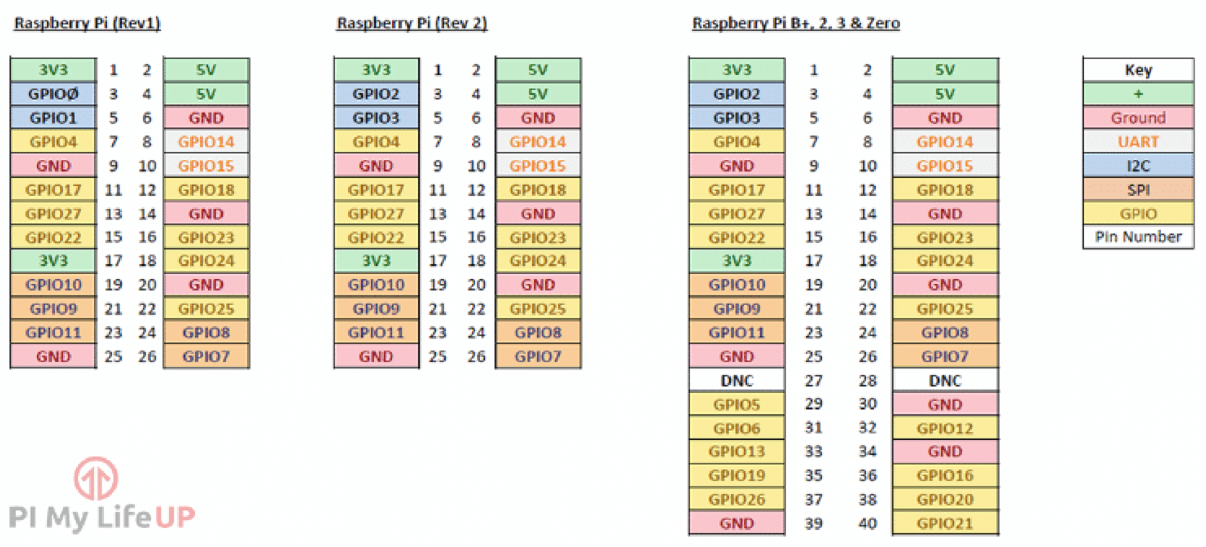
code	status
4xx	Client Error
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Request Entity Too Large
414	Request-URI Too Long
415	Unsupported Media Type
416	Requested Range Not Satisfiable
417	Expectation Failed
5xx	Server Error
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported

Figure 57. HTTP status codes for client response.

Appendix 3 RPi pin configuration



(a)



Raspberry Pi (Rev1)

3V3	1	2	5V
GPIOØ	3	4	5V
GPIO1	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3V3	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7

Raspberry Pi (Rev 2)

3V3	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3V3	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7

Raspberry Pi B+, 2, 3 & Zero

3V3	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3V3	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7
DNC	27	28	DNC
GPIO5	29	30	GND
GPIO6	31	32	GPIO12
GPIO13	33	34	GND
GPIO19	35	36	GPIO16
GPIO26	37	38	GPIO20
GND	39	40	GPIO21

Key
+
Ground
UART
I2C
SPI
GPIO
Pin Number

(b)

Figure 58. RPi pin configuration – (a) RPi 3 Model B [57] & (b) RPi, RPi 2 & RPi 3 [58].