IoT Device Management and Configuration

A Thesis Submitted to the College of

Graduate and Postdoctoral Studies

In Partial Fulfillment of the Requirements

For the Degree of Master of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

YUNXIAO WANG

PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other uses of materials in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

ABSTRACT

As the number of IoT devices grows, the management and configuration of IoT devices becomes crucial in resource constraint networks. It is hard to manage and configure a large amount of heterogeneous resource constraint IoT devices because people need to know how they connect to each other, what internet-enabled services are available to provide, and how people interact with things through the internet.

The thing-centric approach focuses on user experience when engaging things, but the cloud-centric approach switch the focus to IoT services that can process data streams collected from things and applications that help get people joined in the IoT world. To manage IoT populations effectively in a centralized manner, not only does it mean that moving computational power closer to the edge is a way to reduce bandwidth and latency, but it also implies that it is necessary to build an architecture which can scale and manage tons of connected devices by a uniform interface. In particular, RESTful Web services can provide a uniform interface that operates resources by HTTP methods. For example, users can read and write data by a uniform interface, and a flowerpot can write data and be triggered to water plants by a uniform interface. Thus, in the scope of IoT, embedded middleware can implement uniform interface by REST model.

Virtualizing physical things has emerged as a design pattern to build IoT systems. Resource less constraint devices are capable of being virtualized with enough CPU power, memory, networking, but they are more expensive and power consuming. However, resource highly constraint devices take advantage of low energy consumption and cheaper price, but they cannot be virtualized because they do not have ability to even run a single multi-threaded program. Therefore, it is very important to select the right platforms for the right roles. In our case, we use Raspberry Pi 3 as a middleware and Nordic nRF52832 as a BLE endpoint.

In this thesis, a REST-based IoT management system based on Service-Oriented Architecture is built, and the performance of the system has been tested, including the response time of HTTP GET and POST requests of the centralized server in a Fog domain and a script engine onto a BLE-enabled endpoint.

ACKNOWLEDGEMENTS

Hereby, I would like to thank for my family to support my study towards the Master of Science degree in the University of Saskatchewan.

Also, I would like to thank for my supervisor Professor Ralph Deters. Under his supervision, not only did I have a chance of academic review, but I also learnt lots of cutting-edge concepts in the field of Cyber Network and extended practical skills in programming. He is more like a life advisor than a knowledge giver.

Last but not least, I would like to appreciate any help from my MADMUC lab members who shared their experience and the view of problem solving. With the help of the above, I enjoyed my study through this unforgettable period in my life.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADEPT	Autonomous Decentralized Peer-to-Peer Telemetry
BLE	Bluetooth Low Energy
CoAP	Constraint Application Protocol
CREST	Computational Representational State Transfer
CRUD	Create Read Update Delete
GATT	Generic Attribute
HTTP	Hyper Text Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
NFC	Near Field Communication
PAN	Personal Area Network
PoC	Proof of Concept
REST	Representational State Transfer
RTOS	Real Time Operation System
SDN	Software Defined Network
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SoC	System on a Chip
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Universal Resource Identifier
URL	Universal Resource Locator

XML	eXtensible Markup Language
6LoWPAN	IPv6 over Low-Power Wireless Personal Area Network

CHAPTER 1

INTRODUCTION

Rykowski and Wilusz [36] state that “as IoT is very dynamic and heterogeneous, efficient management system for IoT environment should address these features”. Gubbi et al. [32] identify Object-centric architecture and Internet-centric architecture in the vision of IoT and “see Cloud-centric architecture to be the best where cost based services are required”. Kantarci and Mouftah [10] believe that “Cloud-centric IoT can leverage the efficiency of several applications including but not limited to pervasive healthcare, future transportation systems, smart city, and public safety which is a featured application area of the smart city”. However, convergence of IoT and Cloud can foresee new challenges. Biswas and Giaffreda [6] announce that “IoT-Cloud has to provide real-time data processing and service provisioning techniques considering such Big Data. Another issues are to provide more dynamic resources management and orchestration techniques, dynamically offloading from clients / hosts to cloud”.

IoT can be seen as a collection of various protocols and Web services [63]. Ashraf et al. [47] further explore that seamless communication can be achieved by middleware interfaces serving a batch of connected IoT edge devices who are able to run customized applications, so that the status on edge devices can be dynamically controlled. Chowdhury et al. [53] specify that “the data retrieved from the device can be applied to several processes in the network, such as aggregation and abstraction, and the result of these procedures is transmitted to other entities”. Finally, users are able to read processed data, write command to devices, and change code onto devices through RESTful Web services. Meanwhile, HTTP is a Web application protocol that is perfect to work with REST model. In resource constraint environment, REST model can provide a Web-like uniform interface for devices and users, and it effectively shifts the computational power from Cloud to Fog closer to IoT endpoints with the same user experience.

In this thesis, an architecture based on Service-Oriented Architecture to manage and configure IoT devices will be proposed. A Raspberry Pi 3 will be used as Middleware to provide REST Web services and as BLE master to establish BLE connection. A Nordic nRF52832 will be used as a BLE peripheral that provides BLE services to BLE masters. Later, the experiments will evaluate the performance of REST model on a Raspberry Pi 3 and the performance of script engine on Nordic nRF52832.

1.1 Problem Definition

The goal of this research is to propose an architecture that can enable IoT device management. There is a case that lots of flowerpots need to be watered every certain period in a greenhouse. If there is a system that can tell the soil moisture of each flowerpot, what software version is running on it, and can upload new versions at run-time through applications, that would be a great help to water flowers in a timely manner, and that is able to track all behaviors on each flowerpot. Not only can it reduce labor work, but it also boosts the ability of managing and configuring edge things.

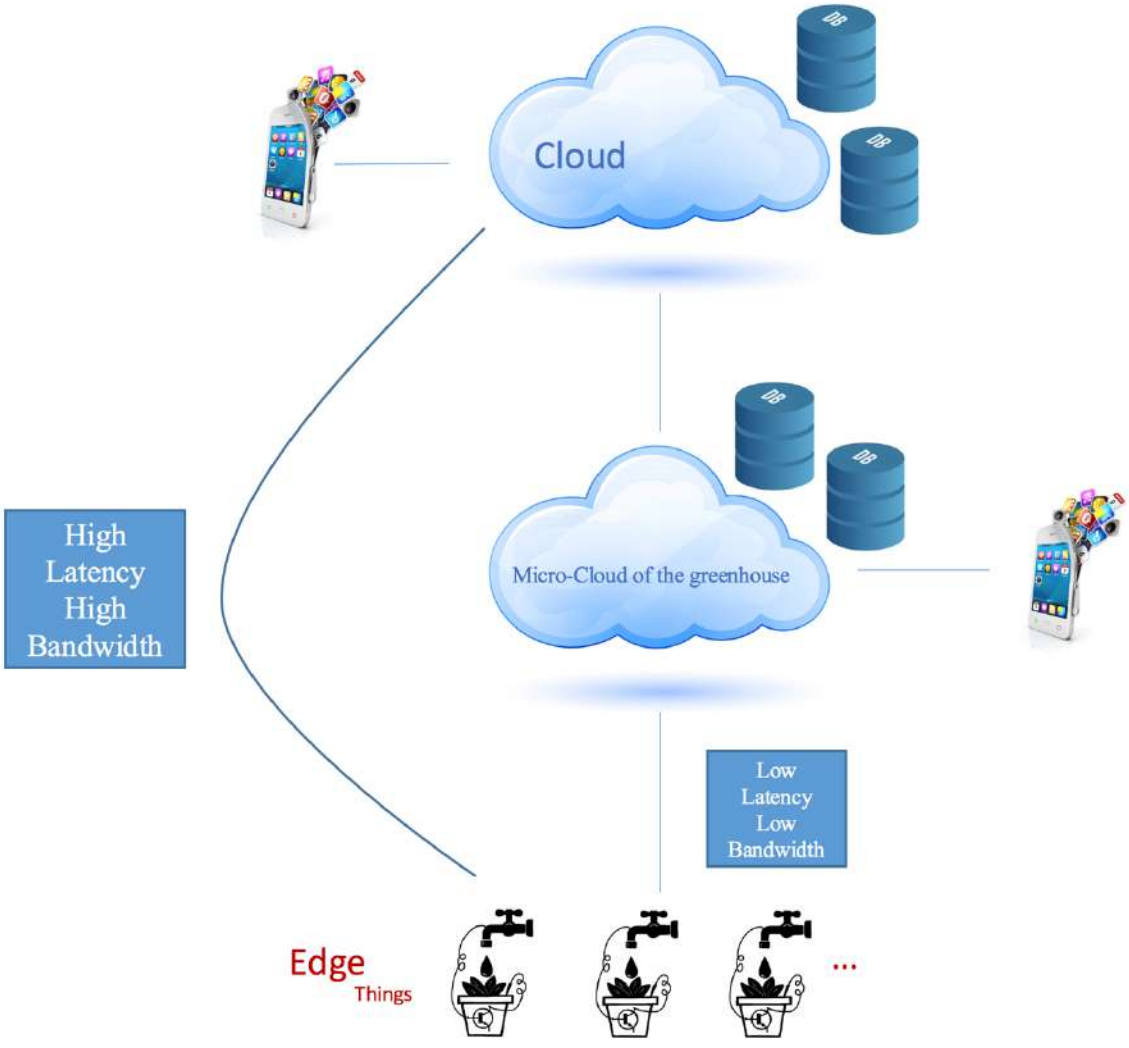

Figure 1-1. Computational power level comparison

In the scope of IoT, figure 1-1 shows that Cloud computing can cause high latency and high bandwidth, whereas Fog (i.e. Micro-Cloud) computing can optimize those important features. However, rather than using traditionally centralized computation (e.g. Cloud), effectively managing IoT devices in Fog domain, as Cherrier et al. [54] identify control flow and different settings for actuators, becomes key challenges as the number of devices increases. Thus, my problems break down to the followings:

1. How to provide uniform web-like interface?
2. How can we use this interface to send command to things?
3. How can we change command / code to run new functionality?

1.1.1 How to provide uniform web-like interface?

It is easy to imagine that there will be tons of IoT devices working at the edge of the network in the near future. However, it is not easy to manage them. For example, users may want to know the latest status of some devices and the configurations those devices are running on. According to Farahzadi et al. [2], "Middleware is a solution for implementing different services in a heterogeneous environment". Therefore, in the context of IoT, a concept of virtualization management system can help. A direct way is to develop a centralized system embedded onto a physical one with a uniform interface supplied to heterogeneous IoT endpoints and users, and Web is the most scalable thing we have.

1.1.2 How can we use this interface to send commands to things?

It is easy to send a complete program to an edge device through an IDE. For example, Kolias et al. [21] relies on a single-board computer programmed in C / C++ to execute sensing and actuating. Typically, an Arduino sketch programmed in C / C++ is compiled by built-in gcc compiler in Arduino IDE, then the hex code is uploaded to an Arduino device by wire or wireless. Therefore, we need a way to send commands that it can support real-time wireless status change and deliver better user experience instead of using an IDE.

However, what if a user just wants to send a simple command to blink an LED at run-time in a low energy device through a pervasive Web-like interface? Kohler et al. [39] have realized that an agent which can be implemented on a physical product can manage the connection to the Web

platform via RESTful or SOAP Web services. Therefore, a script engine mounted onto a physical one is supposed to work in PAN-based low energy environment which is in the coverage of Web services. In particular, sending a command at run-time via Web GUI to change an actuator's status on a BLE-enabled device with a script engine installed for command execution is one of the ways to do so. In spite of 20 bytes of single BLE packet size limit, it is possible to break up a command or a program into up to 20 bytes per packet.

1.1.3 How can we change code to run new functionality?

A mobile app can help directly send a single command in PAN, but it is complicated for users to type code, and also it requires users to be in the range of PAN. However, it is easy for users to type code on a Web GUI and send through a Web application to BLE endpoints wherever they are. Then, the corresponding Web services able to change code are needed. In particular, given the code written in JavaScript, pushing the JavaScript code through Web interface and executing the code by script engine to have new functionality run on the expected device.

1.1.4 Research Goal

The goal of this research is to propose an architecture that includes a middleware embedded onto a resource-rich physical device and a script engine onto a resource highly constraint device, in order to make IoT device management and configuration more functional, accessible, flexible and scalable. To achieve the goal, we will do the followings:

- To provide uniform web-like interface.
- To use this interface to send commands to things.
- To change command / code to run new functionality.

CHAPTER 2

LITERATURE REVIEW

In this Chapter, there are the following areas reviewed. In the first section, we started with the concept of Internet of Things (IoT). In the second section, we compared some typical resource constraint devices to demonstrate what the roles of these hardware is playing in IoT world. In the third section, a REST-based IoT model was introduced. In the fourth section, we introduced underlying protocols, such as HTTP and CoAP, TCP and UDP, and BLE to see where they are a good fit in connectivity. In the fifth section, we proposed solutions to the problems.

2.1 IoT

The term of IoT is coined by Kevin Ashton [37] in 1999, but it was used in supply chain management. As figure 2-1 shows, IoT describes a system where items in the physical world, and sensors within or attached to these items, are connected to the Internet via wireless and wired Internet connections [20]. Basically, there are three things that the Internet of Things will [20]:

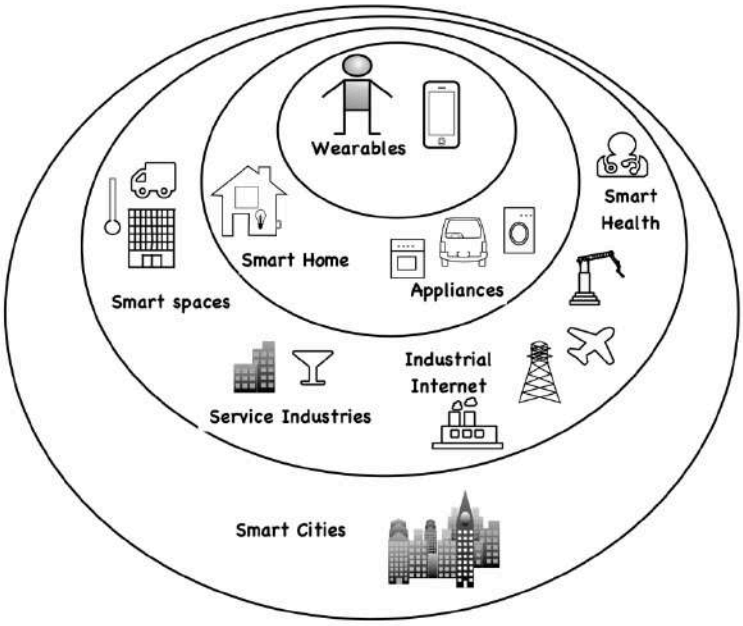

Figure 2-1. The deployment map of IoT [58]

- Connect both inanimate and living things. IoT is a relatively new concept since Industrial 4.0 was presented, but the things (objects) in IoT were widely deployed in industrial

equipment for many years. Today, the things are ubiquitous. It appears and ranges from smart cities to anything micro.

- Use sensors for data collection. RFID and sensor technology enable computers to observe, identify and understand the world—without the limitations of human-entered data [1]. The physical objects are just like human beings who need to have some senses to tell what they feel like. Generally, sensors are functionally designed for different scenarios, such as temperature sensors, humidity sensors, fire alarm sensors, push buttons, buzzers, rotary encoders, etc. Then, the objects hooked up with sensors connect to each other and/or to systems so that desired data can be collected.
- Change what types of item communicate over IP Network. Every physical object cannot be wirelessly accessed unless they are assigned unique identities, and IPv4 and IPv6 are such identities. IPv4 is the most widely deployed Internet protocol used to connect devices to the Internet, whereas IPv6 that provides 128-bit space is the successor to IPv4. With this digital identity, every object can be tracked then.

Once the above three are ready, a centralized interface is required to process the data collected, and this kind of interface could be in large scale or even as small as in the object itself. In the rest of this chapter, we will review the heterogeneity of objects, the necessity of common interface and the connectivity between objects and interface.

Considering heterogeneous IoT devices operating in various solutions, Morabito [51] realizes that the virtualization of physical devices is another important challenge. Such an idea that the capabilities of physical devices could be enhanced by connecting them to remote software components who monitor the status of physical things.

Kim et al. [33] highlight that “The devices in IoT often compose Personal Area Networks (PANs)”, and “it is possible to track locations and states of IoT devices”. Kranz et al. [40] point out that “middleware can be used with embedded interaction to help integrate physical interaction, communication, and data exchange, enabling a holistic approach toward interaction with the Internet of Things”.

2.2 Hardware Platforms

Gartner, Inc. forecasts [22] that up to 8.4 billion of connected things will be in use worldwide in 2017, 31 percent higher than 2016, and will reach to around 20.4 billion by 2020 (see Table 2-1). Total spending on endpoints and services will reach to around $2 trillion in 2018 (see Table 2-2). Obviously, connected IoT devices will be wherever you are, and the economic profit of that will be such huge amount.

Table 2-1. IoT Units Installed Base by Category (Millions of Units) [22]

Category	2016	2017	2018	2020
Consumer	3,963.0	5,244.3	7,036.3	12,863.0
Business: Cross-Industry	1,102.1	1,501.0	2,132.6	4,381.4
Business: Vertical-Specific	1,316.6	1,635.4	2,027.7	3,171.0
Grand Total	**6,381.8**	**8,380.6**	**11,196.6**	**20,415.4**

Table 2-2. IoT Endpoint Spending by Category (Billions of Dollars) [22]

Category	2016	2017	2018	2020
Consumer	532,515	725,696	985,348	1,494,466
Business: Cross-Industry	212,069	280,059	372,989	567,659
Business: Vertical-Specific	634,921	683,817	736,543	863,662
Grand Total	**1,379,505**	**1,689,572**	**2,094,881**	**2,925,787**

IoT devices are also called edge devices which are located at the edge of the network, and they are the necessity of Edge (Fog) Computing that could work independently as a tiny Cloud or work with Cloud service. Choosing proper edge devices is the entry of IoT, and we have to know where each device should be placed.

2.2.1 The Concept of SoC

First of all, let's simply clear up the concept of microprocessor and microcontroller. A microprocessor is a CPU (Central Processing Unit) that is compacted into a chip semiconductor device, whereas a microcontroller (or MCU for microcontroller unit) typically includes small amounts of RAM and ROM, and I/O ports and timers [56]. Microprocessors are used to execute big and generic applications, while a microcontroller will only be used to execute a single task within one application. Some of the benefits of microcontrollers include the following [50]: (1) lower cost due to the relationship between input and output is defined to perform specific tasks; (2) less power consumption due to microcontrollers are generally built using a technology known

as Complementary Metal Oxide Semiconductor (CMOS). This technology is a competent fabrication system that uses less power and is more immune to power spikes than other techniques; (3) All-in-one due to a microcontroller usually comprises of a CPU, ROM, RAM and I/O ports, built within it to execute a single and dedicated task, while a microprocessor needs a lot of peripherals to match. Today different manufacturers produce microcontrollers with a wide range of features available in different versions. Some manufacturers are ATMEL, Microchip, TI, Freescale, Philips, Motorola, etc [50].

There are tons of microcontrollers available in the market, and they are typically categorized by single chips and SoC (System on the Chip) based on capabilities. In our early research, single chips such Arduino Uno equipped with ATmega328P was used. The ATmega328P is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture [7].

However, if there is a system running on the chip, there would be more possibilities coming up. A system-on-a-chip, or SoC, is a computer system that all of whose components are integrated onto a single chip [18]. The components of an SoC are called intellectual property blocks, or IPs in brief. As figure 2-2 shows, IPs of an SoC include one or more processors (called cores) and several peripheral devices [52].

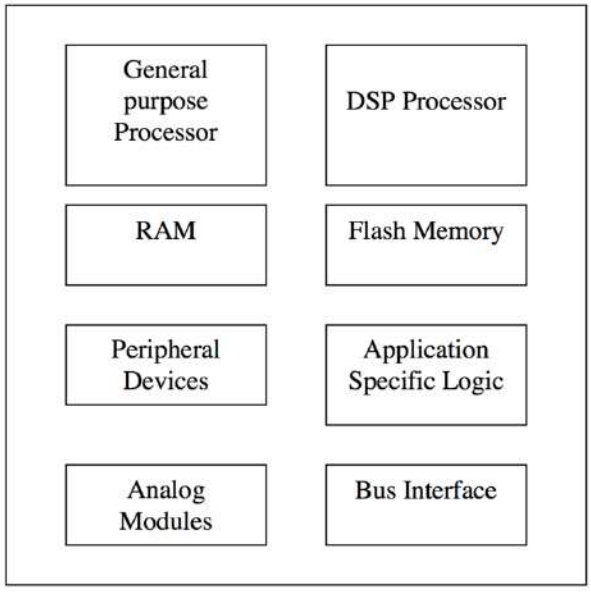

Figure 2-2. Typical components of SoC [15]

2.2.2 Introduction of SoCs

The SoCs involved in our research are Arduino Yun, Raspberry Pi 3, Wemos D1, Arduino 101, BotSpine (TI CC2541), Nordic (nRF52832):

(1) Arduino Yun has one single microcontroller named ATmega32u4 plus one microprocessor named Atheros AR9331 with peripherals externally connected. The Atheros processor supports a Linux distribution based on OpenWrt (a version dedicated for embedded system) named Linino OS [4]. AR9331 chip is widely used in routers, such as TP-Link TL-WR703N [4].

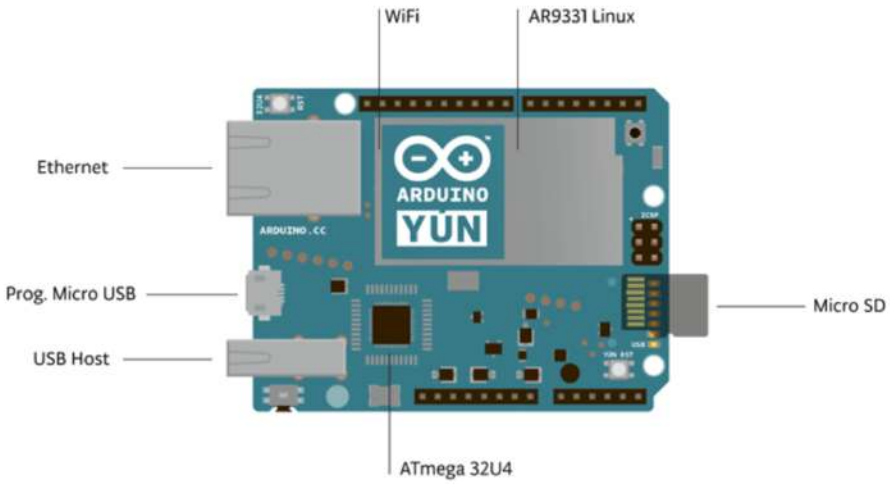

Figure 2-3. Arduino Yun layout [4]

Microcontroller	ATmega32U4
Operating Voltage	5V
Input Voltage	5 V
Digital I/O Pins	20
PWM Output	7
Analog I/O Pins	12
DC Current per I/O Pin	40 mA on I/O Pins; 50 mA on 3,3 Pin
Flash Memory	32 KB (of which 4 KB used by bootloader)
SRAM	2.5 KB
EEPROM	1 KB
Clock Speed	16 MHz

Figure 2-4. Arduino Yun AVR microcontroller specifications [4]

Processor	Atheros AR9331
Architecture	MIPS
Operating Voltage	3.3V
Ethernet	802.3 10/100Mbit/s
WiFi	802.11b/g/n 2.4 GHz
USB Type	2.0 Host
Card Reader	Micro-SD
RAM	64 MB DDR2
Flash Memory	16 MB
SRAM	2.5 KB
EEPROM	1 KB
Clock Speed	400 MHz

Figure 2-5. Arduino Yun microprocessor specifications [4]

Figure 2-3 shows the main components of Arduino YUN. Figure 2-4 shows the specifications of Arduino side, and figure 2-5 shows the specifications of Linux side.

There are a few features worth pointing out:

- The Linino OS installation occupies around 9 MB of the 16 MB available of the internal flash memory [4], and you are discouraged from using the Yún's built-in non-volatile memory, because it has a limited number of writes [4], so Arduino Yun can support microSD card extension if more disk space is needed.
- OpenWRT provides Advanced Configuration Panel (GUI) powered by Luci web interface, where allows users to manage any information on Arduino Yun, including network status, software installation and uninstallation, real-time graphs, CPU power consumption, etc.
- The communication between both two environments is enabled by the Bridge library that provides some classes for variant communication.

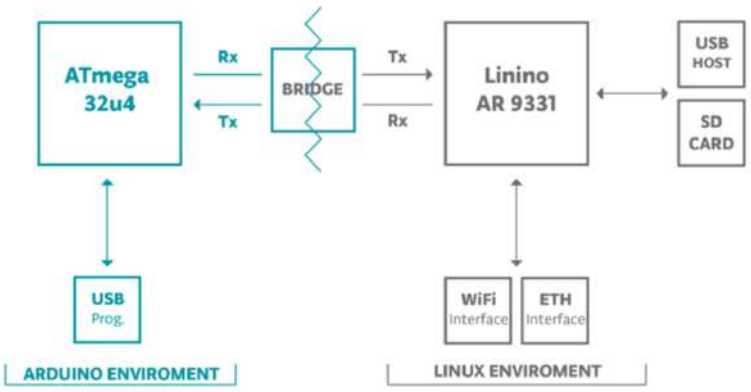

Figure 2-6. Arduino Yun Bridge [27]

Figure 2-6 shows the bridging connection between Arduino side and Linux side through UART series.

(2) Raspberry Pi 3 is the third generation Raspberry Pi. Raspberry Pi 3 is a 64-bit mini Linux computer that can do most tasks as the same as on a Linux desktop. Figure 2-7 shows the components of a Raspberry Pi 3, and figure 2-8 shows the specifications of a Raspberry Pi 3.

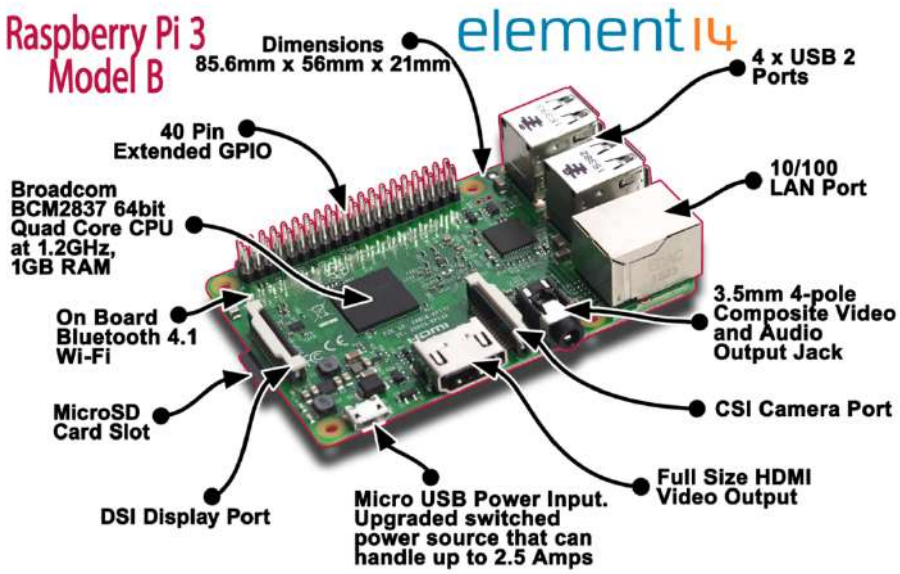

Figure 2-7. Raspberry Pi 3 layout [34]

Specifications

Processor	Broadcom BCM2387 chipset. 1.2GHz Quad-Core ARM Cortex-A53 802.11 b/g/n Wireless LAN and Bluetooth 4.1 (Bluetooth Classic and LE)
GPU	Dual Core VideoCore IV® Multimedia Co-Processor. Provides Open GL ES 2.0, hardware-accelerated OpenVG, and 1080p30 H.264 high-profile decode. Capable of 1Gpixel/s, 1.5Gtexel/s or 24GFLOPs with texture filtering and DMA infrastructure
Memory	1GB LPDDR2
Operating System	Boots from Micro SD card, running a version of the Linux operating system or Windows 10 IoT
Dimensions	85 x 56 x 17mm
Power	Micro USB socket 5V1, 2.5A

Connectors:

Ethernet	10/100 BaseT Ethernet socket
Video Output	HDMI (rev 1.3 & 1.4 Composite RCA (PAL and NTSC)
Audio Output	Audio Output 3.5mm jack, HDMI USB 4 x USB 2.0 Connector
GPIO Connector	40-pin 2.54 mm (100 mil) expansion header: 2x20 strip Providing 27 GPIO pins as well as +3.3 V, +5 V and GND supply lines
Camera Connector	15-pin MIPI Camera Serial Interface (CSI-2)
Display Connector	Display Serial Interface (DSI) 15 way flat flex cable connector with two data lanes and a clock lane
Memory Card Slot	Push/pull Micro SDIO

Key Benefits

- Low cost
- 10x faster processing
- Consistent board format
- Added connectivity

Key Applications

- Low cost PC/tablet/laptop
- Media centre
- Industrial/Home automation
- Print server
- Web camera
- Wireless access point
- Environmental sensing/monitoring (e.g. weather station)
- IoT applications
- Robotics
- Server/cloud server
- Security monitoring
- Gaming

Figure 2-8. Raspberry Pi 3 specifications [49]

Let's have a direct understanding of Raspberry Pi 3 performance by using sysbench software. However, it is no meaningful to test Arduino Yun for a comparison because of obviously distinct CPU architecture.

```
pi@raspberrypi:~ $ lscpu
Architecture:          armv7l
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
Model name:            ARMv7 Processor rev 4 (v7l)
CPU max MHz:           1200.0000
CPU min MHz:           600.0000
```

Figure 2-9. Raspberry Pi 3 CPU profile

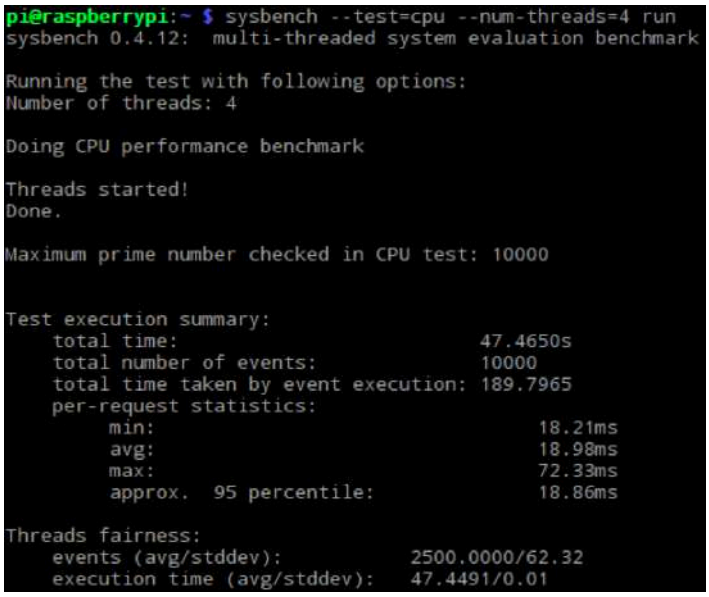

Figure 2-10. Raspberry Pi 3 CPU test at 4 threads

Figure 2-9 shows CPU profile of a Raspberry Pi 3, and figure 2-10 shows a CPU test of a Raspberry Pi 3 at 4 threads by sysbench.

(4) Wemos D1 is equipped with a wifi module on the 32-bit Esp8266 chip running RTOS named FreeRTOS utilized in RTOS SDK that features multi-tasking operations. Considering network connectivity, FreeRTOS includes network lwip API (sockets, IPv4, TCP/UDP, etc) and JSON library. It supports Arduino libraries, where "setup()" and "loop()" are called in "core_esp8266_main.cpp" by "#include <Arduino.h>", which makes Wemos become a wifi version Arduino Uno. As our later research indicates, some other RTOS such as Espruino can also be applied to Esp8266 chip. Figure 2-11 shows the specifications of an Esp8266-12E.

Categories	Items	Values
WiFi Paramters	WiFi Protocles	802.11 b/g/n
	Frequency Range	2.4GHz-2.5GHz (2400M-2483.5M)
Hardware Paramaters	Peripheral Bus	UART/HSPI/I2C/I2S/Ir Remote Contorl
		GPIO/PWM
	Operating Voltage	3.0~3.6V
	Operating Current	Average value: 80mA
	Operating Temperature Range	-40°~125°
	Ambient Temperature Range	Normal temperature
	Package Size	16mm*24mm*3mm
	External Interface	N/A
Software Parameters	Wi-Fi mode	station/softAP/SoftAP+station
	Security	WPA/WPA2
	Encryption	WEP/TKIP/AES
	Firmware Upgrade	UART Download / OTA (via network) / download and write firmware via host
	Ssoftware Development	Supports Cloud Server Development / SDK for custom firmware development
	Network Protocols	IPv4, TCP/UDP/HTTP/FTP
	User Configuration	AT Instruction Set, Cloud Server, Android/iOS App

Figure 2-11. Esp8266-12E parameters [24]

There are a few features worth pointing out:

- Esp8266 supports UART (CH340G USB-serial converter chip) -wifi pass-through that the remote server receives exactly what the serial input is, through wifi module. It is fast and easy to set up a peer-to-peer TCP/UDP communication through a common server in the internet. After connecting a nearby access point, use AT command "AT+CIFSR" to get IP address of Esp8266, then start a UDP connection by "AT+CIPSTART="UDP", "server IP address or DNS", port", followed by "AT+CIPMODE=1" to start pass-through mode, finally send message by "AT+CIPSEND" to the common server. If the connection is successful, then you can start pass-through to a receiver via the internet. NAT traversal is automatically executed in the router.

- Esp8266 supports light sleep mode that the CPU may be suspended in application like wifi switch. Without data transmission, the Wi-Fi Modem circuit can be turned off and CPU suspended to save power according to the 802.11 standard (U-APSD). The power <0.9mA. Modem sleep mode that requires CPU to be working. According to 802.11 standards (like U-APSD), it saves power to shut down the Wi-Fi Modem circuit while maintaining a Wi-Fi connection with no data transmission. The power <15mA. Deep sleep mode that does not require wifi connection to be maintained. The power <10uA [23].

(5) Arduino 101 is equipped with Intel Curie module chip, which features a dual-core architecture, includes a BLE module on the 32-bit Intel Quark SE core and a sensor subsystem (a 6-axis accelerometer and a gyroscope) on 32-bit RISC (reduced instruction set computer) ARC EM4 core with FPU (floating point unit). Intel Quark SE core runs IoT-oriented Zephyr RTOS hosted by the Linux foundation. Figure 2-12 shows the specifications of an Arduino 101.

FEATURE SUMMARY	
Microcontroller	32-bit Intel® Quark™ SE
Operation voltage	7 to 12 VDC
General purpose I/O pins	Twenty 3.3 V I/Os (with 5 V tolerance)
Serial I/Os	One UART, one SPI, one I²C
Analog input pins	Six 10-bit ENOBs
DC current per I/O pin	7 mA
Flash memory	384 KB (192 KB available for Sketch)
Onboard flash	2 MB
SRAM	80 KB (24 KB available for Sketch)
Clock speed	32 MHz
Form factor	Arduino Uno R3 compatible

Figure 2-12. Arduino 101 specifications [26]

(6) BotSpine, which was developed by a local company named EIC (Environmental Instruments Canada Inc.) located in Saskatoon, is equipped with TI CC2541 (8051 MCU) BLE-enabled SoC. The language it uses is Basic interpreted by an interpreter named BlueBasic. The convenience of using BotSpine is that programs can be directly flashed from a mobile app into the chip through BLE, and it provides easier way to prototype in Basic rather than C. However, there are some drawbacks that unfortunately it does not support FPU, BlueBasic only defines 26 variables from A to Z, and currently it can only be a BLE peripheral. However, the existence of those so-called drawbacks is to make it easier for developers. Figure 2-13 shows the profile of a BotSpine.

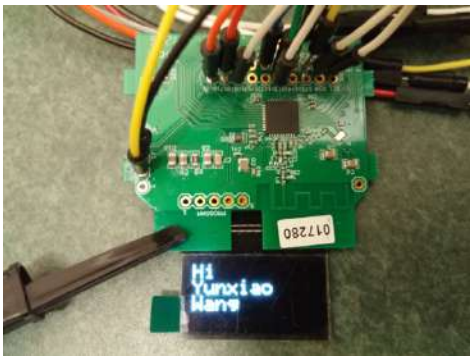

Figure 2-13. BotSpine profile

(7) nRF52832 is a 32-bit ARM Cortex-M4F SoC from Nordic, and it supports BLE and ANT protocol stacks depending on different SoftDevices. Due to its small size, it can be easily embedded into wearables. Figure 2-14 shows the profile of a Nordic nRF52832 development board.

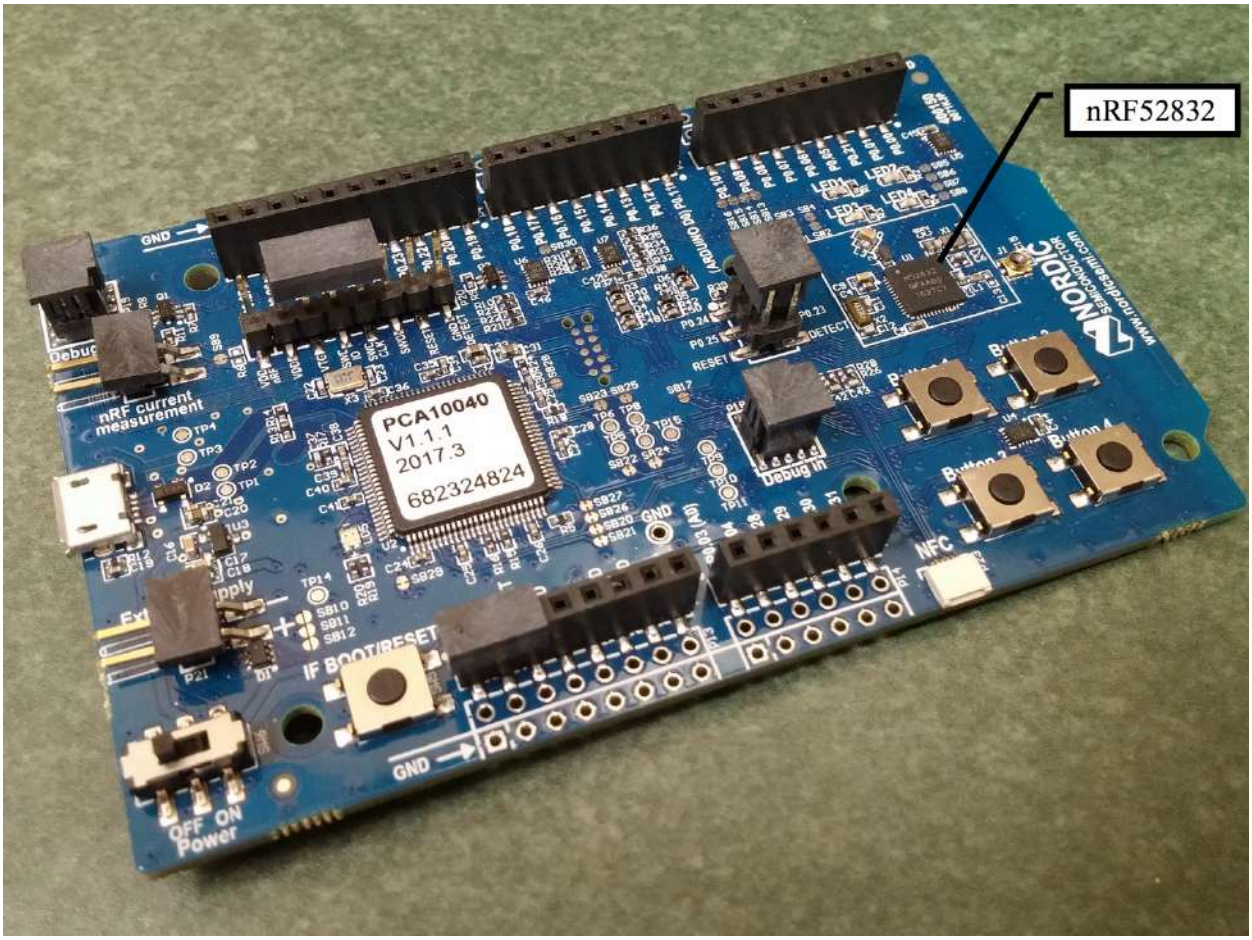

Figure 2-14. Nordic nRF52832 development board

There are a few features worth pointing out:

- nRF52832 is RTOS-independent, so developers can build their customized firmware in 64KB RAM, 512KB flash footprint, such as Zephyr OS, JavaScript engine.
- nRF52 DK includes wireless protocol stack libraries called SoftDevices that are precompiled and linked binary image implementing BLE protocol stack on SoC. The protocol stack and application are independently separated, which means they do not compile and link together. It simplifies application development in a stand-alone manner. The SoftDevice enables the application developers to develop their code as a standard ARM Cortex® -M4 project without having the need to integrate with proprietary IC vendor software frameworks [46]. This means that any ARM Cortex® -M4-compatible toolchain can be used to develop Bluetooth Low Energy applications with the SoftDevice [46]. In comparison, TI CC2541 BLE stack is precompiled to object files, and it requires to be joined with application code at link stage, which means it requires developers to use the same tool-chain to develop application.

2.2.3 Summary

Apparently, compared with single chip, SoC has an inherent dominance that SoC can connect wirelessly while single chip can only do through wire. Deeply, with SoC, the development difficulty can be reduced. For example, an OS coordinates different function programs to do the same work, meanwhile, it isolates each function programs to reduce their coupling. It is convenient for programmers to program each module, and the system architecture is logically clear, especially for complex logic system. Furthermore, with SoC, code readability can be improved for maintenance convenience. For example, developers can manage their codes hierarchically, from driver layer to application layer. The priority of every task and the period of execution are expectable. Moreover, with SoC, code portability can be increased. For example, most logics are already set up in OS, so only porting OS is needed. Due to the above distinctions, only SoC devices are employed for our research. Due to different SoCs offer different interfaces, so a common interface accessible to heterogeneous SoCs is required in the context of IoT.

2.3 IoT Model

Atzoria et al. [38] present that IoT relies on heterogeneous set of objects accessed by its own dialect, so "there is the need for an abstraction layer capable of harmonizing the access to the different devices with a common language and procedure". Yannuzzi et al. [41] assert that "Unfortunately, the requirements and design space of IoT make Cloud Computing unfeasible in numerous scenarios, especially, when the goal is to build a general and multipurpose platform that can serve a wide variety of IoT applications".

2.3.1 IoT Fog

According to Vermesan and Friess [43], "Many Internet of Things applications require mobility support and geo-distribution in addition to location awareness and low latency, while the data need to be processed in "real-time" in micro clouds or fog". A Fog is a Micro-Cloud working closer to the edge of the network. As shown in section 2.2, 20 billion "things" are estimated to be connected to the internet by 2020, and a huge amount of bandwidth is required if the data generated is directly moved to the Cloud. Additionally, Xu and Helal [62] discover that scalability significantly challenges interactions between services and physicals. To be closer to the device, Fog computing plays an important role in helping speed up outputs, boost service quality and increase scalability.

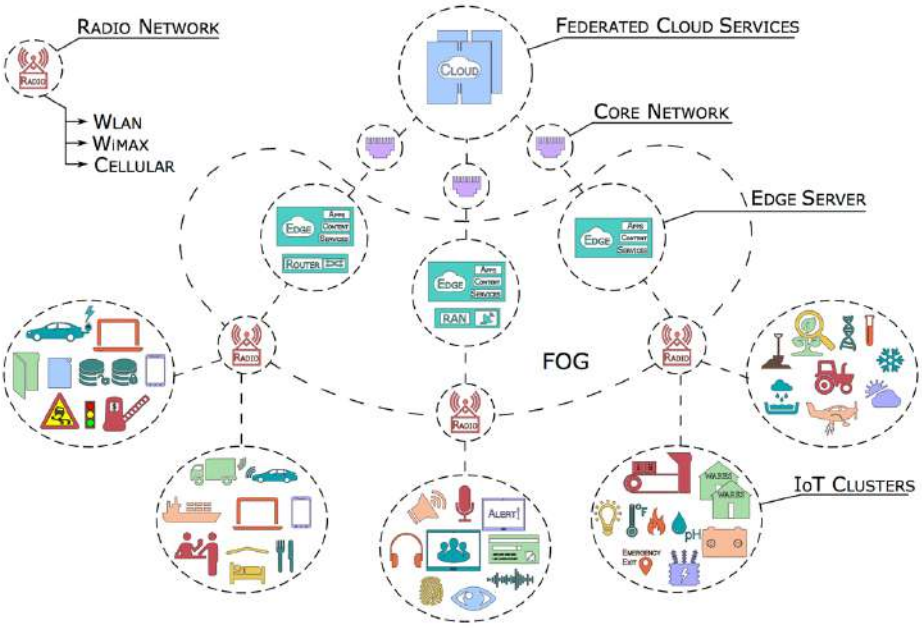

Figure 2-15. Integrated Fog Cloud IoT Architecture [5]

According to Munir et al. [5] who proposed IFCIoT (Integrated Fog Cloud IoT Architecture Paradigm), as depicted in Figure 2-15, a Fog node represents an edge server, and much processing takes place in a fog node. Every Fog node can be deployed locally close to the edge devices, and each one transmits information to a centralized cloud service. "In IFCIoT architecture, each operational Fog node is autonomous to ensure uninterrupted operations of the facility / service it provides" [5]. A Fog node can be connected to other Fog nodes through radio networks (e.g. routers). One of the practical application scenarios is to set up a smart farm. Tons of IoT devices send data to edge servers locally distributed, and edge servers process data and determine the behaviours to IoT devices instead of going to the Cloud. The work in the Cloud is to analyze the data sent from Fog servers for specific applications, then Cloud is able to optimize farming system.

From the above architecture, obviously, Fog computing effectively reduces bandwidth and latency, and increases scalability. A further question may come up: where to place a Fog?

2.3.2 SDN

As Liu et al. [61] state that "Software-Defined Networking (SDN), a novel solution to network configuration and management, has shown great potential to simplify the existing complex and inflexible network infrastructure". Oliveira et al. [17] further identify that "this emerging paradigm uses a logically centralized software to control the behavior of a network". Moreover, Julia and Skarmeta [45] believe that SDN is able to help solve the challenges raised by heterogeneous entities. Rahman [29] proposes an approach of logical clustering against physical clustering where SDN approach is utilized for clustering identifications and managements.

2.3.3 Virtualization of IoT Fog

The virtualization of physical things has emerged in recent years. "Virtualization refers to abstraction of logical resources from their underlying physical characteristics in order to improve agility, enhance flexibility and reduce cost" [9]. As Nastic et al. state [57] that Software-defined IoT (SD-IoT) uses abstraction to simplify provisioning and customization of its components. This approach allows that virtual networks are set up onto a physical device. Even though this approach does not define IoT physicals due to the recognition of costs and processing power of them, we could try to recognize them by their different capabilities for specific roles.

2.3.4 Restful Model

Restful model is the key to connect virtualized IoT Fog node and physicals. All web applications can be developed by provisioning web service, either SOAP (Simple Object Access Protocol) or RESTful (Representational state transfer) web service. SOAP is a protocol, and REST is a style. Both two are not comparable, but it is important to know who they are.

A SOAP message is made by standardized XML document [25]. As shown in figure 2-16, a SOAP envelope contains SOAP header and SOAP body filled with XML messages, and it can be packed in HTTP body and transmitted by HTTP. SOAP and REST are both protocol-independent.

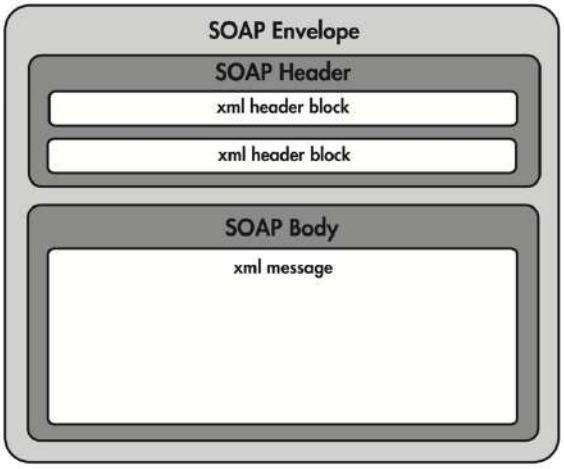

Figure 2-16. SOAP message format [16]

REST is not a standard but a set of constraints that describe three principles [44]:

- Addressability. REST operates data on resources that are identified by URI (Universal Resource Identifier). The resources can be named in any form.
- Uniform interface. The resources are accessible over HTTP standard. Four main REST operations: create, read, update and delete (CRUD) are supported, and they can be implemented by four corresponding HTTP (POST, GET, PUT, DELETE) methods. Uniform interface tells how to operate resources.
- Statelessness. Each request contains all the information that a server needs. The server responds to a new request without referencing any of previous requests.

The most advantage of using REST web service is that you can encode representations in JSON rather than XML the only choice in SOAP, so that you save many bytes, and JSON is easier and faster to parse. According to Potti's experiment [44], "REST wireless response times are comparatively better than SOAP, as the number of simultaneous clients increased", shown in figure 2-17. Therefore, REST would be a better fit in our architecture and experiment.

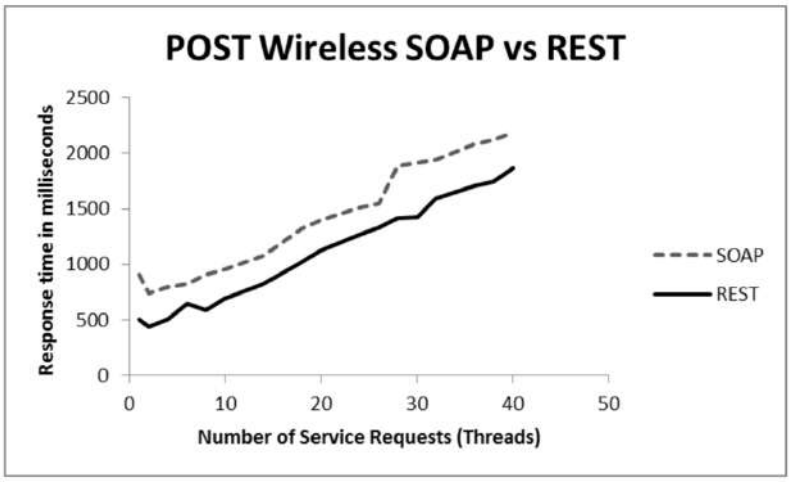

Figure 2-17. SOAP and REST wireless response time [44]

2.3.5 CREST

Furthermore, CREST (Computational REST), which "CREST's framing by explicitly emphasizing computation over information makes it far clearer that these are active resources intended to be discoverable and composable" [35], is "a computation-centric successor to the REST architectural style" [35]. "This style recasts the web from a model where content is the fundamental measure of exchange to a model where computational exchange is the primary mechanism" [35]. For example, a client wants to execute a program and sends a request to a server, and then the server executes the program and return the result. Briefly, a service behavior changes with client conditions.

2.3.6 Summary

In the heterogeneous context, Fog computing allows virtualization network to be feasible in logical cluster against physical cluster by Restful model. The proper management of the virtualized system must be along with proper communication protocols.

2.4 Protocols

Fog environment is comprised of many nodes, so the computation is horizontally distributed, but it can be less energy efficient than in centralized cloud systems [8]. According to Dastjerdi and Buyya [8], “using efficient communications protocols such as CoAP, effective filtering and sampling techniques, and joint computing and network resource optimization can minimize energy consumption in fog environments”. On the other hand, according to Chowdhury et al. [53], “Does HTTP/2 save energy? Yes, when round trip times are above 30ms and when TLS is being used, our tests indicate that HTTP/2 outperforms HTTP/1.1 with TLS in most scenarios”. For example, in the Chowdhury’s tests [53], the Mozilla Firefox Nightly implementation of HTTP/2 consumes less energy than HTTP/1.1 implementation at doing the same work regardless of the webserver used in the tests. Thus, it is hard to compare HTTP and CoAP in the case of energy consumption.

2.4.1 Transport Layer Protocols

In the transport layer, TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are mainly employed over the network layer. The main comparison of TCP and UDP is listed as follows:

- TCP is connection-oriented and reliable (guaranteed delivery), whereas UDP is not. Thus, important packets use TCP because UDP does not react on packet loss.
- TCP establishes a connection before actual data transmission takes place, whereas UDP does not, and also UDP does not do error checking for packets, so UDP is faster than TCP.
- TCP re-arranges packets in an specific order, whereas UDP does not because every packet of UDP is independent of each other. However, application layer can manage the order if required.

2.4.2 Application Layer Protocols

The corresponding Client-Server model based application layer protocols of TCP and UDP are typically HTTP and CoAP in the scope of IoT paradigm.

CoAP is a simplified version of HTTP. It is typically designed for the communication between resource constrained devices, as shown in figure 2-18. Constrained environment and internet environment are bridged through proxy devices.

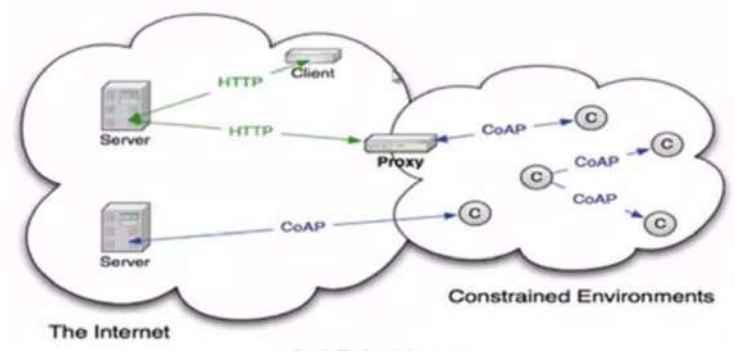

Figure 2-18. CoAP architecture [64]

Figure 2-19 depicts what CoAP message format looks like. CoAP message is written in binary, and it must be initialized with 4 bytes formatted headers. In the message:

```
|       0       |       1       |       2       |       3       |
|7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  TKL  |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Token (if any, TKL bytes) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|    Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2-19. CoAP message format [64]

- “Ver” is a 2-bit unsigned integer. It mentions CoAP version number. “T” is a 2-bit unsigned integer. It indicates message types: CON (0), NON (1), ACK (2), RST (3). “TKL” is a 4-bit unsigned integer. It indicates the length of token (0-8 bytes). “Code” is an 8-bit unsigned integer. It is combined with 3-bit class representing a request (0), a successful response (2), an error response from client (4), or an error response from server (5), and 5-bit detail. The last two bytes are network byte order according to big-endian order.
- Followed by “Token” that associate a request and a response. Then there is 0 option or multi-options. There could be no payload after options.

- If there is a payload, and its length is not 0, there must be a payload identifier 0xFF before payload. 0xFF only happens after the end of an option.

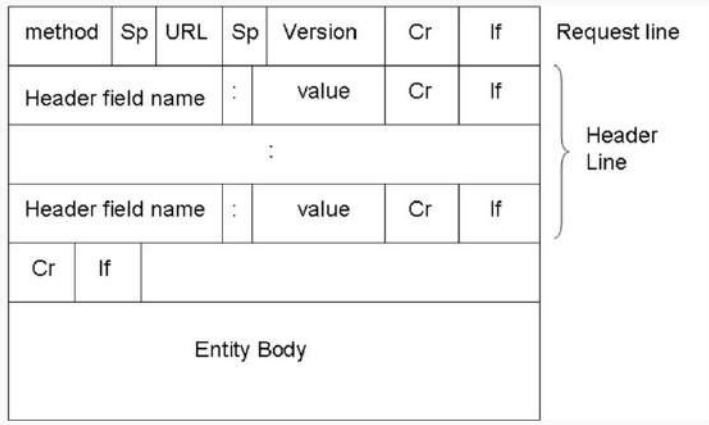

Figure 2-20. HTTP message format [1]

While HTTP message format is depicted in figure 2-20:

- The start line indicates an HTTP method (GET, PUT, POST, DELETE, HEAD, OPTIONS, TRACE, CONNECT where CoAP shrinks it to GET, PUT, POST, DELETE in "Code"), URL, HTTP version.
- HTTP headers are name-value pairs separated by a colon similar to JSON format that is easy to parse.
- Followed by one blank line, then body starts if it is available.

From the above comparison, CoAP has some advantages in the view of application:

- UDP is lightweight (fixed 8-byte header) while TCP has a bigger header (at least 20-byte).
- CoAP supports IP multi-cast and uni-cast communication while TCP only supports point-to-point that is not suitable for notification services.

While HTTP has also some advantages:

- TCP is stream-oriented that is used to transmit a continuous data flow, and all bytes are guaranteed to be received identical as sent, and in a correct order.
- HTTP header and body are plain text that is friendly to programmers, while CoAP header and body are binary that only machine is able to read it.

Both two protocols are used for different purpose. In this thesis, our experiment is to test the performance of the server which can accept multiple requests that requires robust message delivery, big volume of data will be collected, and for ease of parsing header, so HTTP/1.1 is selected for later experiment.

2.4.3 BLE communication layer protocol

According to Litepoint [11], "Bluetooth technology is a short-range communications technology whose robustness, low power, and low cost make it ideal for a wide range of devices ranging from mobile phones and computers to medical devices and home entertainment products". Figure 2-21 shows the history of Bluetooth.

Figure 2-21. Bluetooth history [11]

The latest version of Bluetooth is Bluetooth Low Energy (BLE) or Bluetooth 4.x. According to Tauchmann and Sikora [23], "Bluetooth Low Energy extends the Bluetooth standard in version 4.0 for ultra-low energy applications through the extensive usage of low-power sleeping periods, which inherently difficult in frequency hopping technologies". Sansanayuth [59] indicated that "BLE came with a new design that provided energy consumption 20 times lower than the previous version. Due to the low energy consumption, the maximum data rate for BLE is 100 kbps, which is lower than the Bluetooth classic with EDR mode".

2.4.3.1 BLE protocol stack

Typically, as shown in figure 2-22, a Bluetooth protocol stack is mainly divided into 2 parts: host stack and controller stack. Host stack is composed of the middleware protocols. RFCOMM (Radio Frequency Communications) protocol is used to expose RS-232 serial port to application layer over L2CAP (Logical Link Control and Adaption Protocol) layer which converts data from upper layer into a format that controller stack can understand. SDP (Service Discovery Protocol), which is bound to L2CAP, is used to advertise and discover nearby Bluetooth services. Intermediate HCI transport layer directly accesses to Bluetooth hardware for services execution.

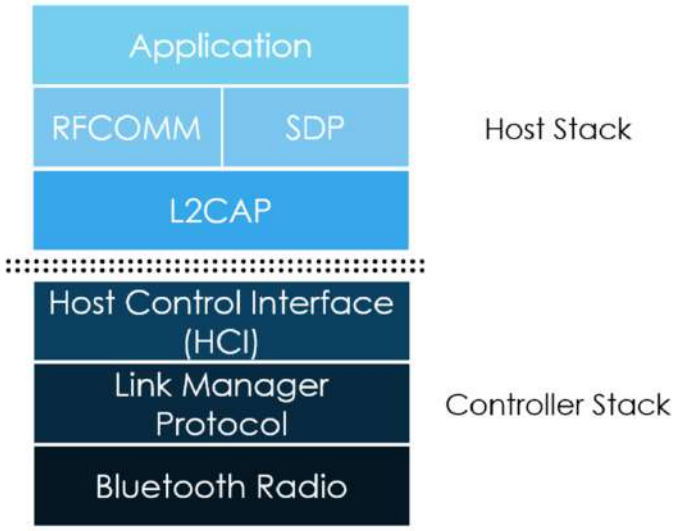

Figure 2-22. a typical Bluetooth stack [13]

Bluetooth specification [13] defines that "Profiles are definitions of possible applications and specify general behaviors that Bluetooth® enabled devices use to communicate with other Bluetooth devices". However, every Bluetooth device has to implement Generic Access Profile (GAP) first. GAP advertises packets (mandatory) and scans response (optional) at up to 31-byte payload for establishing a connection between two Bluetooth devices. Bluetooth specification [13] also points out that "For two Bluetooth devices to be compatible, they must support the same profiles. For Bluetooth LE, developers have the option of using a comprehensive set of adopted profiles, or use the Generic Attribute Profile (GATT) to create new profiles". For example, a Bluetooth Classic peripheral device can use Serial Port Profile (SPP) to be connected and specify RFCOMM channel number and SDP record for communication to a central device; similarly, a BLE peripheral device must use GATT over ATT (Attribute Protocol) who can tell what services and characteristics it can provide to a central device. Thus, any official and customized BLE services cannot be used until GATT has been enabled after the connection is set up by GAP. Figure 2-23 shows Bluetooth profiles and middle-layer protocols.

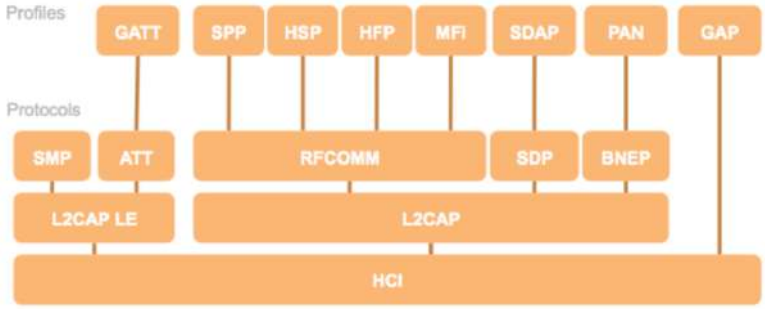

Figure 2-23. Bluetooth profiles and middle-layer protocols [14]

2.4.3.2 GATT

GATT defines two roles: server and client. A server refers to a BLE peripheral device who can provide services, while a client refers to a BLE central device who can connect up to seven peripherals at a time. A GATT transaction is nested by Profiles, Services, Characteristics, as shown in figure 2-24. Profiles do not actually exist but include a collection of services. Each service includes at least one characteristic. A characteristic is the smallest unit in GATT transaction, it is the one who is exactly doing the work. Each characteristic consists of "Properties", "Value" and "Descriptors". "Properties" are required right before "Value" followed by "Descriptors" (optional) that describe "Value". In many cases, "Properties", "Value" and "Descriptors" are all called "Descriptors".

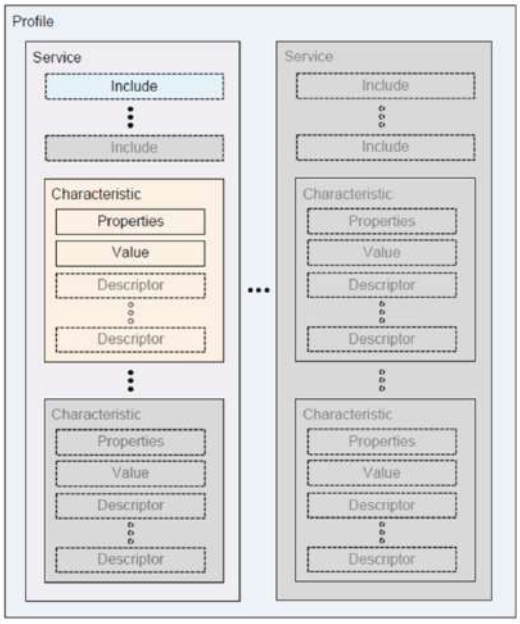

Figure 2-24. GATT profile hierarchy [12]

Table 2-3. A simple example of a service

	Handle	UUID	Attribute value
attribute	0x0001	00002800-0000-1000-8000-00805f9b34fb	Generic Access Service (0x1800)
attribute	0x0002	00002803-0000-1000-8000-00805f9b34fb	Properties (Read), Value Handle (0x2a00)
attribute	0x0003	00002a00-0000-1000-8000-00805f9b34fb	Temperature E.g. 24 degrees
attribute	0x0004	00002803-0000-1000-8000-00805f9b34fb	Properties (Read), Value Handle (0x2a01)
attribute	0x0005	00002a01-0000-1000-8000-00805f9b34fb	Humidity E.g. 90 percent
attribute	0x0006	00002803-0000-1000-8000-00805f9b34fb	Properties (Read), Value Handle (0x2a02)
attribute	0x0007	00002a02-0000-1000-8000-00805f9b34fb	Clock alarm E.g. 10 a.m.
attribute	0x0008	00002803-0000-1000-8000-00805f9b34fb	Properties (Read), Value Handle (0x2a03)
attribute	0x0009	00002a03-0000-1000-8000-00805f9b34fb	Switch state E.g. 1
attribute	0x000a	00002803-0000-1000-8000-00805f9b34fb	Properties (Read), Value Handle 0x2a04
attribute	0x000b	00002a04-0000-1000-8000-00805f9b34fb	Bus arrival E.g. 3 min

Services, characteristics and anything inside characteristics are attributes that are transported by ATT. A GATT server stores Services, Characteristics in a simple lookup table by 16-bit (official) or 128-bit (customized) UUIDs (may not be unique in the profile) for identifying the type (service declaration, characteristic declaration, characteristic value declaration, descriptor declaration) in the table over ATT, and ATT can access attribute value by “Read”, “Write”, “Notify” and “Indicate”, and the maximum single payload size of these four operations is 20

bytes. Each attribute is given a numerical 16-bit handle (must be unique in the profile) for GATT client to access and reference.

Given a service attribute handle ranges from 0x0001 to 0x000b including five characteristics, and service UUID: 0x1800. The example in table 2-3 shows:

- Handle 0x0001 is the service declaration that tells there is a service that starts here, where UUID is always 0x2800.
- Handle 0x0002 (0x0004, 0x0006, 0x0008, 0x000a) is a characteristic declaration that tells there is a characteristic that starts here, where UUID is always 0x2803. Its value declaration is at handle 0x0003 (0x0005, 0x0007, 0x0009, 0x000b), where UUID is 0x2a00 (0x2a01, 0x2a02, 0x2a03, 0x2a04).
- Attribute value reveals where other attributes are. Properties tells the way to interact with characteristic value. (All characteristics of 0x1800 are officially read-only with no descriptors and security, but characteristic values here are fabricated just for a demonstration.)
- The example also proves that UUIDs of services declaration and characteristics declaration are always the same, but attribute handles must be unique in the profile. Additionally, attribute handles are not necessarily consecutive.

The security of BLE is also important. BLE provides three basic security services [12]:

- Authentication and authorization for establishing trusted relationships between devices.
- Encryption and data protection are to protect data integrity and confidentiality.
- Privacy and confidentiality are to prevent device tracking.

2.4.4 Summary

There is no doubt that REST takes obvious dominance for CRUD operation. URLs directly guide users to the target resource with POST, GET, PUT, DELETE operations. In comparison, XML-based SOAP increases complexity of handles with lower performance than REST. After comparison, REST style will be chosen in our architecture design without hesitation. With the

creation of CREST, computational expressions are exchanged, there would be a new evolution of Web services that more contributions would go toward IoT management and configuration.

BLE is expected to be widely used in SoC because of many advantages, such as low power consumption with the use of a cell battery for a year, lower cost than wifi but lower bit rate. It is an ideal communication protocol between IoT middleware and resource highly constraint devices. In the future, IPv6 can be integrated on the top of 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Network) over L2CAP, so that BLE-enabled SoCs are able to be directly connected to the internet without any hub (e.g. a middleware) from vendors.

2.5 Solutions to Problems

Based on the above reviews, we can summarize the solutions to the problems by the table below:

Table 2-4. Solutions to problems

	Problems	**Solutions**
1	How to provide uniform web-like interface?	Rykowski and Wilusz [36] announced that "REST-base framework appeared as better suited for heterogeneous and widely distributed IoT devices and services." In Fog domain, we will virtualize resource-rich physicals as middleware that provisions RESTful Web services, so that the embedded middleware can provide uniform interface for heterogeneous IoT devices and users. In particular, to be accessible to users, the interface can provide web-like experience for users, such as reading data.

2	How can we use this interface to send commands to things?	Huang and Wu [19] realized that "the tasking capability allows other devices or users to actuate devices via the Internet", so users are able to remotely change state onto devices. Bjelica et al. [42] proposed Insight Device Cloud (IDC) that "allows automatic execution of Lua scripts". At the edge of the network, we will port a script engine onto a resource highly constraint BLE-enabled endpoint, and send a command through this uniform interface that then forwards to the script engine for execution at run time. For example, a user can write a simple command to change the state of an LED through this uniform interface.
3	How can we change command / code to run new functionality?	To send code (i.e. computational expression) through this uniform interface by using Computational REST, and to store or update the code into a database for history tracking. Finally, to push the code which covers the previous one and execute a new program to perform new functionality.

CHAPTER 3

ARCHITECTURE

There are three types of existing IoT middleware (i.e. agent) architectures [3]:

- Service-based Architecture that allows developers to deploy heterogeneous IoT devices as services, such as storage, web services interfaces, query manager, etc.
- Cloud-based Architecture that limits developers on the number and type of IoT devices that can be deployed.
- Actor-based Architecture that allows IoT devices to be exposed as reuable actors and distributed in the network.

According to Fersi [28], "The service in SOA is a software that hides its internal implementation details and provides through an invokable interface a public functionality". Thus, application layer and physical layer are able to communicate with each other independently through SOA. Hachem et al. [55] successfully adopt SOA and provide an SOA middleware that supports discovery (registration) of sensing metadata, composition of discovered sensing metadata, and access to any sensing data.

As mentioned in Chapter 1, building a middleware (i.e. uniform interface) that provides services to Clients is the main part of our research goal, so SOA approach is suitable to help reach our goal. To reach the goal, low latency is prerequisite, so the computation must move closer to the edge devices, and virtualization must be able to talk with physicals. Thus, the architecture we propose should only focus on Fog domain, and middleware should only be embedded onto resource-rich physicals. Figure 3-1 shows Fog domain SOA with application and physical layers.

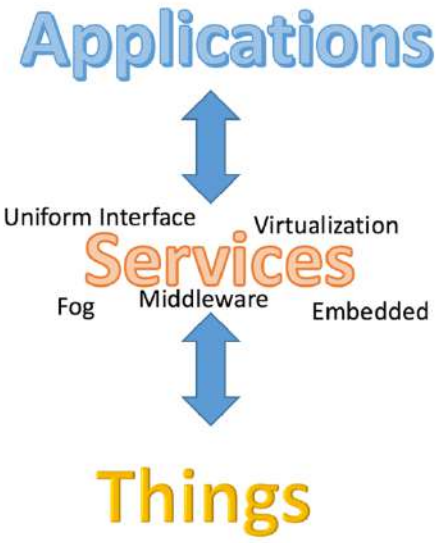

Figure 3-1. Fog domain SOA with application and physical layers

3.1 Proposed System Architecture

In the Fog domain, there are tons of middleware who are virtualized onto physicals, such a Raspberry Pi who is a resource-rich platform able to be multi-threaded. Figure 3-2 shows the proposed system architecture which includes middleware layer that provides services.

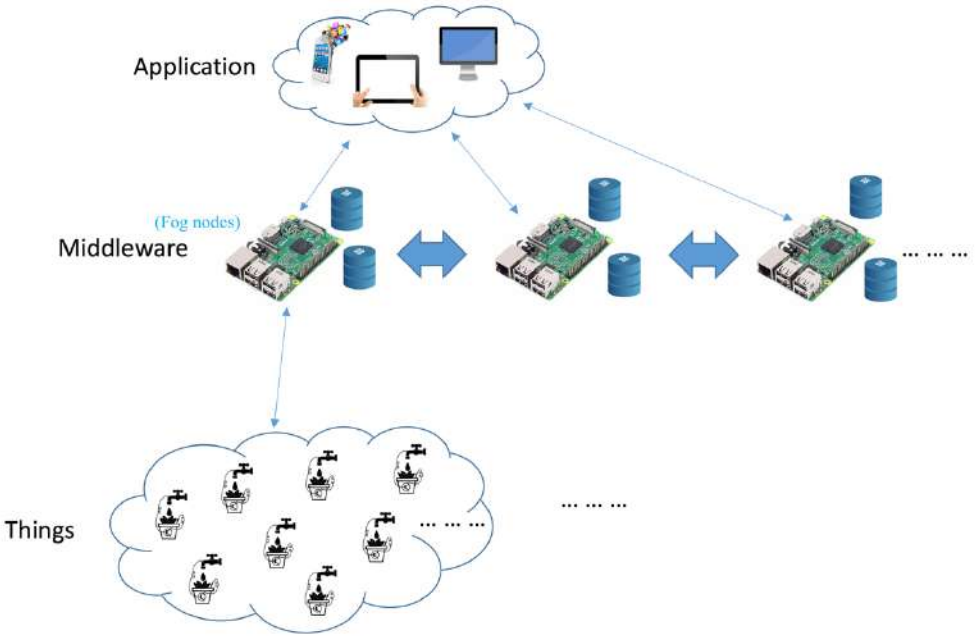

Figure 3-2. Proposed system architecture

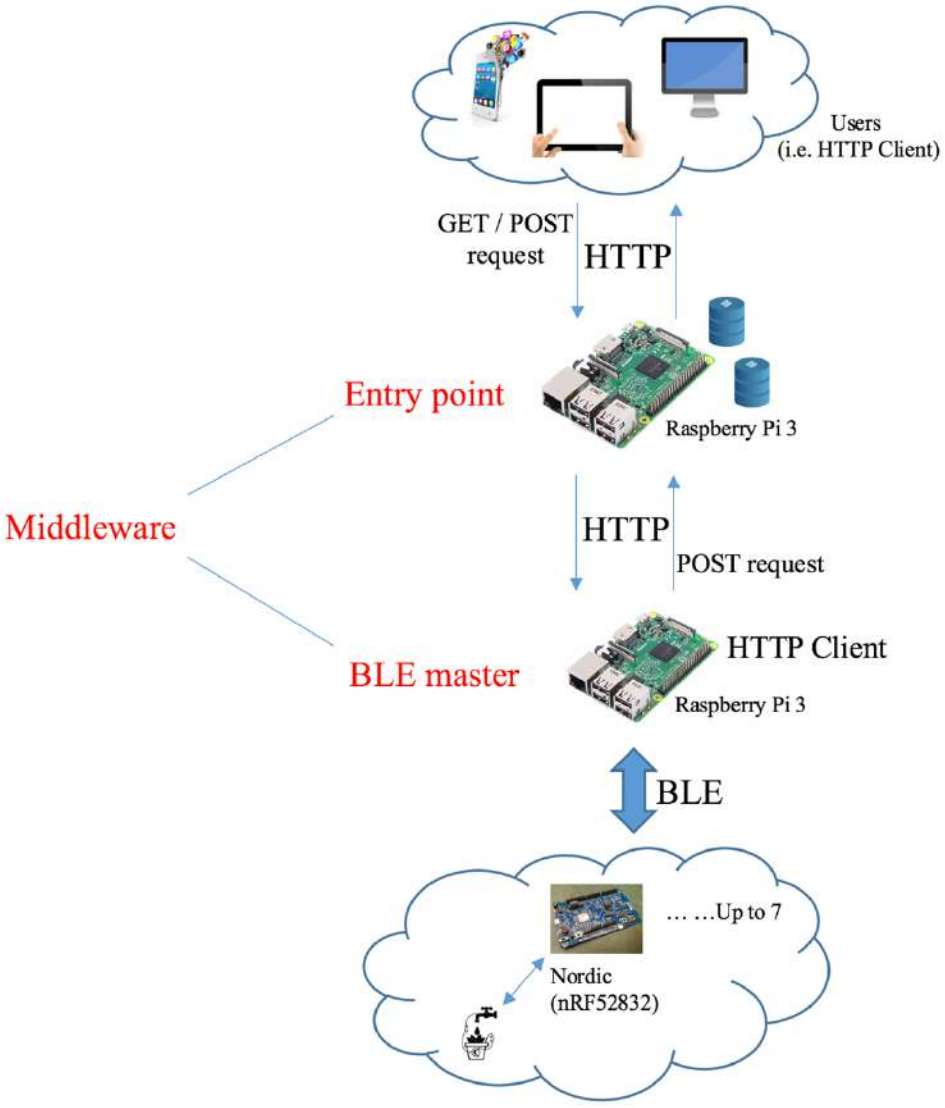

Figure 3-3. Simplified architecture

In our case, there are wirelessly deployed flowerpots in a green house, we need middleware providing uniform interface to manage and configure them. To reduce energy consumption of physicals, we only employ BLE-enabled endpoints in physical layer; however, a BLE master can only connect up to seven BLE endpoints at a time, so we employ BLE masters, such as a Raspberry Pi, directly connect to BLE endpoints. Then, the entry point is free of BLE connections, and it only focuses on HTTP connections, so that scalability can be increased as well. Figure 3-3 shows that the entry point is a Fog server which handles all HTTP requests, and

BLE master takes care of BLE connections and meanwhile handles HTTP requests from the entry point.

3.1.1 Work Flow

The work flow of sending sensor data is shown in figure 3-4 describing that all varieties of BLE endpoints sending sensor data to a BLE master by BLE read operation, then transmitting to an entry point (i.e. Fog server), and finally all sensor data is stored in a database in the Fog node. An BLE endpoint registers in the system since the first successful sending. If the requests failed to send, just sent again and the failure would be kept in log for future analysis. Finally, users can fetch any available data through a Web application running in the entry point.

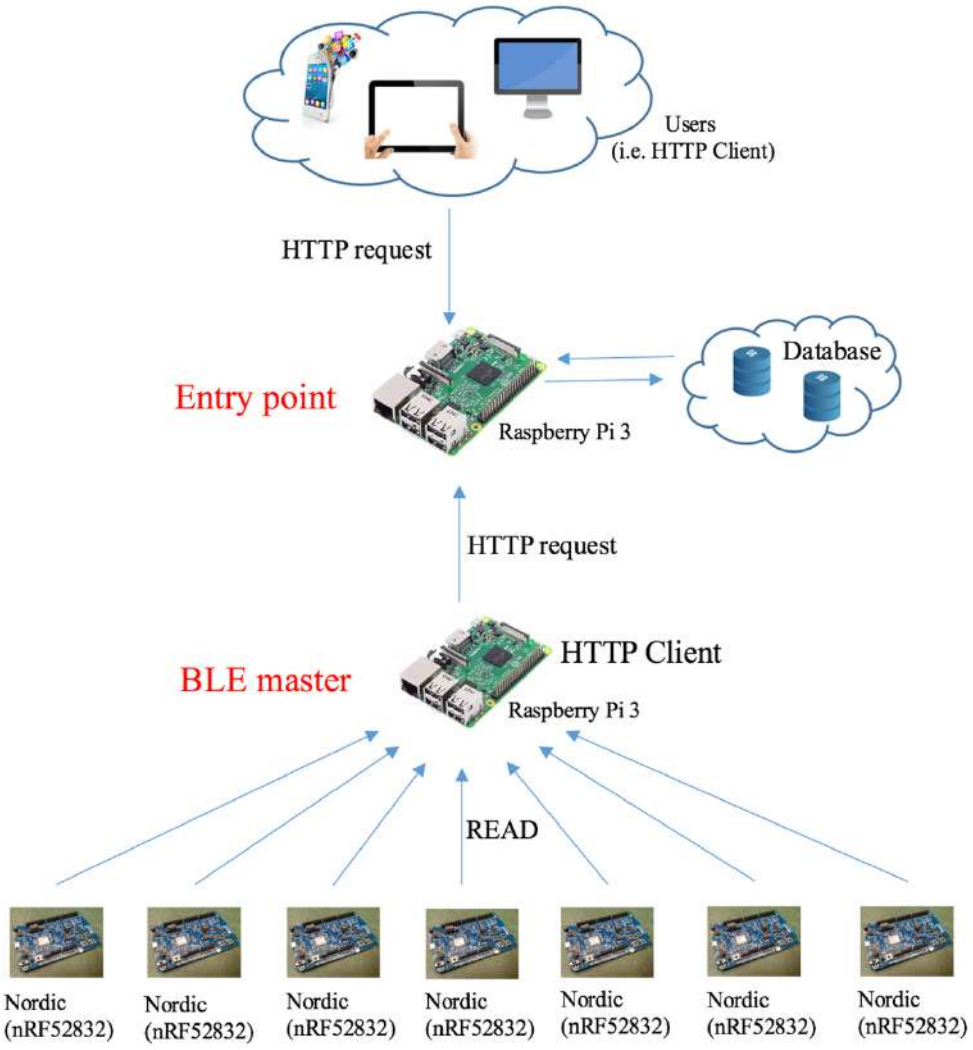

Figure 3-4. Sensing flow

The work flow of triggering actuators is shown in figure 3-5 describing how users to trigger actuators onto BLE endpoints via Web applications. Specifically, the entry point handles HTTP POST requests from users and stores behaviors in the database for history review, and then forwards requests to BLE master. Finally, BLE master triggers BLE endpoints by BLE write operation. It is noted that one BLE master can only connect up to 7 BLE peripherals at a time.

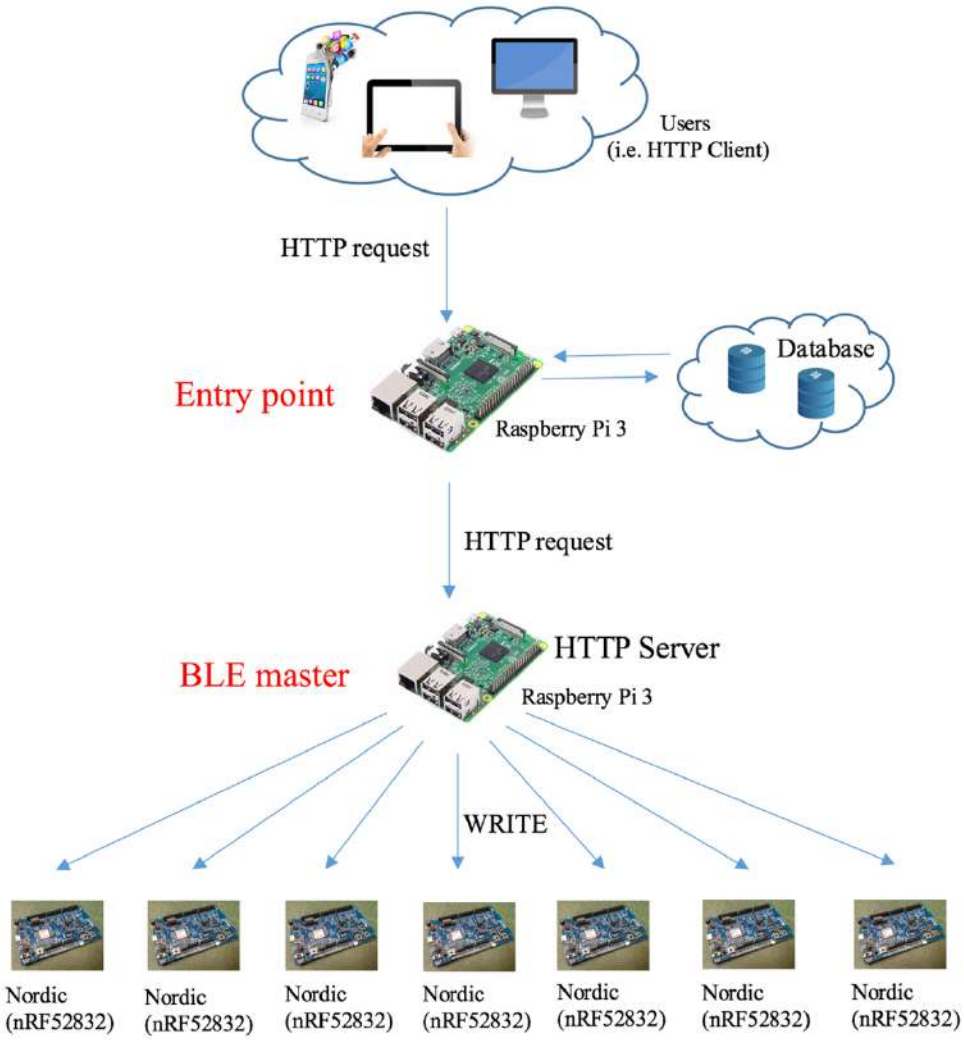

Figure 3-5. Actuating flow

3.1.2 RESTful Web Services

To meet the goal of our research, the middleware should be able to deal with sensing sources, such as sensor data, gateways. Virtualized physicals are in charge of managing Web resources by exposing REST APIs accessible to users and edge devices.

As reviewed in Chapter 2, one of the key principles of REST style is its uniform interface. The resource provisioning is expressed by URIs and operated by four primitives (i.e. CRUD): create, read, update, delete which are mapped to HTTP methods: POST, GET, PUT, DELETE. Table 3-1 shows the mapping from CRUD to HTTP methods.

Table 3-1. Uniform REST interface

CRUD	**HTTP methods**
Create	POST
Read	GET
Update	PUT
Delete	DELETE

An IoT endpoint is represented as a virtual resource exposed to externals by URIs and operated by CRUD uniform interface. Those virtual resources form services middleware may provide. Table 3-2 indicates that the location of resource can be identified by URIs, and the behavior of physical and application layers can be operated by CRUD.

Table 3-2. Examples of REST APIs

HTTP methods	**URIs**	**Semantics**
POST	http://192.168.0.100/IoT/{board_id}	To create (i.e. register) a new board
GET	http://192.168.0.100/IoT/{board_id}	To retrieve information from a specific board
PUT	http://192.168.0.100/IoT/{board_id}	To update a board
DELETE	http://192.168.0.100/IoT/{board_id}	To delete a board

Furthermore, computational REST (CREST) enhances the ability of resource provisioning by exchanging computational expressions, so that any code can be transmitted through RESTful Web services, typically POST and PUT.

For example, a BLE-enabled endpoint sends temperature and humidity data every second to a BLE master, and then the BLE master sends POST requests with payloads to the entry point every second by URL “http://192.168.0.100/IoT/” with payloads, such “boardName”, “value”, “programVersion”, “location” and any other necessary information in JSON string, then the entry point parses JSON string and stores information into database. When users use the Web application and send GET requests to entry point, such URL “http://192.168.0.100/IoT/board_id”. Then the entry point fetches the information of “board_id” from the database and meanwhile inserts the behavior of GET requests into database for later history check. The same principle when sending commands to endpoints. Users send POST requests to the entry point with URL “http://192.168.0.100/IoT/board_id”. Then the entry point inserts this behavior into database for later history check and fetches the IP address of BLE master and MAC address of the corresponding endpoint. Then the entry point sends POST requests to BLE master by URL such “http://192.168.0.102/nRF52832_0/D9:C7:9A:5E:43:30”. Finally, BLE master parses URL and payload, and then connects and writes commands to the endpoint with MAC address “D9:C7:9A:5E:43:30”.

Sending a computational expression is exactly the same way to do, but with setting different characteristic UUID. PUT and DELETE requests are similar to POST. In our case, PUT requests are used to update command / code, and DELETE requests are used to delete history records.

3.1.3 Script Engine

A script engine can help users change state of an endpoint at run-time by simply executing a command / code. For example, Xie et al. [48] had successfully implemented Tapper which is a lightweight scripting engine for highly constrained wireless sensor nodes. In our case, we will hire Espruino, which is compatible with Nordic nRF52832 chip, as script engine to execute JavaScript command / code at run-time. Consequently, Espruino can accept on-demand computational expressions and execute them immediately over BLE communication.

3.2 Summary

In this Chapter, we proposed an architecture that is able to:

- provide a uniform interface by RESTful Web services.
- provide an access for sending computational expressions by CREST.
- execute command / code in JavaScript at run-time by Espruino.
- accept computational expressions sent from middleware by REST APIs.

CHAPTER 4

IMPLEMENTATION

In this chapter, we will implement REST model which shows how to use GET, POST, PUT and DELETE to solve the problems. Also, we will implement BLE communication between a Nordic nRF52832 chip and a smart phone through a mobile app.

4.1 RESTful Web services in Embedded Middleware

A Raspberry Pi 3, which is a Linux platform, plays the role in implementing software components that provides RESTful Web services to externals. The entry point implements Fog server for requests from both application and physical layers. The BLE master implements a lightweight server that only handles requests forwarded by the Fog server, and also implements a BLE handler that operates BLE write and read with Nordic nRF52832 chip.

The sensing data of Nordic endpoints represents virtual resources that are read by BLE master, and then the BLE master POST data as payload in JSON format to the entry point at every certain period. Finally, an end point registers in the Fog server that stores all information into database. The URL exposed by the entry point represents an API for a BLE master to access. The payload is written in JSON format as name / value pair for efficient and easy data parsing. Figure 4-1 is an example of a request from an endpoint, including a URL, a JSON-formatted payload, and an API to send a HTTP POST request.

```
url='http://192.168.0.104:9999/IoT/'
value='{"Temperature":"%d %s","Humidity":"%d %s"}' % (T, "Celsuis", H, "%")
payload={'value':value, 'boardName':'nRF52832_0', 'programVersion':'v1.0.0', 'location':'MADMUC-RM178.9', 'remoteHostPath':'/nRF52/', 'mac':'D9:C7:9A:5E:43:30'}
r=requests.post(url, data=payload, timeout=sys.maxint)
```

Figure 4-1. Example of a request from an endpoint

We use Python Flask framework to form servers in both the entry point and BLE master, and Werkzeug server is the base of Flask. Werkzeug is easy to parse HTTP header and payload, and it integrates routing system to match URLs. Additionally, Flask naturally supports multi-threaded mode. Then, Flask is able to run concurrent Web services.

```
@app.route('/IoT/', methods=['GET', 'POST', 'PUT', 'DELETE'])
def IoT_from_board_to_server():
    if request.method == 'GET':
        return 'This is GET request.'
    if request.method == 'POST':
        boardName=request.form['boardName']
        value=request.form['value']
        programVersion=request.form['programVersion']
        location=request.form['location']
        remoteHostPath=request.form['remoteHostPath']
        mac=request.form['mac']
    if request.method == 'PUT':
        return 'This is PUT request.'
    if request.method == 'DELETE':
        return 'This is DELETE request.'
```

Figure 4-2. Example of HTTP parsing in the server

On the server side, as an example shown in figure 4-2, the server parses the HTTP header including the URL and the HTTP method, and also parses the payload. Then, data processing would be followed by.

After registration of an end point, users can look up the specific virtual resource or composition of several ones via a Web application. To fulfill the scalability of RESTful Web services, HTTP GET responses naturally support caching and parallelization on URIs. Based on HTTP GET, users are able to check the history of an endpoint by selecting available dates. See figure 4-3, figure 4-4, figure 4-5 and figure 4-6.

Figure 4-3. Example of GET request on Web GUI

Result:

Data:

The latest data from **nRF52832_0**

Date: 2017-06-17
Time: 16:07:49

Table 1. Value(s)

Temperature	Humidity
24 Celsuis	27 %

Program version: v1.0.0
Location: MADMUC-RM178.9
IP address: 192.168.0.102

Status: success

Figure 4-4. Example of GET result on Web GUI

Check a device history?

--select a date-- (Only the dates the selected device was operated are shown)

See history

Figure 4-5. Example of GET history request on Web GUI

Result:

Table 2. History of nRF52832_0 on 2017-06-17

Time	HTTP method	Code
16:41:32	GET	N/A
16:41:42	POST	reset();
16:42:27	POST	var on=false; setInterval(function(){ on=!on; LED2.write(on); LED3.write(on); digitalWrite(D11,on); },500);
16:43:00	POST	empty

Note:
If HTTP method is GET, code is "N/A" due to no body part;
If HTTP method is POST, and nothing was posted, code is "empty"

Status: success

Figure 4-6. Example of GET history result on Web GUI

To exchange computational expressions between application and physical layers, POST and PUT are to create and update payloads that could be computational expressions, as shown in figure 4-7. It allows users to change the state and configure endpoints on-demand. Also, DELETE can remove an endpoint, but the behavior will be logged for later history check.

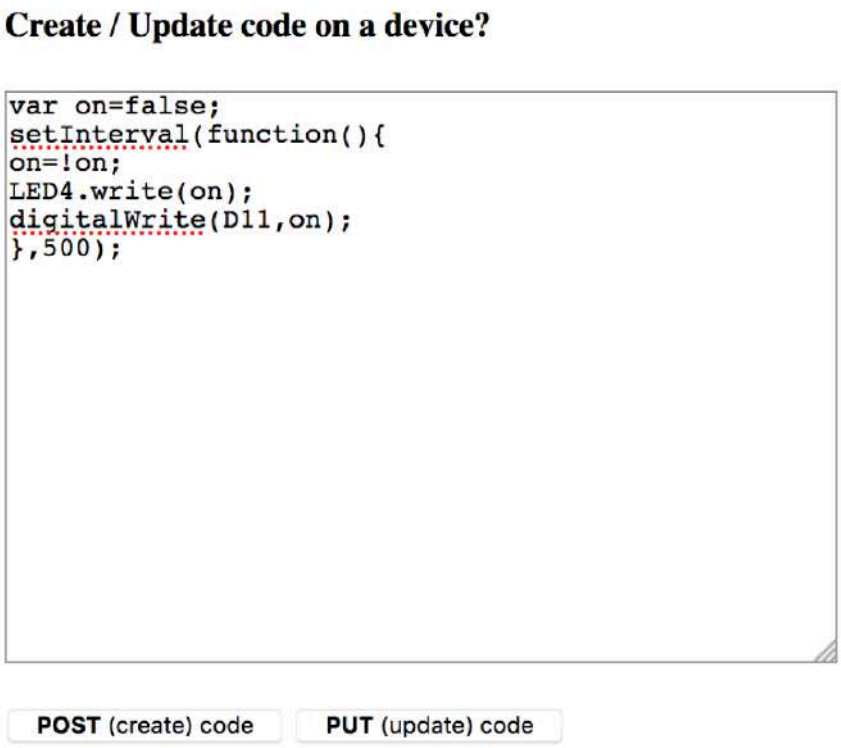

Figure 4-7. Example of POST and PUT computational expressions

4.2 JavaScript Execution in BLE endpoints

Nordic nRF52832 chip cannot be virtualized due to highly constraint resource (i.e. CPU power, memory, networking, etc.), but it is capable of hosting a single program that monitor inbound low-energy connections. Then, the third parties are allowed to push scripts onto it. To do so, we developed a mobile app to access low-energy connections, particularly with Nordic nRF52832 chip. The application we use over BLE protocol stack is Espruino (i.e. JavaScript Interpreter) that executes JavaScript code.

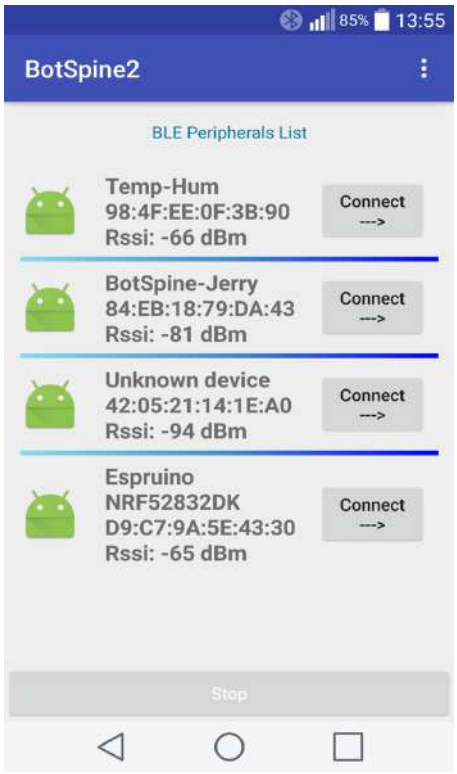

Figure 4-8. Example of BLE scanning in a mobile app

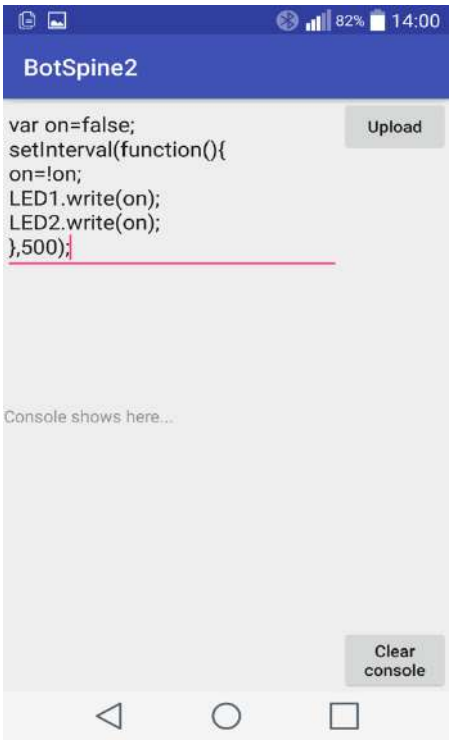

Figure 4-9. Example of BLE write in JavaScript

Figure 4-8 and figure 4-9 show the first activity that performs BLE scanning, and the second activity that performs BLE write operation once the service is discovered.

As long as the command / code directly sent from a mobile app can be successfully executed at run time, then it is possible to accept computational expressions sent from application layer through middleware at run time, as figure 4-7 shows. Additionally, Espruino allows users to customize APIs exposing to JavaScript, which brings users more flexibility towards applications' functionality setup.

4.3 Summary

This Chapter implements the proposed architecture and reaches the goal of our research.

RESTful Web services provide uniform interface represented by CRUD pattern, such GET, POST, PUT, DELETE. Not only does it fulfill the requirement of retrieving, creating, updating, and deleting virtual resources, but computational REST can also exchange computational expressions with low-energy nodes.

With Espruino (i.e. JavaScript interpreter), Nordic nRF52832 chip is not only a role of a BLE peripheral, but it also extends the possibility for users to access an BLE endpoint through Web application. By doing this, there would not be any distance limit between users and low-energy physicals.

CHAPTER 5

EXPERIMENT

To know the performance of middleware (i.e. uniform interface) and script engine, in this chapter, as shown in Figure 5-1, there are three experiments that will be covered: the round trip time of:

1. Users GET data stored from MySQL database in the entry point.
2. BLE master POST data collected from Nordic chip to the entry point with MySQL database process.
3. BLE communication between a BLE master (i.e. a central) and a Nordic chip (i.e. a peripheral).

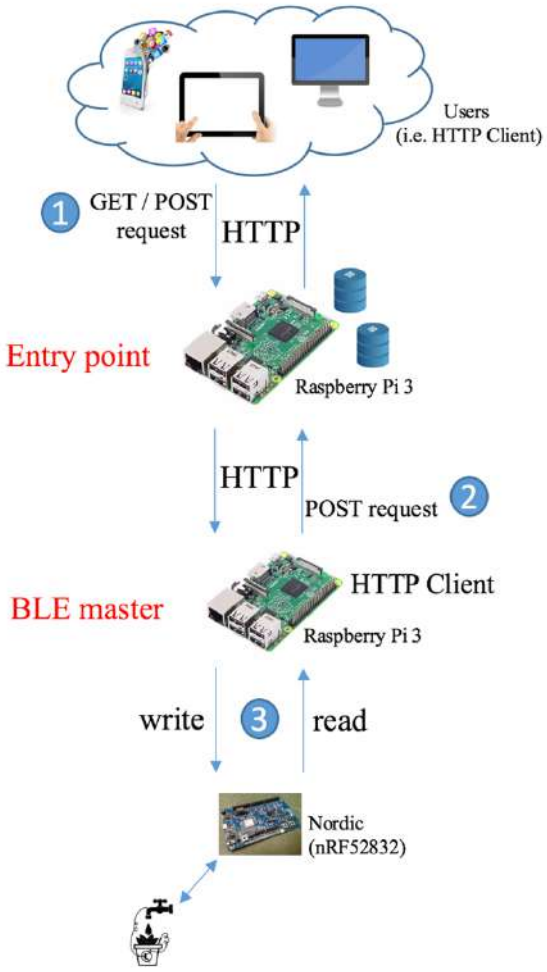

Figure 5-1. Experiment profile

To ensure the network speed and stability, we will use Ethernet access instead of wireless connection. The core of this experiment is to test the performance of the RESTful Web services provisioning in Raspberry Pi 3 and the performance of JavaScript interpreter in Nordic chip.

5.1 Performance of Middleware

Since we will test the concurrent capability of the middleware, a multi-thread simulator named JMeter will be hired as a load generator.

5.1.1 Performance of GET

The first set of tests focuses on users as Client that retrieves the state of a device by implementing GET request processed with MySQL database, and the URL is "/IoT/nRF52832_0". Please note that concurrent Clients are represented by threads that refer to different colors. Every setting runs three rounds. Every request sends 208 bytes and receives 5183 bytes. Since GET requests are cacheable, the plots below indicate the performance of the cache that is hosted in the Raspberry Pi 3 entry point. The cache is updated every second by reads that emanate from the endpoint. Nordic nRF52832 chip is the endpoint.

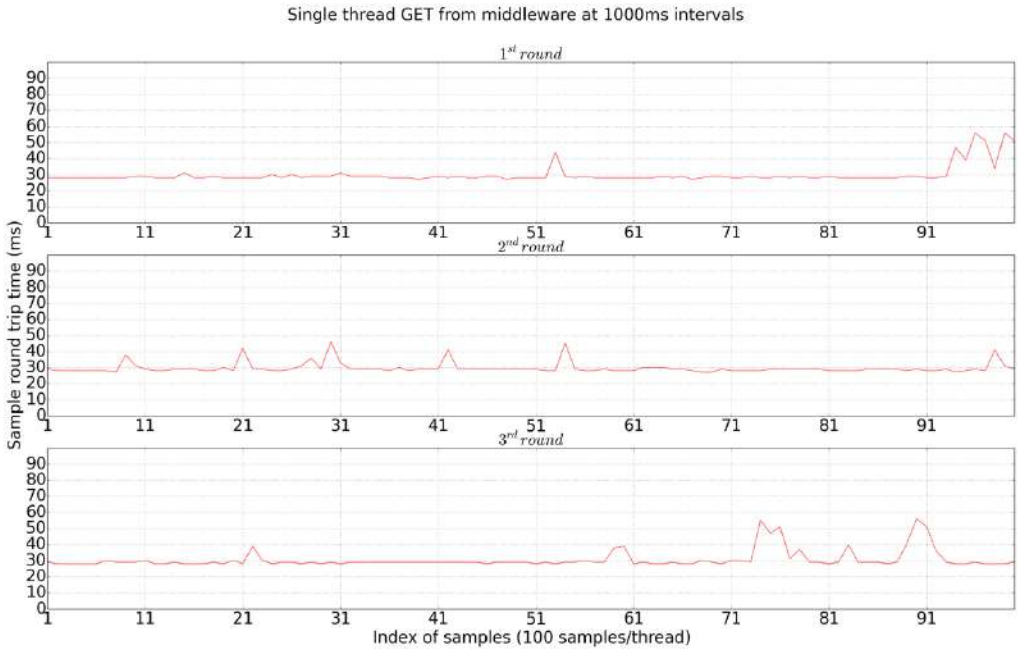

Figure 5-2. One thread sending 100 GET requests (1000 milliseconds delay)

In figure 5-2, there are some spikes around sample #53 and #93~#100 in the first round. There are some spikes around sample #9, #21, #30, #42, #54 and #98 in the second round. There are some spikes around sample #22, #60, #75, #83 and #90 in the third round. Thus, the plot looks flat with few random spikes probably because of noise. The average round trip time is at around 30 milliseconds.

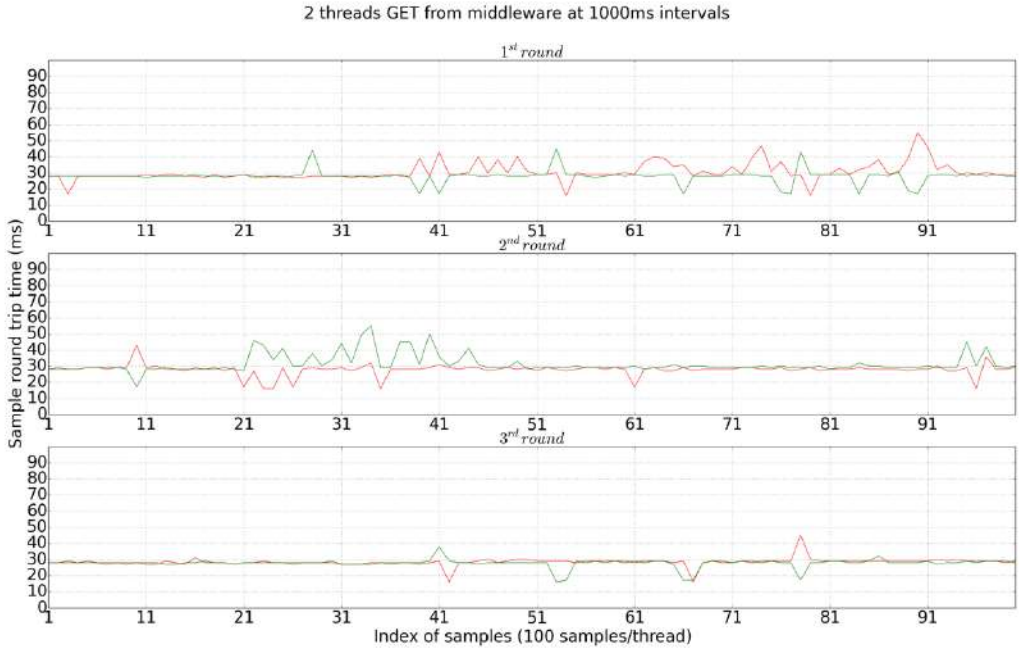

Figure 5-3. Two threads sending 100 GET requests (1000 milliseconds delay)

In figure 5-3, the second thread was added. The first thread is represented by red line, and the second thread is represented by green line. In the first round, there are some points where one thread goes faster at the similar amount of time as the other thread goes slower at around sample #40, #53, #65, #77 and #90. In the second round, the same phenomenon happens at round sample #10, #22, #25, #34 and #95. In the third round, it just happens a couple times at around sample #41 and #78. The reason of that is not known, but it is not surprising that the plot has more fluctuations than single thread has because CPU must fork more power for the second thread. Similarly, the average round trip time is at around 30 milliseconds.

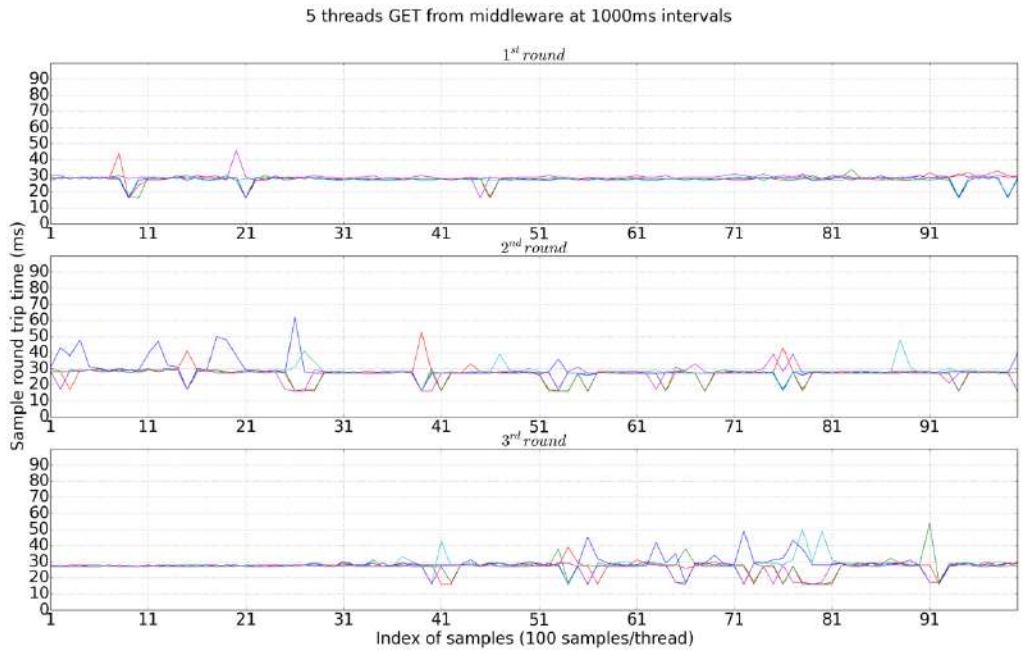

Figure 5-4. Five threads sending 100 GET requests (1000 milliseconds delay)

In figure 5-4, the same phenomenon happens when the number of thread increases to five with sending requests at the same sending delay (i.e. arrival rate), and the performance of handling five threads is similar to handling two threads.

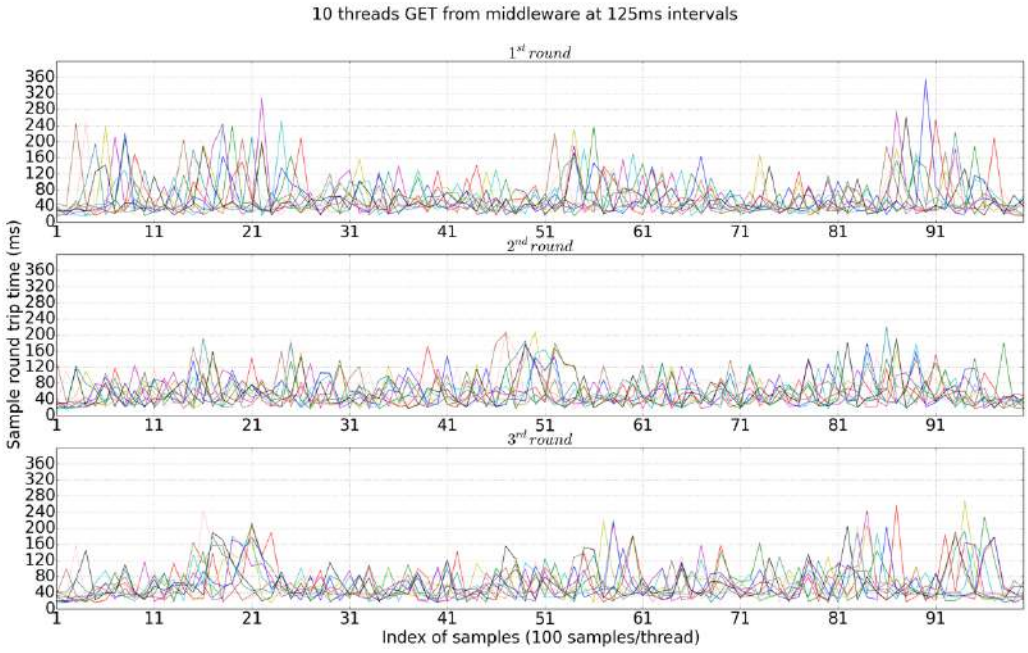

Figure 5-5. Ten threads sending 100 GET requests (125 milliseconds delay)

In figure 5-5, when the number of thread increases to ten and the sending delay decreases to 125 milliseconds, the average round trip time is three times higher than the one of five threads at 1000 milliseconds. It is not surprising that the performance of a resource-constraint device becomes critical as the number of thread increases and the sending delay decreases.

In figure 5-6, when the number of thread increase to twenty, the plot of each round looks very chaotic, and the server becomes overloaded due to the Raspberry Pi 3 is resource-constraint in CPU power, memory and networking, even though it is resource-rich, compared to resource highly constraint devices.

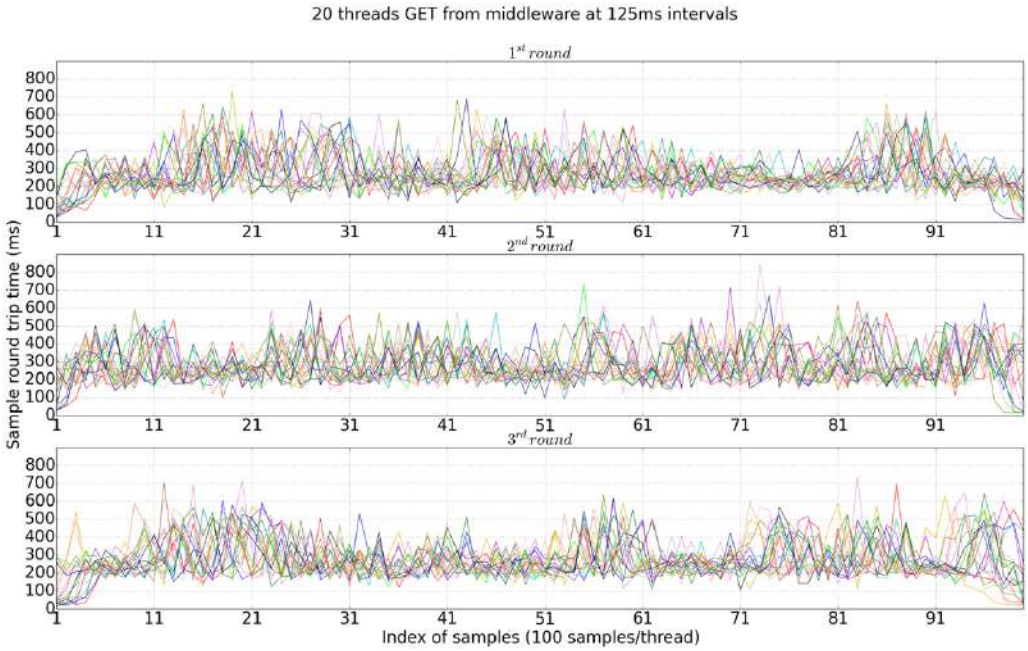

Figure 5-6. Twenty threads sending 100 GET requests (125 millisecond delay)

To sum up, from figure 5-2 to figure 5-4 show that at 1000 millisecond sending delay up to 5 concurrent threads do not impact the middleware so much. However, from figure 5-5 to figure 5-6, as the number of concurrent thread increases and the sending delay decreases, we can see a distinct decline in the middleware performance. The more the number of concurrent thread increases, the closer the middleware reaches the critical point. It is suggested that a load balancer can act as a reverse proxy and distributes network traffic loads across multiple machines.

5.1.2 Performance of POST

The second set of tests focuses on a BLE master (i.e. Raspberry Pi 3) as Client POST data from an endpoint to the entry point (i.e. Fog server) with MySQL database process. The URL is "/IoT/". HTTP body in JSON format shows in figure 5-7. Every request sends 469 bytes and receives 159 bytes. It is noted that POST cannot be cached.

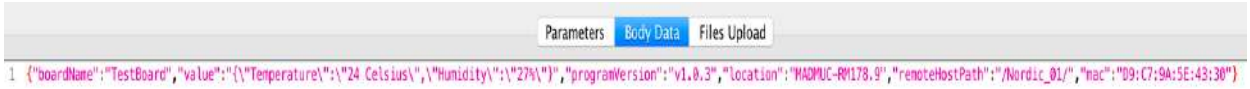

Figure 5-7. HTTP body in JSON format

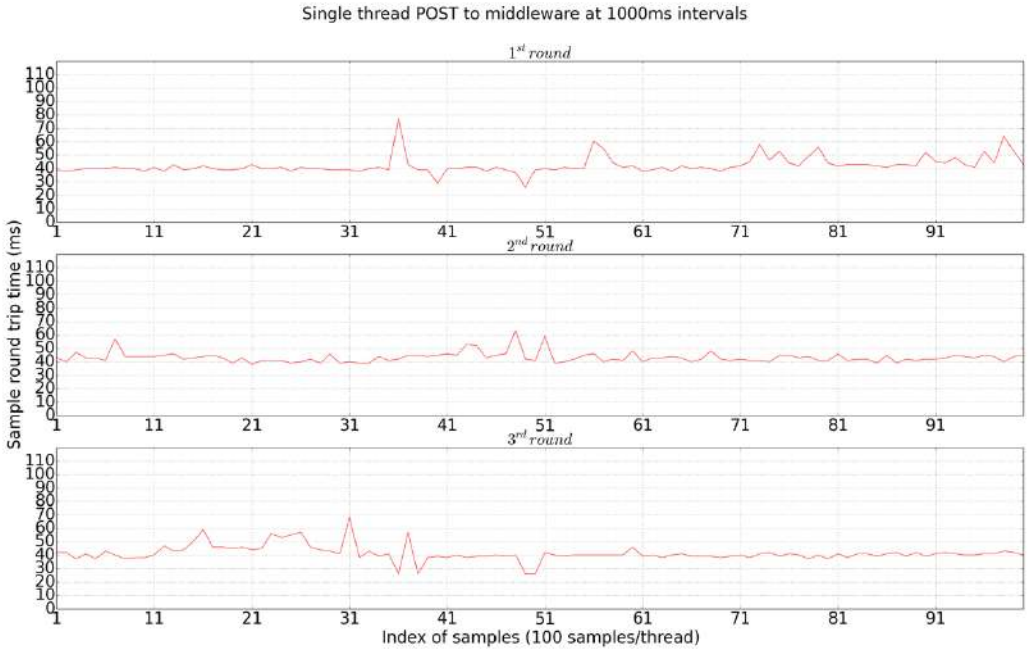

Figure 5-8. One thread sending 100 POST requests (1000 millisecond delay)

In figure 5-8, there are some spikes at around sample #35, #55, #75 and #95 in the first round. There are a couple of spikes at around sample #5 and #46 in the second round. There are some continuous spikes at the first half of the third round. Compared to figure 5-2, the plot is more fluctuated than the one of GET requests test as a whole. The reason of that might be because POST requests cannot be cached, while the GET requests in figure 6-2 retrieve caches.

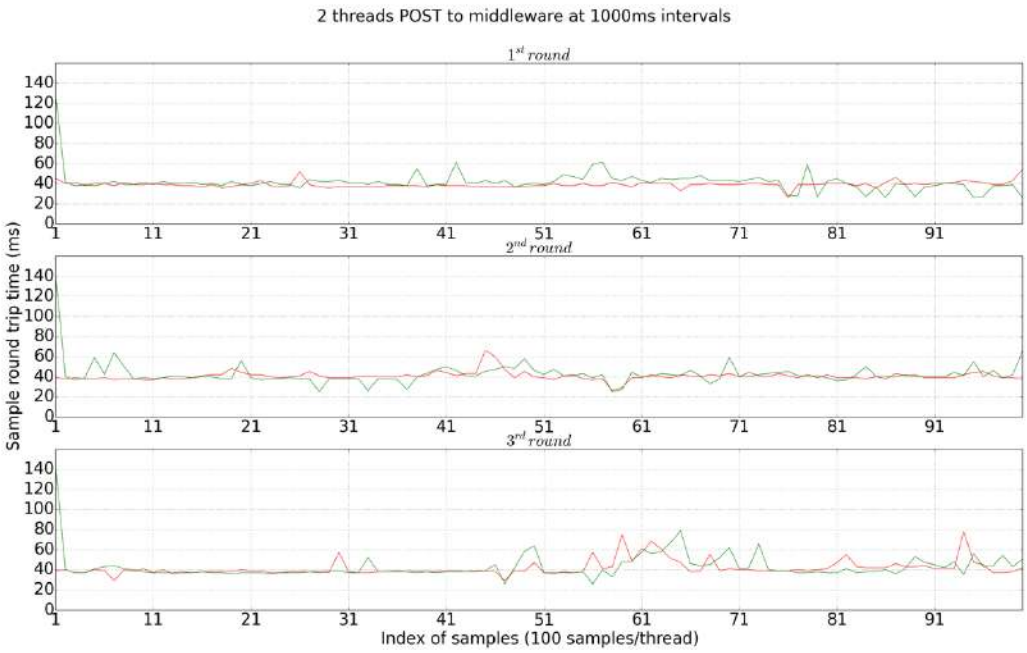

Figure 5-9. Two threads sending 100 POST requests (1000 millisecond delay)

In figure 5-9, in the first round, the first thread in red looks more stable than the second thread in green, and the second thread has some spikes at around sample #1, #38, #42, #55 and #76. In the second round, there are two fluctuations for the first thread at around sample #44 and #56, and the second thread has some spikes at around sample #1, #5, #20, #26, #33, #35, #48, #56, #70, #83 and #95. In the third round, the first thread has some spikes at around sample #6, #30, #45, #55~#66, #82 and #93, and the second thread has some spikes at around sample #1, #32, #50, #55~#73. When the second thread is involved, according to the plot, there is a wake-up period and its sending delay is at around 140 milliseconds for the second thread at the beginning of each round.

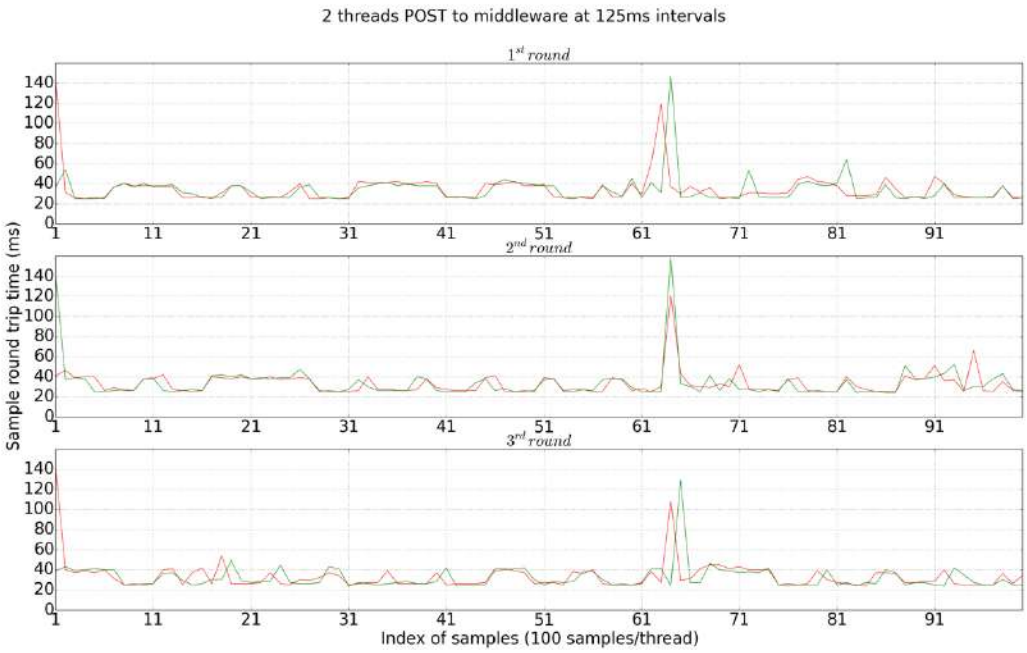

Figure 5-10. Two threads sending 100 POST requests (125 millisecond delay)

In figure 5-10, as the sending delay decreases to 125 milliseconds, the plot looks very different. In the first round, the first thread has a couple significant spikes at around sample #1 and #62, and the second thread has one significant spike at around sample #63. In the second round, the first thread has one significant spike at around sample #63, and the second thread has two significant spikes at around sample #1 and #63. In the third round, the phenomenon is similar to the first round. It is interesting that the significant spike at around sample #63 of the second thread is always higher than the one of the first thread in each round. Also, there is a wake-up period for either thread at the first request in each round.

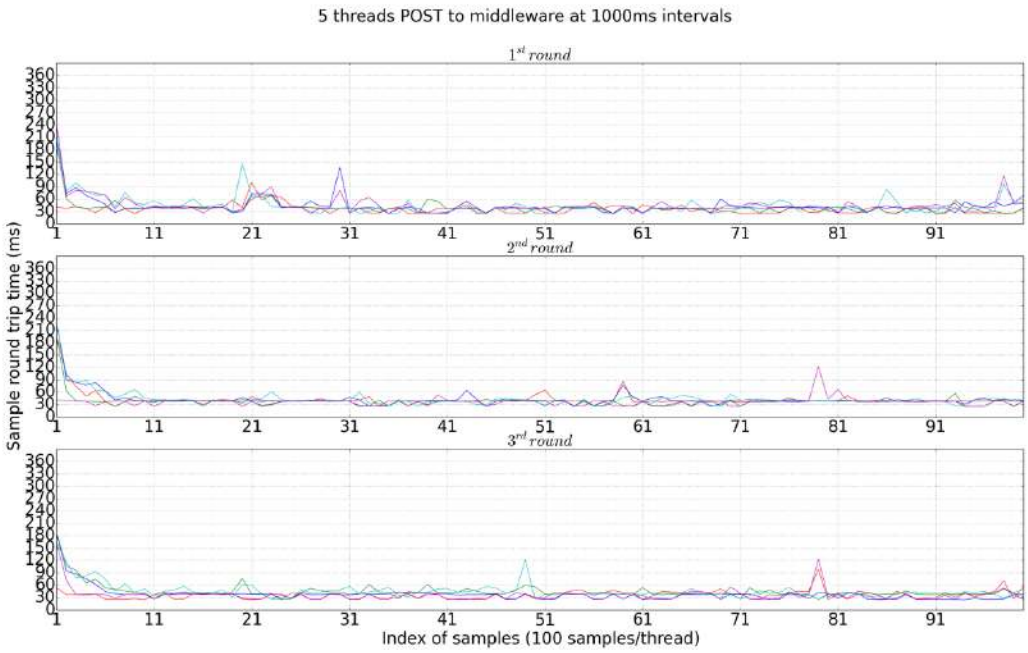

Figure 5-11. Five threads sending 100 POST requests (1000 millisecond delay)

In figure 5-11, in the first and the third round, there is a significant wake-up period for each thread at around sample #1~#10, except the first thread; however, in the second round, the first thread has a wake-up period at the first request.

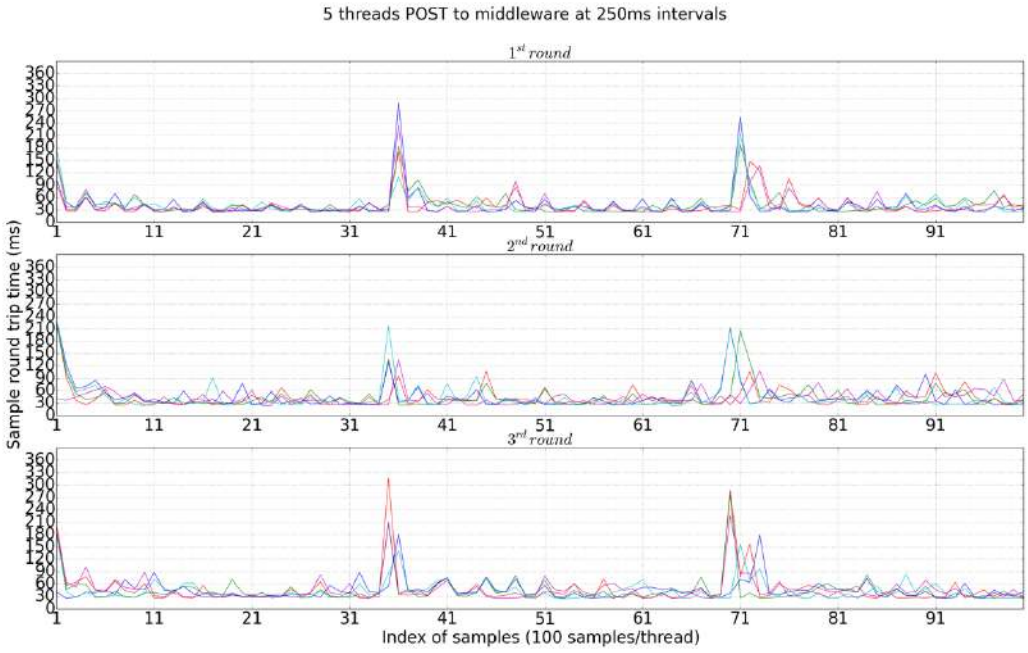

Figure 5-12. Five threads sending 100 POST requests (250 millisecond delay)

In figure 5-12, in each round, there are two significant spikes at around sample #34 and #71, when the sending delay decreases to 250 milliseconds. Compared to figure 5-11, the round trip time of the first request in both two plots are roughly at 200 milliseconds, and there are two critical peaks coming up when the sending delay goes down to 250 milliseconds. The reason might be too much memory use at transient time.

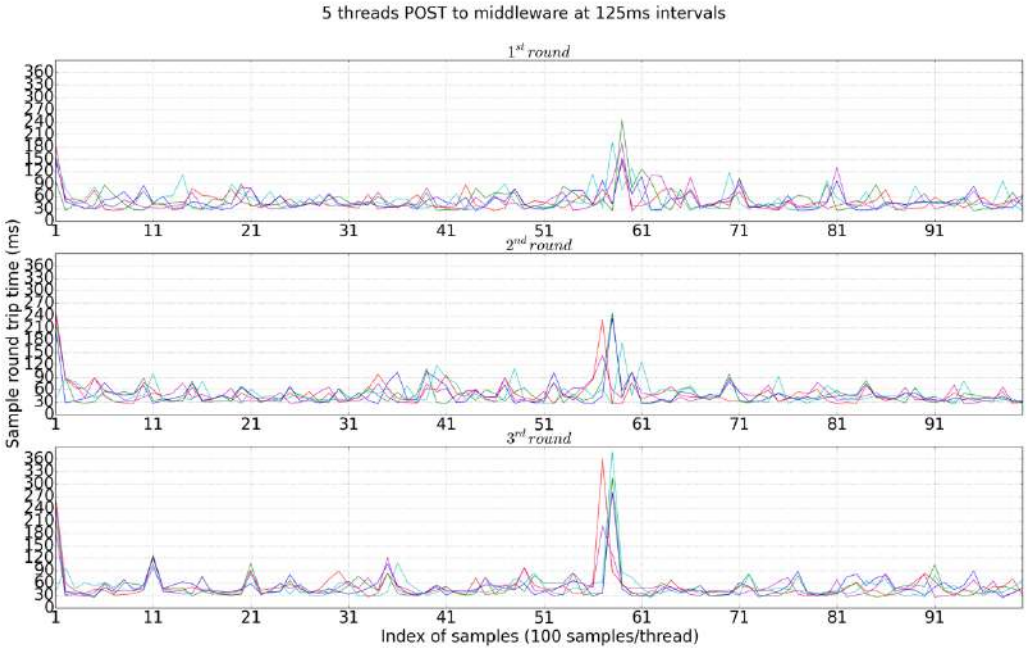

Figure 5-13. Five threads sending 100 POST requests (125 millisecond delay)

In figure 5-13, in each round, it is not surprising that there is a wake-up period at the first request, but there is only one significant spike at around sample #57 when the sending delay further goes down to 125 milliseconds. Also, the spike at around sample #57 in the third round shows a higher sending delay than the one in the first and the second rounds. Compared to figure 5-12, the round trip time of the first request goes up to roughly 240 milliseconds in figure 5-13, and the entire curves of each round look sharper than the ones in figure 5-12. Therefore, from figure 5-11 to figure 5-13, it is clear that the shorter the sending delay is, the more impact the middleware has toward the performance.

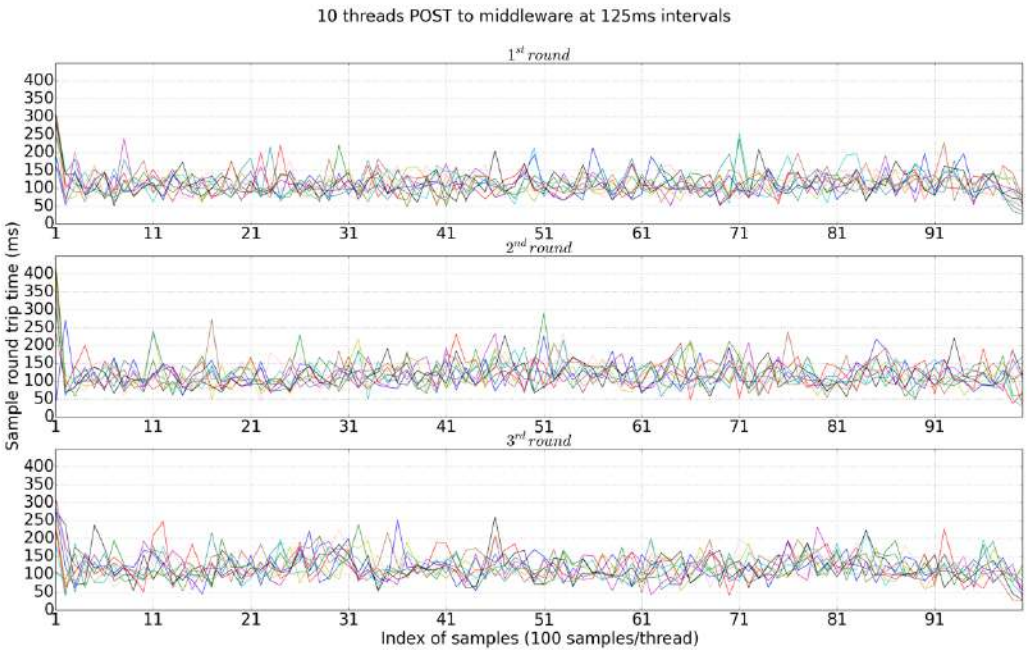

Figure 5-14. Ten threads sending 100 POST requests (125 millisecond delay)

In figure 5-14, compared to figure 5-13, when the number of thread doubles at the same sending delay, the performance of middleware gets pervasive impact. The average round trip time is at 100 milliseconds.

In figure 5-15, when the number of thread doubles again, the performance of middleware becomes critical, and the plot looks chaotic. The average round trip time rises up to around 300 milliseconds. According to the plot indication, there is no need to test more than twenty threads, and we believe that there will be a critical point that the middleware is out of capability to handle requests.

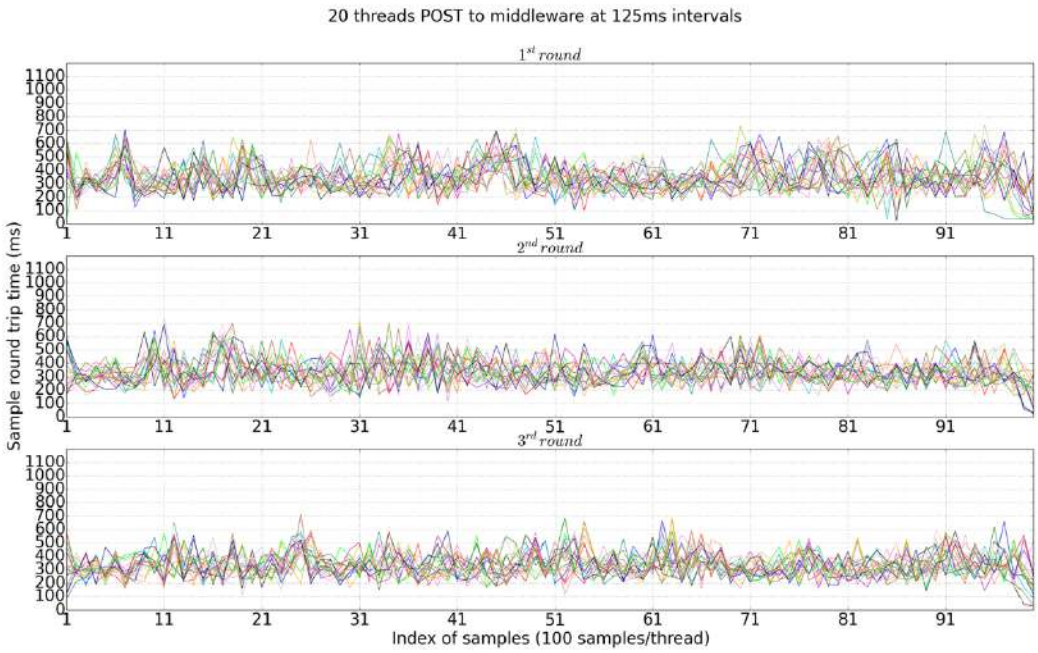

Figure 5-15. Twenty threads sending 100 POST requests (125 millisecond delay)

To sum up, figure 5-10, figure 5-12 and figure 5-13 show that there are some significant spikes coming up when the sending delay becomes shorter at multiple threads. Given that more endpoints are involved in sending POST requests via BLE masters, it is not surprising that the round trip time significantly increases that the performance dramatically declines at higher loads.

5.2 Performance of Script Engine

The third set of tests are to know the performance of Espruino (i.e. JavaScript interpreter). The test will be 100 writes from BLE master to a Nordic chip, and 100 reads from the Nordic chip, and 100 writes plus reads. In this experiment, the size of every write and read is in one packet (i.e. up to 20 bytes). For example, a command such as "reset();", a temperature value such as "24". The purpose of this experiment is to evaluate the latency caused by BLE master writes and reads to / from endpoints.

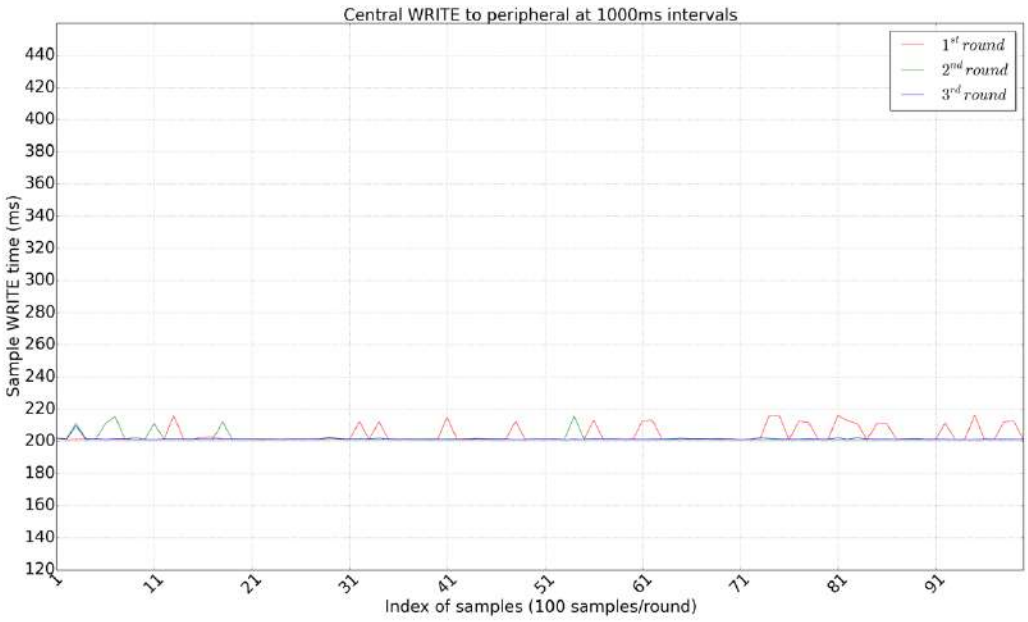

Figure 5-16. 100 sequential writes to an endpoint (1 second delay)

In figure 5-16, the BLE "write" operations have some spikes at around sample #13, #33, #41, #47, #55, #61, #75, #82 and #95 in the first round. In the second round, there are four spikes at around sample #5, #11, #18 and #53. In the third round, there is only one spike at around sample #3. The average trip time is at around 200 milliseconds. It is noted that one BLE connection can only be set up by one BLE master and one BLE peripheral at a time, so there is impossible for either of two to be multi-threaded.

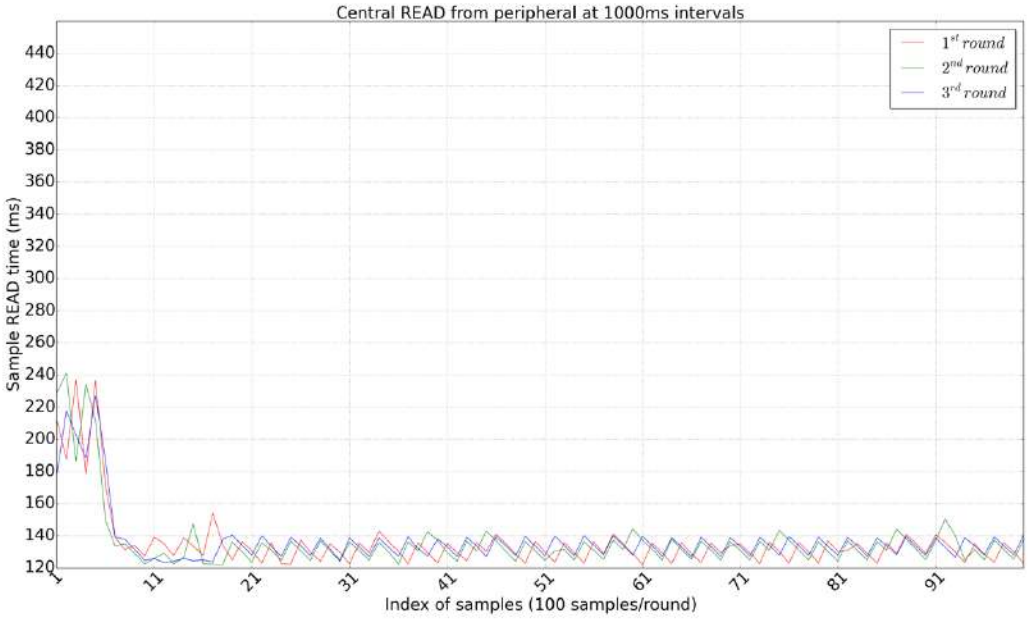

Figure 5-17. 100 sequential reads from an endpoint (1 second delay)

In figure 5-17, there is a wake-up period at the beginning of each round from sample #1 to roughly sample #5, and the trip time of wake-up period is in range of 180 to 240 milliseconds. The average trip time is at 130 milliseconds. The plot of BLE "read" operations looks very different from the one of BLE "write" operations.

In figure 5-18, "write" plus "read" operations are tested. It is expected that the average trip time is at around 330 milliseconds. The plot looks like overlaying figure 5-16 and figure 5-17.

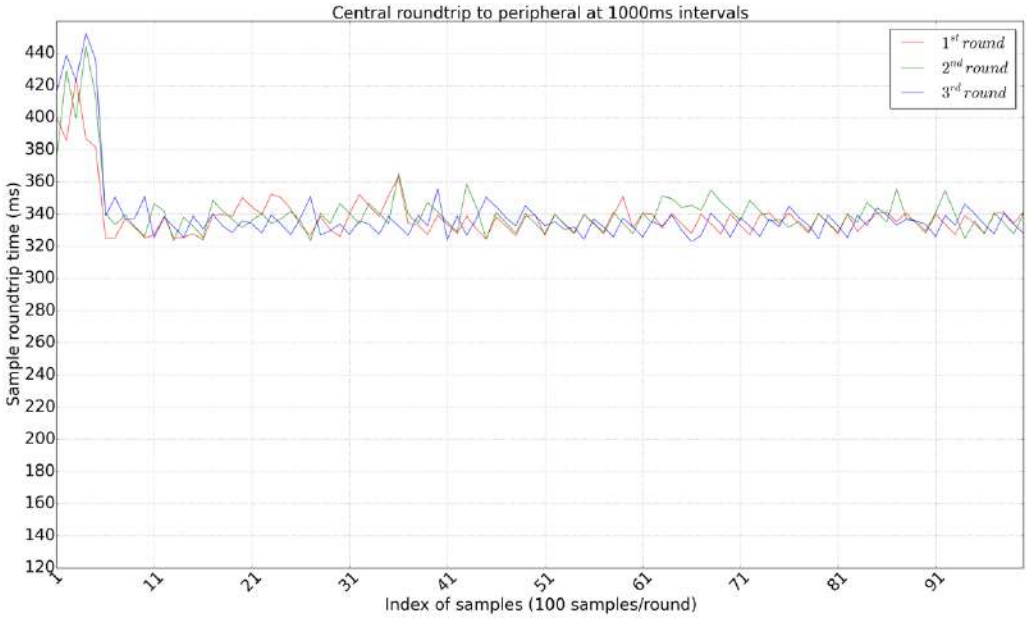

Figure 5-18. 100 sequential round trip to an endpoint (1 second delay)

To sum up, as results show that it requires around 200 milliseconds to change the state of the endpoint, while reading the state requires only around 130 milliseconds. It is observed that there is a wake-up period at the beginning of each round of reads, while writes do not have.

5.3 Summary

Generally, the three experiments indicate the performance of middleware (i.e. uniform interface) and script engine. In particular, the experiments of GET and POST show how virtual resources are presented in the middleware and their ability of handling loads. According to the experiments of both GET and POST, since the number of threads increases to ten, the plots become chaotic, and there must be a critical point at a certain point when further increasing the number of threads. It is not necessary to find out where their critical points exactly are, but it is important to know the trend of the performance of middleware. Also, these two experiments show the difference of middleware performance between when the data requested is cached and non-cached. Even though HTTP PUT and DELETE requests are applied to our case, we do not have to test them due to the similarity of HTTP POST requests.

The experiments of BLE writes and reads show the capability of handling computational expressions in Espruino. The experiment of BLE writes show fast trip time and stable

performance, whereas the experiment of BLE reads must start with a wake-up period and be followed by fluctuating but faster performance. The reasons of what plots look like need to be explored in the future.

After doing three experiments, we can make sure that it is possible to virtualize a resource-constraint physical device to be a middleware for providing uniform web-like interface, and a script engine onto a BLE-enabled endpoint works well to execute code sent through this uniform interface at the edge of the network.

CHAPTER 6

CONCLUSION

Centralized network can help IoT leverage data collected from heterogeneous endpoints. The traditional Cloud-centric system becomes hard to talk with Things because of the cost of high latency and high bandwidth requirement. However, Fog computing can address the problems by moving computational power closer to Things. IoT-Fog can provide real-time data processing and service provisioning, and dynamic resources management. Virtualization of resource constraint physicals allows Client to interact with physicals by software-defined networking.

This research proposed an architecture based on Service-Oriented Architecture that includes embedded middleware and script engine for low-energy endpoints. With the help of REST model, the architecture effectively addressed the challenges of service provisioning in virtualization environment and accessibility of virtual resources in PANs, and the state exchange onto virtual resources (i.e. endpoints) by Web applications. Particularly, the service provisioning relies on RESTful Web services (i.e. REST APIs) who provide uniform interface represented by URIs and CRUD mapping to HTTP methods. Furthermore, the research proofs that computational expressions are transferable to BLE-enabled virtual resources free of distance limitation via uniform web-like interface.

The experiments indicate that there is a decent concurrent performance in embedded middleware when handling HTTP requests from application and physical layers, which means that real-time REST Web services provisioning is not a problem in IoT-Fog when the number of thread is less than the critical point, and users can dynamically manage resources by using HTTP GET, POST, PUT, DELETE without significant network latency. The experiments also indicate that Espruino has a low interpretation latency at every single packet, which means that it enhances the capability of real-time processing in middleware, facilitates users' convenience for the ease of configurations, and working with BLE protocol stack reduces overhead of endpoint deployment in energy-saving manner. Additionally, Espruino is portable to other compatible resource constraint endpoints with small footprint. By our research, proposed architecture, implementation and experiment, we believe that a resource constraint physical is able to have required computing power at lower cost and / or lower energy to work in IoT environment.

CHAPTER 7

FUTURE WORK

The proposed architecture can be improved in the following aspects.

7.1 Decentralization with Access Control

Fog-centralized network is controlled by a single entity no matter how many proxy layers it involves. The most advantage of Fog-centralized network is service provisioning by its uniform interface under the control of a central. However, if the central server stops working, tracking will stop immediately that may cost some losses. Therefore, a distributed peer-to-peer network is a solution to avoid single point of failure. IBM and Samsung have unveiled proof-of-concept (PoC) for ADEPT (Autonomous Decentralized Peer-to-Peer Telemetry) fully distributed. According to IBM Blockchain [31], "Blockchain is a shared, immutable ledger for recording the history of transactions. It fosters a new generation of transactional applications that establish trust, accountability and transparency". However, Blockchain has underlying security and privacy issues. Access Control allows only eligible users can access resources by their roles and attributes.

A future work in middleware layer will employ Blockchain with Access Control to manage virtual resources in embedded Fog. In particular, if a user created, read, updated and deleted some data, Blockchain can help track who behaved what at when and where because Blockchain keeps the entire ledger that is shared to eligible nodes, which means that Blockchain never loses a chain of records because any other nodes are informed a copy of a transaction after validation. Eventually, Blockchain creates a ledger which only allows to append new records and track IoT devices through such manufacturing, transporting, deploying, and any other steps involved. Based on Blockchain, decentralized transaction processing among IoT devices will bring IoT management and configuration to a new world. In the future, we will work on building Blockchain-based decentralized solution in middleware layer, including peer-to-peer connection between nodes, efficient transaction processing, and the security of access, and finally we will need to know the performance of the middleware.

7.2 NFC (Near Field Communication)

A future work in physical layer focuses on a new communication protocol named NFC. According to Saeed et al. [30], “NFC is becoming widely more popular due to its ease of use, affordability, and extremely power efficient data communication”. Nowadays, most mobile phones are equipped with NFC technology. It is possible to tap onto an NFC tag to read sensor data and request to a central middleware. Fortunately, NFC can also work with IP networking in physical layer. Choi et al. [60] realize that NFC-based devices should use TCP/IP network for communication with BLE-enabled devices by adaptation layer and also for secure data transferring instead of by directly tapping on tags.

For example, there could be an IPv6 adaptation layer for NFC over NFC protocol stack as the same as an IPv6 adaptation layer for BLE over BLE protocol stack, and then network layer could be over IPv6 adaptation layer for internet connection. This approach could also be applied to application layer where users can use a mobile device to tap a tag and actuate IoT devices. In the future, we will need to know the performance of NFC communication, and we will compare the result to BLE communication performance. Finally, we can know which protocol is more efficient, secure and universal at cost-saving and energy-saving manner.

REFERENCES

[1] A. Arkatkar, K. Kudchadkar, H. Vaghela, K. Nawal, P. Gandhi, S. Gavali, . . . Y. Chen. (2016). *Communication Networks*. Retrieved from https://en.wikibooks.org/wiki/Communication_Networks/HTTP_Protocol

[2] A. Farahzadi, P. Shams, J. Rezazadeh and R. Farahbakhsh, "Middleware Technologies for Cloud of Things-a survey", *Digital Communications and Networks*, University of Technology Sydney, April 2017. doi: 10.1016/j.dcan.2017.04.005.

[3] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal and Q. Z. Sheng, "IoT Middleware: A Survey on Issues and Enabling Technologies," in *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 1-20, Feb. 2017. doi: 10.1109/JIOT.2016.2615180.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7582463&isnumber=7847463

[4] "Arduino Yún LininoOS." *Arduino Yún LininoOS*. N.p., 2016. Web. https://www.arduino.cc/en/Main/ArduinoBoardYun.

[5] A. Munir, P. Kansakar and S. U. Khan, "IFCIoT: Integrated Fog Cloud IoT: A novel architectural paradigm for the future Internet of Things.," in *IEEE Consumer Electronics Magazine*, vol. 6, no. 3, pp. 74-82, July 2017. doi: 10.1109/MCE.2017.2684981.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7948854&isnumber=7948824

[6] A. R. Biswas and R. Giaffreda, "IoT and cloud convergence: Opportunities and challenges," *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Seoul, 2014, pp. 375-376. doi: 10.1109/WF-IoT.2014.6803194.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6803194&isnumber=6803102

[7] ATMEL 8-BIT MICROCONTROLLER WITH 4/8/16/32KBYTES IN-SYSTEM PROGRAMMABLE FLASH DATASHEET. (2015). Retrieved from http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf

[8] A. V. Dastjerdi and R. Buyya, "Fog Computing: Helping the Internet of Things Realize Its Potential," in *Computer*, vol. 49, no. 8, pp. 112-116, Aug. 2016. doi: 10.1109/MC.2016.245.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7543455&isnumber=7543413

[9] B. Golden. (2008). *Virtualization for dummies*. Hoboken, NJ: Wiley Publishing, Inc.

[10] B. Kantarci and H. T. Mouftah, "Sensing services in cloud-centric Internet of Things: A survey, taxonomy and challenges," *2015 IEEE International Conference on Communication Workshop (ICCW)*, London, 2015, pp. 1865-1870. doi: 10.1109/ICCW.2015.7247452.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7247452&isnumber=7247062

[11] Bluetooth Low Energy. (2012). Retrieved from http://litepoint.com/whitepaper/Bluetooth%20Low%20Energy_WhitePaper.pdf

[12] Bluetooth SIG Generic Attributes (GATT) and the Generic Attribute Profile. (2017). Retrieved April, 2017, from https://www.bluetooth.com/specifications/gatt/generic-attributes-overview

[13] Bluetooth SIG Profile Overview. (2017). Retrieved April, 2017, from https://www.bluetooth.com/specifications/profiles-overview

[14] BTstack Protocols. (n.d.). Retrieved from https://bluekitchen-gmbh.com/btstack/protocols/

[15] B. Sirpatil. "Software Synthesis of SystemC Models." Thesis. Virginia Polytechnic Institute and State University, 2002. Print.

[16] B. Suda. (2003). *SOAP Web Services*. University of Edinburgh. Retrieved from http://suda.co.uk/publications/MSc/brian.suda.thesis.pdf

[17] B. T. de Oliveira, C. B. Margi and L. B. Gabriel, "TinySDN: Enabling Multiple Controllers for Software-Defined Wireless Sensor Networks", *2014 IEEE Latin-America Conference on Communications (LATINCOM)*, p 1-6, Nov 2014

[18] C. A. M. Duenas. Verification and test challenges in SoC designs. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design,* pages 9–9, New York, NY, USA, 2004. ACM.

[19] C. Huang and C. Wu, "A Web Service Protocol Realizing Interoperable Internet of Things Tasking Capability," *Sensors. 2016*; 16:1395. doi: 10.3390/s16091395.

[20] Cisco (2013). "An Introduction to the Internet of Things (IoT)". Retrieved from https://www.cisco.com/c/dam/en_us/solutions/trends/iot/introduction_to_IoT_november.pdf

[21] C. Kolias, A. Stavrou, J. Voas, I. Bojanova and R. Kuhn, "Learning Internet-of-Things Security Hands-on," in *IEEE Computer and Reliability Societies*, January/February 2016. URL: https://iot.ieee.org/images/files/pdf/LearningIoTSecurityJan2016.pdf

[22] Consumer Applications to Represent 63 Percent of Total IoT Applications in 2017. (2017). Retrieved from http://www.gartner.com/newsroom/id/3598917

[23] D. Tauchmann & A. Sikora. (2015). Experiences and Measurements with Bluetooth Low Energy (BLE) Enabled and Smartphone Controlled Embedded Applications. *International Journal of Electronics and Electrical Engineering*, 3, 4th ser., 292-296. doi:10.12720/ijeee.3.4.292-296.

[24] "ESP-12E WiFi Module." Shenzhen Anxinke Technology Co.,LTD, 2015. Web. https://mintbox.in/media/esp-12e.pdf.

[25] F. Belqasmi, J. Singh, S. B. Melhem & R. H. Glitho. (2012). SOAP-Based Web Services vs. RESTful Web Services for Multimedia Conferencing Applications: A Case Study. Retrieved from http://spectrum.library.concordia.ca/980040/1/PersonalCopy-SOAPvsREST.pdf

[26] "Genuino* 101 with Intel® CurieTM Module." N.p., 2016. Web. http://docs-europe.electrocomponents.com/webdocs/149d/0900766b8149d34c.pdf.

[27] "Getting Started with the Arduino Yún LininoOS." *Arduino Yún LininoOS*. N.p., 2016. Web. https://www.arduino.cc/en/Guide/ArduinoYunLin.

[28] G. Fersi, "Middleware for Internet of Things: A Study," *2015 International Conference on Distributed Computing in Sensor Systems*, Fortaleza, 2015, pp. 230-235. doi: 10.1109/DCOSS.2015.43.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7165049&isnumber=7164869

[29] H. Rahman. "Self-Organizing Logical-Clustering Topology for Managing Distributed Context Information." *Stockholm University*, 2015.

[30] H. Saeed, A. Shouman, M. Elfar, M. Shabka, S. Majumdar and C. Horng-Lung, "Near-field communication sensors and cloud-based smart restaurant management system," *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, Reston, VA, 2016, pp. 686-691. doi: 10.1109/WF-IoT.2016.7845440.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7845440&isnumber=7845389

[31] IBM and Samsung unveil ADEPT blockchain proof of concept for IoT. (January 23, 2015). April, 2017. Retrieved from http://rethinkresearch.biz/articles/ibm-samsung-unveil-adept-blockchain-proof-concept-iot-security/

[32] J. Gubbi, R. Buyya, S. Marusic and M. Palaniswami. "Internet of Things (IoT): A vision, architectural elements, and future directions." *Future Generation Computer Systems* 29, no. 7 (2013): 1645-1660.

[33] J. Kim, J. J. Jang and I. Y. Jung, "Near Real-Time Tracking of IoT Device Users," *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Chicago, IL, 2016, pp. 1085-1088. doi: 10.1109/IPDPSW.2016.218.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7529985&isnumber=7529833

[34] J. Man. "Raspberry Pi 3 Model B Technical Specifications." Element14Community, 2016. Web. https://www.element14.com/community/docs/DOC-80899/l/raspberry-pi-3-model-b-technical-specifications.

[35] J. R. Erenkrantz. (2009). Computational REST: A new model for decentralized, internet-scale applications (Order No. 3372349). Available from ProQuest Dissertations & Theses Global. (304850580). Retrieved from https://search.proquest.com/docview/304850580?accountid=14739

[36] J. Rykowski and D. Wilusz, "Comparison of architectures for service management in IoT and sensor networks by means of OSGi and REST services," *2014 Federated Conference on Computer Science and Information Systems*, Warsaw, 2014, pp. 1207-1214. doi: 10.15439/2014F324.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6933156&isnumber=6932982

[37] K. Ashton, "That 'Internet of Things' thing," *RFiD J.*, vol. 22, pp. 97–114, 2009.

[38] L. Atzoria, A. Ierab and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54 (15): 2787–2805, Oct. 2010. doi:10.1016/j.comnet.2010.05.010.

[39] M. Köhler, D. Wörner and F. Wortmann. (2014). Platforms for the internet of things–an analysis of existing solutions. *In 5th Bosch Conference on Systems and Software Engineering (BoCSE)*. Retrieved from http://cocoa.ethz.ch/downloads/2014/02/1682_20140212%20-%20Bocse.pdf

[40] M. Kranz, P. Holleis and A. Schmidt, "Embedded Interaction: Interacting with the Internet of Things," in *IEEE Internet Computing*, vol. 14, no. 2, pp. 46-53, March-April 2010. doi: 10.1109/MIC.2009.141.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5342400&isnumber=5427387

[41] M. Yannuzzi, R. Milito, R. Serral-Gracià, D. Montero and M. Nemirovsky, "Key ingredients in an IoT recipe: Fog Computing, Cloud computing, and more Fog Computing," *2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, Athens, 2014, pp. 325-329. doi: 10.1109/CAMAD.2014.7033259.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7033259&isnumber=7033190

[42] M. Z. Bjelica, N. Ignjatov, I. Papp and N. Teslic, "Device cloud platform with script based agents

for “anywhere access” applications development," *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, 2014, pp. 1061-1065. doi: 10.1109/MIPRO.2014.6859726.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6859726&isnumber=6859515

[43] O. Vermesan and P. Friess. Internet of Things – From Research and Innovation to Market Deployment. Denmark: River Publishers, 2014. Print.

[44] P. K. Potti. (2011). *On the Design of Web Service: SOAP vs. REST*. UNIVERSITY OF NORTH FLORIDA. Retrieved from http://digitalcommons.unf.edu/etd/138

[45] P. M. Julia and A. F. Skarmeta. White paper on “Extending the Internet of Things to IPv6 with Software Defined Net- working”

[46] "Product overview" N.p., 2016. Web. http://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.s130.sds%2Fdita%2Fsoftdevices%2Fs130%2Fs130sds.html

[47] Q. M. Ashraf, M. H. Habaebi, M. R. Islam and S. Khan, "Device discovery and configuration scheme for Internet of Things," *2016 International Conference on Intelligent Systems Engineering (ICISE)*, Islamabad, 2016, pp. 38-43. doi: 10.1109/INTELSE.2016.7475159.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7475159&isnumber=7475106

[48] Q. Xie, J. Liu and P. H. Chou, "Tapper: a lightweight scripting engine for highly constrained wireless sensor nodes," *2006 5th International Conference on Information Processing in Sensor Networks*, Nashville, TN, 2006, pp. 342-349. doi: 10.1145/1127777.1127829.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1662476&isnumber=34795

[49] "Raspberry Pi 3 Model B Technical Specifications." N.p., 2016. Web. http://docs-europe.electrocomponents.com/webdocs/14ba/0900766b814ba5fd.pdf.

[50] R. Mary. "Microcontrollers." N.p., 2010. Web.

[51] R. Morabito, "Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation," in *IEEE Access*, vol. 5, no. , pp. 8835-8850, 2017. doi: 10.1109/ACCESS.2017.2704444.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7930383&isnumber=7859429

[52] R. Sinha. *Automated Techniques for Formal Verification of SoCs*. Thesis. The University of Auckland, 2009. Auckland: ResearchSpace@Auckland, 2009. Print.

[53] S. A. Chowdhury, V. Sapra and A. Hindle (2015) Is HTTP/2 more energy efficient than HTTP/1.1 for mobile users? PeerJ PrePrints 3:e1280v1 https://doi.org/10.7287/peerj.preprints.1280v1

[54] S. Cherrier, Z. Movahedi and Y. M. Ghamri-Doudane, "Multi-tenancy in decentralised IoT," *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Milan, 2015, pp. 256-261. doi: 10.1109/WF-IoT.2015.7389062.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7389062&isnumber=7389012

[55] S. Hachem, A. Pathak and V. Issarny. (2014). Service-oriented middleware for large-scale mobile participatory sensing, Pervasive and Mobile Computing, Volume 10, Pages 66-82, ISSN 1574-1192, doi: 10.1016/j.pmcj.2013.10.010.

[56] Single chip microcontroller. (n.d.) *McGraw-Hill Dictionary of Scientific & Technical Terms, 6E.*

(2003). Retrieved April 2 2017 from
http://encyclopedia2.thefreedictionary.com/Single+chip+microcontroller

[57] S. Nastic, S. Sehic, D. H. Le, H. L. Truong and S. Dustdar, "Provisioning Software-Defined IoT Cloud Systems," *2014 International Conference on Future Internet of Things and Cloud*, Barcelona, 2014, pp. 288-295. doi: 10.1109/FiCloud.2014.52.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6984208&isnumber=6984143

[58] S. Perera (2015). Taxonomy of IoT Usecases: Seeing IoT Forest from the Trees. Retrieved from https://iwringer.wordpress.com/2015/10/08/taxonomy-of-iot-usecases-seeing-iot-forest-from-the-trees.

[59] T. Sansanayuth. (2015). *Development of a Wireless Interface for Fitting, Training, and Monitoring of Advanced Prosthetic Limbs*. CHALMERS UNIVERSITY OF TECHNOLOGY. Retrieved April, 2017, from http://publications.lib.chalmers.se/records/fulltext/219124/219124.pdf

[60] Y. Choi, Y. Choi, D. Kim and J. Park, "Scheme to guarantee IP continuity for NFC-based IoT networking," *2017 19th International Conference on Advanced Communication Technology (ICACT)*, Bongpyeong, 2017, pp. 695-698. doi: 10.23919/ICACT.2017.7890182.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7890182&isnumber=7890033

[61] Y. Liu, A.Y. Ding and S. Tarkoma, “Software-Defined Networking in Mobile Access Networks”, Department of Computer Science, Series of Publications C, no. C-2013-1, University of Helsinki, Department of Computer Science, Helsinki

[62] Y. Xu and A. Helal, "Scalable Cloud–Sensor Architecture for the Internet of Things," *in IEEE Internet of Things Journal*, vol. 3, no. 3, pp. 285-298, June 2016. doi: 10.1109/JIOT.2015.2455555.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7155463&isnumber=7467596

[63] Z. Shelby and C. Bormann, 6LoWPAN: The Wireless Embedded Internet, John Wiley & Sons, 2011.

[64] Z. Shelby, K. Hartke and C. Bormann. The constrained application protocol (coap). Technical report, 2014.