University of Calgary

PRISM Repository **https://prism.ucalgary.ca**

The Vault Open Theses and Dissertations

2014-05-02

Implementation and Evaluation of Interoperable Open Standards for the Internet of Things

Jazayeri, Seyyed Mohammad Ali

Jazayeri, S. M. (2014). Implementation and Evaluation of Interoperable Open Standards for the Internet of Things (Master's thesis, University of Calgary, Calgary, Canada). Retrieved from https://prism.ucalgary.ca. doi:10.11575/PRISM/26467

http://hdl.handle.net/11023/1482

Downloaded from PRISM Repository, University of Calgary

UNIVERSITY OF CALGARY

Implementation and Evaluation of Interoperable Open Standards for the Internet of Things

By

Seyyed Mohammad Ali Jazayeri

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF GEOMATICS ENGINEERING

CALGARY, ALBERTA

APRIL, 2014

© Seyyed Mohammad Ali Jazayeri 2014

Abstract

Recently, researchers focused on a new use of the Internet called *Internet of Things* (*IoT*), in which capable electronic devices can be remotely accessed over the Internet. All around the world, IoT applications are emerging exponentially with various functionalities in order to monitor and control the environment. For example, Wemo switch, Philips Hue light bulb, Ninja Blocks and Air Quality Egg are samples of the existing IoT applications which make environmental dynamics accessible via the Internet. Each application is developed based on the developer's desire of the device. That means the number of proprietary protocols is growing as the number of IoT devices increases. Moreover, IoT devices are intuitively heterogeneous in terms of the hardware capabilities and communication protocols. Therefore, ensuring interoperability is an important step to integrate various devices together. In this research, we focus on the communication challenges of the IoT objects to make the network suitable for a wide scale of IoT devices. To do this, we implement open standards in different communication layers on a resource constraint IoT object. The standard protocols developed in this research are OGC PUCK over Bluetooth, TinySOS (a lightweight profile of the OGC SOS), SOS over CoAP, and OGC SensorThings API. To the best of our knowledge, these implementations are the world's first contribution for the IoT objects. Eventually, we benchmark the efficiency of the implemented protocols by a comprehensive performance analysis in terms of memory occupation, request size, response length and response latency. As a result, by hosting the aforementioned open standard protocols on IoT devices, not only the devices become self-describable, self-contained, and interoperable, but also innovative applications can be simply developed by standardized interfaces.

Acknowledgements

There are a number of people without whom this thesis might not have been written, and to whom I am greatly indebted. My sense of gratitude to one and all who, directly or indirectly, have lent their helping hand in this venture.

First, I would like to thank my supervisor, Dr. Steve Liang, for encouraging my research and for giving me the opportunity to pursue my Masters degree in University of Calgary. Your advice on both research as well as on my career have been priceless.

I would also like to thank my committee members, Dr. Mea Wang, Dr. Xin Wang, and Dr. Andrew Hunter. I want to appreciate you for letting my defense be an enjoyable moment, and for your brilliant comments and suggestions.

I also give thanks to the entire research group, the GeoSensorweb Lab, for helping me with my research, and providing invaluable feedbacks. I owe Alec Huang thanks for reviewing my thesis, but more importantly for your academic support during my graduate degree. I give special thanks to Tania Khalafbeigi, Mahdi Oraei, and Reza Malek, who took it upon yourselves to help me in revising my thesis content.

I want to give a special thanks to my family including my parents, my siblings and my grandma. Your love and support have made it possible for me to excel in school and without you, I would not be where I am today. Words cannot express how grateful I am to the sacrifices that you have made on my behalf.

At the end, I wish to dedicate this thesis to the person I love and who has changed my life for the better in every way possible. To Setareh Sajadi, with all the love I have to give. You spent sleepless nights with and were always my support in the moments when there was no one to answer my queries. I will be grateful forever for your love.

Table of Contents

Abstract ii
Acknowledgements iii
Table of Contents iv
List of Tables vii
List of Figures and Illustrations viii
List of Symbols, Abbreviations and Nomenclature xi

CHAPTER ONE: INTRODUCTION 1
1.1 Background 1
1.2 Internet of Things 2
1.3 Existing IoT Applications 5
1.4 Interoperability 8
1.5 Problem Definition, Motivation and Solutions 10
1.6 Objective and Contributions 13
1.7 Development Platform 14
1.8 Definition of Terms 15
1.9 Thesis Organization 17

CHAPTER TWO: PUCK OVER BLUETOOTH 18
2.1 Introduction 18
2.2 Related Works 19
2.3 Architecture 20
2.3.1 OGC PUCK 20
2.3.2 Sensor Protocol 22
2.3.3 System Architecture 24
2.4 Implementation 25
2.4.1 Service Layer 25
2.4.2 Additional Software Components 26
2.5 Discussion 27
2.6 Summary 27

CHAPTER THREE: TINYSOS 29
3.1 Introduction 29
3.2 Related Works 32
3.3 Architecture 34
3.3.1 TinySOS 34
3.3.2 System Architecture 35
3.3.3 Resource Discovery 37
3.4 Implementation 39
3.4.1 Tiny Web Server 39
3.4.2 XPU Algorithms 41
3.5 Discussion 44
3.6 Summary 45

CHAPTER FOUR: SOS OVER COAP 47

4.1 Introduction....47
4.2 Related Works....49
4.3 Architecture....50
4.3.1 CoAP Specification....50
4.3.2 Device Architecture....53
4.3.3 SOS Integration to CoAP....54
4.4 Implementation....56
4.4.1 SOS Request to a CoAP Server....56
4.4.2 CoAP Request to a SOS Server....57
4.5 Discussion....57
4.6 Summary....58

CHAPTER FIVE: OGC SENSORTHINGS API....59
5.1 Introduction....59
5.2 Related Works....60
5.3 Architecture....62
5.3.1 API Components and Ecosystem....62
5.3.2 Data Model....63
5.3.3 System Architecture....64
5.4 Implementation....66
5.4.1 Device Registration....67
5.4.2 Observations Uploading....69
5.4.3 Actuator Tasking....69
5.5 Discussion....72
5.6 Summary....73

CHAPTER SIX: EVALUATION AND RESULTS....74
6.1 Introduction....74
6.2 Performance Evaluation....75
6.2.1 Memory Occupation....75
6.2.2 Request Size....77
6.2.3 Response Length....78
6.2.4 Response Latency....81
6.3 Summary....83

CHAPTER SEVEN: CONCLUSIONS AND FUTURE WORKS....84
7.1 Introduction....84
7.2 Conclusions....85
7.3 Future Works....87

REFERENCES....89

APPENDIX A....99
A.1 Sample requests and responses used in Section 6.2....99
A.1.1 Simple Web Server....99
A.1.2 PUCK over Bluetooth....100
A.1.3 TinySOS....101

A.1.4 SOS over CoAP 102
A.1.5 OGC SensorThings API 104

List of Tables

Table 2.1 Command set of the OGC PUCK 21

Algorithm 3.1 Naive pattern matching 42

Algorithm 3.2 Revised pattern matching 43

Table 4.1 Mapping SOS operations to CoAP requests 56

Table 4.2 CoAP responses to SOS requests 57

Table 5.1 Device registration procedres 67

Table 5.2 Response codes of the response engine 71

Table 6.1 RAM and ROM memory occupation 77

List of Figures and Illustrations

Figure 1.1 The Internet of Things scheme 3

Figure 1.2 Technical overview of the IoT [15] 4

Figure 1.3 Existing IoT applications: (a) Wemo switch(http://blessthisstuff.com); (b) Philips Hue light bulb (http://theverge.com); (c) Ninja Blocks (http://ninjablocks.com); (d) Air Quality Egg (http://airqualityegg.wikispaces.com) 6

Figure 1.4 Internet protocol graph [28] 10

Figure 1.5 The placement of PUCK, TinySOS, and CoAP in the IoT 13

Figure 1.6 Netduino Plus mainboard [31] 15

Figure 2.1 The overall workflow of accessing to the sensor measurements 19

Figure 2.2 PUCK memory 22

Figure 2.3 Procedures of the sensor protocol 23

Figure 2.4 The system architecture supporting PUCK protocol 24

Figure 2.5 The high level workflow of the service layer 26

Figure 3.1 The system architecture supporting TinySOS protocol [30] 36

Figure 3.2 Resource discovery process [30] 39

Figure 3.3 Code size comparison [30] 40

Figure 3.4 The high-level workflow of the XPU [30] 41

Figure 3.5 Using a SWE client to access a TinySOS device 44

Figure 4.1 High level view of the SOS over CoAP strategy 48

Figure 4.2 CoAP message format [55] 51

Figure 4.3 CoAP client-server interaction: (a) CON request; (b) NON request [55] 52

Figure 4.4 Empty ACK because of response deferral 53

Figure 4.5 The device architecture supporting CoAP protocol 53

Figure 4.6 The architecture of SOSCoAP Proxy 55

Figure 5.1 Ecosystem of the OGC SensorThings API 62

Figure 5.2 URI Components 63

Figure 5.3 Data Model 64

Figure 5.4 The device architecture supporting OGC SensorThings API 65

Figure 5.5 An example of registration request 68

Figure 5.6 An example of IoT service response 68

Figure 5.7 Tasking request triggered from user to IoT service 70

Figure 5.8 Workflow of the response engine 71

Figure 6.1 Different components of our development platform 74

Figure 6.2 Request size evaluation for the get observation request 78

Figure 6.3 Response size vs. the number of sensor readings 79

Figure 6.4 Response size vs. the number of sensor readings (removed the OGC SensorThings trend) 80

Figure 6.5 Response latency vs. the number of sensor readings 82

Figure 6.6 Response latency vs. the number of sensor readings (removed TinySOS) 83

Figure A.1.1 HTTP GET request to the simple web server 99

Figure A.1.2 Wireshark screenshot to analyze the requests sent to the simple web service 99

Figure A.1.3 GETREADING request and its response to a PUCK-enabled Netduino Plus through Bluetooth 100

Figure A.1.4 Statistics of the GETREADING request 100

Figure A.1.5 GetObservation request to TinySOS by using 52 North test client tool 101

Figure A.1.6 Wireshark screenshot to analyze the requests sent to TinySOS 101

Figure A.1.7 GetObservation request to the SOSCoAP proxy 102

Figure A.1.8 Get observation request sent to a CoAP server (*i.e.,* Netduino Plus) 102

Figure A.1.9 Wireshark screenshot to analyze the requests sent to the CoAP server 103

Figure A.1.10 Details of the get observation request sent to the CoAP server 103

Figure A.1.11 HTTP GET request/response to the OGC SensorThings 104

Figure A.1.12 Wireshark screenshot to analyze the request sent to the SensorThings service .. 104

Figure A.1.13 Response of the SensorThings to multiple readings request 105

Figure A.1.14 Summarized response of the SensorThings 106

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
API	Application Programming Interface
CoAP	Constraint Application Protocol
CPU	Central Processing Unit
EEPROM	Electronically Erasable Programmable Read Only Memory
EXI	Efficient XML Interchange
GPRS	General Packet Radio Service
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
IETF	Internet Engineering Task Force
IoT	Internet of Things
ITU	International Telecommunication Union
IP	Internet Protocol
JSON	JavaScript Object Notation
LAN	Local Area Network
MAC	Medium Access Control
MTU	Maximum Transmission Unit
M2M	Machine to Machine
OS	Operating System
OData	OASIS Open Data Protocol
OGC	Open Geospatial Consortium
OWS	OGC Web Services
O&M	Observation and Measurement
PC	Personal Computer
P2P	Peer-to-Peer
RAM	Random Access Memory
ROM	Read Only Memory
RFID	Radio Frequency Identification
SBC	Single Based Computer
SensorML	Sensor Modeling Language
SID	Sensor Interface Descriptors
SOS	Sensor Observation Service
SPOF	Single Point Of Failure
SPS	Sensor Planning Service
SSL	Secure Socket Layer
SWE	Sensor Web Enablement
SWG	Standards Working Group
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UPnP	Universal Plug and Play
URI	Uniform Resource Identification
URN	Uniform Resource Name
URL	Uniform Resource Locator

UUID	Unique Universal Identification
VGI	Volunteered Geographic Information
WSN	Wireless Sensor Network
XML	Extensible Markup Language
XPU	XML Processor Unit

Chapter One: **Introduction**

1.1 Background

The term “Sensor Web” was first used by Kevin Delin in 1997 [1] to describe a wireless sensor network (WSN) architecture where sensors can cooperate as a whole. Nowadays, we are witnessing the increasing number of deployments of WSNs and Sensor Web on the Earth. These networks consist of spatially distributed autonomous sensors, each of which is used to monitor the physical or environmental conditions (e.g., temperature, humidity, sound, motion, etc.) and to cooperatively pass their data through the network to a main location (end user) [2]. WSNs have been involved in many traditional applications, including habitats monitoring systems [3], environment observation systems [4], structure health monitoring systems [5], health applications [6], and fire emergency response systems [7].

Although the traditional monitoring systems (e.g., sensor networks) can provide precise and accurate measurements, the deployment of these systems is usually labour-intensive and challenging [8]. Therefore, a new paradigm called *Citizen Sensing* or *Volunteered Geographic Information* (*VGI*) has been proposed to involve the general public into the monitoring system [9, 10, 11]. With citizens measuring environmental properties voluntarily, scientists are able to observe the environment with a much higher spatial and temporal resolution. A key to realize the above citizen sensing vision is to empower citizens with low-cost and easy-to-use sensor systems. Similar to the fact that the affordable and user-friendly PC democratized computing [12], such cost-effective and easy-to-use sensor systems would be widely used in environmental monitoring.

One of the fundamentals of the Citizen Sensing vision is the Internet connectivity to provide online access to the sensor observations as they are measured [10]. Recently, researchers

focused on a new use of the Internet called *Internet of Things* (*IoT*), in which capable electronic devices can be remotely accessed over the Internet. Since the IoT is made of different kinds of objects, such device heterogeneity will pose challenges in terms of interoperability. Thus, the goal of this research is to address the interoperability issues of the IoT.

Among the existing Internet-enabled devices, *sensor* is one of the key enablers in the IoT paradigm [13]. For instance, sensors allow objects to *sense* the environment around them such as thermometer, water gauges, cameras, etc. Since the IoT and sensors are tightly integrated, the vision of the IoT and the *World Wide Sensor Web* [14] are similar. One immediate solution to the IoT interoperability challenge can be using the interoperable protocols for the sensor networks. The other solution can be to design a new specific protocol for the Internet of Things.

This chapter gives a brief presentation of the research topic by first introducing the Internet of Things paradigm and the existing progress in the IoT applications. Next, we define interoperability as a research motivation followed by problem definition and solutions. It then states the research objectives and our contributions to overcome the introduced problem. In addition, this chapter mentions our development platform, and a brief definition of the terms used in the next chapters. Lastly, the remaining chapters are outlined.

1.2 Internet of Things

The Internet connected services are growing rapidly. A great number of people use the Internet for web surfing, multimedia accessing, sending and receiving emails, playing games, shopping, social networking and many other daily tasks. Consequently, World Wide Web can intuitively be a good candidate to involve citizens into the sensing systems.

Therefore, the concept of Internet of Things (IoT) emerged as a networking infrastructure to interconnect electronic objects over the medium of the Internet. The prime goal of the IoT is to

capture the observations from sensors, control IoT devices, and finally make those devices easily available through the Internet. As illustrated in Figure 1.1, all electronic devices which are capable of Internet connectivity (e.g., sensors, actuators, machines, and computers) can be visited through the Internet browsers (e.g., web browsers and cell phone applications).

Figure 1.1 The Internet of Things scheme

To technically define IoT, we echo the definition provided by International Telecommunication Union (ITU) [15]: "Internet of Things is a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies". Figure 1.2 depicts this concept by mapping the physical world to the digital world across communication networks. According to this technical viewpoint, IoT would certainly affect on different aspects of the potential user's life and behaviour. For example, assisted living, e-health, enhanced learning, automation and industrial manufacturing, and intelligent transportation

systems are only a few examples of possible application scenarios in which the new paradigm will play a leading role in the near future.

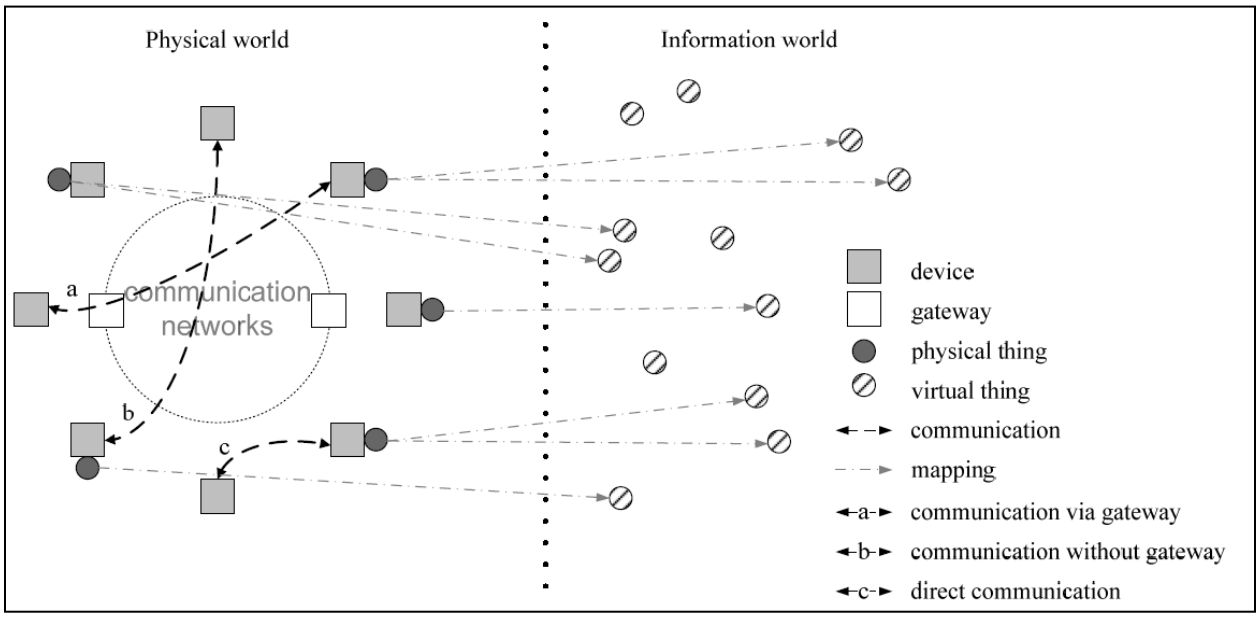

Figure 1.2 Technical overview of the IoT [15]

Referring to the definition provided by ITU, a *Thing* is also described as a uniquely identifiable instance of the physical world or the information world, which can be integrated into communication networks [15]. In this research, a Thing denotes a physical device in the physical world with the mandatory capabilities of communication and the optional features of sensing, actuation, data capture, data storage and data processing. Bormann et al. [16] worked within IETF[1] analyzed and categorized IoT objects into three categories with respect to their communication capabilities: *class-0* devices (*i.e.*, impossibly limited devices), *class-1* devices (*i.e.*, devices with about 10 Kbytes of RAM and 100 Kbytes of code space), and *class-2* devices (*i.e.*, devices with about 50 Kbytes of RAM and 250 Kbytes of code space). Bormann et al. [16] argue that the class-0 devices require extra help to communicate with other devices; the class-1

[1] IETF: Internet Engineering Task Force

devices cannot easily communicate with other devices or applications through traditional XML-data representations and protocols (e.g., HTTP and Transport Layer Security (TLS)); and the class-2 devices are able to communicate with the traditional transfer protocols and data encodings. Based on these arguments, we focus on the relatively inexpensive class-1 IoT devices in this research. Thus, this approach allows us to explore the lower bound of the resources that are required for IoT applications. In that way, we ensure that our design choices can deliver an efficient implementation suitable for a broader application domain.

Here, we identify and emphasis on two of the major issues of the Internet of Things. First, since objects in the IoT act independently, each IoT object needs to be *self-describable* and *self-contained* in order to communicate with other objects or sensors. That is, a Thing should be able to describe and advertise both itself and its capabilities, which in general is the metadata of the Thing. Second, since objects are developed to satisfy a particular need, their communication protocols and data encodings are usually different from each other. This *heterogeneity* consequently obstructs the communication and cooperation between objects. Besides the two aforementioned issues, there are other issues in the IoT, such as limited power supply, privacy, and security concerns. While these issues are important, we do not address them here as they are out of the scope of this research.

1.3 Existing IoT Applications

IoT projects are dramatically growing in different areas, specifically in energy optimization. Ericsson and Cisco IBSG[2] predicted there will be 25 billion Internet-connected devices by 2015

[2] Internet Business Solutions Group

and more than 50 billion by 2020 [17, 18]. Some of the available IoT projects recently released to the IoT market are shown in Figure 1.3.

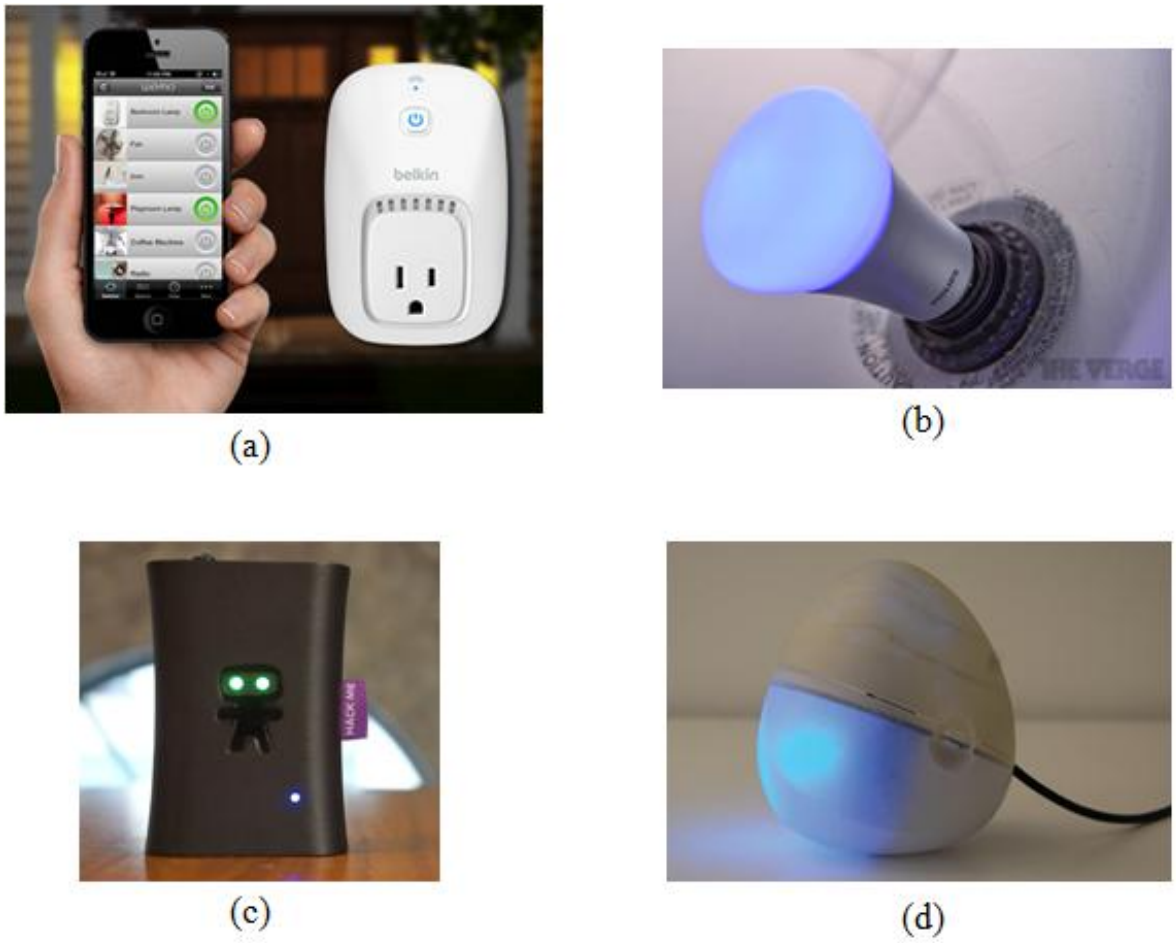

Figure 1.3 Existing IoT applications: (a) Wemo switch(http://blessthisstuff.com); (b) Philips Hue light bulb (http://theverge.com); (c) Ninja Blocks (http://ninjablocks.com); (d) Air Quality Egg (http://airqualityegg.wikispaces.com)

Traditionally, network peripherals have not been easy to install. Recent standards such as *Universal Serial Bus* (*USB*) and *Plug-and-Play*[3] have improved the situation so that devices are

[3] http://www.pcguide.com/ref/mbsys/res/pnp-c.html

automatically detected and device drivers are automatically installed. Yet, networked devices, such as Internet gateways and networked printers still require manual setup and configuration.

Wemo switch[4] (Figure 1.3(a)) lets electronic devices to be remotely turned on/off. It is an IoT application that follows the *Universal Plug and Play* (*UPnP*) [19] protocol in its communication. Using UPnP, when a device is plugged in and turned on, it "just works". However, the Wemo application has to be installed on an Android or iOS device, in order to transfer the network settings to the Wemo switch. Furthermore, the Wemo switch cannot be controlled from outside the network it exists (*i.e.*, Internet).

Philips Hue light bulb[5] (Figure 1.3(b)) is a wireless light which can display different tones of white light from warm yellow white to vibrant blue white. The Hue light bulb works similarly by means of a Hue router as a bridge between the bulb and the Hue app. Later on, it is possible to talk to the light bulb by Hue bridge across the Internet. Furthermore, the Hue application and Hue bridge are required to support the remote communication to the device.

Ninja Blocks[6] (Figure 1.3(c)) are cloud-enabled components including sensors (e.g., temperature, humidity, motion, window and door contact) and actuators (e.g., lights, power sockets) to monitor and control the environment. Ninja Blocks are more accessible than the two previous IoT apps by integrating the Ninja Block to Ninja clouds on the Internet. The connection between the Ninja Block and Ninja clouds is established automatically based on an API which has already hard-coded on the Ninja Blocks. The cloud also provides a web interface for the clients to aggregate sensor data in a repository. However, if a new device is added to the

[4] http://www.belkin.com
[5] https://www.meethue.com
[6] http://ninjablocks.com

network, the device needs to follow the Ninja cloud API to be able to interact with the Ninja server.

Air Quality Egg[7] (Figure 1.3(d)) is an air quality monitoring system which provides online access to its observations. The egg is composed of a sensing device that measures the air quality in the environment and a gateway that shares the collected data in real-time. The Air Quality Egg immediately uploads the collected data to an open database service named *Xively*[8] (formerly Cosm and before that Pachube). Although Xively provides online access to the sensor-derived data, users can register their egg at the Air Quality Egg portal[9] to visualize the data on a map. Similar to the Ninja Blocks, the Air Quality Egg follows the robust API provided by Xively.

Accordingly, IoT is creating innovative applications by assembling the IoT sensing and controlling capabilities from different sources in effective ways. However, IoT service providers are developing their own proprietary software interfaces for their devices. As mentioned above, even these four instances of IoT applications do not apply the same protocol, application, and communication style in data exchange. This means the number of proprietary interfaces is growing as the number of IoT devices increases. Consequently, an effort is required to interconnect various IoT devices with a shared interface to be globally accessed on the Internet.

1.4 Interoperability

Towards the first issue mentioned in Section 1.2, devices should somehow provide web services to advertise the devices capabilities and information in the network. For the second issue, the devices need to be interoperable in their communications. Based on the IEEE definition [20],

[7] http://airqualityegg.com
[8] https://xively.com
[9] http://airqualityegg.com/

syntactic interoperability means the ability of interoperation and information exchange in a system; that is, devices should be able to interactively communicate with a common protocol and data format. Beyond the syntactic definition of the IEEE, devices should exhibit *semantic interoperability* as well. To clarify, semantically interoperable devices can interpret the exchanged data, and generate meaningful result which is understandable by both sides. Although interoperability has a broader scope, we focus on the syntactic and semantic interoperability in this research.

According to Rodriguez et al. [21], Sensor Web and WSNs play an important role in the IoT. In order to provide global interoperability for all IoT devices, we point to the open standard interfaces defined for WSNs. One of the pioneers in the standardization of WSNs is the *Open Geospatial Consortium* (*OGC*). OGC has been supporting geospatial interoperability since 1994. Among all OGC standards, the Sensor Web Enablement (SWE) is a suite of standards to enable sensor network interoperability. SWE standards include *Observations & Measurements* (*O&M*) [22], *Sensor Model Language* (*SensorML*) [23], *Sensor Interface Descriptors (SID)* [24], *Sensor Observation Service* (*SOS*) [25], *Sensor Planning Service* (*SPS*) [26], and *PUCK* protocol [27]. As a result, one possible solution to achieve the interoperable IoT is the development of these OGC open standards on the IoT devices. Therefore, one of the objectives is to implement the suitable SWE standards for IoT devices, and then we can evaluate whether the SWE standards are suitable for IoT devices or not. In addition, as the SWE standards are designed for scientific grade sensor systems rather than for resource-constrained low-cost IoT devices, there might be a need to define a specific standard for the IoT objects.

1.5 Problem Definition, Motivation and Solutions

The information communication, including tasking IoT objects and retrieval of spatio-temporal observations from distributed IoT devices, is a key function in the Internet of Things. The traditional interaction models in the Internet are based on the request/response communication style between network entities. The *Internet protocol suite* is the networking model for the Internet which contains four layers: *Application Layer*, *Transport Layer*, *Internet Layer*, and *Link Layer* [28] as depicted in Figure 1.4.

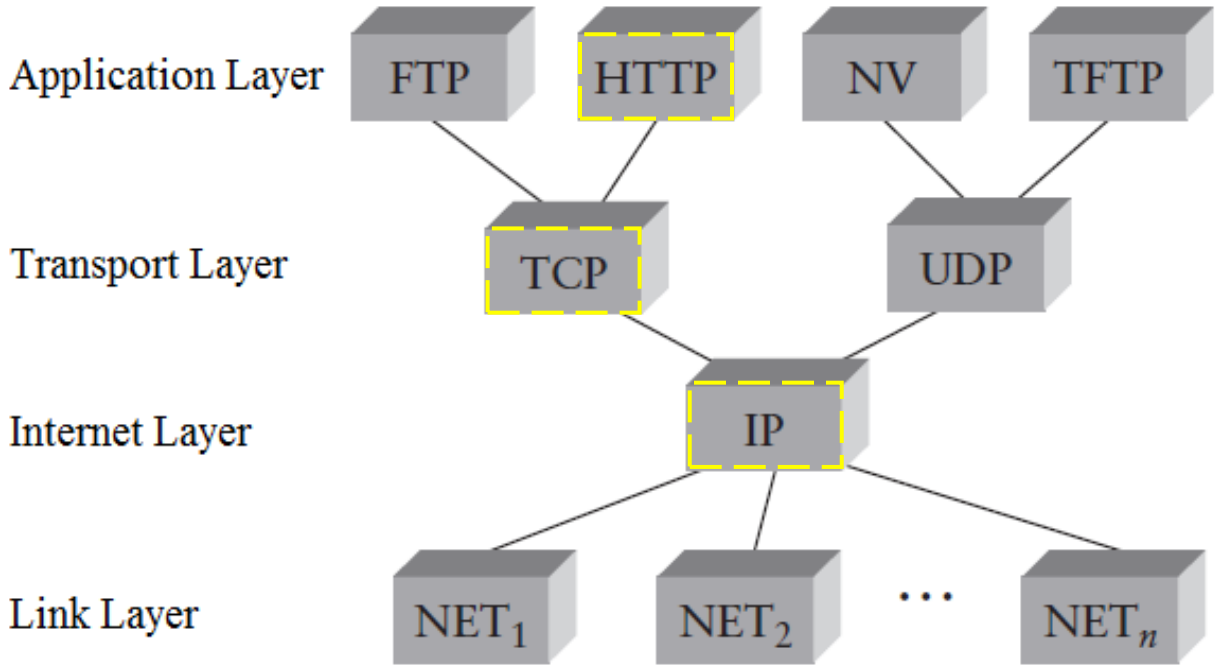

Figure 1.4 Internet protocol graph [28]

Other than the Link Layer which is significantly related to hardware equipments, *Internet Protocol* (*IP*) in the Internet Layer, *Transmission Control Protocol* (*TCP*) in the Transport Layer and *Hyper Text Transfer Protocol* (*HTTP*) in the Application Layer are mostly used to communicate between computers on the World Wide Web. Therefore, a prominent candidate for the IoT would be the WWW protocols that are very scalable, robust, and ubiquitous [30].

However, the existing Internet protocol suite may not be appropriate for the IoT because of the following reasons:

1. *Internet access problem*: Many IoT devices are not capable of accessing the Internet. This problem occurs because IoT devices do not meet the hardware requirements in order to connect to the Internet or do not have the stable power supply to be continuously connected to the network. Consequently, IoT should be flexible in integrating the Internet protocol with other protocols in order to make those devices available on the Internet. Involving new protocol(s) to the available network infrastructure typically requires new software and hardware requirements (e.g., gateway) to facilitate the seamless integration of those devices with mobile communication networks or Internet. For example, when a Hue light bulb is going to communicate with a client (e.g., a web browser); the Hue bridge needs to be installed on the network.
2. *Lack of standard protocol for IoT data representation*: HTTP is a foundation of data communication for the World Wide Web to transfer hypertext across the Internet. However, data representation considerably differs from one device to another because there is no standard defining the data representation in the IoT. For example, *plain text*, *Hypertext Markup Language* (*HTML*), *JavaScript Object Notation* (*JSON*), and *Extensible Markup Language* (*XML*) are possible response formats. In order to supply interoperable access between heterogeneous IoT objects, we need to define a standard protocol on top or in parallel of HTTP.
3. *Constrained resources of IoT devices:* As we already mentioned in Section 1.2, we focus on resource limited class-1 objects in this research. The Internet protocol suite itself might be inappropriate for the constrained network or objects of the Internet of Things.

Therefore, efficient and compressed data encodings in terms of computational resource consumption (e.g., memory, CPU, bandwidth) are required for these devices.

To alleviate the above deficiencies for the existing Internet protocols of the World Wide Web, an alternative standard protocol(s) needs to be considered. The protocol(s) should be effectively compliant with the requirements of IoT participants (*i.e.*, objects, users, applications, networks, gateways, proxies) in order to make the IoT devices interoperable in the network. The selected protocol(s) can encourage people to participate in the IoT by connecting their sensors and actuators to the network. In this case, IoT will provide real-time sensor data streams with a much higher spatial and temporal resolution. Also, end users can remotely command their daily devices by means of web browsers or mobile applications.

As OGC SOS is a commonly-used web service interface in the Sensor Web, we first connect users to IoT devices based on that protocol. This connection may be established directly through TinySOS protocol [30] which is a compressed implementation of the OGC SOS [25] on the IoT objects. On the other hand, the connection protocol to the device can be modified to OGC PUCK [27] or CoAP [16] which requires intermediary nodes (*i.e.*, proxy, gateway) for protocol conversions (Figure 1.5). The OGC PUCK provides access to the driver code, installation procedures, communication port configuration, and metadata of the device. The CoAP also employs the basic features of HTTP to the constrained network while maintaining a low overhead.

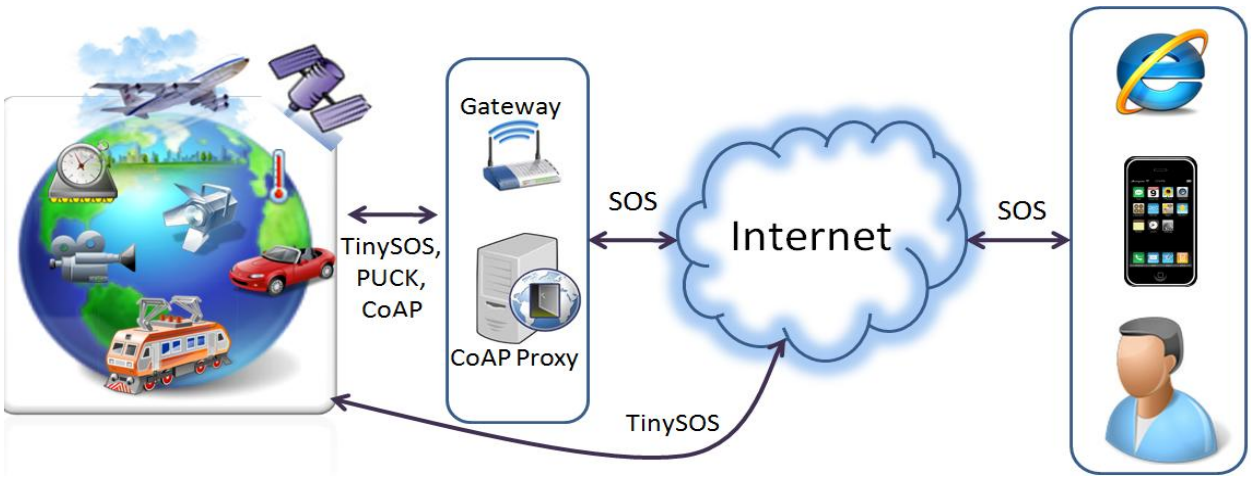

Figure 1.5 The placement of PUCK, TinySOS, and CoAP in the IoT

In addition to using the existing standards for IoT, there is an ongoing effort of defining a standard *Web Application Programming Interface* (*API*) for the IoT. This API, namely *OGC SensorThings*, is built on HTTP protocols and applies the widely-used *Representational State Transfer* (*REST*) style to access the system's components.

1.6 Objective and Contributions

The main objective of this research is to address the IoT interoperability issues. To achieve this major goal, we first investigate the current progress on this aspect of the IoT. Then, we implement four standard protocols on a class-1 IoT device including PUCK over Bluetooth, TinySOS, SOS over CoAP, and OGC SensorThings. Finally, we evaluate the four different protocols. To summarize, this thesis makes the following contributions:

- We implement the OGC PUCK on a Bluetooth-enabled class-1 IoT object. To make the sensor data available on the Internet, we also integrate the OGC SOS protocol with the PUCK-enabled IoT object.

- We implement the heavy-weight OGC SOS and SensorML standards on a resource-constrained sensor (class-1 IoT device). In order to overcome the hardware constraints, we introduce an efficient XML parser algorithm.
- To interconnect a CoAP-enabled IoT object with other sensors on the Web, we integrate this protocol to other standards of the WSNs (e.g., OGC SOS) as an interoperable infrastructure for the IoT. Therefore, we implement the commonly-used SOS standard over CoAP on a CoAP proxy which has enough computational resources.
- We design a specific RESTful protocol for the Internet of Things called OGC SensorThings API which communicates with IoT objects based on their own defined protocols.
- At the end, we complete our contributions by evaluating the performance of the four aforementioned protocols (*i.e.*, PUCK over Bluetooth, TinySOS, SOS over CoAP, and OGC SensorThings) in terms of memory occupation (ROM and RAM), request length, response size and response latency.

The major contribution of this research is to explore the possible approaches to achieve interoperability between class-1 IoT objects. Furthermore, we expect that the direction addressed in this research can be a motive to establish a better infrastructure for the future of IoT.

1.7 Development Platform

In this research, we use a sensor compatible *Single Board Computer* (*SBC*), namely *Netduino Plus*. This electronic framework is a low-price (59$) open source hardware platform built by *Secret Labs Company* [31]. The board features a 32-bit Atmel microcontroller with 48 MHz speed, 28 Kbytes main memory (*i.e.*, *RAM*), and 64 Kbytes code storage. In this case, Netduino Plus belongs to the class-1 device category in the framework of Bormann et al. [16].

Furthermore, Netduino Plus supports micro SD memory (up to 4 GB) as a permanent memory to store necessary information such as configuration files, capabilities document, sensor observations, etc. The network connectivity of the board is established by an Ethernet cable, but its mainboard can support other network alternatives (e.g., Wi-Fi, Bluetooth, Zigbee, and GPRS). As shown in Figure 1.6, the mainboard also supports 20 I/O pins (14 digital and 6 analog) where sensors and actuators can be simply attached to. From the software viewpoint, codes developed on this device should be written in C# .Net Micro Framework. Netduino Plus can run the codes directly without any needs for operating systems (OS).

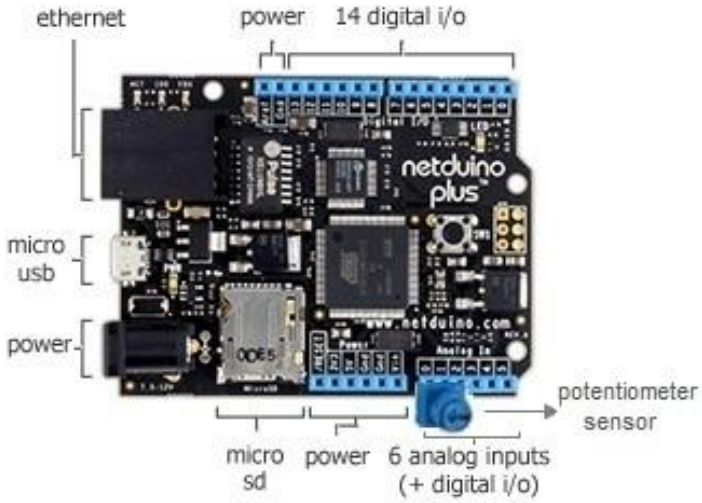

Figure 1.6 Netduino Plus mainboard [31]

1.8 Definition of Terms

For clearer understanding of the terms used in this study, terms and their definitions are as follows:

Actuator- It refers to a transducer that accepts an electrical signal and converts it into a physical action [32].

Feature of Interest- This describes a feature (so a representation of a real-world object) that carries the property which is observed. This can be either a domain feature (a.k.a. sampled feature) such as “Mississippi”, or a sampling feature like “water gage X" at Mississippi river. [33]

Gateway- It refers to a device used to connect two different networks, especially a connection to the Internet [29].

Observation Offering- It groups collection of observations which are somehow similar such as the observations produced by a specific procedure. [25].

Observed Property- Facet or attribute of an object referenced by a name which is observed by a procedure [25].

Phenomenon- It is an event in the real world which will be measured. A phenomenon may be a physical property (such as temperature, length, etc.), a classification (such as species), frequency or count, or an existence indication [34].

Procedure- This involves method, algorithm, instrument, sensor, or system of these which may be used in making an observation [25].

Proxy server- In computer networks, a proxy server is a server (a computer system or an application) that acts as an intermediary for requests from clients seeking resources from other servers. A client connects to a proxy server, requesting some services, such as a file, connection, web page, or other resources available from a different server. Then, the proxy server evaluates the request as a way to simplify and control its complexity. Proxies were invented to add structure and encapsulation to distributed systems [35].

Sensor- It is an entity that provides information about an observed property as its output. A sensor uses a combination of physical, chemical or biological means in order to estimate the

underlying observed property. At the end of the measuring chain, electronic devices produce signals to be processed [25].

Sensor Web Enablement (SWE) - Among the OGC working groups, SWE focuses on integrating sensors, transducers, and sensor data storages discoverable, accessible and useable via the Web. The OGS SWE standards include: Sensor Observation Service (SOS), Sensor Planning Service (SPS), PUCK, Sensor Model Language (SensorML), and Observations & Measurements (O&M) [36].

1.9 Thesis Organization

Chapter 2, 3, 4, and 5 overview PUCK over Bluetooth, TinySOS, SOS over CoAP, and OGC SensorThings API, respectively. Therefore, Chapter 2 to Chapter 5 will independently explain a specific protocol, each of which contains introduction, literature review, architecture, methodology, discussion, and summary sections. Then, Chapter 6 evaluates our implementations by comparing the four protocols in terms of performance analysis. Finally, conclusions and future work are given in Chapter 7.

Chapter Two: **PUCK over Bluetooth**

2.1 Introduction

Among the large scope of OGC standards, we first choose PUCK which is a simple command protocol. The PUCK contains a set of standard commands to access the device memory, read the device metadata, and write data on the memory. The prime purpose of the OGC PUCK is to provide interoperability for devices connected through serial cables or Ethernet. In order to enable sensors to be accessible via wireless connections, we analyze possible radio communication technologies. The choice of the radio highly matters since it influences either energy consumption or software design. Comparing to Zigbee and RF transceiver alternatives applied in WSNs or Sensor Webs, Bluetooth is more popular because it has been widely supported by many daily devices (e.g., cell phone and notebook). In addition, Bluetooth is more energy-efficient in comparison with Wi-Fi. Therefore, we integrate the Bluetooth protocols to the PUCK standard in order to raise the interoperability between various types of sensors and actuators, namely IoT devices.

PUCK standard is efficiently designed to be applied on devices supporting different protocols. It considers two modes: *PUCK mode* for processing the PUCK commands, and *instrument mode* for handling instrument-specific operations. Since the PUCK itself has no support for retrieving and publishing the sensor measurements on the Internet, we use other OGC standards, SID and SOS, to provide users the access to the measurements. The workflow is shown in Figure 2.1 and is elaborated in Section 2.4.

Figure 2.1 The overall workflow of accessing to the sensor measurements

To wrap up, the first contribution of this chapter is that we initially enable sensors to be accessible through Bluetooth technology. Then, we integrate Bluetooth protocol and PUCK as an open standard wireless protocol to raise the interoperability of IoT devices.

The remainder of this chapter is organized as follows. In Section 2.2, literature review and related works are stated. Section 2.3 and Section 2.4 present the proposed architecture and implementation, respectively. In Section 2.5, we discuss about the PUCK over Bluetooth idea and its consequent issues. Finally, a summary about this chapter is offered in Section 2.6.

2.2 Related Works

Bluetooth has already been utilized in Sensor Web [37] to let sensors upload their readings to a data repository. Leopard et al. [38] achieved this by introducing a tiny Bluetooth stack that allows TinyOS [39] applications to be executed on Bluetooth enabled sensor nodes. While

Leopard et al. [38] focused on the efficient network processing and system architecture design, their research did not consider the interoperability issues between various sensors.

Since the Bluetooth radio range is over a couple of meters [40], the system developed by Leopard et al. [38] does not provide the world wide access to the sensor measurements. To overcome this problem, Ferrari et al. [41] proposed a new architecture for the sensor networks to integrate the Bluetooth-enabled sensors with Internet-connected computers. As a result, these Bluetooth-enabled sensors are essentially connected to the Internet. Although this implementation successfully demonstrated the possibility of combining Bluetooth sensor nodes to the web interfaces, the communication protocol between sensors and computers was proprietary and did not consider the interoperability issues.

Nevertheless, to the best of our knowledge, there is no standard protocol based on the Bluetooth that enables embedded sensors and IoT devices to be connected in an interoperable manner. Therefore, we believe that the integration of Bluetooth and OGC standards for IoT devices that this chapter presents is a pioneer in this field.

2.3 Architecture

Here, we briefly introduce the OGC PUCK protocol. Next, we explain the sensor protocol for retrieving sensor observations from the device. Finally, we present the high-level architecture of our proposed system.

2.3.1 OGC PUCK

The PUCK protocol provides access to the driver code, installation procedures, communication port configuration, command protocol, and metadata such as OGC SensorML. In general, this standard protocol mainly consists of two parts: *PUCK commands*, and *PUCK memory*.

- *PUCK commands*: The protocol has a command-response style in which commands are considered as ASCII strings. Upon successful execution, the device executing PUCK protocol will return the characters: PUCKRDY<CR>. If the PUCK-enabled instrument is unable to execute a command successfully, it will issue a specific error. Table 2.1 shows a summary of the PUCK commands.

Table 2.1 Command set of the OGC PUCK

Command	Description
PUCKRM	Read from PUCK memory
PUCKWM	Write to PUCK memory
PUCKFM	End PUCK write session
PUCKEM	Erase PUCK memory
PUCKGA	Get address of PUCK internal memory pointer
PUCKSA	Set address of PUCK internal memory pointer
PUCKSZ	Get the size of PUCK memory
PUCKTY	Query PUCK type
PUCKVR	Get PUCK version string
PUCK	Null command
PUCKIM	Put PUCK into instrument mode
PUCKVB	Verify baud rate support
PUCKSB	Set PUCK-enabled instrument baud rate

- *PUCK Memory*: PUCK memory provides a space for device information, and a *memory pointer* referring to the memory address that will be read or written by the relevant PUCK commands. Figure 2.2 indicates partitions of the PUCK memory which is mainly divided into two parts: *PUCK datasheet* and *PUCK payload*. PUCK datasheet contains a small standard datasheet including a *Universally Unique Identifier* (*UUID*), manufacturer ID, PUCK version, header size, and several device related information such as name, version, model ID, and serial number. On the other hand, the optional PUCK payload stores additional information needed to operate the device such as device driver code, SensorML, and so forth.

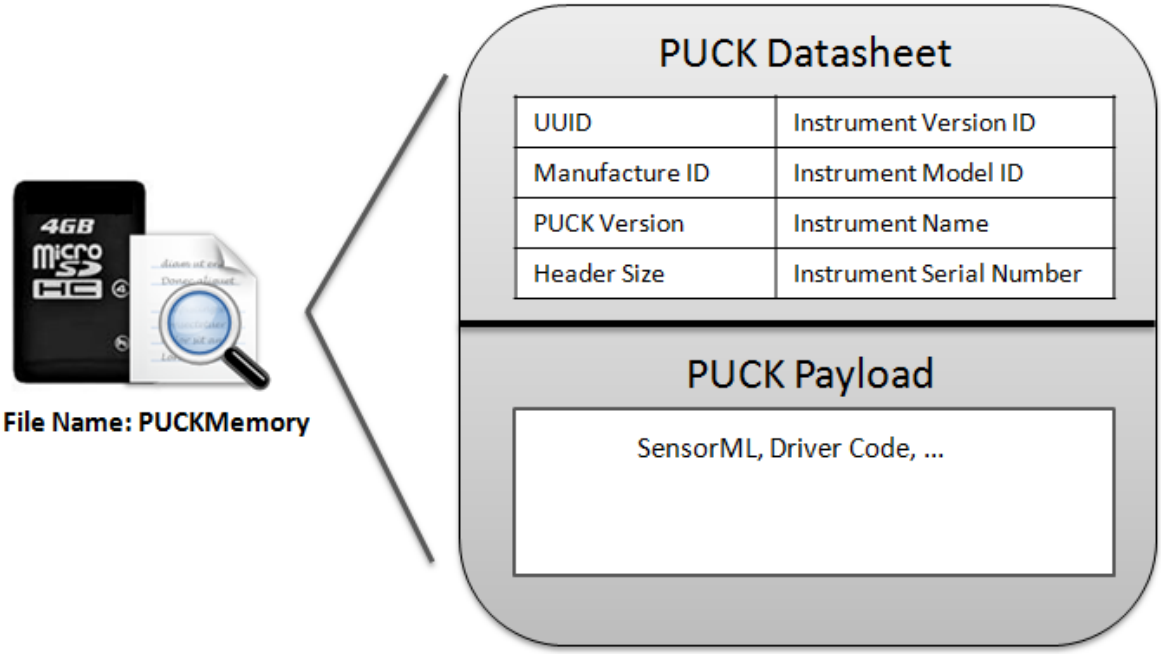

Figure 2.2 PUCK memory

2.3.2 Sensor Protocol

The purpose of sensor protocol is to allow users to simply query sensor capabilities, observations, and presentations of observed features in the instrument mode. As the device we

used in this research does not provide any sensor protocol, we define a protocol based on the concept of OGC SOS [25] to serve the demonstration purpose. Most of the terms used in this part follow the terminology in the OGC SOS. Because of the limited resources in IoT instruments, the command and response formats should be considered as simple as possible. Therefore, unlike the SOS applying XML as the format, this protocol simply defines "separators" (e.g., {#, :, |}) to format requests and responses (Figure 2.3). Similar to the OGC SOS, we define *GETCAPABILITIES* operation in order to show the capabilities of the device. The response includes the unique IDs of the sensors attached to the device, the phenomena IDs which are measured by the sensors, and the unit of measurements. Next, the other operation, *GETREADING*, can be sent to retrieve sensor readings. Figure 2.3 depicts the procedures of the sensor protocol.

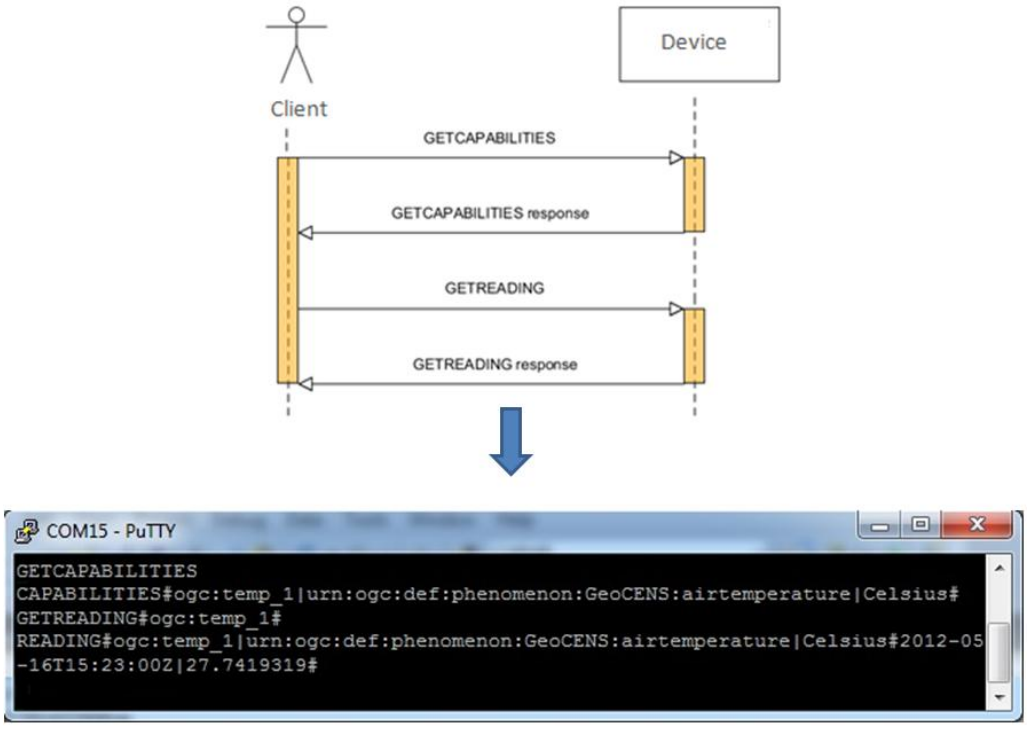

Figure 2.3 Procedures of the sensor protocol

2.3.3 System Architecture

As shown in Figure 2.4, the architecture we proposed for the device follows a layered structure which has three major layers: *Communication Layer*, *Service Layer*, and *Sensor Layer*.

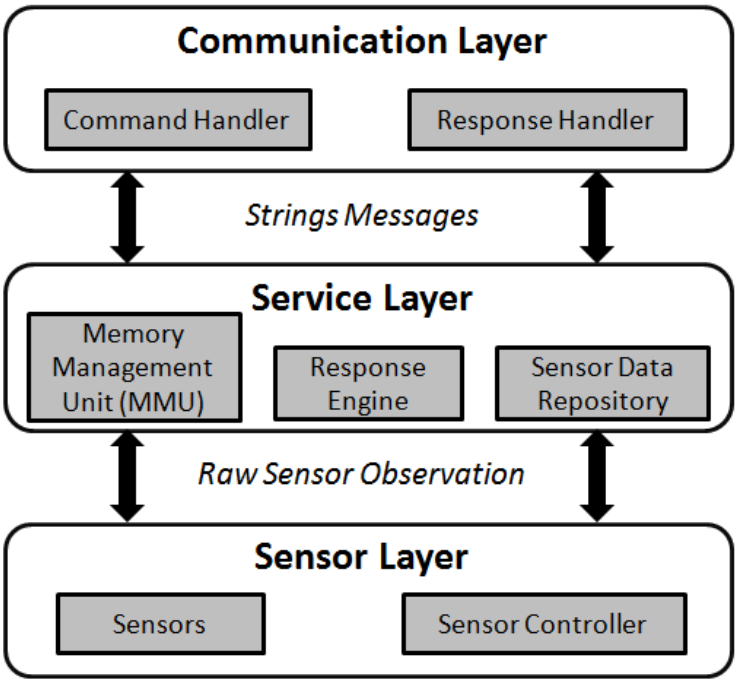

Figure 2.4 The system architecture supporting PUCK protocol

- *Communication Layer*: This layer includes the Bluetooth hardware and its protocol. When a request is received, the layer forwards the request string to the service layer for processing. After the service layer finishes processing the request, a response string is returned to the communication layer to send back to the client.
- *Service Layer*: The service layer handles business logic of the system. This layer itself consists of three modules: *sensor data repository*, *response engine*, and *memory management unit* (*MMU*). More details about the service layer are presented in Section 2.4.

- *Sensor Layer*: The sensor layer consists of the physical sensors and the sensor controller. The sensor controller tasks sensors to collect sensor observations. Next, it sends the retrieved sensor observations to the sensor data repository of the service layer.

2.4 Implementation

In this section, we explain the service layer in detail. Then, we introduce the required software components to connect the IoT device to the Internet.

2.4.1 Service Layer

In order to parse the commands and compose response messages on the small memory of IoT devices, we propose the response engine. This unit is equipped with a *buffering* mechanism to handle the large contents. By the way, the maximum memory consumption at any time for reading and writing a document is equal to the buffer size. In our implementation, the buffer size is considered 1 KB which is more than enough for the commands of PUCK and sensor protocol.

The high-level workflow of the service layer is illustrated in Figure 2.5. As the response engine encountered with a carriage return operator, it tries to match the command with the hard-coded commands (*i.e., pattern*). After pattern matching, the response engine processes the request by retrieving necessary information from the MMU (if the command relates to PUCK memory), or the sensor data repository (if the request contributes to the sensor protocol). Finally, the response engine packages the result in buffers to be sent to the communication layer. The key features in the service layer are the buffering and pattern matching approaches. By these features, we could successively parse and compose large commands (e.g., 100KB) on devices with limited resources (e.g., 25 KB RAM).

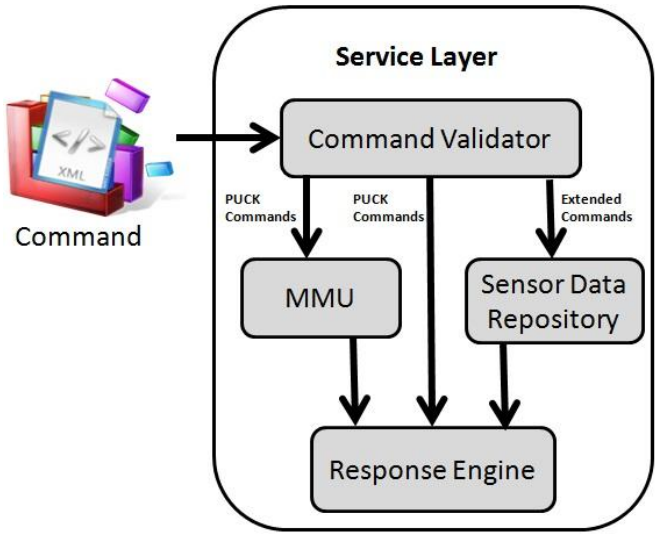

Figure 2.5 The high level workflow of the service layer

2.4.2 Additional Software Components

As the proposed IoT devices follow the PUCK open standard, users are able to connect to the devices using the PUCK commands. For example, users can develop a *PUCK detector* application that establishes Bluetooth connections and sends out PUCK *Null command* (*i.e.*, PUCK<CR>) to discover PUCK-enabled devices. After a successful discovery process, the client can send other PUCK commands to the devices through Bluetooth.

As PUCK provides access to the data in the PUCK memory, PUCK does not support communications in the device protocol. Therefore, we apply another OGC standard, SID, to handle the communications in the device protocol. First, we store a SID file which contains the necessary information about the device protocol, in the PUCK payload for client applications. Then, a client application can use a *SID Interpreter* [24] to retrieve sensor readings, and upload the observations to an online SOS (Figure 2.1).

2.5 Discussion

PUCK over Bluetooth was presented in this chapter as a Bluetooth protocol allowing physical sensors to be interoperable in the Bluetooth established network. Although PUCK protocol was not designed to be hosted on devices with limited resources, we designed and implemented this standard to host on class-1 IoT devices. The developed system has demonstrated that it is feasible to have an interoperable and standard Bluetooth protocol for entire IoT devices. In this case, the Sensor Web can be easily integrated to our daily devices such as mobile phones or notebooks. In spite of popularity of the Bluetooth radio in our daily electronic devices, there are several issues for the aforementioned system.

One issue is that Bluetooth radio has a short frequency range which clearly confines the users to be in proximity to the sensor (e.g., 10m). Although other wireless technologies (e.g., Wi-Fi or RF transceiver) might cover this inconvenience, they lack power conservation, or compatibility with our daily devices.

Moreover, one of the most challenging issues points to the security and privacy concerns. This issue can be solved by considering a passkey on the sensor's Bluetooth modem which is requested during the pairing process. Also, secure connection can be achieved by leveraging existing standard mechanisms. For example, the current Bluetooth modem uses an encrypted connection to protect the message content's integrity and confidentiality.

2.6 Summary

In this chapter, we presented the PUCK over Bluetooth protocol, as a wireless profile of the OGC PUCK for IoT devices. Thereafter, we defined the OGC SOS-like commands to query the capabilities document, and sensor readings. Furthermore, to address the world-wide access to the

sensor readings, we proposed the PUCK detector and SOS service developed on the host, which is able to establish Internet connectivity.

By hosting open standard Bluetooth protocol on the IoT devices, not only the devices become interoperable and easily plugged-and-played, but also the collected observations are accessible via our daily devices as soon as they are measured. In this case, we can easily make sensors available whenever wherever leading to a part of our tomorrow's daily life.

Chapter Three: **TinySOS**

3.1 Introduction

The basic concept of the IoT is the ubiquitous existence of various *things* or *objects* that can communicate and cooperate with each other in order to achieve shared goals [42]. By giving objects the possibility to interact with each other, the IoT is attracting a wide range of applications. For example, Giusto et al. [42] categorized IoT applications into five categories: transportation and logistics, healthcare, smart environments, personal and social, and futuristic applications.

In general, this chapter addresses the issues from the *decentralized* and *heterogeneous* nature of IoT objects and sensors. The main idea is basically inspired by two papers published by Priyantha et al. [43] and Bormann et al. [16]. Priyantha et al. [43] proposed a *tiny web service* for sensors and an application-level interface which have three advantages. First, each sensor becomes self-describable and self-contained by providing web interfaces for applications to retrieve sensor's capabilities. Second, some sort of privacy is preserved for device owners by direct connections to their devices. In addition, the sensor deployment and maintenance are easier with interfaces for updating a sensor's metadata. However, in order to achieve the interoperability between sensors and applications, one solution is to use standard-based web service interfaces and widely-used data encodings in information communication. However, Priyantha et al. [43] defined their own *ad-hoc* interfaces rather than implementing existing standards. On the other hand, Bormann et al. [16] proposed the *Constrained Application Protocol* (*CoAP*) as a lightweight transfer protocol for IoT objects. To develop a lightweight protocol, they used User Datagram Protocol (UDP) [28] to simplify the information exchange between the CoAP nodes. UDP is an alternative to the Transmission Control Protocol (TCP) [28]

which keeps track of packet delivery. In order to enable CoAP with this advantage of TCP, CoAP applies a re-transmission mechanism for lost packets. However, as most web applications are using HTTP, an extra proxy that translates HTTP and CoAP is required for applications to communicate with IoT objects. Chapter 4 thoroughly explains the specification of the CoAP and its contribution to this research.

According to the two above papers, one effective way to make IoT objects self-describable and self-contained is to implement web services on IoT objects. Moreover, the web services and the communication protocols have to be lightweight enough to be executed on objects with limited resources. Both the tiny web service paper [43] and CoAP paper [16] present a concrete idea about how to address the decentralized and heterogeneous issues of IoT and Sensor Web. However, we argue that the only drawback of these two papers is that they do not take advantage of the existing open standards to address the interoperability issues.

The Open Geospatial Consortium (OGC) established Sensor Web Enablement (SWE) as a group of open standards related to sensors, sensor data models, and Sensor Web services [44]. Similar to the W3C standards, the OGC SWE specifications are consensus-based open standards defined by any individual who is willing to participate. In principle, by following the SWE standards, we can achieve interoperability for the Sensor Web. However, the SWE standards are defined under the concept that web services are intermediaries between end-user applications and the sensors, and the SWE web services are based on HTTP and XML data representation. Lightweight Sensor Web services are not in the scope of SWE[10]. Thus, to the best of our knowledge, there is no existing work that evaluates the feasibility of constructing a SWE web

[10] A new OGC Standards Working Group (SWG) was formed in June 2012 called the Internet of Things REST API SWG. It focuses on developing an OGC standard for access to sensors in an IoT environment.

service directly on an object with limited resources. The reason this evaluation is important is that if SWE web services can be hosted on IoT objects, the IoT objects will not only be self-describable and self-contained, but also they will inherit the comprehensive SWE conceptual model directly. In this case, the IoT objects can interoperate with each other as well as the existing OGC SWE applications. Moreover, some sort of privacy might be preserved by removing the gateways in the path between the applications and devices.

Among the SWE specifications, we choose the Sensor Observation Service (SOS)[11] which defines a web service interface for accessing sensor observations and metadata [25], to be implemented on a class-1 IoT object. Our implementation of the SOS is termed *TinySOS* [30] that supports a lightweight profile of OGC SOS suitable for limited resources IoT objects.

Moreover, to address the issue of discovering IoT objects, we implement a *sensor registry service* that not only allows a sensor to register and advertise itself, but also lets consumers (e.g., other IoT objects, sensors, or end-user applications) to search for available IoT resources.

In summary, this chapter makes the following contributions:

1. We present TinySOS as an open standard Sensor Web service on the IoT devices. With the aim of doing this, TinySOS enables average users to deploy low-cost sensor systems easily.
2. Instead of using the traditional web service container, we develop a tiny web service whose code size is four times smaller than that of the traditional web service container.

[11] In this thesis, Sensor Observation Service (SOS) refers to the SOS version 1.0 [OGC, 2007]

3. To parse and compose potential large XML documents on an IoT device, we implement an XML processor unit (XPU) equipped with buffering mechanism to efficiently read and write XML documents. Therefore, the TinySOS allows a highly constrained device to handle very large XML documents.
4. Finally, to address the resource discovery issue, we implement the sensor registry service that acts not only as a catalog service, but also as a proxy to forward requests and responses between clients and TinySOSs with dynamic IP addresses.

The remainder of this chapter is organized as follows. Section 3.2 reviews the OGC SWE and the literature of integrating SWE and IoT. Section 3.3 and Section 3.4 present the proposed architecture and implementation, respectively. This is followed by a discussion about our findings and other issues about the IoT in Section 3.5. Finally, we this chapter provides a summary in Section 3.6.

3.2 Related Works

There have been some existing IoT projects applying proprietary protocols, such as Microsoft's HomeOS [45], Xively[12] (previously known as Cosm and before that Pachube), MicroStrain's SensorCloud[13], and Wovyn[14]. Many of them provide a web portal for users to manipulate the data collected by their sensors. We refer to these web portals as *the IoT portals*. Most of the IoT portals allow users to visualize the time-series data collected by sensors or publish the data with their own *Application Programming Interfaces* (*APIs*). However, in this case, IoT objects, that support only one type of proprietary APIs, form a "silo", and cannot interoperate with objects in other silos. Consequently, the development of various IoT silos obstructs the development of the

[12] https://xively.com
[13] http://www.sensorcloud.com/
[14] http://www.wovyn.com/

IoT. Therefore, in order to break down these silos and achieve the vision of an open IoT environment, following open standard protocols is necessary.

Sensor Web Enablement (SWE) as an OGC working group defines open standards related to Sensor Web. The prominent standards in the SWE frameworks are Observations & Measurements (O&M) [22], Sensor Model Language (SensorML) [23], Sensor Observation Service (SOS) [25], Sensor Planning Service (SPS) [26] and PUCK Protocol Standard (PUCK) [27]. O&M defines the standard models and XML schema for observations and measurements collected by sensors. The SensorML specification includes the standard models and XML schema for representing the metadata of sensor systems and processes. SOS presents the standard web service interface for requesting, filtering, and retrieving observations and sensor system information. An SOS service is the intermediary between a client and sensor observation repositories. The SPS specification provides the standard web service interface for users to *task* sensors to make observations. The PUCK standard which is introduced in Chapter 2 is a low-level protocol to retrieve SensorML documents, sensor driver code, and other information from sensors.

The SOS and SPS are the two SWE specifications defining the standard web service interfaces. For implementing a SWE web service on an IoT object, we choose the SOS in this chapter due to the fact that the SPS service requires customized implementations depending on each sensor's capabilities.

In fact, there have been some initiatives on integrating SWE and IoT. For example, presentations and talks such as "SWE and IoT"[15], "Sensor Web Standards and the IoT"[16],

[15] "SWE and IoT," Mike Botts, Botts Innovative Research, SWE-IoT ad-hoc during OGC TC, March 2012.

"Bringing IoT to the mass market - what should a standard do?"[17], and "Collaborative development of open standards for expanding GeoWeb to the Internet of Things"[18] were given in workshops and OGC Technical Committee meetings to discuss the possibility of applying SWE standards on the IoT. In addition, a new OGC working group was formed in June 2012 to define open standards for integrating SWE and IoT [46]. Moreover, Broring et al. [47] implemented *SenseBox*, which utilizes the O&M standard in their web service API. However, the web service on their SenseBox does not follow SWE standards. Furthermore, Resch et al. [48] did implement SWE standards (including SOS) on an embedded sensing device. However, their sensor hardware has 512 Mbytes RAM and 32 Mbytes flash memory, which even much more powerful than the class-2 device mentioned in Bormann et al. [16]. Therefore, we argue that it is still necessary to evaluate the feasibility of implementing SWE standards on a relatively inexpensive *class-1-like* device.

3.3 Architecture

In this section, we introduce the lightweight profile of SOS – TinySOS. Next, we present the high-level system architecture of TinySOS for class-1 IoT objects, and finally discuss our proposed sensor registry service for IoT resource discovery.

3.3.1 TinySOS

As mentioned earlier, class-1 devices have limited resources. In order to host web services on class-1 devices, the web service needs to be lightweight enough. Therefore, in this

---

[16] "Sensor Web Standards and the IoT," Scott Fairgrieve, Northrop Grumman, Expanding GeoWeb to IoT workshop during COM.Geo, 24 May 2011.
[17] "Bringing IoT to the Mass Market - What should a standard do?" Ben Pirt, Pachube, IoT Workshop at OGC TC, November 2011.
[18] "Collaborative Development of Open Standards for Expanding GeoWeb to the Internet of Things," George Percivall, OGC, COM.Geo, Expanding GeoWeb to IoT workshop during COM.Geo, 24 May 2011.

implementation, we only select the mandatory operations of the SOS (*i.e.*, the core operations) for the TinySOS. There are three mandatory operations in the SOS, namely *GetCapabilities*, *DescribeSensor*, and *GetObservation*.

The *GetCapabilities* operation provides access to metadata and detailed information about the available capabilities of the service. The *GetCapabilities* request can be sent either by HTTP GET or POST request type to retrieve the service metadata as an XML file (*i.e.*, the Capabilities document). The XML file contains metadata about this service, such as unique sensor identifiers, logical groupings of sensor observations (*i.e.*, the ObservationOfferings in the SWE terminology), and the URIs of physical phenomena (*i.e.*, the ObservedProperties) that sensors are measuring. Users can use the information in the Capabilities document to retrieve the sensor metadata and the observations with the other two core operations.

The *DescribeSensor* operation allows users to retrieve sensor metadata with a unique sensor identifier specified in the Capabilities document. If the *DescribeSensor* request is valid (*i.e.*, the service has sensor matches the unique identifier), the SOS returns the sensor metadata in the SensorML format.

The *GetObservation* operation provides access to the observations made by the sensors. Users can use the ObservationOffering and ObservedProperty in the *GetObservation* request as criteria in querying sensor observations. According to the criteria specified in the request, the SOS returns the sensor observations in the O&M format.

3.3.2 System Architecture

As we can see from the previous sub-section, to support the three core operations of SOS, an IoT object needs the functionalities of validating the HTTP request type (*i.e.*, GET, POST), content type (*i.e.*, text/xml), parse the XML request, and create the XML response. To achieve these

functionalities, here we present the proposed system architecture of the TinySOS service. There are three major layers in the TinySOS service (Figure 3.1), including *Communication Layer*, *Service Layer*, and *Sensor Layer*.

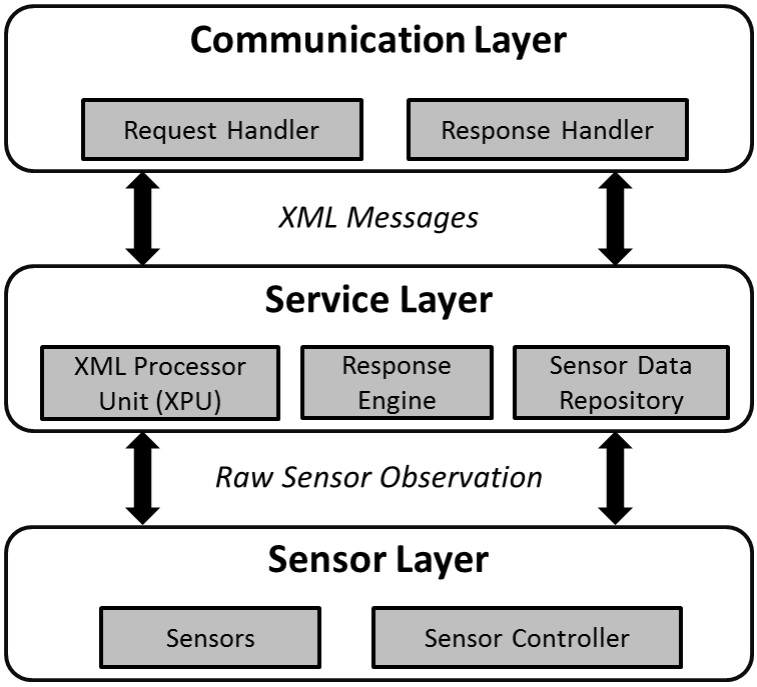

Figure 3.1 The system architecture supporting TinySOS protocol [30]

1) *Communication Layer:* The communication layer is responsible for managing the HTTP requests and responses, including the network related protocols and hardware (e.g., Network Interface Card). When a request is received by a TinySOS service, the communication layer forwards the XML request to the service layer for further processing. After the service layer finishes the task, an XML response is returned to the communication layer, and then sent back to the client.

2) *Service Layer:* The service layer handles the business logic of TinySOS. This layer consists of three modules: *XML processor unit* (*XPU*), *response engine*, and *sensor data repository*. As the XML documents are essentially too large to be stored in the memory

of class-1 devices, the TinySOS service needs a new way to parse XML documents. Therefore, unlike the traditional XML parsers that load the whole XML document into memory, we propose the XML processor unit (XPU) which reads and parses XML documents buffer by buffer. The XPU not only extracts the request criteria parameters, but also composes the *GetObservation* responses. More details about the XPU are presented in Section 3.4. The request criteria extracted by the XPU are forwarded to the response engine. If it is a *GetCapabilities* request or a *DescribeSensor* request, the response engine retrieves a predefined XML file (e.g., the Capabilities document and the SensorMLs) from the permanent memory, and forwards it to the communication layer. Otherwise, if the request is a *GetObservation* operation, the response engine tasks the XPU to compose the *GetObservation* response according to the criteria, and forwards the response to the communication layer. In addition, as an SOS should have the ability to return the historical observations, the TinySOS stores sensor measurements in a sensor data repository. Depending on the device, the sensor data repository could be located in the main memory (RAM) or the permanent memory (e.g., micro SD card).

3) *Sensor Layer:* The sensor layer consists of the physical sensors and the sensor controllers. The sensor controllers closely work with sensors. For example, a sensor controller can task sensors to collect sensor observations and send the collected sensor observations to the sensor data repository in the service layer. The sensor controller would play an important role in supporting SPS on IoT objects.

3.3.3 Resource Discovery

For the decentralized environment such as the IoT, resource discovery is always an issue. In our case, each sensor has a TinySOS web service which allows users to directly connect to sensors.

However, users still need to know the service's Internet location (e.g., IP address) in the first place.

In order to address the resource discovery issue, we propose a *sensor registry service*. Sensor registry service is similar to search engines and catalog services [49] that stores the metadata of web services and allows users to search services with criteria on metadata. However, in addition to the functionalities of a catalog service, the proposed sensor registry service is enhanced to support web services without a static IP address. This is because getting a static unique IP is not always possible, especially for embedded devices such as IoT objects. Therefore, in order to make a good use of IoT objects with dynamic IP addresses, we enhance the TinySOS and the sensor registry service to maintain a live connection together. In this case, the sensor registry service can act as a proxy redirecting requests and responses between users and TinySOS services.

The overall resource discovery process is shown in Figure 3.2. First, a TinySOS device with a static IP address registers itself to the sensor registry service by sending its IP address and service metadata. For a TinySOS device with a dynamic IP address, it not only transmits the service metadata to the sensor registry service, but also maintains a live connection with the sensor registry service. After the registration process, a client can send search requests to the sensor registry service. If the TinySOS that matches the search criteria has a static IP address, the sensor registry service returns the IP address to the client, and then the client can connect to the TinySOS through the SOS protocol. If the matched TinySOS has a dynamic IP address, the sensor registry service returns the IP address of itself (i.e., the sensor registry service) with a *subpath* of the unique identifier of that TinySOS (*i.e.*, http://IP of the sensor registry service/unique ID of TinySOS). In this case, the client can directly send SOS requests to the

sensor registry service, and the requests will be forwarded to the TinySOS. Similarly, the sensor registry service will redirect the SOS responses from the TinySOS to the client.

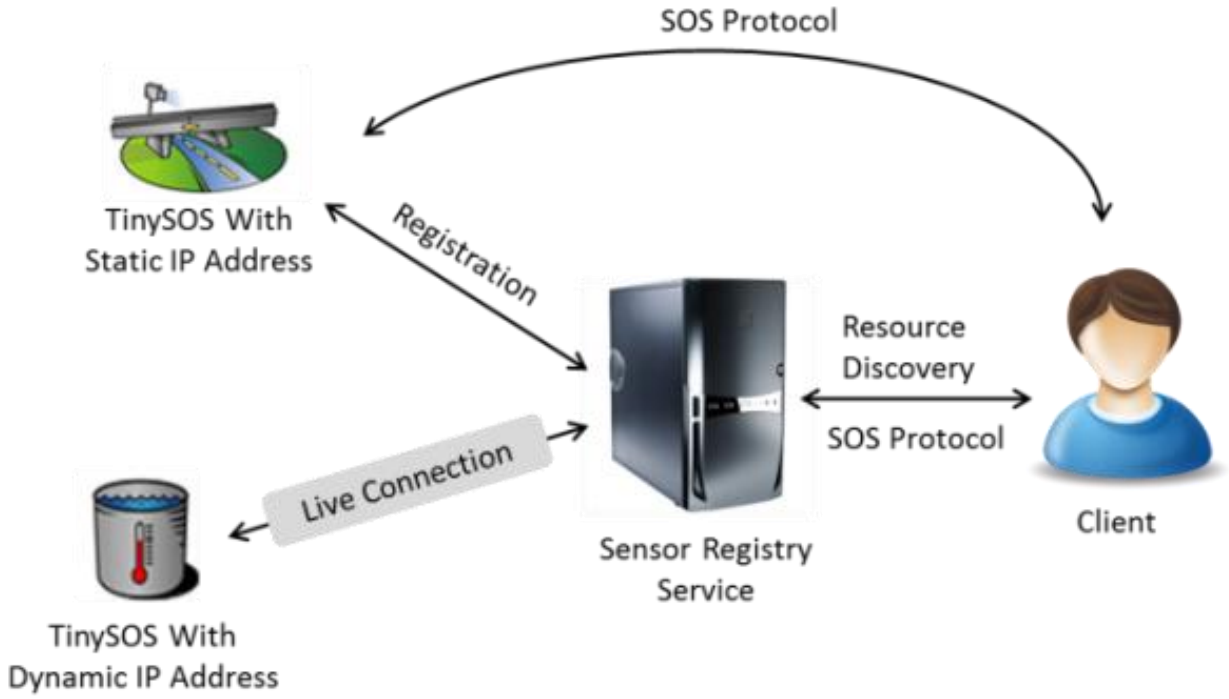

Figure 3.2 Resource discovery process [30]

3.4 Implementation

In this section, we point to a lightweight web server implementation, and later we explain the XPU algorithms in parsing XML documents.

3.4.1 Tiny Web Server

As regular web service containers (e.g., Apache web server) would be too heavy for Netduino Plus, we develop a "Tiny Web Server" as a container for TinySOS. This web server implements the basic features of an HTTP server which includes getting requests from clients and returning response data streams.

While the classes and libraries in C# .Net Micro Framework are relatively convenient to use, they consume a considerable amount of the code storage and memory footprint [50]. Hence,

instead of using the predefined C# libraries, we develop most functions by ourselves to decrease the code space and memory usage. For example, we develop another HTTP request handler to replace the .NET Micro HTTP library resulting in 35% less code storage occupation (from 17 KB to 11 KB). At the end, our tiny web server implementation only takes 11.72 KB of the code storage, which is much smaller than a regular SOS server occupying tens of Mbytes of code space[19]. Figure 3.3 shows the comparison between the code size of the simple web server and the three layers of TinySOS (communication layer, service layer, and sensor layer).

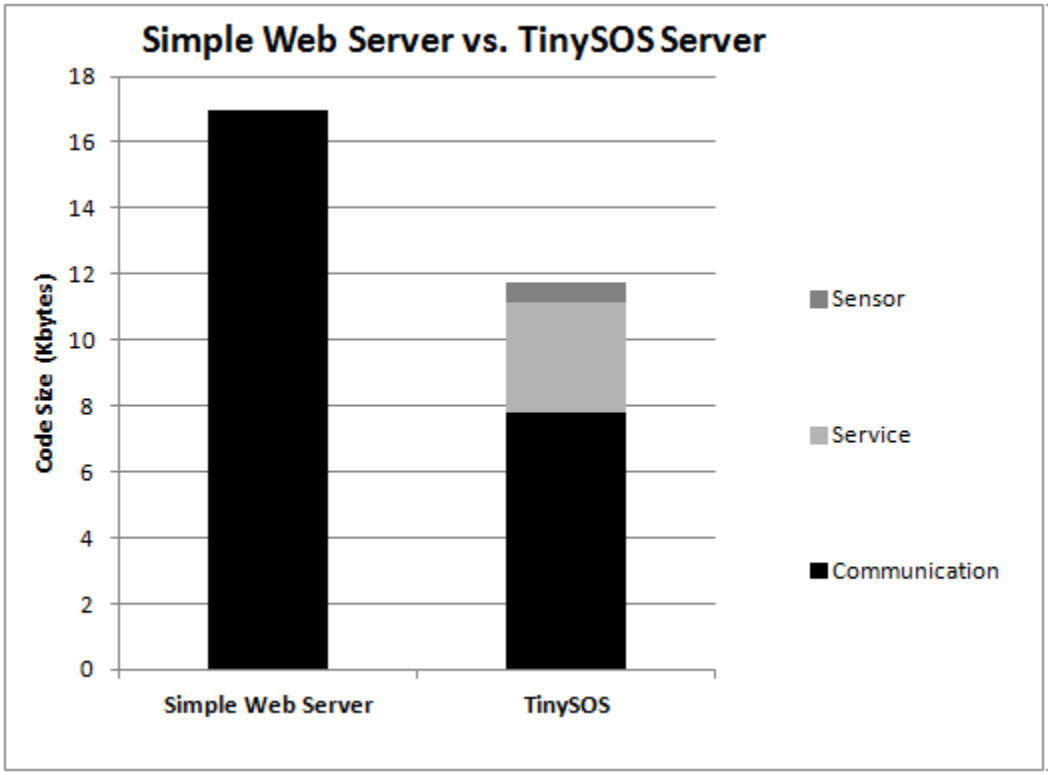

Figure 3.3 Code size comparison [30]

To clarify more the performance of the TinySOS, the simple web server is not able to process the requests including contents larger than 4 KB. In contrast, our TinySOS efficiently

[19] http://wiki.geocens.ca/sos/Installation, and http://52north.org/communities/sensorweb/sos/

responds requests up to 82 KB in length. Additionally, the simple web server cannot parse any XML files due to lack of enough memory.

3.4.2 XPU Algorithms

Since the memory of IoT objects is usually small, the XML request and response documents cannot fit into the memory. In order to parse and compose XML documents, we propose the XML processor unit (XPU) to read and write XML documents by utilizing a buffer mechanism. A buffer refers to a certain physical memory allocated to hold data temporarily. Therefore, by reading an XML document buffer by buffer, the maximum memory consumption at any time is equal to the buffer size. In our implementation, the buffer size is considered 1 KB.

Figure 3.4 depicts the high-level workflow of the XPU. First, when the XPU receives an XML request document straight over the sockets, the XPU calls the buffer reader iterater to read the document buffer by buffer. Then, in order to extract the information of the request from the buffer, XPU uses the *data extractor* to match the bytes in the buffer between some predefined bytes of patterns.

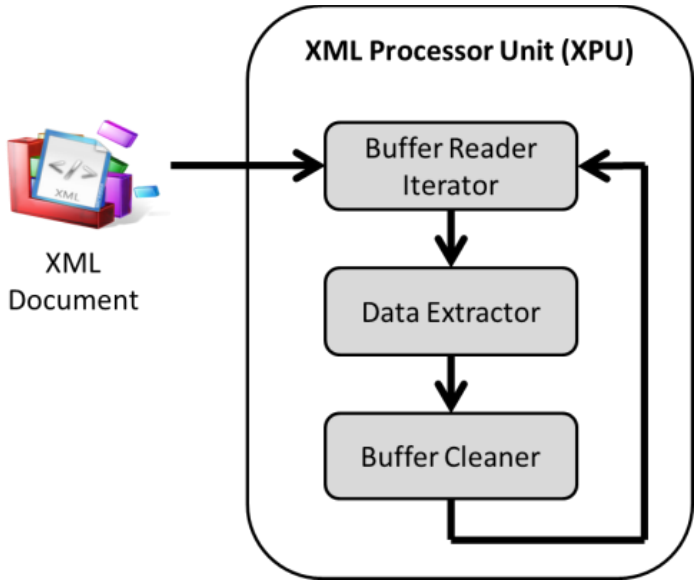

Figure 3.4 The high-level workflow of the XPU [30]

Since SOS defines specific XML elements and attributes for the SOS requests, we can understand the requests by searching the predefined words. For instance, the data extractor matches the bytes of the buffer with the bytes of the following words: “XML”, “request”, “service”, “version”, “GetCapabilities”, “DescribeSensor”, “GetObservation”, “offering”, “procedure”, “observedProperty”, and “<”. By treating the bytes of the predefined words as *patterns*, Algorithm 3.1 shows the naive approach of how the data extractor searches for these key words. After finding the location of the predefined XML elements or attributes, we can extract the values of them by simply loading the bytes after the elements/attributes.

Algorithm 3.1 Naive pattern matching

```
Function Match(pattern, buffer): matched
1:  m ← pattern.length()
2:  n ← buffer.length()
3:  matched ← false
4:  i ← 0
5:  FOR i to n - m
6:      IF pattern [1..m] is equal to buffer [i+1..i+m] THEN
7:          matched ← true
8:      END IF
9:  END FOR
10: RETURN matched
```

However, although the naive approach (Algorithm 3.1) works well in many cases, it fails on the case that patterns exist across two buffers. To overcome this problem, we merge the previous buffer to the current buffer for all iterations. In fact, each buffer except the first and last buffers is processed twice. Since the predefined patterns are all less than 1 KB, the revised version of the naive pattern matching approach (Algorithm 3.2) can extract all necessary information.

Algorithm 3.2 Revised pattern matching

```
Function Match_Revised (pattern, buffer_current, buffer_next): matched
1:  buffer← CONCATENATE (buffer_current, buffer_next)
2:  m ← pattern.length()
3:  n ← buffer.length()
4:  matched ← false
5:  i ← 0
6:  FOR i to n - m
7:      IF pattern [1..m] is equal to buffer [i+1..i+m] THEN
8:          matched ← true
9:      END IF
10: END FOR
11: RETURN matched
```

After the data extractor finishes the matching process on each buffer, the *buffer cleaner* removes the buffer memory and continues the iterations until the whole XML document is read. On the other hand, the same buffering approach is applied to compose XML responses. For a TinySOS responding to an SOS request, the XPU iteratively fills the response message into a buffer. When the buffer is full, its content is streamed to the client and then the XPU cleans the buffer. Consequently, with the buffering and pattern matching approaches, we successively address the issues of parsing and composing large XML document on devices with limited resources.

To sum up the proposed system, we demonstrate that by hosting an SOS service on IoT devices, we can use existing SWE applications to retrieve sensor data from the IoT devices. Figure 3.5 shows the GeoCENS [51] SWE client and the sensor data retrieved from a TinySOS device.

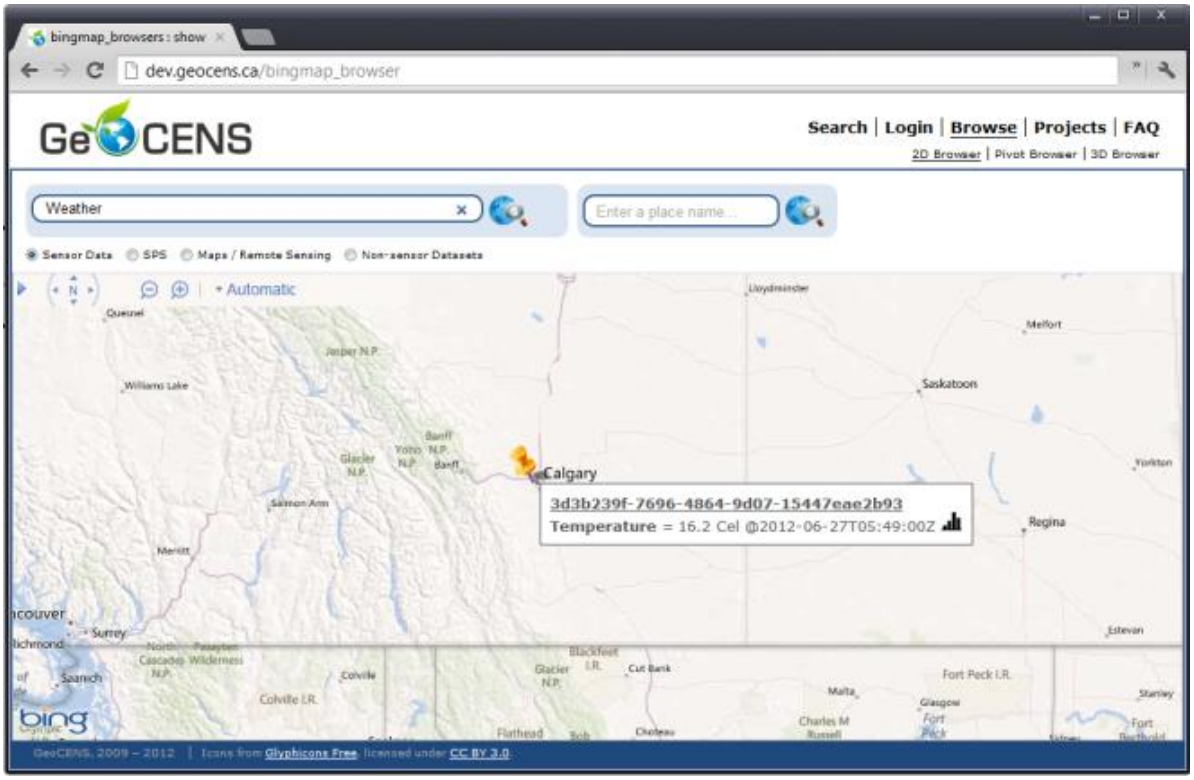

Figure 3.5 Using a SWE client to access a TinySOS device

3.5 Discussion

The proposed TinySOS system allows physical sensors to be interoperable in the OGC SWE framework. Although SOS was not designed to be hosted on devices with limited resources, we design and implement the tiny web service container and XPU to host SOS on IoT devices. The proposed system has demonstrated that it is feasible to host a SWE web service on class-1 devices. In this case, the Sensor Web can provide real-time sensor data streams with a much higher spatio-temporal resolution. However, we also observe some potential issues on the TinySOS system.

The first immediate issue is that each IoT device should have a stable and constant Internet connection in order to receive requests from clients, which is currently unavailable. One potential solution already discussed in Chapter 2 is to utilize the OGC PUCK standard protocol

on IoT devices. However, the PUCK protocol was not designed to be a web service serving sensor observations. Additional web interfaces are still needed in this case.

The second issue is about the update of metadata, such as the SensorML or sensor location. As these information can be manually stored in the permanent memory (e.g., SD card), a standard way provided by SOS is the transaction operations. By supporting the transaction operations of SOS, sensor owners can register SensorMLs into the SOS. However, to automatically measure sensor locations, attaching a Global Positioning System (GPS) sensor on the device may be a better choice.

In addition, privacy and security are the important items as well. The privacy issue is about whether the sensor owners want their devices to be discoverable or not. In the case of this implementation, as the resource discovery is handled by search engines or catalog services such as the sensor registry service, mechanisms to preserve privacy should be implemented in these resource discovery services. Information security means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, perusal, inspection, recording or destruction [52]. This can be achieved by leveraging existing standard mechanisms. For example, the current TinySOS implementation uses the Transport Layer Security (TLS) and Secure Socket Layer (SSL) to protect the message content's integrity and confidentiality.

3.6 Summary

In this chapter, we presented the TinySOS service, a lightweight profile of OGC SOS for IoT devices. In order to host an SOS service on devices with limited resources, we developed a tiny web service container to handle HTTP requests/responses; and we proposed the XML processor unit to parse and compose XML documents with small memory consumption. Furthermore, to address the resource discovery issue, we developed the sensor registry service which can serve

not only as a catalog service, but also as a proxy between clients and devices with dynamic IP address.

By hosting open standard web services on IoT devices, not only the devices become self-describable, self-contained, and interoperable, but also the collected observations are accessible via the Internet as soon as they are measured. In this case, the Sensor Web can provide real-time sensor data streams in a much higher spatio-temporal resolution, which allows users to observe phenomena that were previously unobservable.

Chapter Four: **SOS over CoAP**

4.1 Introduction

IoT devices are usually limited in power, network, memory and processing capabilities [16]. The aforementioned standard protocols, PUCK and TinySOS, have not typically been designed with power and network efficiency in mind. In battery-operated WSN nodes, the radio transceiver is certainly the most power-consuming component [53], so power-efficiency translates into optimized radio duty cycling. Since the IoT and WSN share similar visions, the same scenario exists in the IoT. The naive solution is enforcing the battery-powered device to keep its radio off as much as possible. Another solution is to minimize the network load by which not only the bandwidth is dramatically saved, but also the radio transceiver can fulfill its task faster resulting in more sleeping [54].

To achieve this, we select the IETF protocol designed for constrained nodes and networks (e.g., WSNs), and named *Constrained Application Protocol* (*CoAP*) [55]. This protocol employs the basic features of HTTP to the constrained network while maintaining a low overhead. HTTP is based on the *Representational State Transfer* (*REST*) style [56]; in which the web resources are identified by URIs. Thus, CoAP enables interoperability in machine to machine (M2M) communications at the application layer through RESTful web services. REST only relies on the HTTP methods such as GET and POST. Unlike HTTP, CoAP operates over the UDP and applies an efficient retransmission mechanism instead of complicated congestion control as used in standard TCP.

The CoAP can easily be translated to HTTP to make the seamless integration of constrained networks with the Web. To do this, CoAP proxies are employed to convert CoAP

messages to HTTP packets. The main interest in making CoAP nodes part of the Internet is to allow various nodes to interact with each other using the existing web technologies.

Since we have already demonstrated the integration of OGC SOS to the IoT, we combine this protocol with CoAP in order to make CoAP nodes interoperable to other IoT components. As we already discussed in Chapter 3, SOS is not originally designed for limited resources IoT objects. On the other hand, CoAP cannot validate SOS requests which are definitely larger than the CoAP upper bound for the message size (1280 byes for IPv6 datagram) [55]. Therefore, one possible solution is to combine SOS and CoAP on the CoAP proxy which has enough resources. Therefore, the contribution of this chapter is that we are the first to bind the OGC SOS to the CoAP Proxy denoted as *SOSCoAP proxy*. According to Figure 4.1, the SOSCoAP proxy can communicate through CoAP regulations to CoAP nodes (*i.e.*, IoT devices) from one side, and it can speak through the SOS standard from another side. As a result, we achieve the interoperability while maintaining minimal resource consumption on IoT devices.

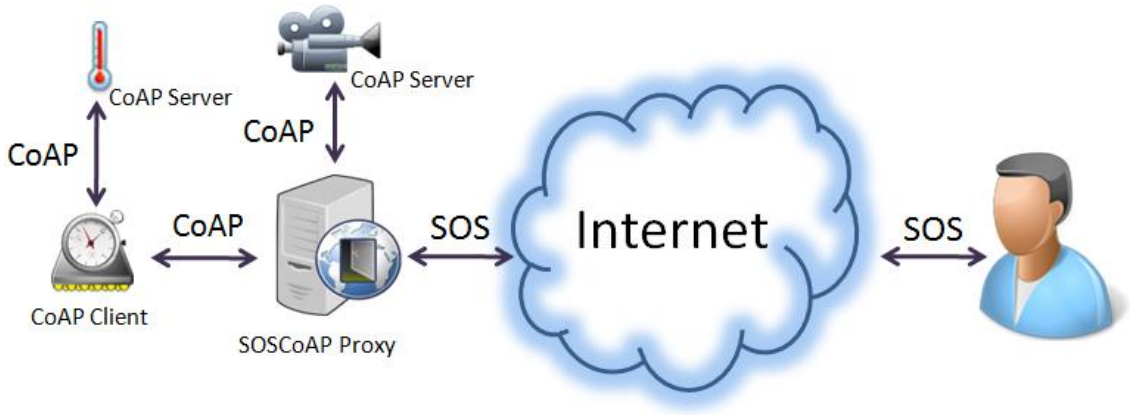

Figure 4.1 High level view of the SOS over CoAP strategy

The remaining sections of this chapter are organized as follows. In Section 4.2, existing literature about the CoAP implementation is reviewed. Section 4.3 and Section 4.4 present the proposed architecture and implementation, respectively. Section 4.5 provides a discussion about

the SOS over CoAP strategy and its challenges. Finally, this chapter is briefly described in Section 4.6.

4.2 Related Works

CoAP has been already implemented in the most popular operating systems for WSNs such as LibCoap for TinyOS [57] and CoapBlip for Contiki [58]. These research efforts mainly addressed the possibility of the CoAP on target platforms with only tens of KB RAM and ROM.

Later on, some research improved the CoAP implementations for WSNs in case of energy consumption, memory usage and network latency [59]. Although CoAPBlip [59] has been previously included in the TinyOS as a CoAP library, Ludovici et al. [59] introduced TinyCoAP as a more efficient implementation of the CoAP for TinyOS. The TinyCoAP is implemented only for the devices supporting TinyOS which conflicts with the aim of interoperability between all kinds of IoT devices.

There are also a few efforts to make the CoAP compliant to the World Wide Web standards. For example, *Simple Object Access Protocol* (*SOAP*) standard [60] for the data exchange of web services was bound in CoAP in [61]. This research could successfully transport SOAP messages in resource constrained environments resulting in deployments of web services in WSNs. However, there is a negative point in combining SOAP and CoAP because SOAP messages are encapsulated in the XML format which leads to complex message processing. Since the overhead of data transfer between SOAP-based web services is significantly higher than the RESTful web services [62] [63], the authors of [64] focused on the combination of RESTful CoAP and XML to make it more standardized. Thus, they proposed CoAP to supply RESTful communications among applications, and EXI (Efficient XML Interchange) format [65] to make their system more standardized according to the World Wide Web Consortium

(W3C) [66]. The weakness of this design is that the interoperability issue of the IoT objects was not touched at all.

Lerche et al. [62] give an overview of the current CoAP implementations and present the results of an interoperability meeting organized by the European Telecommunications Standards Institute (ETSI) [67]. In this research, 18 CoAP server and 16 CoAP client implementations were tested against each other. Although this is a preliminary step towards the interoperability assessment between CoAP nodes, the use of CoAP solely in the IoT has not been definitely confirmed yet.

According the above literature, we are not the first to argue the benefits of the CoAP and its implementation challenges, but we are the first to demonstrate the integration of this protocol to other standards of the WSNs (e.g., OGC SOS) as an interoperable infrastructure for the IoT.

4.3 Architecture

In this section, the CoAP specification is technically discussed first. Then, the proposed architecture for a CoAP-enabled IoT device is described. At the end, we also present the architecture of the SOSCoAP proxy.

4.3.1 CoAP Specification

The CoAP was originally released by the Constrained RESTful Environment (CoRE)[20] working group at IETF as a reliable lightweight protocol for the Internet of Things. The CoAP is lightweight because it keeps the message length as short as possible, and it transmits the packets over the network by using UDP. The CoAP specification provides an upper bound to the message size. Since the messages larger than an *IP fragment* [28] result in undesired packet

[20] http://tools.ietf.org/wg/core/

fragmentation, a CoAP message should fit within a single IP packet (*i.e.*, avoid IP fragmentation) and must fit within a single IP datagram. In cases that the *Maximum Transmission Unit* (*MTU*) of a path is not known for a destination, an IP MTU of 1280 bytes is assumed for the CoAP message size. If nothing is known about the size of the headers, an upper bound of 1152 bytes for the message size and 1024 bytes for the payload size must be considered [55].

In general, the CoAP message is composed of a *header* with at least 4-byte length, a *token*, several *options*, and a *payload*. To have a better understanding, Figure 4.2 depicts the format of a CoAP message. The 4-byte header includes CoAP version, message Type (*confirmable*, *non-confirmable*, *acknowledgement*, *reset*), token length, code (*request*, *success response*, *client error response*, *server error*), and message ID (for detection of message duplication). The header is followed by a token value (to correlate request and response), options (if any), and payload (if any) which are all variable-length.

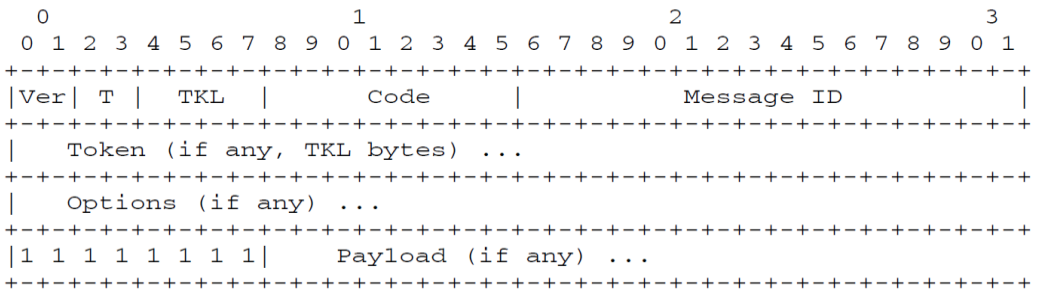

Figure 4.2 CoAP message format [55]

CoAP messages may be *Confirmable* (*CON*) or *Non-confirmable* (*NON*). In spite of using UDP in the request/response interactions, reliability is provided when messages are labelled as Confirmable through end-to-end stop-and-wait retransmissions mechanism [55]. That is, a CoAP server receiving a CON request must acknowledge its receipt to the client. Until the acknowledgement (ACK) is received by the client, the previous request will be retransmitted to

the server with exponential back-off. Sometimes a request might need further processing to be responded; so the server sends an empty ACK to indicate that the response will be deferred. Consequently, the client must also acknowledge the arrival of the server's CON response. On the contrary, Non-confirmable messages are used to allow sending requests that does not require reliability. Figure 4.3 exemplifies a client-server interaction for a CON request and a NON request.

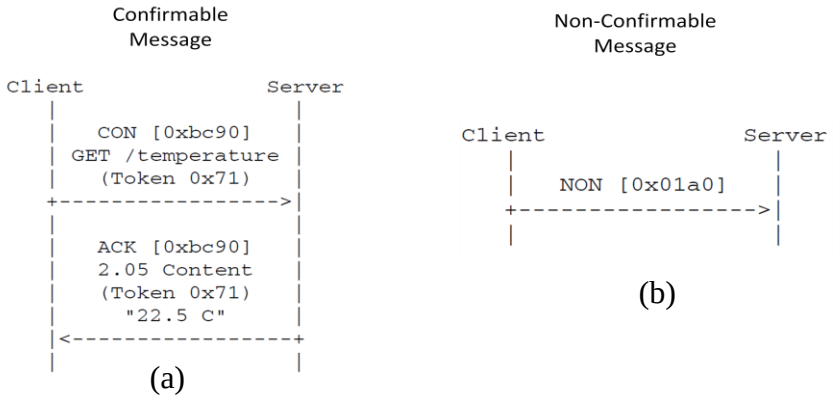

Figure 4.3 CoAP client-server interaction: (a) CON request; (b) NON request [55]

Furthermore, CoAP is able to detect duplicate messages by matching requests to responses. This is done by checking the message ID of each request which is already generated by the client. The detection of duplicated messages is available in CON as well as in NON messages. Finally, the token value (*i.e.*, request ID) is used for distinguishing concurrent requests. The server must echo the token value of a client request in any relevant responses to that request (Figure 4.4).

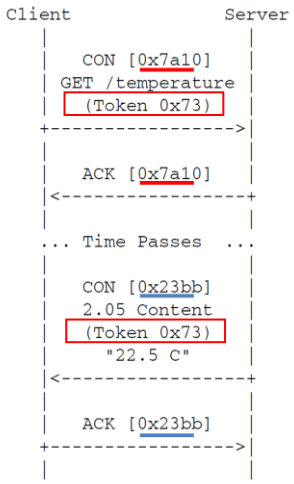

Figure 4.4 Empty ACK because of response deferral

4.3.2 Device Architecture

As depicted in Figure 4.5, we have integrated a full protocol stack necessary for an IoT device in order to communicate through the CoAP.

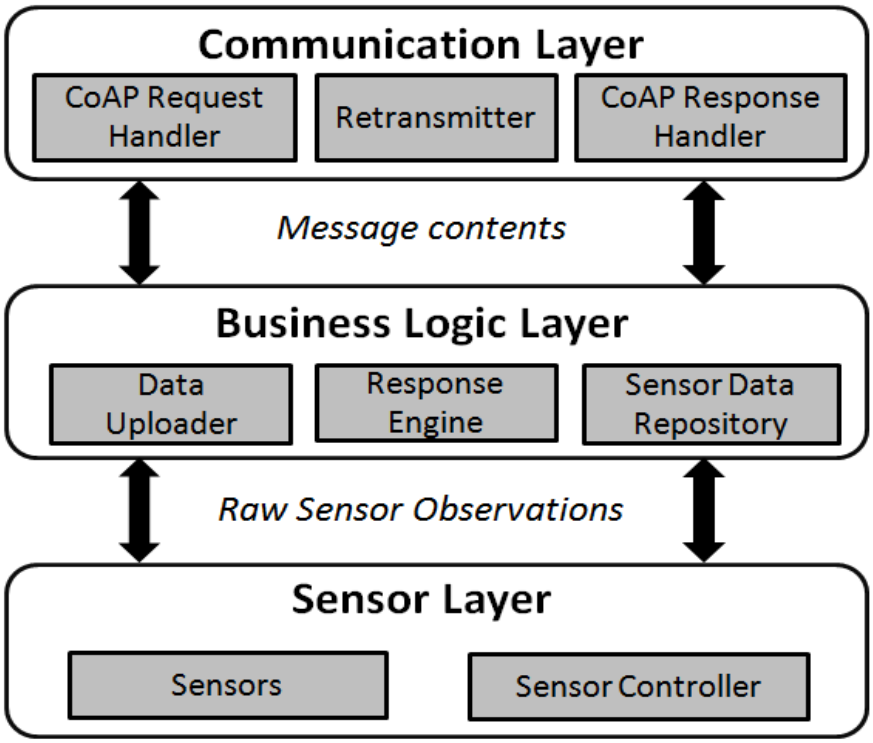

Figure 4.5 The device architecture supporting CoAP protocol

The sensor layer pretty remained unchanged comparing to TinySOS and PUCK. The business logic layer is partially similar to the service layer of the two prior protocols. The significant highlight in this layer is the *Data Uploader* component (*i.e.,* client) in order to frequently upload the sensor observations to a pre-defined CoAP proxy. When a CoAP request is received in the communication layer, it is directly forwarded to the response engine. The response engine composes the content, and posts the message to the communication layer to be packaged in the CoAP message format. Furthermore, as a user may request historical observations, the sensor readings are dynamically stored in a sensor data repository.

More importantly, CoAP focuses on efficiency in data transmission, so the communication layer on the device is totally modified from the two previous protocols. The most fundamental change points to the usage of UDP instead of TCP in the transport layer with retransmission mechanism.

4.3.3 SOS Integration to CoAP

The SOSCoAP proxy is a regular web service placed in the CoAP network infrastructure as illustrated in Figure 4.1. One of the responsibilities of this proxy is to interconnect CoAP endpoints to users via the OGC SOS protocol. As a result, this proxy should be capable of converting the two protocols together (*i.e.,* CoAP-to-SOS, or SOS-to-CoAP). As shown in Figure 4.6, we propose the following architecture for the SOSCoAP proxy.

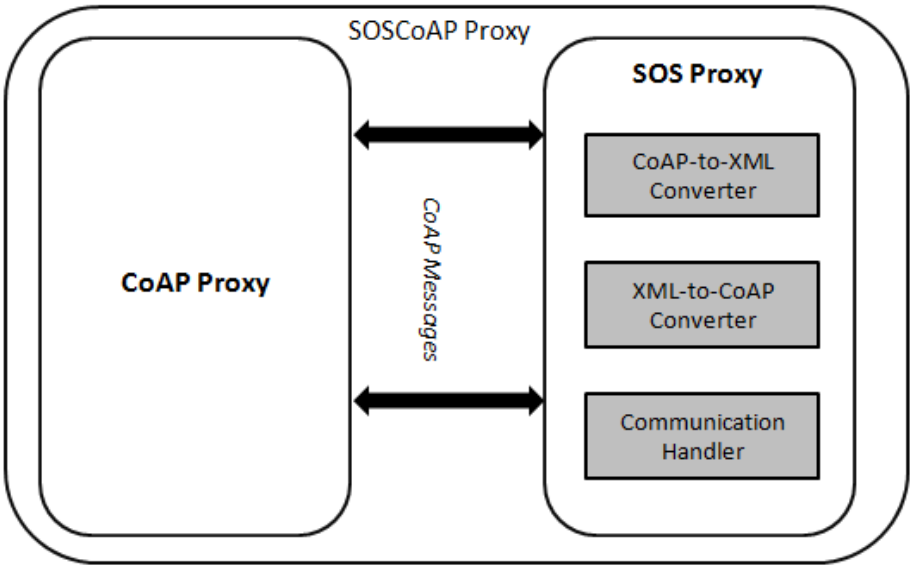

Figure 4.6 The architecture of SOSCoAP Proxy

The SOSCoAP proxy consists of a *CoAP proxy* and a *SOS proxy*. For the CoAP proxy, we use jCoAP[21] which is an open source Java library. While the CoAP proxy is important, we do not address its components in this chapter as they are pretty unchanged from the CoAP specification. On the other hand, we develop the SOS proxy that consists of three components:

- *XML-to-CoAP Converter*: This component receives the core SOS requests (GetCapabilities, DescribeSensor, and GetObservation) from user. As those requests are encoded in XML, they need to be formatted to plain text requests encapsulated in UDP message.
- *CoAP-to-XML Converter*: This component receives the CoAP messages and it coverts them to the SOS responses. As the CoAP messages are plain texts, they need to be encoded in XML format to be sent back to the user.

[21] https://code.google.com/p/jcoap/

- *Communication Handler*: The communication handler checks the user requests in terms of compatibility to the SOS operations. If the request is validated, the relevant SOS response is sent to the user.

4.4 Implementation

The CoAP implementation itself is straight forward on our development platform. The CoAP part of the SOSCoAP proxy is also provided from jCoAP library. Thus, this section mainly highlights the data exchange between a user and a CoAP node through the SOSCoAP proxy.

4.4.1 SOS Request to a CoAP Server

In our implementation, we only consider the three core SOS operations: GetCapabilities, DescribeSensor, and GetObservation. First, the SOS part of the SOSCoAP proxy retrieves XML-encoded SOS requests through the Internet. Since the XML body requires a complex and expensive message processing [59], the request is encoded to a simpler format according to Table 4.1. Then, the SOS part packages the mapped request for the CoAP part of the proxy to send the simplified request to the CoAP server (IoT device).

Table 4.1 Mapping SOS operations to CoAP requests

SOS request (XML)	CoAP request (plain text)
GetCapabilities {...}	Get /capabilities
DescribeSensor {...}	Get /describeSensor?procedure=prodecureValue
GetObservation {...}	Get /observation?observedProperty=observedPropertyValue&offering=offeringValue

When one of the three requests of Table 4.1 is received on the CoAP server (*i.e.*, IoT object), the relevant response is generated according to Table 4.2. The rest is the same as CoAP message processing which is not the contribution of this research.

Table 4.2 CoAP responses to SOS requests

SOS request converted to CoAP message (plain text)	CoAP response content (plain text)
Get /capabilities	offering: offeringValue, observedProperty: observedPropertyValue, procedure: procedureValue
Get /describeSensor ...	sensorID: URN, unitOfMeasurement: unitValue
Get /observation ...	sensorID: URN, observation(s): resultValue_1|observationTime_1#resultValue_2| observationTime_1#...

4.4.2 CoAP Request to a SOS Server

Although the sensor measurements are slightly cashed on the limited data repository of the CoAP server, the data uploader component can be tasked to submit the data to the proxy for historical record. The content of such request is similar to the response content of the get observation request (Table 4.2). Later on, the CoAP proxy of SOSCoAP proxy will convert the requests to HTTP messages accepted by the SOS component of SOSCoAP proxy. In addition to data uploading on the CoAP server, the SOSCoAP proxy is capable of conveying the sensor data to a predefined SOS clouds (e.g., GeoCENS [51]) through standard SOS requests. The configurations of these clouds are recorded in the communication handler component of the SOSCoAP proxy.

4.5 Discussion

The SOS over CoAP protocol is considered as a simple but efficiently integrated protocol for the IoT. Although the SOS standard is overkill for resource constraint devices, its implementation on the CoAP proxy does not cause any difficulties. Apart from common issues with the previous protocols such as power supply, Internet connectivity, and metadata update, some other issues exist as follows.

First, the connection between each IoT device and the rest of the Internet would be indirectly through a proxy. Although the proxy can have a more stable and constant Internet

connection, the *single point of failure* (*SPOF*) [68] issue should be highlighted. The SPOF is referred to a system component that its failure affects the entire system. Since the proxy is the only interface between users and IoT objects, it should be equipped enough to guarantee a constant connection to all IoT devices it relates.

Moreover, a request passes four levels to be delivered to a user: CoAP server, CoAP proxy, SOS proxy, and client. If the number of requests increases on a single SOSCoAP proxy, the response time may be affected. One potential solution for this problem can be deployment of multiple *cloud services* for IoT devices. The cloud services [69] involve a large number of computers connected through the Internet. In the other words, cloud services rely on sharing the computational resources to offer a utility over a network which can solve the aforementioned problem in an efficient way.

4.6 Summary

In this chapter, we counted some of the strengths of the Internet Engineering Task Force (IETF) approach. To this end, the resource efficient CoAP was implemented on our class-1 development platform. Then, the interoperability issue for the newborn CoAP was challenged.

In order to have the potential of popularity of OGC SOS, and the efficiency mechanisms of CoAP, we detailed the realization of simple but powerful SOSCoAP proxy for the IoT applications. The SOSCoAP proxy was supposed to establish a connection between CoAP network through CoAP messages and the rest of the Internet through the OGC SOS standard.

By combining the lightweight CoAP protocol and popular OGC SOS in the IoT network, a great range of devices can be interoperable together as TinySOS devices, PUCK-enabled instruments, and SOS services.

Chapter Five: **OGC SensorThings API**

5.1 Introduction

In the previous chapters, we significantly demonstrated that the existing protocols of Sensor Web and WSNs can be implemented on resource constraint IoT objects. While these efforts are moving the Internet of Things toward greater interoperability, they do not fit well in the IoT devices in case of processing load or interconnection with the other Internet nodes. In an attempt to address both deficiencies of the previous protocols, there is an ongoing effort of defining a standard *Web Application Programming Interface* (*API*) for the IoT.

This API, namely *OGC SensorThings*, is an OGC candidate standard for monitoring and controlling IoT devices (sensors and actuators) over the Web. The API is built on HTTP protocols, and applies the widely-used *Representational State Transfer* (*REST*) architectural style [56] to access a system's components. REST considers the system as a black box with a high level view regardless of the component details and their functionalities. REST only focuses on the status of the components and their relationship to each other. Web services complying with the REST principles are called RESTful. To exemplify, a camera device has a light sensor and also a LED actuator. When the camera is being accessed through a RESTful protocol, the camera, light sensor and LED are considered as the system components in which the LED and light sensor are attached to the camera component.

This API interconnects IoT services and applications over the Web through *Java Script Object Notation* (*JSON*) data format. The JSON is one of the text formats designed for representing simple data structures, data collections, and of course data exchange over a network connection. Therefore, as an alternative to the heavy Extensible Markup Language (XML) format, we use the simple JSON format to efficiently present the data on the server. Since our

ultimate goal in defining this standard is an easy-to-use and easy-to-implement for global IoT devices, we use plain text in the device-server interactions.

The OGC SensorThings service interface differs from the existing OGC web services in case of RESTful interface and JSON data encoding. This API is essentially inspired by the OASIS Open Data Protocol (OData)[22], which defines a general-purpose RESTful service interface. Besides the OData, the RESTful service interface also leverages the existing and widely-implemented OGC standards. For example, the capabilities part of the API service interface adapts several elements from the GetCapabilities response defined in *the OGC Web Service (OWS) Common Standard* [70] by converting the XML encoding into the JSON encoding.

The SensorThings API was mainly developed by a group of researchers in University of Calgary including Dr. Steve H. L. Liang, Dr. Chih –Y Huang, Tania Khalafbeigi and me. I was involved in the design and implementation of the device-side protocol that covers the interactions between IoT objects and the IoT RESTful service. The rest of this protocol focuses on the users and the IoT server communications which can be found from here: http://ogc-iot.github.io/ogc-iot-api/.

5.2 Related Works

Linking the Web and physical objects is not a new idea. As we can see, three protocols have been discussed in the previous chapters. The key idea of those protocols was to provide a virtual counterpart of the physical objects on the Web. With advances in computing technology, most devices are enabled with tiny web services [43, 71, 72]. However, the interoperability problem

[22] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=odata

still exists in most of them because lack of a specific standard in the IoT for communication protocol and data representation.

Several systems for integration of sensor systems with the Internet have been proposed such as SenseBox [47] and Xively, which offer a platform for people to share their sensory readings using web services. This sharing is performed by transmitting the data onto an online repository. Unlike the OGC SensorThings, these approaches exclusively support the sensing profile, and devices are considered as passive actors only able to push data.

Kindberg et al. [73] developed Cooltown project which associates web pages and URIs to people, places and things. Kindberg et al. also implemented scenarios where this information could be physically discovered by scanning infrared tags in the environment. We would like to go a step further to truly make IoT objects part of the Web so that they proactively serve their functionality in an interoperable manner.

Similar to our RESTful web interface, T. Luckenbach et al. [74] and W. Drytkiewicz et al. [75] consider the use of REST-like architectures for sensor networks. However, to make the API interoperable, we extend the model with the use of other standards (e.g., OGC SOS, OGC SPS and OData).

In essence, the OGC SensorThings provides a RESTful web interface allowing users and application developers to apply a common API to retrieve the things' profiles, and sensor observations. This protocol will facilitate a generic adapter for integration of devices to the IoT server, so interoperability between things will become simpler.

5.3 Architecture

In this section, we first elaborate the API components and its ecosystem. Then, we present the data model of this open standard. Finally, we describe the system architecture like the previous chapters.

5.3.1 API Components and Ecosystem

The SensorThings API follows a RESTful web service interface to access the registered resources on the server. Each resource is assigned a uniquely identification (UID) by the server. The API supports the four basic operations of the persistent storage, namely CREATE, READ, UPDATE, and DELETE (CRUD) to any resources of the service. The API also consists of two major profiles: *Sensing Profile* and *Tasking Profile*. The Sensing Profile is designed based on the OGC Sensor Observation Service (SOS) specification, in which defines an interoperable framework to manage and access sensors and observations. The Tasking Profile is based on the OGC Sensor Planning Service (SPS) specification, in which defines an interoperable way to submit tasks to control sensors and actuators. Figure 5.1 depicts the ecosystem of this API.

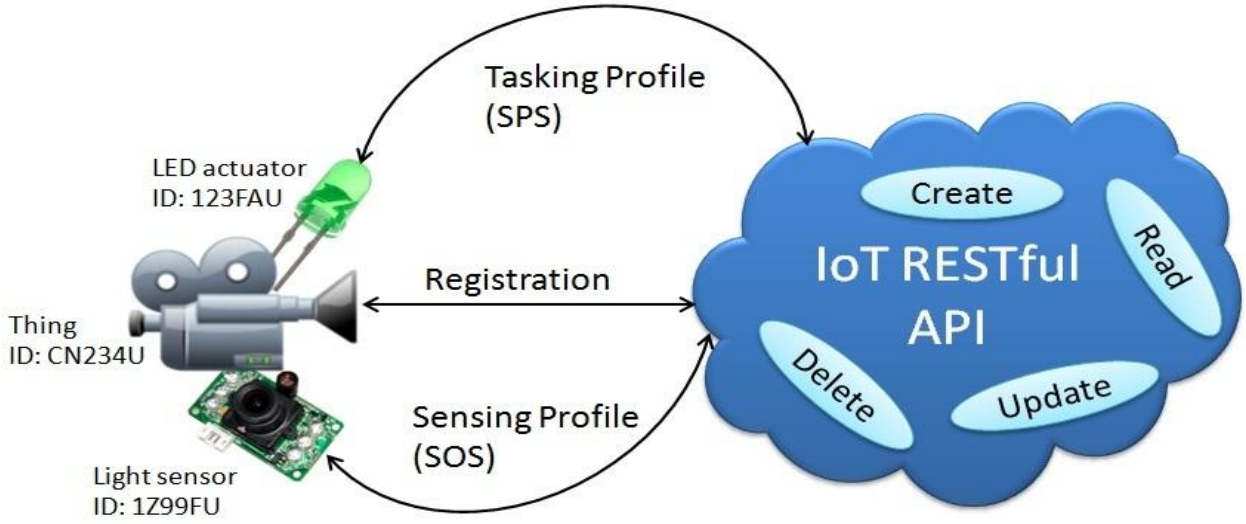

Figure 5.1 Ecosystem of the OGC SensorThings API

To address the resources, we define the protocol of retrieving the *capabilities document* which includes the service metadata about a specific service instance. By this document, the service advertises the supported functions and any constraints on using these functions. After a client retrieves the capabilities document, he/she understands how to perform CRUD actions to the target resource(s) through URI. There are three major URI components used in this API: the *service root URI*, the *Resource Path*, and the *Query Options*. The service root URI is the address of the IoT RESTful service. By attaching the resource path after the service root URI, users can address different resources (e.g., *collection or specific entity*). In order to facilitate information retrieval for the READ action, users can apply query options to the resource path, such as sorting by properties and filtering with criteria. Figure 5.2 demonstrates the URI components.

Figure 5.2 URI Components

5.3.2 Data Model

The OGC SensorThings API describes a data model for the resources and their connections as shown in Figure 5.3. The core of the data model is a Thing. Since the geographical positions of IoT objects may dynamically change, we record multiple locations in place and time for each Thing. As we mentioned in Chapter 5.3.1, the IoT data model consists of a *Sensing Profile* and a *Tasking Profile*. The Sensing Profile allows IoT devices and applications to perform CRUD operations on the gathered data from sensors. On the other hand, Tasking Profile provides the functions to control IoT devices and actuators. According to the data model illustrated in Figure 5.3, each Thing can also have several Datastreams and Tasking Capabilities, which form the core

of the Sensing Profile and Tasking Profile respectively. Datastream relates to observed properties, and also sensor observations. Each instance of the observation entity is also linked to a specific sensor. Since sensor observations can be performed in a location different from the Thing location, Features of Interest is also considered to record the place that observation occurred. On the other hand, Tasking Capabilities is linked to actuator metadata, and tasks triggered from client.

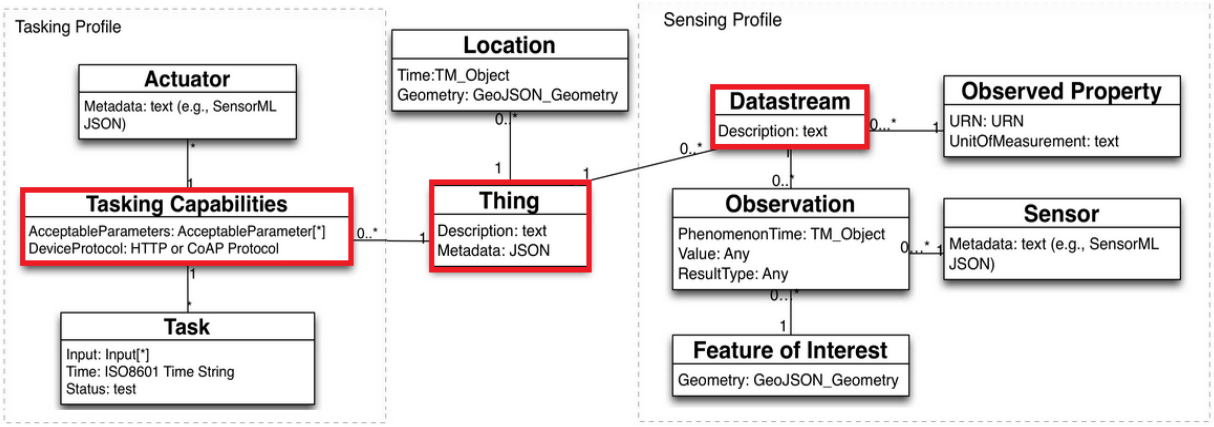

Figure 5.3 Data Model

Since more explanation of the data model is out of the scope of this Chapter, we skip to the system architecture which is related to the IoT device structure.

5.3.3 System Architecture

Like the former chapters, devices supporting the OGC SensorThings API follow a system architecture to process requests and responses. In this section, we describe the proposed system architecture of IoT devices displayed in Figure 5.4. In this architecture, you can see the three common layers including *Communication Layer*, *Business Logic Layer*, and *Sensor/Actuator Layer*.

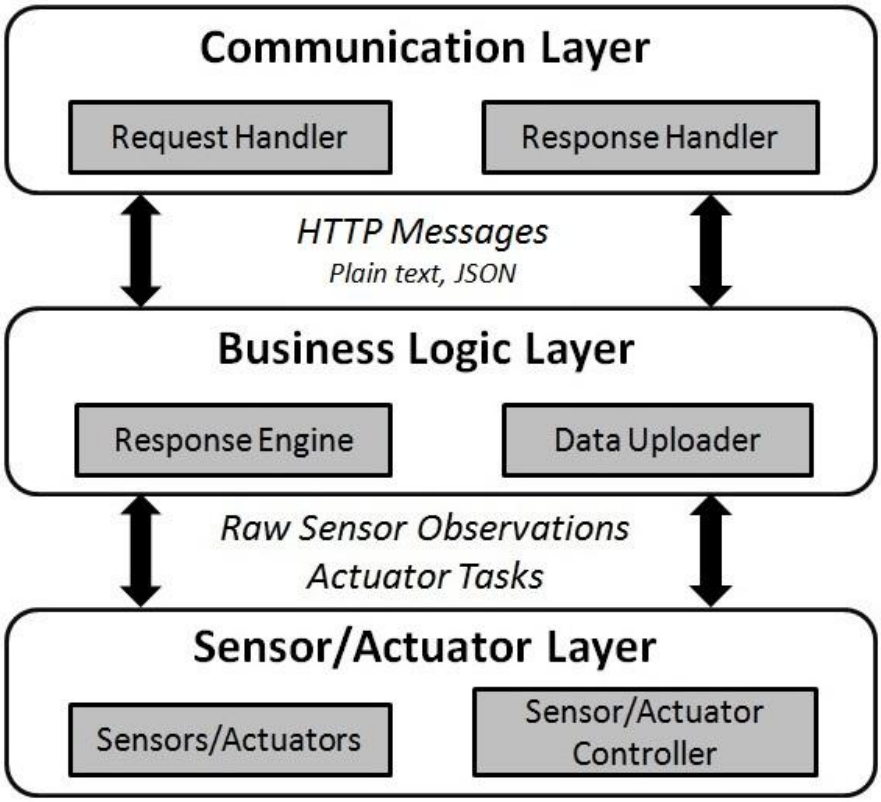

Figure 5.4 The device architecture supporting OGC SensorThings API

1) *Communication Layer:* Similar to the previous protocols, the communication layer contributes in device interactions over the network. Unlike the TinySOS which uses heavy XML, and CoAP message that is included into UDP packet, the OGC SensorThings API applies plain text in all communications except for its registration. When a Thing is registering itself on the server, the requests are formatted in JSON which are already hardcoded in device's memory. Our API uses JSON format only for the registration requests from the device in order to transmit a bunch of data to the RESTful data service. In other cases, the communication is based on the plain text format that is more comfortable for IoT devices to process plain texts with no need to a parser.

2) *Business Logic Layer:* The business logic layer can have the function of both client and server simultaneously. The client role is because a Thing demands to interact with IoT server in order to register itself, and to upload sensor observations. The *data uploader unit* plays the client role once for the registration steps, and frequently for publishing the sensor measurements. To accept tasking requests from clients, the Thing should also contain a server which is named the *response engine* component in this architecture. Similar to the TinySOS and CoAP, the response engine reads HTTP requests buffer by buffer. After processing the requests, the task might be sent to the sensor/actuator layer, and the relevant response is forwarded to the communication layer. Since in OGC SensorThings API a Thing is always connected to a data service, the Thing does not need to record the sensor readings on its own memory. Therefore, unlike the other protocols, on the device architecture of this API (Figure 5.4), the "sensor data repository" component was removed.

3) *Sensor/Actuator Layer:* The sensor/actuator layer consists of the physical sensors, actuators, and their controllers. The sensor controller manages the sensors and actuators. For example, the sensor controller can command sensors to collect sensor measurements, or task actuators to do an action.

5.4 Implementation

One of the main advantages of the SensorThings API is simplicity in case of network communication and device computation. According to Figure 5.1, this API defines three different types of interactions between IoT object and IoT RESTful service: 1) *device registration*, 2) *observation uploading*, and 3) *actuator tasking*. In this section, we describe the implementation of these interactions on a class-1 IoT device.

5.4.1 Device Registration

As we have already described the data model of the OGC SensorThings API, the IoT server should contain IoT devices' information. To do this, when a Thing is connected to the Internet, it automatically registers its resources and properties on the IoT server though the sequences shown in Table 5.1.

Table 5.1 Device registration procedres

Procedure	Response code	Definition
1	Thing	Description
2	Datastreams	Thing ID, Description
3	Tasking Capabilities	Thing ID, Description, Tasking Parameters (Parameter ID, Necessity, Definition, Input Type, Unit, and Range), Protocol (HTTP Method, Resource Path, and Message Body)
4	Sensors	Metadata
5	Actuators	Tasking Capability ID, Metadata
6	Observed Properties	Datastream ID, Unit of Measurement, URN
7	Feature of Internets	Description, Geometry (Type, and Coordination)
8	Location	Thing ID, Time, Geometry (Type, and Coordination)

Despite of XML and JSON which are widely used encoding data formats for human and computer, the SensorThings API uses plain text for the requests sent to a Thing. Thus, the Thing does not require any parser or complicated processing for the incoming messages. On the contrary, the IoT server interacts with users by JSON standard. As the IoT device can conveniently encapsulate JSON requests into string values, we force the device to send its registration requests to the server in the JSON format. Obviously, there is no processing load on the device because those requests have been previously saved on the device's code storage as string variables. Figure 5.5 describes an example where JSON is used to create a Tasking Capability resource on the IoT service.

```
{
        "Thing": {"ID": 10},
        "Description": "This is a TaskingCapability",
        "Parameters":
        {
                "ParameterID": "paramOn",
                "Description": "on",
                "Use": "mandatory",
                "Definition":
                {
                        "InputType": "Integer",
                        "UnitOfMeasurement": "brightness degrees",
                        "AllowedValues": [{ "Min": "10", "Max": "15" }]
                }
        },
        "Protocols":
        {
                "HTTPMethod": "POST",
                "AbsoluteResourcePath": "http://path.com",
                "MessageBody": "lamp-id = 1 & on = {paramOn}"
        }
}
```


Figure 5.5 An example of registration request

However, the responses of those requests are still in plain text format including resource ID, and resource location on the IoT RESTful server (*i.e.*, URL). Figure 5.6 represents a response to a Tasking Capability registration request.


```
Response:  Created
Server: Apache-Coyote/1.1
Location: http://136.159.122.160/SWIOT_V0.9_M/TaskingCapabilities(59)
Access-Control-Allow-Origin: *
Content-Type: application/xml;charset=utf-8
Content-Length: 0
Date: Fri, 25 Oct 2013 01:41:45 GMT
```


Figure 5.6 An example of IoT service response

To prevent an IoT object from repetitive registrations on all its powering, we record the retrieved resource IDs and URLs on the device permanent memory (In our implementation, we

used a micro SD). Therefore, at the time of powering up, the device checks its registration status to decide about the next steps. If all IDs are available, there is no need for new registration on the service.

According to the device architecture (Figure 5.4), the data uploader component provides the aforementioned requests, forwards them to the communication layer, and finally receives responses from the communication layer.

5.4.2 Observations Uploading

Not only does the data uploader perform the registration operations, but also it cooperates in the sensing profile. Therefore, the data uploader that plays the role of a client is responsible to dynamically collect sensor readings, and upload them to the IoT service based on a preset *frequency* (saved on the device). Similar to the registration requests, the sensor observation request is in JSON format, too. The observation request carries datastream ID, sensor ID, feature of interest, observation time, result value, and also result type (e.g., measure). Accordingly, the data service acknowledges the request by messaging the location (URL) of the recorded observation on the data service. If that response does not contain any location value, the observation request will be immediately re-submitted to the IoT service.

5.4.3 Actuator Tasking

Tasking requests are mostly triggered from users to IoT service. As shown in Figure 5.7, that request is encoded in JSON based on the specification of the SensorThings API. The tasking request contains tasking capability ID (to retrieve the device-defined protocol), input parameters and trigger time (when the task should be sent to the device).

```
{
        "TaskingCapability":{"ID":5},
        "Inputs":[
        {
                "ParameterID":"paramOn",
                "Value":200
        }
        ],
        "Time":"2013-04-18T16:15:00-0700"
}
```


Figure 5.7 Tasking request triggered from user to IoT service

During device registration, the tasking capability request introduces the device communication protocol to the data service (Figure 5.5). The device protocol is hidden from the public access and application developers which enables some sort of security and privacy for controlling the device. Additionally, according to the simplicity approach of the SensorThings API, the IoT service can effectively convert the user request from JSON to something simpler (e.g., plain text) to follow the device protocol. In the registration request shown in Figure 5.5, the request to task the lamp actuator should be sent to the specified resource path including the required message body by HTTP POST method.

Unlike the sensing profile, the tasking profile merely acts as a simple web server implemented on an IoT device. Tasking requests delivered by the communication layer are forwarded to the response engine for further processing. As we have already mentioned, all requests to a Thing are in the form of plain text (Figure 5.7).

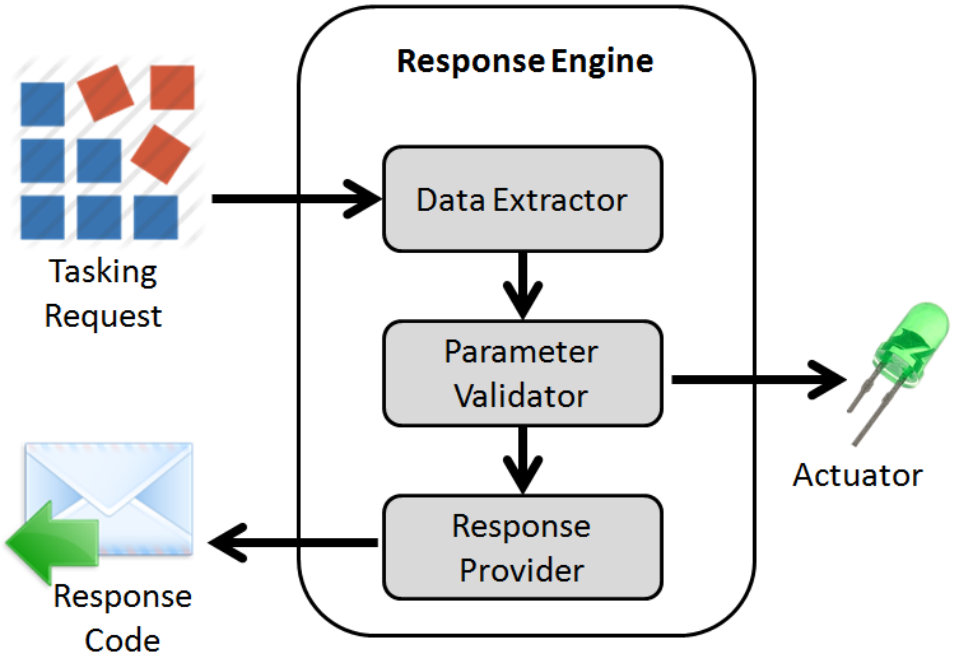

Figure 5.8 Workflow of the response engine

Based on Figure 5.8, the tasking request is sent to the data extractor component to extract the tasking capability ID, parameter name, and parameter value. Then, the request validator module examines the retrieved values with the device actuators in case of resource availability, parameter validation, and also request type. After validation process is performed, the actuator might be tasked, and the relevant response code is delivered to the communication layer. The response codes of the response engine are listed in Table 5.2.

Table 5.2 Response codes of the response engine

Response code	Definition
600	Confirmed
610	Resource is busy
620	Parameters are missing
630	Out of range parameters
640	Invalid parameters
650	Invalid actuator
660	Invalid request type

Depending on the request parameters, the appropriate response in JSON encoding is forwarded to the communication layer. As a server should always remain online, the response engine does not sleep at all in order to listen to incoming messages.

5.5 Discussion

The OGC SensorThings API establishes an easy-to-develop and easy-to-use protocol for the resource constraint IoT devices. This protocol was mainly inspired from OData, and OGC SWE standards (SOS, and SPS). In spite of simplicity demonstrated in this API, there would be several issues listed as follows.

The first immediate issue is that each IoT device should have a stable and constant Internet connection in order to receive tasks from outside, and to upload sensor readings to the server. Obviously, there is no obligation for a Thing to upload its sensor readings to the IoT service. On contrary, the server side of the Thing is expected to be always online. One potential solution for this is to consider the IoT service as a forward proxy between user and IoT device. By the way, the proxy can check the device availability on the network before any interactions. If the Thing is detected out of the network, the proxy can notify the user from this situation.

The second issue of the proposed API is that it forces the limited resource devices to interact through HTTP standard and TCP packets. As we illustrated in CoAP, UDP is significantly more efficient than TCP in packet transmissions. In order to address this issue, one potential solution is to combine partially the CoAP and OGC SensorThings API. In other words, a bit change from TCP to UDP in SensorThings API will catch a large achievement in the future.

Since the IoT RESTful service is an intermediary node between user and IoT device, the privacy is strongly supplied by the IoT service. On the other hand, the security issue is still remained because no strategy is considered for the data transmission. To overcome this problem,

we can simply define a bidirectional rules for the message encryption/decryption to guarantee message integrity and confidentiality.

5.6 Summary

In this chapter, the OGC SensorThings API was elaborated as a prospective open standard for the Internet of Things. In this protocol, we reduced the complexity on the device by simplifying the message format, and lessening message size. The transactions were basically established thorough JSON language, except the ones sent to the IoT device, so the device does not need any resource consuming parser. Likewise, the IoT device transmits the requests to the server in hard-coded JSON format only because JSON is more suitable for the IoT RESTful service.

By hosting the simple open standard API on the IoT devices, not only a great range of devices can apply that protocol, but also innovative applications can be developed more conveniently by means of a standard interface.

Chapter Six: **Evaluation and Results**

6.1 Introduction

The objectives of this chapter is as follows: (1) to benchmark the efficiency of the implemented protocols on a class-1 IoT object, (2) to provide a quantitative guideline for developers to choose the interoperable protocol that is suitable to their applications. In general, this chapter evaluates the four standard protocols developed in this research. We assess the *performance* of those protocols on a class-1 IoT object. By performance, we mean the measurement of the degree to which a system accomplishes its functions within given constraints such as CPU speed, memory, bandwidth, and so forth [76].

In our test environment, we choose Netduino Plus introduced in Section 1.7 as our development platform as a class-1 IoT object. In order to demonstrate how different components work together, multiple meteorological sensors (temperature, humidity, carbon monoxide, hydrogen monoxide, and dust), sound pressure sensor and LED actuator are connected to our Netduino Plus (Figure 6.1).

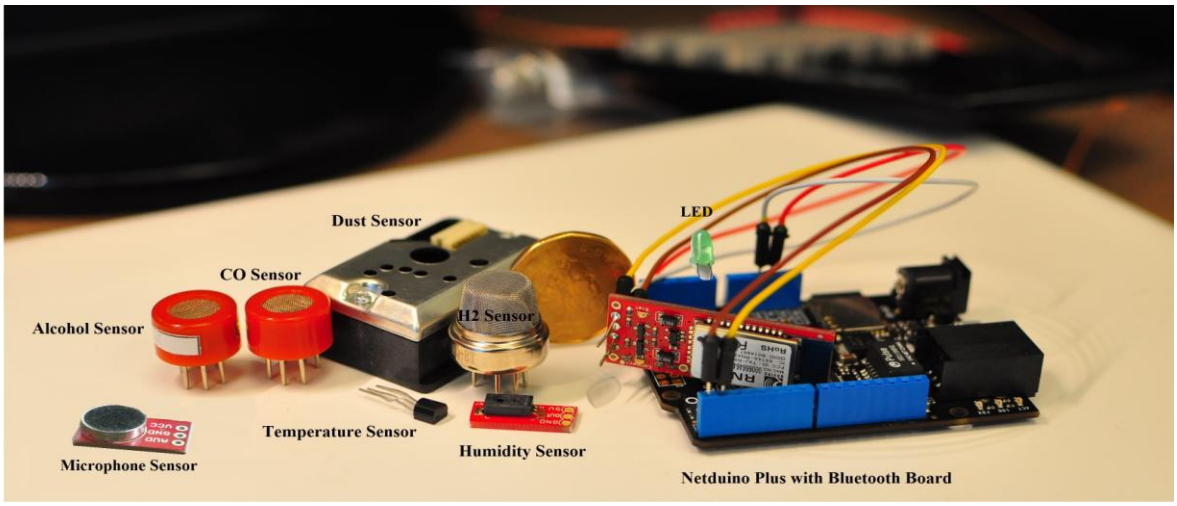

Figure 6.1 Different components of our development platform

Although more powerful IoT objects exist, they are also more expensive. The reason we focused on the constrained nodes in this research is because they are more cost-effective and will be more widely deployed in real world. By using the resource-constrained and cost-effective nodes, it allows us to explore the lower bound of the resources that are required for IoT applications. In that way, we ensure that our design choices can deliver an efficient implementation suitable for a broader application domain.

6.2 Performance Evaluation

This section evaluates each protocol using a service prototype (*i.e.*, server), a gateway (where applicable), and a client. The metrics selected for this evaluation are as follows: (1) code storage (EEPROM[23]) occupation, (2) main memory (RAM) usage, (3) request length of an operation, (4) response size of an operation, and (5) response latency. In all cases except SensorThings, the tests are carried out using a Netduino Plus as the server and a PC as the client. The two are connected via Ethernet cable to the Internet.

6.2.1 Memory Occupation

The first experiment is about memory occupation (*i.e.*, ROM and RAM usage). The results obtained in this experiment demonstrate memory management's importance in terms of resource consumption. We also include a HTTP web server in our tests as a reference. The HTTP web server is implemented on Netduino Plus and responds in plain unstructured text format. This web service can be a reference because it is purely developed by using C# HTTP libraries with no enhancement on the code efficiency.

[23] Electronically Erasable Programmable Read Only Memory

First, we measure the occupied code space after code deployment from the development environment (e.g., a PC) to the EEPROM of the Netduino Plus. The occupation of ROM can serve as an indicator of the required code's complexity for each implementation. For example, according to Table 6.1, the OGC SensorThings API and SOS over CoAP need more ROM in comparison to the other implementations, because both not only need to handle server-side operations but also should support client-side functions. The simple web service is in the third place of ROM usage as the classes and libraries in the C# .Net Micro Framework consume a considerable amount of code storage [50]. Comparing to the simple web server, TinySOS is more efficient because of two reasons: (1) rather than using the C# .Net Micro Framework's libraries, we implemented our own HTTP libraries; (2) we recorded the XML responses on the micro SD card instead of ROM. The OGC PUCK is the most efficient protocol in terms of ROM usage because PUCK specification does not require any heavy parser (e.g., XML parser, JSON parser), retransmission mechanism (e.g., CoAP-To-HTTP), and data uploader component. Although the OGC PUCK requires PUCK memory, SensorML and driver code, we are able to use the device's permanent memory (micro SD card) to keep those necessary data.

Moreover, Table 6.1 shows the amount of RAM allocated at compile time for each implementation. A code with a small memory footprint would allow adding extra capabilities such as resources that the server could provide to clients. Although PUCK occupies the least code space, this protocol is highly inefficient in RAM usage. It is possible that the memory management unit or data transceiver of the Bluetooth module requires more memory in comparison with other components of this protocol. After the OGC PUCK, TinySOS consumes a lot of RAM likely due to the XML parser and request validator units. SOS over CoAP and OGC SensorThings are similar in terms of RAM usage. On the other hand, the simple web server acts

better than others in this experiment since it is simple in case of request validation and response generation.

Table 6.1 RAM and ROM memory occupation

	Simple Web Service	PUCK over Bluetooth	TinySOS	SOS over CoAP	OGC SensorThings
ROM (kB)	16.08	8.48	11.72	29.13	26.11
RAM (kB)	9.54	13.15	11.33	10.36	10.21

6.2.2 Request Size

Both IoT devices and the network they use are highly constrained [16]. And that means the payload packet size is very important. To identify the efficiency of the above standard protocols, we record the request size generated for a specific use case (*i.e.*, get one sensor measurement) that is possibly most widely used. To do this, we use Wireshark[24], a network protocol analyzer software for all tests except PUCK. That is Wireshark is unable to monitor the serial ports that are the communication ports of the PUCK. Thus, to measure the PUCK request size, we simply count the characters of its plain text request. According to Figure 6.2, PUCK generates the smallest request since the request is made of a short string of characters with no header, description or complicated format. Also, CoAP request is at least 67% smaller in comparison with other Internet-based protocols. This efficiency is because of using UDP instead of TCP in the transport layer which makes the header size extremely smaller. The simple web service communicates through HTTP GET request with no request content. Therefore, only the header

[24] http://www.wireshark.org

features of the HTTP GET request (350 bytes) are calculated for the simple web service. The OGC SensorThings requires several parameters embedded in the request body besides the header features. Therefore, SensorThings is ranked after the HTTP protocol in this experiment. On the contrary, the requests of the SOS protocol are at least 47% larger than other protocols since they are packaged in XML format. In order to ensure that the tested SOS request is compatible with the OGC SOS standard, we used a test client tool developed by 52 North SOS[25].

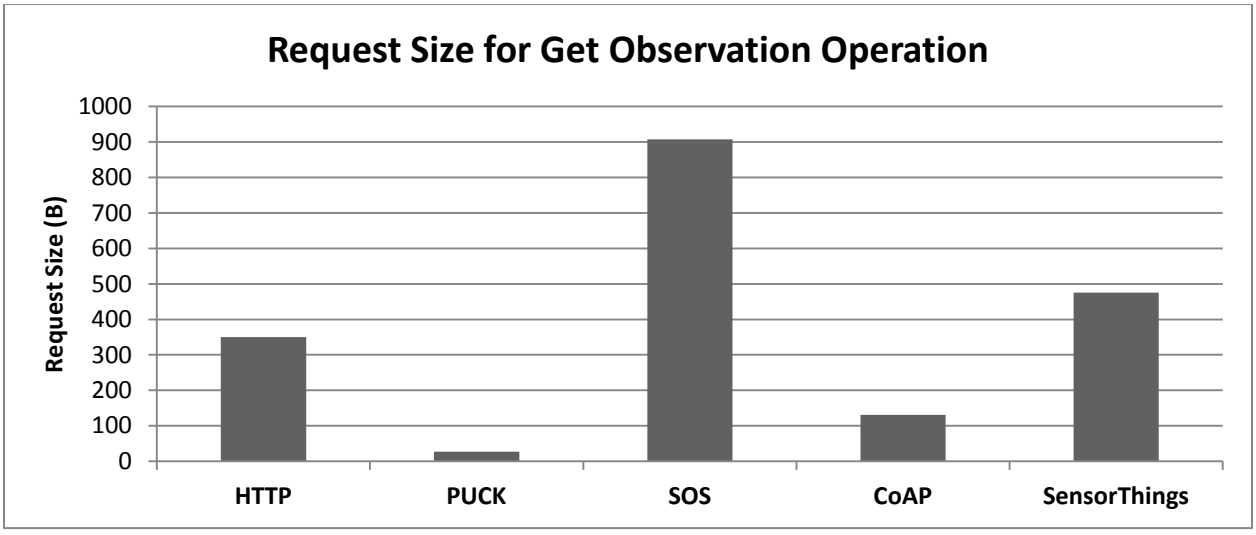

Figure 6.2 Request size evaluation for the get observation request

6.2.3 Response Length

Apart from the requests comparison among the standard protocols, we also evaluate the response length generated by our implementations. Figure 6.3 depicts the response length trend of different implementations versus the number of sensor readings requested (from 1 to 100). Since the specification of OGC SensorThings conveys the sensor related requests to a RESTful data service, we send the get observation request to that data service[26] (a regular PC) instead of

[25] http://sensorweb.demo.52north.org/52nSOSv3.2.1/
[26] http://demo.student.geocens.ca:8080/SensorThings_V1.0

Netduino Plus. According to Figure 6.3, OGC SensorThings and TinySOS provide larger responses in comparison with other protocols. One possibility of this difference can be the output formatting which is in JSON and XML, respectively. After looking at the responses generated by OGC SensorThings data service, we faced several JSON attribute-value pairs (e.g., observation ID, request type, feature of interest, sensor profile, and data stream information) repeated in all sensor readings (Appendix A). Based on the capabilities of the SensorThings data service, we are able to retrieve only the sensor measurement and the observation time in JSON format (Appendix A). As a result, the response length would be 71% less in average comparing to the previous responses of the SensorThings API. On the other hand, TinySOS follows the OGC SOS specification for the response generation by embedding the observation values and time in the existing response file. Accordingly, the response size will not be as large as OGC SensorThings protocol with repetitive attribute-value pairs. As a trade-off, end users can simply parse the SensorThings responses by a JSON parser while for the TinySOS responses, a new parser needs to be developed to extract the required data from the XML file.

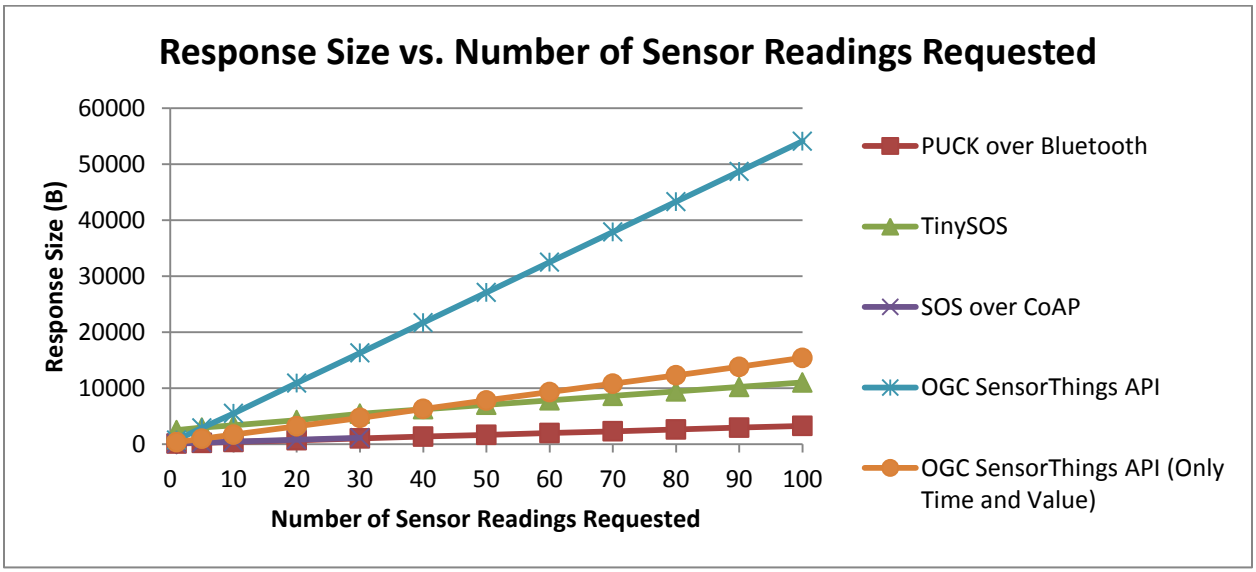

Figure 6.3 Response size vs. the number of sensor readings

To better understand the trends of other implementations, we remove the OGC SensorThings trend in Figure 6.4. The SOS over CoAP and PUCK over Bluetooth follow each other closely since the protocols defined to retrieve the sensor readings are similar for both. According to the CoAP specification elaborated in Section 4.3.1, CoAP messages should not exceed 1024 bytes [55]. That explains why the green line representing SOS over CoAP in Figure 6.4 has not gone any further than point 30 in which the response size was 1019 bytes. However, the required response header of CoAP makes the CoAP response size a bit larger than the one outputted by PUCK for cases with equal number of sensor readings.

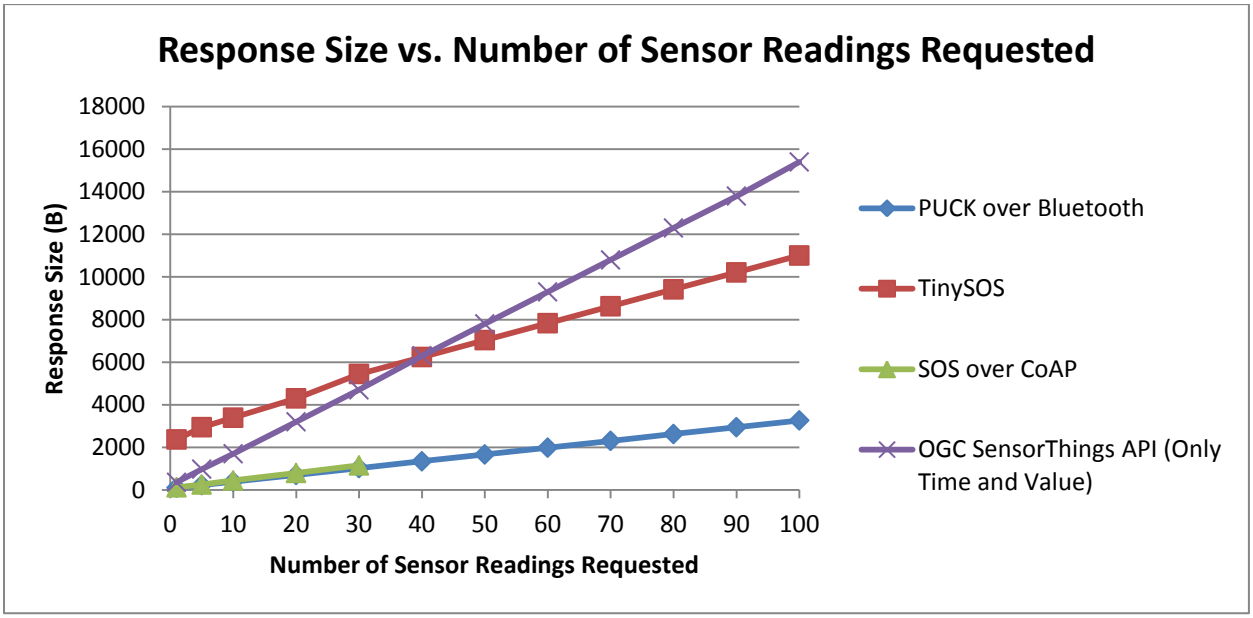

Figure 6.4 Response size vs. the number of sensor readings (removed the OGC SensorThings trend)

According to Figure 6.4, TinySOS and OGC SensorThings API generate the same response size for forty senor observations. Due to the fact that the observation values and time are the same for the two protocols, we can conclude that the size of XML tags of the SOS

response is equal to the total length of the JSON attribute-value pairs of the SensorThings response (i.e., “time”, “result value”, “self-link”)[27].

6.2.4 Response Latency

To wrap up our performance evaluation, we record the end-to-end response latency. The experiment is conducted by a PC client to retrieve sensor data from a Netduino Plus-based service or from a PC-based IoT data service. We define latency as the time elapsed from the moment the PC client sends a request until the moment it receives the response. Figure 6.5 shows the latency trend based on our experiments. Each point on Figure 6.5 represents the latency value of successful request/response transactions. Number of sensor readings ranges from 1 to 100. In this way, the differences between the other implementations can be better appreciated. Low latency values can notably improve the user experience and benefit the implementations that work in real-time.

TinySOS behaves worse than others in this experiment as its communications are in XML data encoding. Thus, the Netduino Plus server has to parse the XML request, read the XML response file from the micro SD card, embed the sensor reading(s) into the response body, and forward the XML file to the client. All these functions are performed on a device with 48 MHz CPU speed and 28 KB memory leading to high latency.

[27] Appendix A provides a sample of such response.

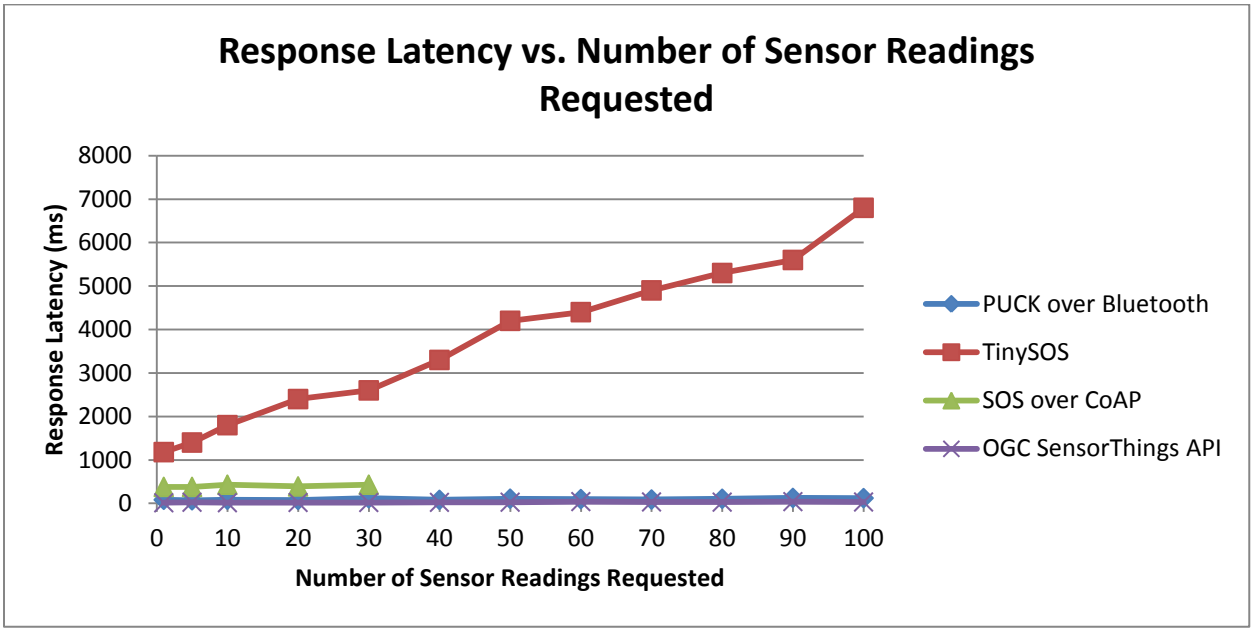

Figure 6.5 Response latency vs. the number of sensor readings

Figure 6.6 removes the TinySOS trend in order to determine the behaviour of other implementations. The SOS over CoAP has more latency than PUCK since the CoAP communicates over the World Wide Web. As we explained in Section 6.2.3, CoAP stops at point 30 because of the CoAP limitation for the message size. Due to the fact that the SensorThings data service is a regular PC, if we ignore this protocol, the PUCK over Bluetooth is the most efficient implementation in this experiment. For the PUCK evaluation, we applied Device Monitoring Studio software[28] in order to monitor the serial port of the PC. Since PUCK over Bluetooth is a wireless protocol, the distance between the pairs affects the response latency. In our experiments, the Netduino Plus (server) and the notebook (client) were placed close to each other (less than 1 meter).

[28] http://www.hhdsoftware.com/device-monitoring-studio

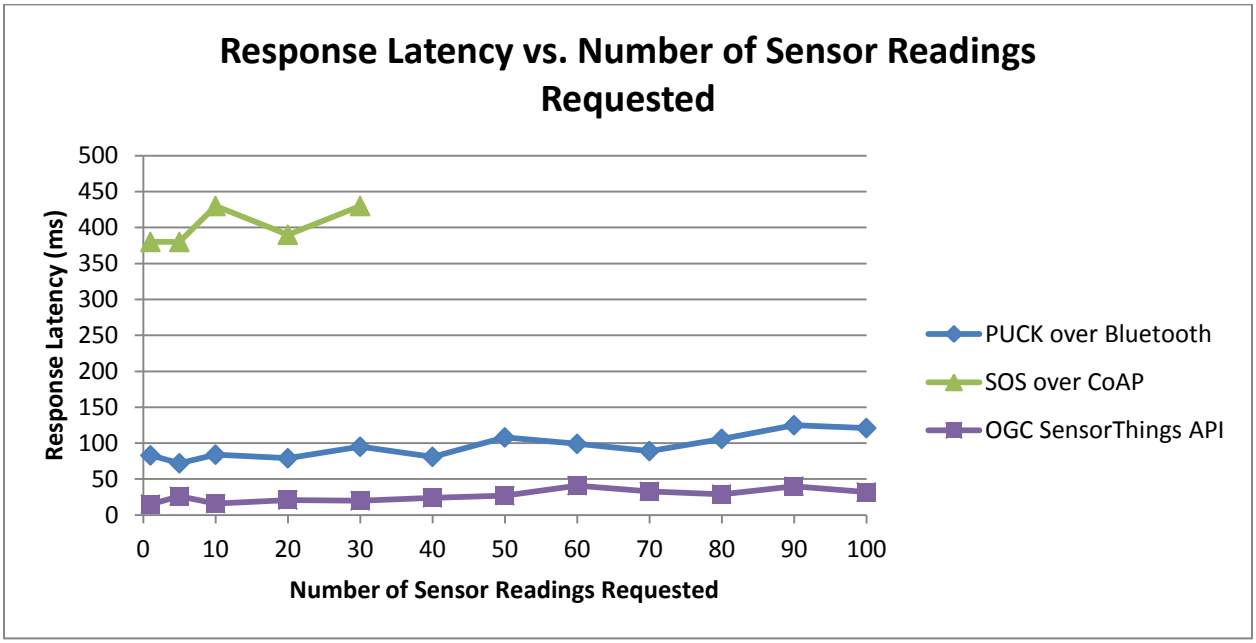

Figure 6.6 Response latency vs. the number of sensor readings (removed TinySOS)

6.3 Summary

We demonstrated the effectiveness of our approaches by a comprehensive performance evaluation. To conduct a comparative study on the applied protocols, we compared the test results together in terms of code storage occupation, memory usage, request size, response size, and finally response latency.

Chapter Seven: **Conclusions and Future Works**

7.1 Introduction

We conclude this research in the final chapter by summarizing the research work that has been carried out and outlining the conclusions drawn out of the result. It also comments on the limitations and proposes areas for the future work.

This research has contributed in the interoperability approach of the Internet of Things by adapting existing standards defined for the Sensor Web (PUCK and SOS), implementing the newly introduced protocol for the Internet of Things (CoAP), and eventually designing a specific RESTful protocol for the Internet of Things (OGC SensorThings). Besides, a real-time meteorological system has been developed in this research as a proof of concept to evaluate the adequacy and efficiency of the interaction.

In general, our contributions could overcome the three problems mentioned in Chapter 1. For the Internet access problem, we applied PUCK over Bluetooth protocol. We also implemented the four standard protocols to solve the lack of standardization in data representation. As a solution for the third problem, we demonstrated the possibility of implementing open standards on a resource constrained IoT object.

To summarize, Chapter 1 provided a brief introduction regarding the topic of this research, development platform and outlined the research problem and the key objectives. Chapter 2 presented PUCK over Bluetooth as a wireless standard protocol for the Internet of Things. Chapter 3 described TinySOS as a lightweight profile of the OGC SOS suitable for IoT devices. Chapter 4 also discussed about the possibilities of combining CoAP which is mainly designed for the Internet of Things, and OGC SOS which is commonly used in WSNs. Moreover, Chapter 5 introduced our defined protocol under the OGC license supporting REST

and JSON, namely OGC SensorThings. Finally, Chapter 6 presented several scenarios to evaluate the performance of the implemented protocols on a class-1 IoT object.

7.2 Conclusions

All around the world, the IoT applications are emerging exponentially with various functionalities. Each application is developed based on the developer's desire of the device. That means the number of proprietary protocols is growing as the number of IoT devices increases. Consequently, standardized interfaces are required to interconnect different IoT devices for innovative applications.

In this research, we presented our contribution in the interoperability aspect of the Internet of Things by developing PUCK over Bluetooth, TinySOS, SOS over CoAP, and finally the OGC SensorThings API. Our implementations were the world's first contribution for the IoT objects.

At the beginning, we chose a class-1 IoT object as categorized in the framework of Bormann et al. [14] for our development platform. First, we equipped the class-1 IoT device with a Bluetooth transceiver in order to establish wireless network in a limited range. We standardized its connection by means of OGC PUCK. Due to the fact that Internet access is a key requirement for IoT objects, we applied additional software components to farther enhance this functionality for the Bluetooth-enabled PUCK instrument.

In the second stage, we removed the intermediary gateway in the path between the user and IoT object by developing a web service on a Thing itself. Since different device owners or manufacturers might have their own design for the data representation, we introduced a lightweight version of the OGC SOS, TinySOS. As a result, the sensor measurements could be accessed remotely in a standardized way simply through a web browser.

According to the complicated nature of the OGC SOS for resource constraint IoT objects, we proposed another approach which is more suitable for the class-1 IoT devices. Thus, the third contribution of this research was integrating CoAP into the OGC SOS. The point of differentiation between this approach and the previous effort (TinySOS) lies in the connection of the device to the network. In the previous approach, the device was required to tolerate the huge load of SOS requests/responses formatted in XML. In the new approach, the device supported CoAP which is a constrained application protocol for the IoT. Moreover, the SOS operations were processed on the CoAP proxy which is essentially a regular computer with enough computational resources.

Due to the UDP transmission, CoAP could not establish a direct connection to the Internet components without the deployment of CoAP proxies. As the IoT will eventually follow the Internet protocol suite model, it is recommended to adjust the connections compatible with the Internet standard protocols. Moreover, the IoT infrastructure needs a specific standard protocol. As a result, we designed our own standard application programming interface called OGC SensorThings API. The use cases of this API started with IoT device registrations to the service. For the sensing devices, registration information contained the phenomenon that was observed. After registration, sensing devices could start uploading their observations to the data service. From the tasking point of view, actuators could also register and publish their tasking capabilities to the data service. As a result, users were able to access those observations and also send controlling tasks to the devices through the service. All the communications with the data service followed the RESTful architecture.

Finally, the four implementations on Netduino Plus were assessed comparatively. To do so, each implementation was evaluated according to memory occupation (RAM and ROM),

request size, response length and response latency. As a case study, we embedded multiple meteorological sensors, sound pressure sensor and LED actuator to our Netduino Plus in order to demonstrate how different components work together.

In the following section, we will discuss our future works on the IoT. Several challenges in the context of IoT will be discussed and a future road map will be presented.

7.3 Future Works

This thesis takes a practical approach to the interoperability in the Internet of Things. There are several ways in which this research can be improved and extended. In this section, some of the major issues that can be later investigated and guidelines of the future work is addressed.

First of all, the aforementioned interoperable protocols follow the client-server architectural style which has the single point of failure (SPOF) issue [77]. In order to address this issue, one potential solution is to design a peer-to-peer (P2P) architecture as it has been proven reliable and effective. In this case, devices can form an overlay network to discover resources and forward requests; so a centralized component such as the sensor registry service, CoAP proxy and IoT RESTful server would be no longer needed.

In this research, Bluetooth and Ethernet were considered as the network enablement technologies for IoT devices. Since Wi-Fi is being dominant in network communications [78], the study on Wi-Fi communications in the IoT is highly suggested. One immediate issue in Wi-Fi connection emerges about transmission of network configuration to the IoT device which has no display equipment and input peripheral.

In addition to Wi-Fi as wireless enablement for the IoT, another research should be started to improve energy saving on the IoT objects. The first assumption in this research was that IoT devices having unlimited power resource while this assumption may not be true in many

cases. Some sensor nodes will be battery-operated [52], so energy is perhaps the most notable constraint for the IoT devices. Furthermore, achievement in Wi-Fi connection of IoT objects leads to removing wires and cords from devices. Therefore, their battery charge must be efficiently conserved to extend the life of the individual sensor node, and consequently the entire IoT network.

More potential future works pertain privacy and security for IoT devices. We efficiently implemented existing security and privacy mechanisms of the information technology and computer networks on class-1 IoT devices [79]. Although an acceptable level of secure connection in IoT can be achievable, we believe IoT would require specific rules and mechanisms for the successful implementation of this approach.

This research mainly concentrated on the way the data is transferred from inexpensive class-1 IoT objects. However, we do not know how reliable the retrieved data is. According to the IEEE Standard Computer Dictionary [20], *reliability* is defined as the capability of a sensor to perform its measurements under stated conditions for a specific time period. By this definition, we can intuitively link a sensor's reliability to *accuracy* and *precision*. Accuracy denotes the closeness of a measurement to the actual value, and precision characterizes the reproducibility of the generated value. In order to achieve the reliability in the IoT, both precision and accuracy should be considered. For future directions, we believe that data reliability is an important and interesting approach for the IoT that is worth to be further investigated.

References

[1] Mike Botts, Alex Robin (Oct. 2007). "Bringing the Sensor Web Together". *Geosciences*. pp. 46–53.

[2] Dargie, W. and Poellabauer, C., "Fundamentals of wireless sensor networks: theory and practice", *John Wiley and Sons*, 2010. ISBN 978-0-470-99765-9, pp. 7-10.

[3] Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D., and Anderson, J., 2002. "Wireless Sensor Networks for Habitat Monitoring." In *2002 ACM International Workshop on Wireless Sensor Networks and Applications*. Atlanta, US.

[4] Hart, J.K. and Martinez, K. 2006. "Environmental Sensor Networks: A revolution in the earth system science?" *Earth Science Reviews*. 78, 177-191.

[5] Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S., and Turon, M., 2007. "Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks." In *the 2007 ACM International Workshop on Information Processing in Sensor Networks*. Massachusetts, USA.

[6] Schwiebert, L., Gupta, S.K.S., and Weinmann, J., 2001. "Research Challenges in Wireless Networks of Biomedical Sensors." In *the 2001 ACM International Workshop on Mobile Computing and Networking*. Rome, Italy, 151-165.

[7] Kassab, A., Liang, S., and Gao, Y., 2010. "Real-Time Notification and Improved Situational Awareness in Fire Emergencies using Geospatial-based Publish/Subscribe." *International Journal of Applied Earth Observation and Geoinformation*. 12, 6, 431-438.

[8] Lynch, Jerome P., et al. "The development of a wireless modular health monitoring system for civil structures." *MCEER Mitigation of Earthquake Disaster by Advanced Technologies (MEDAT-2) Workshop*. 2000.

[9] M.F. Goodchild, "Citizens as sensors: the world of volunteered geography," *GeoJournal 69* (2007), pp. 211-221.

[10] Boulos, Maged N. Kamel, et al. "Crowdsourcing, citizen sensing and sensor web technologies for public and environmental health surveillance and crisis management: trends, OGC standards and application examples." *International journal of health geographics 10.1* (2011): 67.

[11] A. Sheth. "Citizen Sensing, Social Signals, and Enriching Human Experience." *IEEE Internet Computing*, 13(4):87-92, 2009.

[12] Declan Butler, (2006), Virtual globes: The web-wide world, Nature. [online]. Available: http://www.nature.com/nature/journal/v439/n7078/full/439776a.html [Accessed 1 August 2013].

[13] Aggarwal, Charu C., Naveen Ashish, and Amit Sheth. "The Internet of Things: A Survey from the Data-Centric Perspective." *Managing and Mining Sensor Data*. Springer US, 2013. 383-428.

[14] Liang, S.H.L., Croitoru, A., and Tao, C.V., 2005. "A Distributed Geospatial Infrastructure for Sensor Web." *Computers and Geosciences*. 31, 2, 221-231.

[15] TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU, *Overview of the Internet of things, Recommendation ITU-T Y.2060*, 2012.

[16] Bormann, C., Castellani, A.P., Shelby, Z, 2012. "CoAP: An Application Protocol for Billions of Tiny Internet Nodes," *IEEE Internet Computing*, Volume: 16, Issue: 2, Page(s): 62-67.

[17] Ericsson: “More than 50 Billion Connected Devices”, [online]. *White Paper*, February 2011, Available: http://www.ericsson.com/res/docs/whitepapers/wp-50-billions.pdf [Accessed 1 August 2013].

[18] Evans, Dave. "The Internet of Things: How the next evolution of the internet is changing everything." *CISCO white paper* (2011).

[19] Jeronimo, Michael, and Jack Weast. "UPnP design by example." Vol. 158. Intel Press, 2003.

[20] Institute of Electrical and Electronics Engineers (1990), "IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries." New York, NY. ISBN 1-55937-079-3.

[21] Rodriguez, M. J., et al., 2013. "Combining Wireless Sensor Networks and Semantic Middleware for an Internet of Things-Based Sportsman/Woman Monitoring Application." *Sensors*, 13(2), 1787-1835.

[22] Cox, S., 2011. Observations and Measurements - XML Implementation [online]. *Open Geospatial Consortium*. Available from: http://portal.opengeospatial.org/files/?artifact_id=41510 [Accessed 1 September 2013].

[23] Botts, M., 2007. OpenGIS® Sensor Model Language (SensorML) Implementation Specification [online]. *Open Geospatial Consortium*. Available from: http://portal.opengeospatial.org/files/?artifact_id=21273 [Accessed 1 February 2012].

[24] Broering, A., Below, S., 2010. OpenGIS® Sensor Interface Descriptors (SID) [online]. *Open Geospatial Consortium.* Available from: http://portal.opengeospatial.org/files/?artifact_id=39664 [Accessed 1 September 2013].

[25] Na, A., Priest, M., 2007. Sensor Observation Service [online]. *Open Geospatial Consortium.* Available: http://www.opengeospatial.org/standards/sos [Accessed 1 September 2013].

[26] Open Geospatial Consortium, 2011, OGC® Sensor Planning Service Implementation Standard [online]. *Open Geospatial Consortium*. Available: http://portal.opengeospatial.org/files/?artifact_id=38478 [Accessed 1 September 2013].

[27] O'Reilly, T., 2012. OGC® PUCK Protocol Standard Version 1.4 [online]. *Open Geospatial Consortium*. Available: https://portal.opengeospatial.org/files/?artifact_id=47604 [Accessed 20 August 2013].

[28] Peterson, Larry L., and Bruce S. Davie. *Computer networks: a systems approach*. Elsevier, 2007.

[29] Lerche, C., Hartke, K., Kovatsch, M. "Industry Adoption of the Internet of Things: A Constrained Application Protocol Survey." In *Proceedings of the 7th International Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE 2012)*, Kraków, Poland, 17–21 September 2012.

[30] Mohammad Ali Jazayeri, Chih-Y Huang, and Steve H. L. Liang. "TinySOS: design and implementation of interoperable and tiny web service for the internet of things." In *Proceedings of the First ACM SIGSPATIAL Workshop on Sensor Web Enablement*. ACM, 2012.

[31] Netduino Plus, Secret Labs LLC, 2010. [online]. Available: http://www.netduino.com/, 2011 [Accessed 1 September 2013].

[32] Woods, Stan, M. Geipel, and F. Gen-kuong. "IEEE-P1451. 2 smart transducer interface module." In *Proceedings Sensors Expo Philadelphia*. 1996.

[33] OGC Network (n.d.). "How to model your observation data in SOS 2.0?." *Open Geospatial Consortium, Inc*. 4 October 2013. [online]. Available: http://www.ogcnetwork.net/sos_2_0/tutorial/om [Accessed 1 September 2013].

[34] Whiteside, A. "Definition identifier URNs in OGC namespace." *OpenGIS Best Practice document*, OGC (2009).

[35] Shapiro, Marc. "Structure and encapsulation in distributed systems: the proxy principle." *icdcs*. 1986.

[36] Open Geospatial Consortium (n.d.). "Sensor Web Enablement (SWE)." *Open Geospatial Consortium, Inc.* 27 January 2014. [online]. Available: http://www.opengeospatial.org/ogc/markets-technologies/swe [Accessed 1 February 2014].

[37] Delin, K.A., and Jackson, S.P., 2001. "The Sensor Web: A New Instrument Concept." *Symposium on Integrated Optics: International Society for Optics and Photonics*, 15 May 2001, USA: SPIE, 1-9.

[38] Leopold, M., et al., 2003. "Bluetooth and sensor networks: A reality check." In *Proceedings of the 1st international conference on embedded networked sensor systems*, 5-7 Nov. 2003 Los Angeles, LA: ACM, 103-113.

[39] Hill, J., et al., 2000, "System architecture directions for networked sensors." *ACM SIGOPS operating systems review*, 34(5), 93-104.

[40] Bluetooth SIG, 2013. Bluetooth Basics [online]. *Bluetooth SIG, Inc*. Available: http://www.bluetooth.com/Pages/Basics.aspx [Accessed 2 September 2013].

[41] Ferrari, P., et al., 2005. "A Bluetooth-based sensor network with web interface. Instrumentation and Measurement," *IEEE Transactions on*, 54(6), 2359-2363.

[42] A. Giusto, G. Morabito, and L. Atzori, The Internet of Things. *Springer*, 2010.

[43] Nissanka B. Priyantha, Aman Kansal, Michel Goraczko, Feng Zhao, 2008. "Tiny Web Services: Design and Implementation of Interoperable and Evolvable Sensor Networks," In *Proceedings SenSys '08 Proceedings of the 6th ACM conference on Embedded network sensor systems*, Pages 253-266.

[44] Botts, M., Percivall, G., Reed, C., and Davidson, J., 2007. OGC Sensor Web Enablement: Overview and High Level Architecture (OGC 07-165). *Open Geospatial Consortium white paper*, 28 Dec. 2007.

[45] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A.J. Brush, Bongshin Lee, Stefan Saroiu, and Victor Bahl, The Home Needs an Operating System (and an App Store), In *HotNets IX, ACM*, 20 October 2010.

[46] George Percival, Steve Liang, 2012. "Charter for a SWG: Internet of Things (IoT) REST API [online]. Available: https://portal.opengeospatial.org/files/49608 [Accessed 2 October 2013].

[47] A. Broring, A. Remke, and D. Lasnia, "SenseBox - A Generic Sensor Platform for the Web of Things." In *Proceedings of MobiQuitous*, 2011, pp.186-196.

[48] Bernd Resch, Manfred Mittlboeck, and Michael Lippautz, 2010. "Pervasive Monitoring—An Intelligent Sensor Pod Approach for Standardised Measurement Infrastructures," *Sensors 2010*, 10(12), 11s440-11467.

[49] Open Geospatial Consortium, 2007, OpenGIS® Catalogue Services Specification [online]. Available: http://portal.opengeospatial.org/files/?artifact_id=20555 [Accessed 20 September 2013].

[50] Cuno Pfister, Getting Started with the Internet of Things, O'Reilly, pp. 30-100, 2011.

[51] Liang, S. H. L., S. Chen, C. - Y. Huang, R. - Y. Li, D. Y. C. Chang, J. Badger, and R. Rezel, "GeoCENS: Geospatial Cyberinfrastructure for Environmental Sensing ", *GIScience 2010*, Zurich, Switzerland, 09/2010.

[52] Ralph C. Merkle, "A Certified Digital Signature," In *Gilles Brassard, ed., Advances in Cryptology – CRYPTO '89*, vol. 435 of Lecture Notes in Computer Science, pp. 218–238, Spring Verlag, 1990.

[53] Tsiftes, Nicolas, Joakim Eriksson, and Adam Dunkels. "Low-power wireless IPv6 routing with ContikiRPL." In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 2010.

[54] Van Dam, Tijs, and Koen Langendoen. "An adaptive energy-efficient MAC protocol for wireless sensor networks." In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*. ACM, 2003.

[55] Shelby, Zach, Klaus Hartke, and Carsten Bormann. "Constrained Application Protocol (CoAP)." *draft-ietf-core-coap-18*, 2013.

[56] Fielding, Roy T., and Richard N. Taylor. "Principled design of the modern Web architecture." *ACM Transactions on Internet Technology (TOIT) 2.2* (2002): 115-150.

[57] Kuladinithi, K.; Bergmann, O.; Potsch, T.; Beckera, M.; Gorg, C. Implementation of CoAP and its Application in Transport Logistics. In *Proceedings of Extending the Internet to Low power and Lossy Networks (IP+SN 2011)*, Chicago, IL, USA, 11 April 2011.

[58] Kovatsch, M.; Duquennoy, S.; Dunkels, A. A Low-Power CoAP for Contiki. In *Proceedings of Eighth IEEE International Conference on Mobile Ad-Hoc and Sensor Systems*, Valencia, Spain, 17–22 October 2011; pp. 855–860.

[59] Ludovici, Alessandro, Pol Moreno, and Anna Calveras. "TinyCoAP: a novel constrained application protocol (CoAP) implementation for embedding RESTful web services in wireless sensor networks based on TinyOS." *Journal of Sensor and Actuator Networks* 2.2 (2013): 288-315.

[60] Curbera, Francisco, et al. "Unravelling the Web services web: an introduction to SOAP, WSDL, and UDDI." *Internet Computing, IEEE* 6.2 (2002): 86-93.

[61] Moritz, Guido, Frank Golatowski, and Dirk Timmermann. "A lightweight SOAP over CoAP transport binding for resource constraint networks." *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*. IEEE, 2011.

[62] Lerche, C.; Laum, N.; Moritz , G.; Zeeb, E.; Golatowski, F.; Timmermann, D. Implementing Powerful Web Services for Highly Resource-Constrained Devices. In *Proceedings of 7th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, Seattle, WA, USA, 21–25 March 2011; pp. 332–335.

[63] Groba, H.; Clarke, S. Web Services on Embedded Systems: A Performance Study. In *Proceedings of 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, Mannheim, Germany, 29 March—2 April 2010; pp.726–731.

[64] Castellani, A.P.; Gheda, M.; Bui, N.; Rossi, M.; Zorzi, M. Web Services for the Internet of Things through CoAP and EXI. In *Proceedings of IEEE International Conference on Communications Workshops (ICC)*, Kyoto, Japan, 5–9 June 2011; pp 1–6.

[65] J. Schneider and T. Kamiya, "Efficient XML Interchange (EXI) Format 1.0," *W3C Working Draft*, 2008. [Online]. Available: http://www.w3.org/TR/2008/WD-exi-20080919 [Accessed 2 October 2013].

[66] World Wide Web Consortium (W3C), "XML Technology." [Online]. Available: http://www.w3.org/standards/xml [Accessed 2 October 2013].

[67] ETSI. 1st CoAP Plugtest. *Technical Report CTI Plugtest Report 1.1.1* (2012-03), ETSI, 2012.

[68] Lynch, Gary S. Single point of failure: The 10 essential laws of supply chain risk management. *John Wiley and Sons*, 2009.

[69] Mariana Carroll, Paula Kotze, Alta van der Merwe (2012). "Securing Virtual and Cloud Environments". *Cloud Computing and Services Science*. Springer New York, 2012. 73-90.

[70] Greenwood, Jim. "OGC Web Services Common Standard." [Online]. (2010). Available: http://portal.opengeospatial.org/files/?artifact_id=38867 [Accessed 19 February 2014].

[71] J. W. Hui and D. E. Culler, "IP is dead, long live IP for wireless sensor networks," In *SenSys '08: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems.* New York, NY, USA: ACM, 2008, pp. 15–28.

[72] S. Duquennoy, G. Grimaud, and J. Vandewalle, "The Web of Things: Interconnecting Devices with High Usability and Performance," In *Proceedings of the International Conference on Embedded Software and Systems (ICESS '09)*, Hangzhou, Zhejiang, China, 2009, pp. 323–330.

[73] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, and M. Spasojevic, “People, places, things: web presence for the real world,” *Mob. Netw. Appl.*, vol. 7, no. 5, pp. 365–376, 2002.

[74] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim, “TinyREST - A protocol for integrating sensor networks into the internet,” *In Proceedings of the Workshop on Real-World Wireless Sensor Network (REALWSN)*, Stockholm, Sweden, 2005.

[75] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin, “pREST: a REST-based protocol for pervasive systems,” *In Proceedings of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS’04)*, Lauderdale, Florida, 2004, pp. 340–348.

[76] Smith, Connie U., and Murray Woodside. "Performance validation at early stages of software development." *System Performance Evaluation: Methodologies and Applications* (1999).

[77] Lynch, Gary S. "Single Point of Failure: The 10 Essential Laws of Supply Chain Risk Management." *John Wiley and Sons*. Oct 7 2009. ISBN 978-0-470-42496-4.

[78] Al-Alawi, Adel I. "Wi-Fi technology: Future market challenges and opportunities." *Journal of computer Science 2.1* (2006): 13.

[79] M. A. Jazayeri, and S. H. L. Liang, "Security and Privacy: Two Prominent Aspects for the Internet of Things", In *Proceedings of the Spatial Knowledge and Information Canada 2014*, Banff, Feb 2014.

Appendix A

A.1 Sample requests and responses used in Section 6.2

In this section, requests and responses of the get observation operation for each of the four implemented protocols are presented. In addition to the message contents, the screenshots of the Wireshark software and message summary are also provided to demonstrate the network analysis of those requests.

A.1.1 Simple Web Server

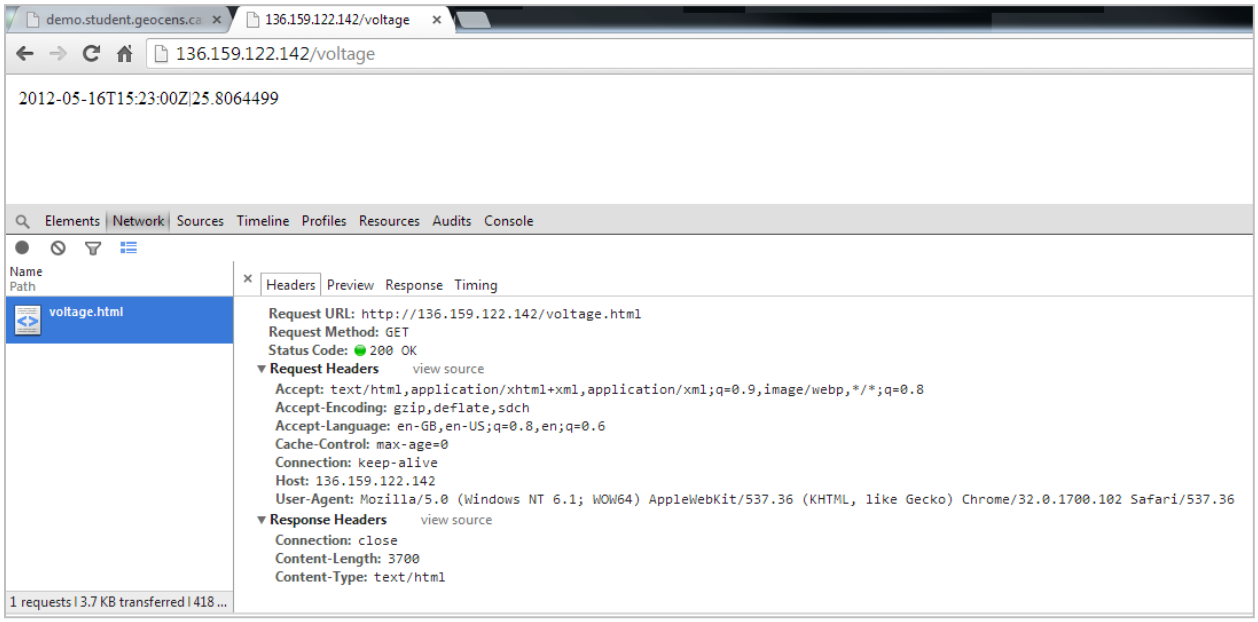

Figure A.1.1 HTTP GET request to the simple web server

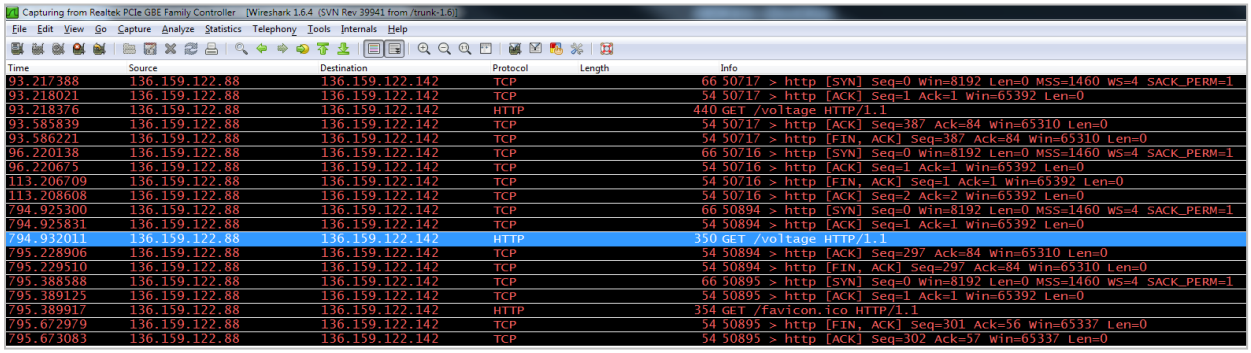

Figure A.1.2 Wireshark screenshot to analyze the requests sent to the simple web service

A.1.2 PUCK over Bluetooth

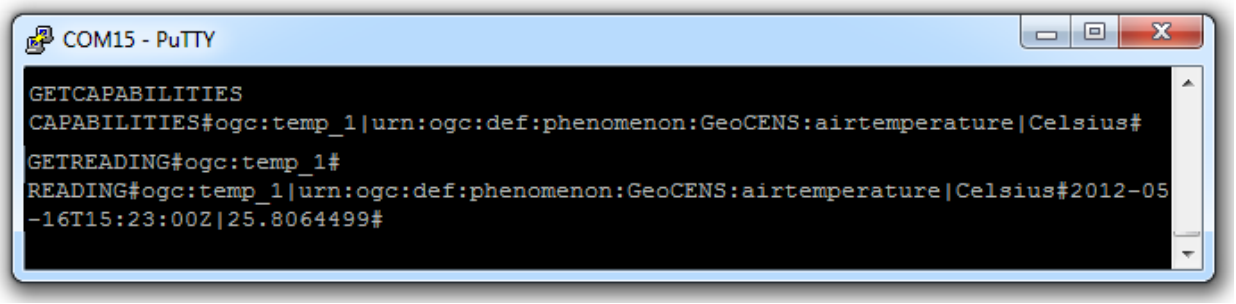

Figure A.1.3 GETREADING request and its response to a PUCK-enabled Netduino Plus through Bluetooth

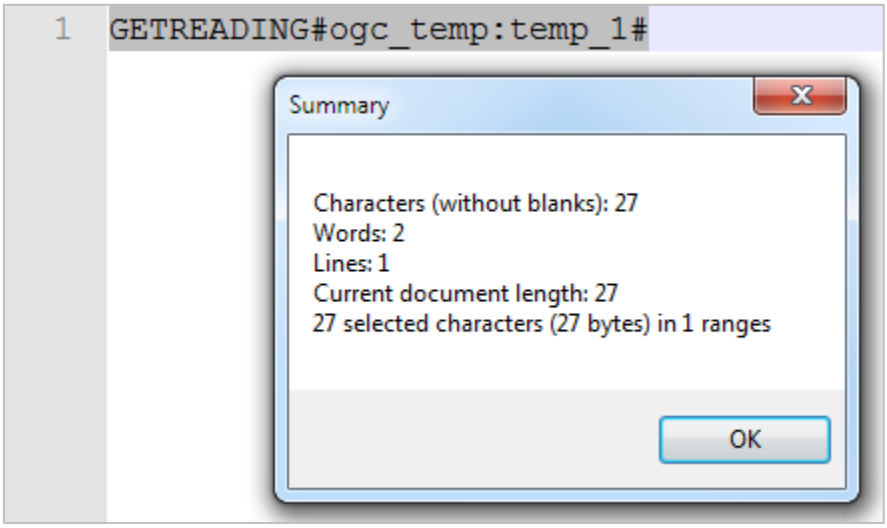

Figure A.1.4 Statistics of the GETREADING request

A.1.3 TinySOS

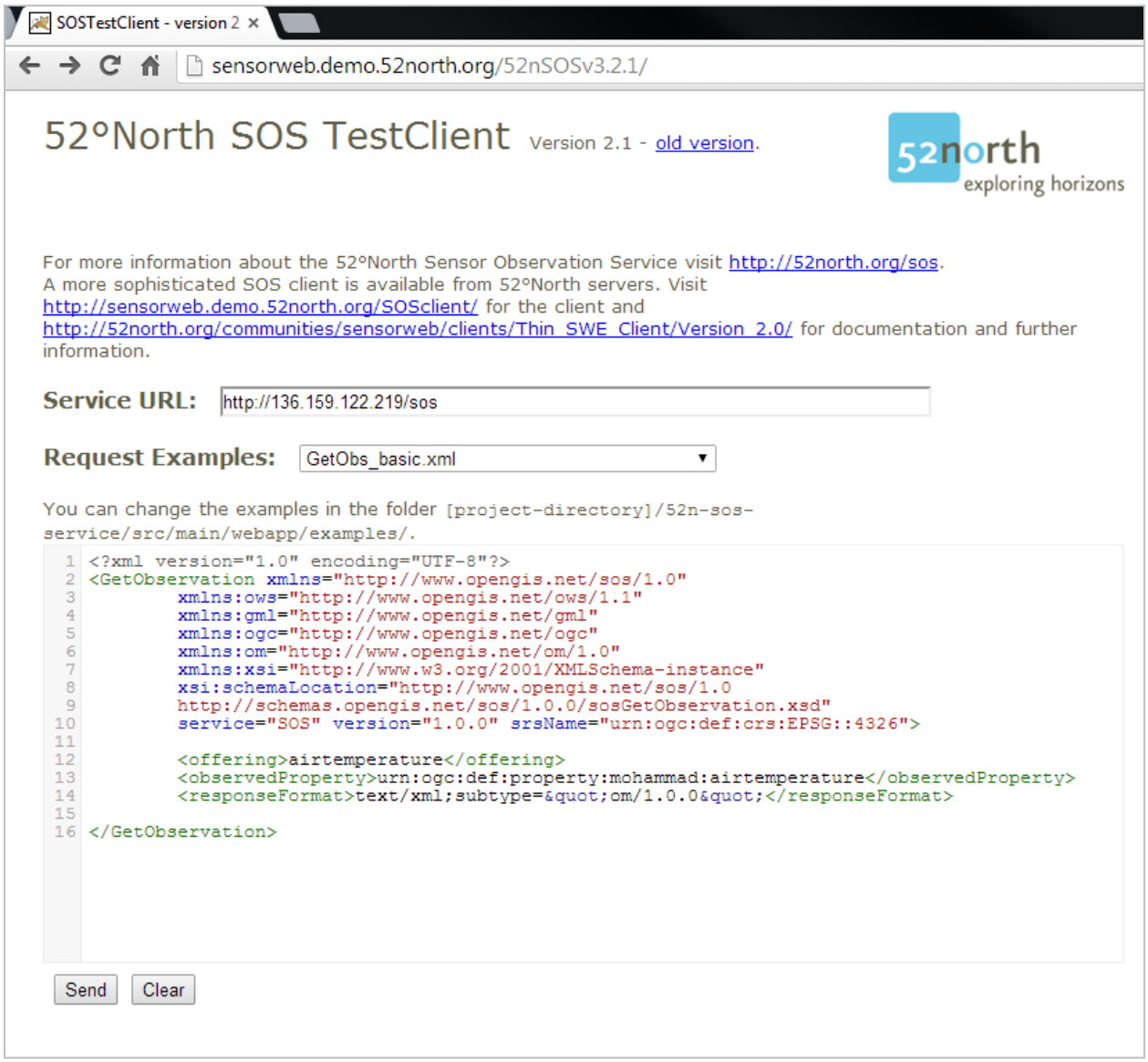

Figure A.1.5 GetObservation request to TinySOS by using 52 North test client tool

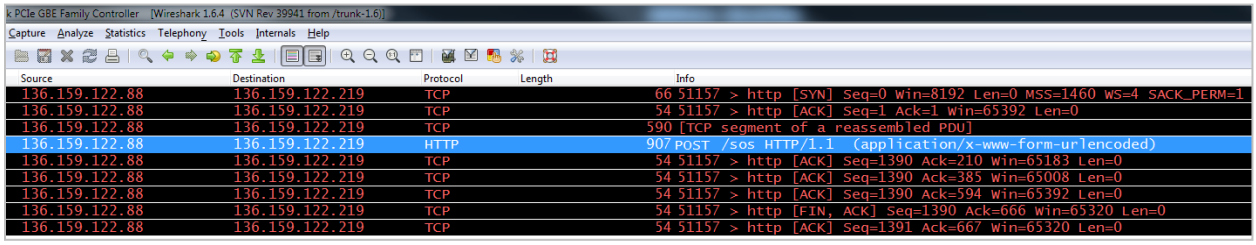

Figure A.1.6 Wireshark screenshot to analyze the requests sent to TinySOS

A.1.4 SOS over CoAP

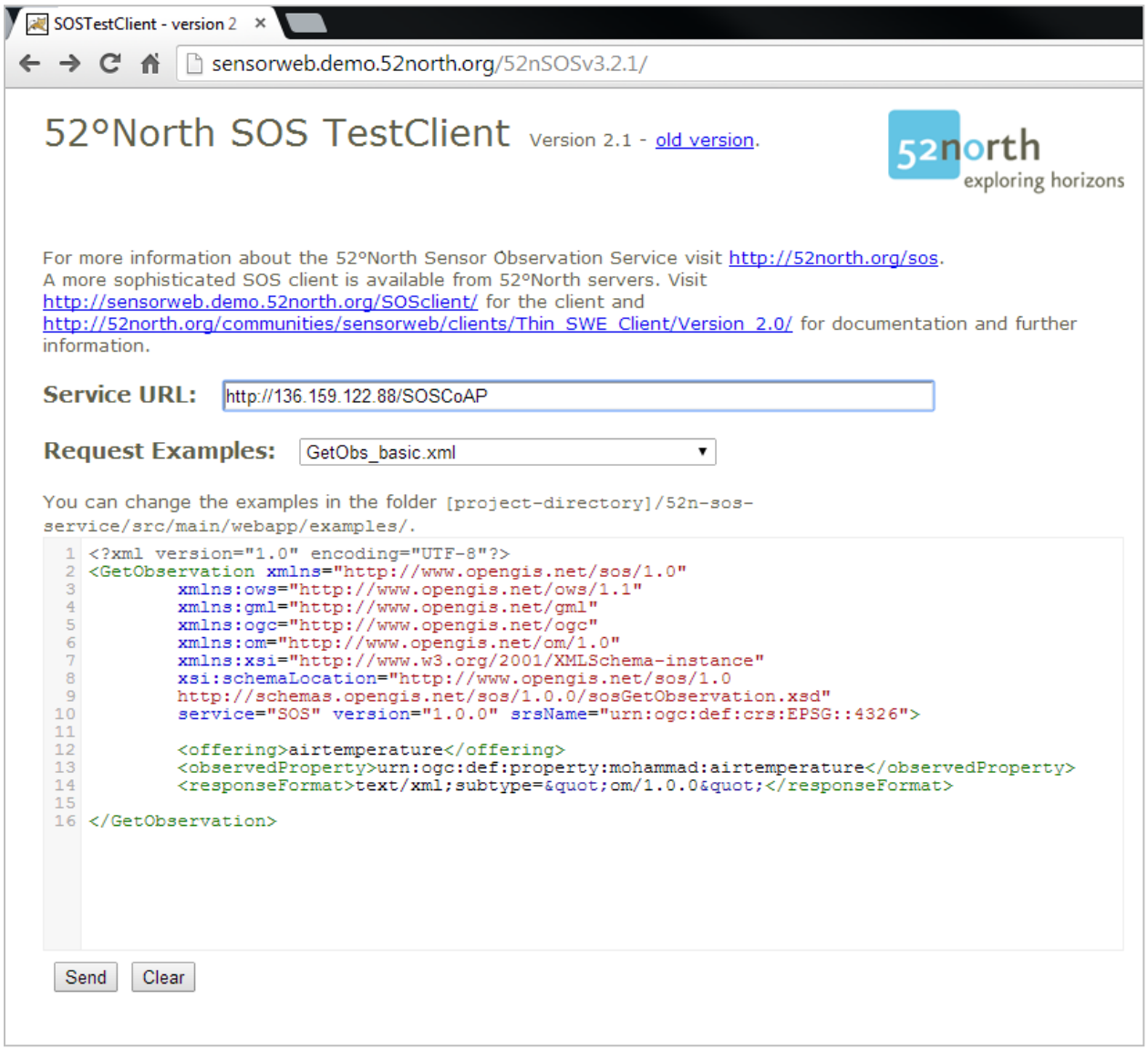

Figure A.1.7 GetObservation request to the SOSCoAP proxy

```
136.159.122.144/observedProperty=urn:ogc:def:property:mohammad:airtemperature&offering=airtemperature
```

Figure A.1.8 Get observation request sent to a CoAP server (*i.e.*, Netduino Plus)

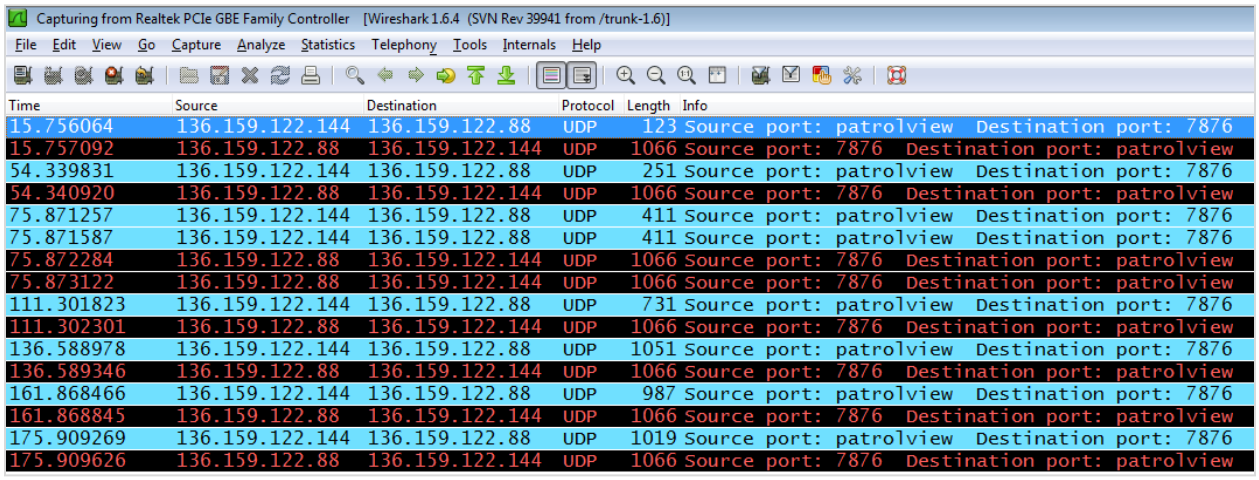

Figure A.1.9 Wireshark screenshot to analyze the requests sent to the CoAP server

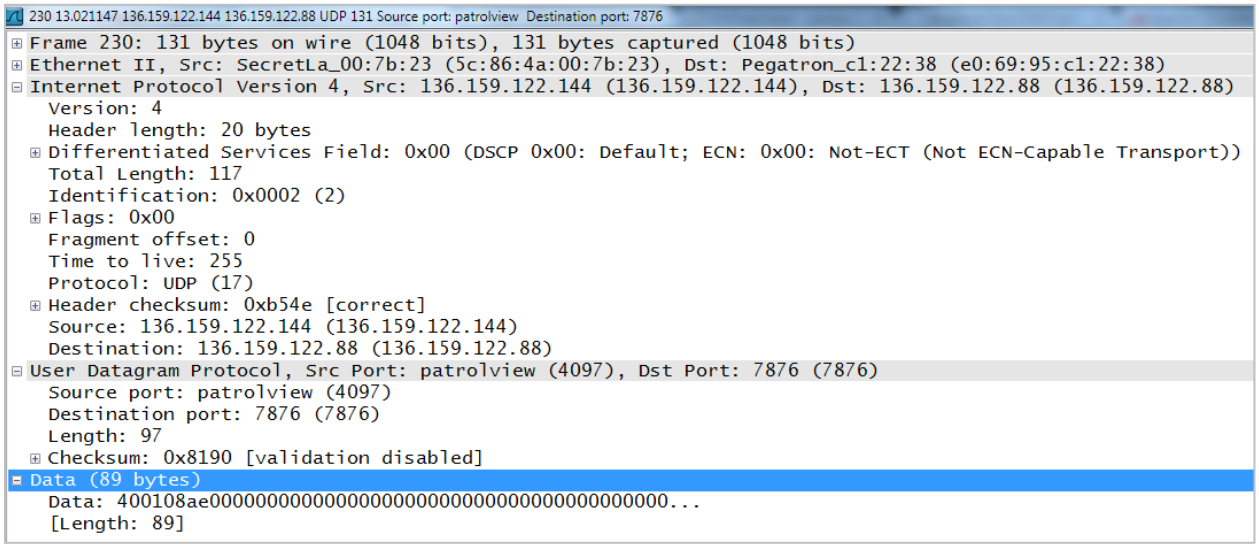

Figure A.1.10 Details of the get observation request sent to the CoAP server

A.1.5 OGC SensorThings API

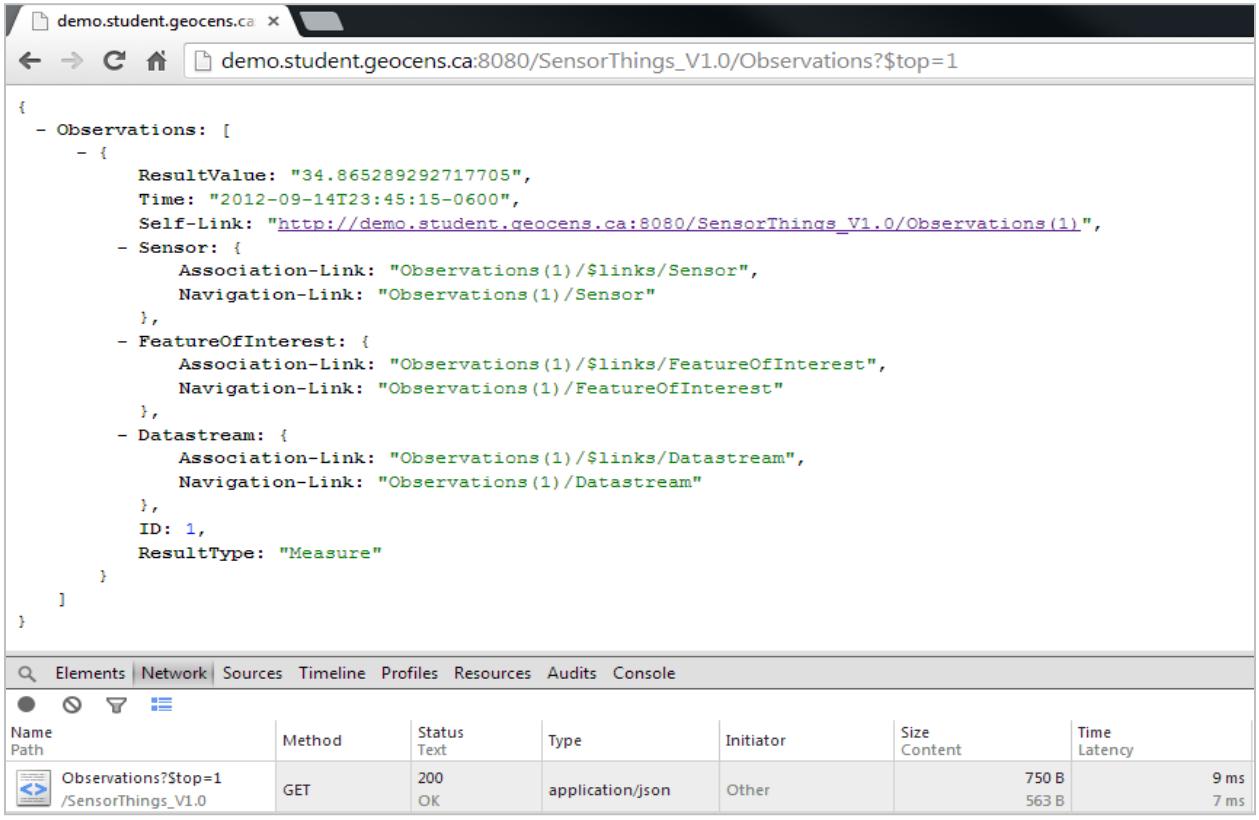

Figure A.1.11 HTTP GET request/response to the OGC SensorThings

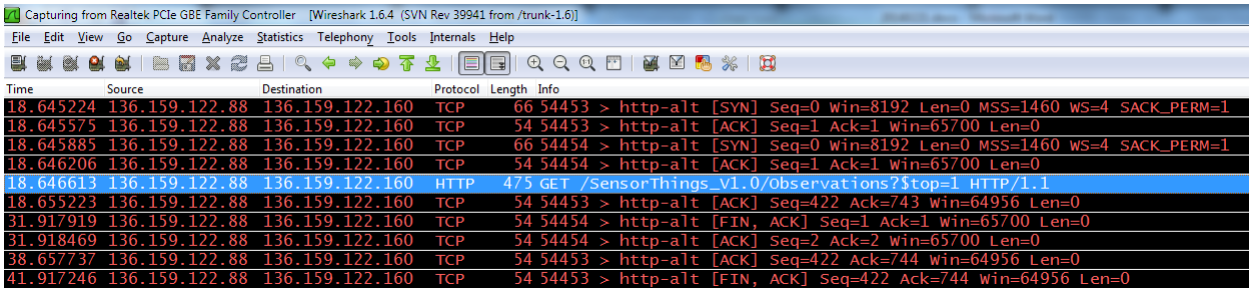

Figure A.1.12 Wireshark screenshot to analyze the request sent to the SensorThings service

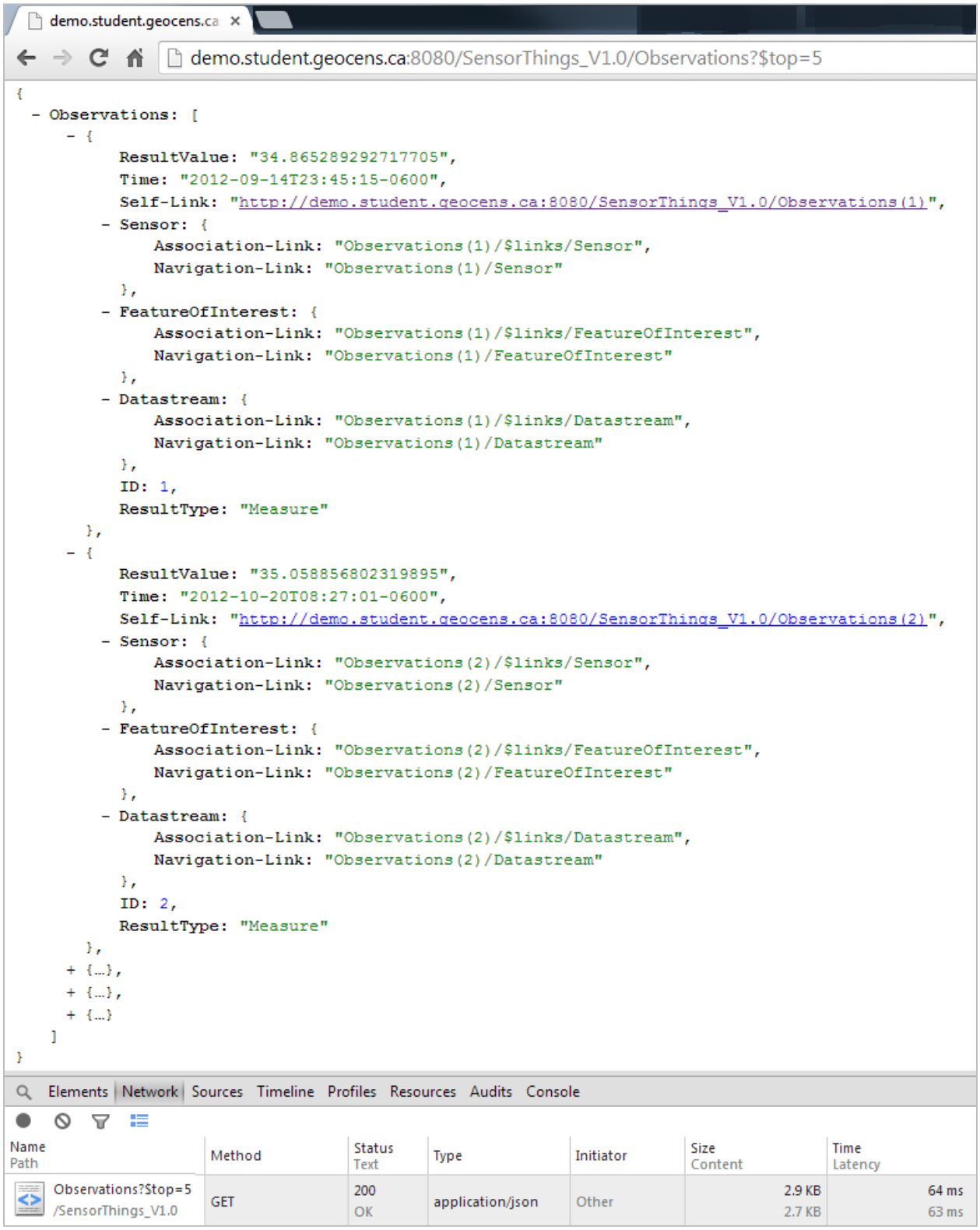

Figure A.1.13 Response of the SensorThings to multiple readings request

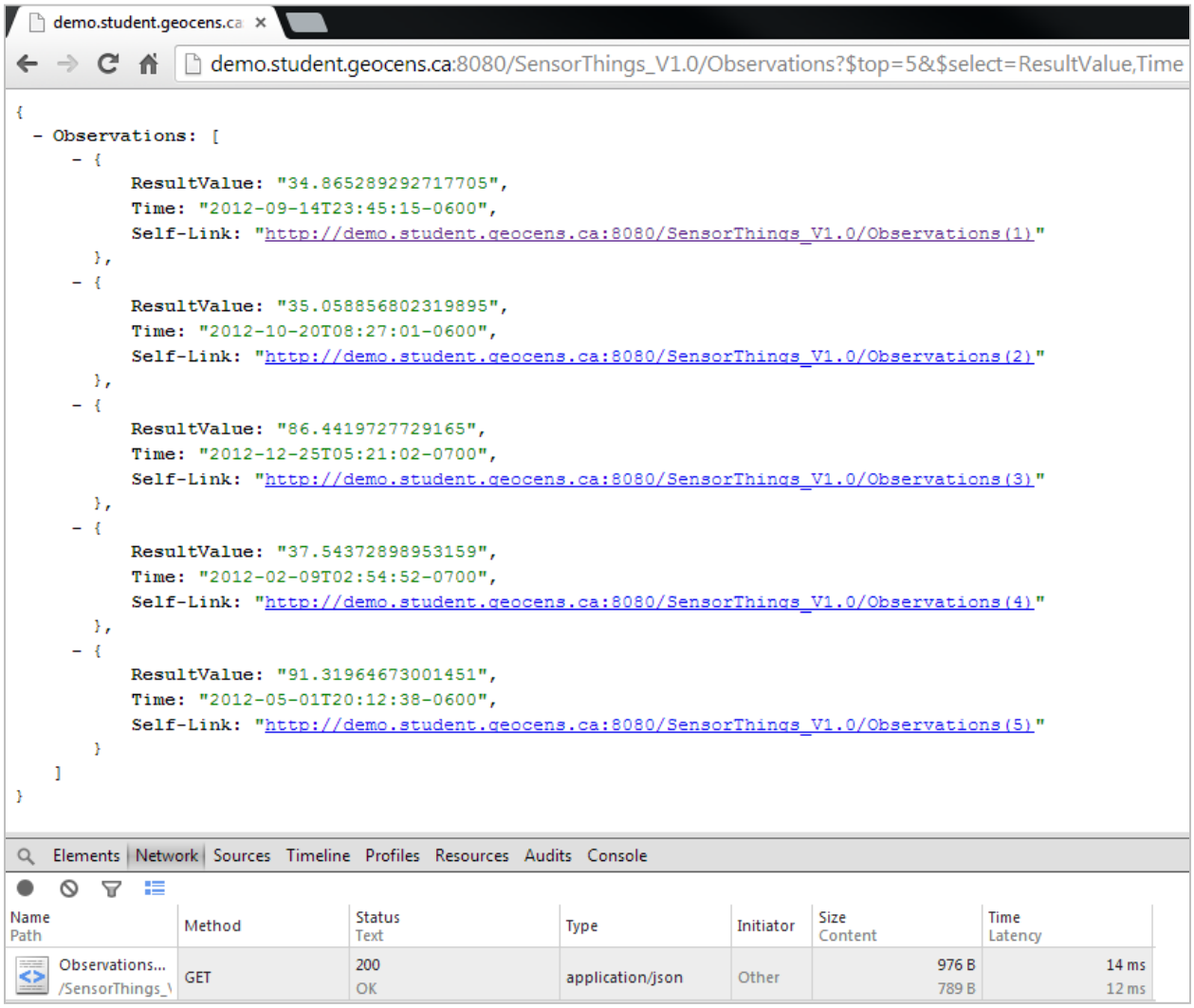

Figure A.1.14 Summarized response of the SensorThings