UNIVERSITY OF CALIFORNIA,
IRVINE

A Centralized IoT Middleware System for Devices Working Across Application Domains Using Self-descriptive Capability Profile

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Chengjia Huo

Dissertation Committee:
Professor Pai Chou, Chair
Professor Phillip Sheu
Professor Rainer Dömer

2014

DEDICATION

To my parents, my finacee and my friends

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **viii**

ACKNOWLEDGMENTS **ix**

CURRICULUM VITAE **x**

ABSTRACT OF THE DISSERTATION **xii**

1 Introduction **1**
1.1 Description of Device's Capability 2
1.2 Devices Working Across Application Domains 3
1.3 Security and Privacy Enforcement 4
1.4 Low Power Requirement 5
1.5 Contributions 5
1.6 Disseration Structure 6

2 Background: Protocols for Networks of Things **7**
2.1 NoT Protocols 7
2.1.1 Bluetooth Low Energy 8
2.1.2 ANT+ 15
2.1.3 ZigBee 15
2.2 BLE's GATT Profile Hierarchy 16
2.2.1 Properties of Characteristic 16
2.2.2 Descriptors of Characteristic 18

3 Background and Related Work: IoT **20**
3.1 Middleware 20
3.2 Device Capability Abstraction 24
3.2.1 External-descriptive Solutions 24
3.2.2 Self-Descriptive Solutions 26
3.3 Task Composition 26
3.3.1 Task Composition of NoT 27
3.3.2 Composition of Web Services 27

3.4 Cloud and IoT Middleware 29

4 Rimware: System Overview 33

4.1 Overview of Proposed Firmware-Gateway-Central System 33
4.2 Firmware 34
4.3 Gateway 36
4.3.1 NoT Device Adapter 36
4.4 Centralized Components of Rimware 37
4.4.1 Knowledge Base 38
4.4.2 Adaptive Application-level Security Enforcement with Knowledge Base . . 40
4.4.3 Device-initiated Adaptive Privacy Protection with Knowledge Base 41
4.4.4 Task Scheduler 43
4.4.5 Data Store 44

5 Device Capability Profile 45

5.1 Device Capability Profile Model 45
5.1.1 Functionality 46
5.1.2 Attribute 47
5.1.3 *fact*-typed Attribute 49
5.1.4 *stream*-typed Attribute 49
5.1.5 *property*-typed Attribute 50
5.2 Syntax Expression of Device Capability Profile 52
5.3 Device Capability Profile in Firmware 54
5.3.1 Functionality, Attribute, and Attribute's Description in Firmware 54
5.3.2 Access Control on Device Capability Profile 56
5.4 Security and Privacy Specification using Device Capability Profile 56
5.4.1 Security Description in Device Capability Profile 57
5.4.2 Privacy Description in Device Capability Profile 59

6 Query and Task Scheduling 61

6.1 Access on Attributes 61
6.1.1 Web APIs-based method 62
6.1.2 Query-based method 63
6.1.3 Search for Device's KB Record 64
6.2 Task Scheduling 64
6.3 Task Optimization 66
6.3.1 Optimization on Tasks Affected by Periodic Actions 67

7 Implementation: BlueRim 70

7.1 BLE Device and Emulator 70
7.1.1 GATT-based Capability Profile 71
7.1.2 EcoBT 72
7.1.3 BLE Device Emulator by MacBook 75
7.2 Gateway 75
7.3 Cloud-base Centralized Components 76

8 Evaluation and Case Studies **77**
8.1 Evaluation 77
8.1.1 Code Size 78
8.1.2 Responsive Time 79
8.1.3 Power Consumption 79
8.2 Case Studies 81
8.3 ECG Recorder 82
8.4 Infant Monitor 82
8.5 Water Pipe Monitor 83

9 Conclusions and Future Work **85**

Bibliography **87**

LIST OF FIGURES

Page

2.1 BLE Protocol Stack 9
2.2 States in BLE Link Layer 10
2.3 GATT Profile Hierarchy 17

3.1 NoT device's firmware structure with NoT middleware 21
3.2 Classification of middleware of NoT 22
3.3 Position of the centrailized middleware for IoT with respect to NoTs and applications 23
3.4 Application domains formed by clouds 30

4.1 Relationship of rimware, NoTs and application Domains 34
4.2 Overview of rimware 35
4.3 Structure of centralized components of rimware and the interaction with gateway and application domains 39

5.1 An example of a device capability profile model 46
5.2 Functionalities of an infant monitoring device 47
5.3 Example of a device's functionality model 48
5.4 A complete device capability profile of the infant monitoring device 53
5.5 An example of infant monitoring device interacting on the communication channel with detailed description on heart-rate monitor functionality 55
5.6 An example of applying a security policy based on a symmetric key cryptography algorithm 58
5.7 An example of applying a security policy based on a public-key cryptography algorithm 59
5.8 An example of authentication checking based on two different mechanisms 60

6.1 The table structure used for KB records maintenance in a relational database 63
6.2 A workflow of the optimization on tasks affects by periodic actions 68

7.1 (a) EcoBT Platform for ECG recording and infant monitoring, (b) EcoBT with ECG module, and (c) EcoBT with modules for Infant Monitoring 73
7.2 Software framework of EcoBT 74
7.3 BlueRim Interactive Shell Interface 76

8.1 An screenshot of the ECG recorder applicaition 83
8.2 An screenshot of the infant monitoring applicaition 84

8.3 Locations of deployed devices 84
8.4 The collected result of a specific location 84

LIST OF TABLES

Page

2.1 Comparison of common NoT Protocols 8
2.2 Structure of Attribute 12
2.3 Security Modes and Levels in BLE 13
2.4 Characteristic properties 17
2.5 Descriptor Types of Characteristic 18

5.1 Required access permissions of different types of attributes 54

6.1 Mappings between access type and standard HTTP methods 62
6.2 An example of searching for an online device base on various conditions 64

7.1 Mappings of components between a device capability profile and a GATT-based profile 71
7.2 Mappings between attribute types and characteristic properties 71
7.3 The structure of 'Characteristic Presentation Format' descriptor used to describe an attribute of a BLE device's capability 72

8.1 Code sizes of each component in BlueRim 78
8.2 Impact on responsive time from security and privacy functionalities in BlueRim for the demo device 80
8.3 Impact on power consumption from security and privacy functionalities in BlueRim for the demo device 81

ACKNOWLEDGMENTS

This work was sponsored in part by a National Institute of Standards and Technology (NIST) Technology Innovation Program (TIP) Grant 080058, and National Institute of Health (NIH) grants 11R41HL112435-01A1, and 1R4HD074379-01.

I would like to extend my deepest gratitude towards my advisor Pai H. Chou, and committee members Professor Phillip C-Y Sheu, Professor Rainer Dömer for their patient guidance and insightful advices. Professor Pai H. Chou has supervised me through the entire work. I appreciate all of his thoughtful directions and encouragements.

I dedicate this work to my parents, Wuqin Huo and Fengqin Liu for their steadfast love and encouragement. I am also indebted to my fiancee, Pinglin Huang. She always supports me and made me concentrated with her love.

I am thankful to all my co-workers: Ting-Chou Chien, Jun Luan, Ke Hao, Qi Wang for their ideas, collaborations, and friendship, which have made my Ph.D study here a wonderful experience.

I have found the Electrical Engineering and Computer Science Department at the University of California, Irvine to be a great place to study.

CURRICULUM VITAE

Chengjia Huo

EDUCATION

Doctor of Philosophy in Computer Engineering **2014**
University of California, Irvine *Irvine, California*

Master of Science in Computer Engineering **2008**
University of California, Irvine *Irvine, California*

Bachelor of Science in Computer Science **2006**
Dalian University of Technology *Dalian, China*

RESEARCH EXPERIENCE

Graduate Research Assistant **2008–2014**
University of California, Irvine *Irvine, California*

RESEARCH OF INTEREST

Internet of Things (IoT), Cyber-physical system (CPS), software modeling, online analytical processing (OLAP)

REFEREED JOURNAL PUBLICATIONS

Chengjia Huo, Ting-Chou Chien, and Pai H. Chou. "Middleware for IoT-Cloud Integration Across Application Domains", *IEEE Design & Test*, Volumn 31, Issue 3, Pages 21-31, 2014.

Ke Hao, Zhiyuan Gong, **Chengjia Huo**, and Phillip C.-Y. Sheu. "Semantic Computing and Computer Science", *International Journal of Semantic Computing* (IJSC), Volume 05, Issue 1, Pages 95-120, 2011.

REFEREED CONFERENCE PUBLICATIONS

Ting-Chou Chien, **Chengjia Huo**, and Pai H. Chou, "A Modular Backend Computing System for Continuous Civil Structural Health Monitoring", in *Proceedings of the Nondestructive Characterization for Composite Materials, Aerospace Engineering, Civil Infrastructure, and Homeland Security 2014*, Volume 9063, March 9-13, 2014.

INVITED TALKS

"Security and Trust Enforcement Using Profile-based Handler in a Highly Mobilized IoT", The

10th IEEE International Conference on Distributed Computing in Sensor Systems, Marina Del Rey, California, May 26, 2014.

ABSTRACT OF THE DISSERTATION

A Centralized IoT Middleware System for Devices Working Across Application Domains Using Self-descriptive Capability Profile

By

Chengjia Huo

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2014

Professor Pai Chou, Chair

The Internet of Things (IoT) has been receiving growing attention in recent years as the next wave of computing revolution made possible by all types of networks of things (NoTs), where devices powered with low-cost, miniature low-power systems-on-chip (SoC) with computing and communication capabilities, and are bridged to the Internet with the assistance of gateways. More and more NoT device are designed to provide more than one functionalities to fulfill different requirements from the application domains. We believe that the true power of IoT is that functionalities of devices can work across application domains. In order to reveal the potential of IoT, the description of a device's capability needs to represent the functionalities that the device can provide. We discover the previous solutions on describing a device's capability focus mainly on hiding the vendor-specific interfaces made by different manufacturers, but they do not reflect different functionalities that a device provides.

In this thesis, the concept of device capability profile is proposed. Different from the previous solutions, the device capability profile specified in the firmware of a device allows the device to work across different application domains. Together with device capability profile, a centralized IoT middleware framework, called *rimware*, is proposed. Rimware tracks every device's capability and state in a centralized manner and provides different ways for application domains to query against the device's functionalities. In addition, rimware utilizes the device capability profile to carry out the enforcement of the security and privacy throughout the communication with the devices. Moreover, tasks can be scheduled through the rimware which enables functionalities from multiple devices to work together to fulfill the requirements from application domains. Optimization is applied on cases that one device working for multiple task simultaneously. An implementation of rimware that is specifically designed for BLE devices, called BlueRim, which takes advantages of BLE's very long battery life on the device side and the cloud functionality on the centralized side is provided. The fundamental features of rimware have been validated in several real-world applications from different different domains while incurring minimal code size and communication overhead on BLE devices. We believe that our approach represents an important technology in taking IoT closer to realizing the full potentials.

Chapter 1

Introduction

The Internet of Things (IoT) has been rapidly gaining ground in the scenarios of pervasive computing and automated human life. The basic idea of this concept is to present pervasively a variety of things or devices that are around us through some unique addressing schemes so that those things or devices can interact with each other and cooperate with each other to meet the users' requirements. Manifold definitions of IoT were proposed from either the "Internet" perspective or the "things" perspective, depending on their interests and backgrounds. We follow the definition that raises the IoT vision from both two perspectives as defined in [51], which is, "*Internet of Things (IoT) is an integrated part of Future Internet including existing and evolving Internet and network developments and could be conceptually defined as a dynamic global network infrastructure with self configuring capabilities based on standard and interoperable communication protocols where physical and virtual "things" have identities, physical attributes, and virtual personalities, use intelligent interfaces, and are seamlessly integrated into the information network*". Physical things in IoT include devices with direct Internet access and devices from Networks of Things (NoTs) that are inter-networked by the Internet. An NoT is a network that makes use of short-range communication protocols to enable devices to interact with their neighboring devices within the same network. The wireless sensor network (WSN), which creates low-rate wireless personal area networks for the communications of the low-cost, energy-efficient sensor devices based on the standards on different layers of the wireless network, such as IEEE 802.15.4 for physical and MAC layers, and ZigBee for the network layer and higher layers, is considered as one example of an NoT. The Internet is the global network system of interconnected computer network that use the standard TCP/IP protocol. Many application services are built on top of the Internet, such as World Wide Web that uses Hypertext Transfer Protocol (HTTP), email services that use Simple Mail Transfer Protocol (SMTP), and sharing services that use peer-to-peer (P2P) technology. According to our definition, an NoT is considered as part of the IoT. Unlike an NoT, things in the IoT can work together even though they are not deployed in the same NoT because connections of things in IoT are supported by the Internet, which does not have the limit of locale. The IoT takes care of much more than just the connectivity issue, such as low-cost, miniature size, naming, object abstraction, service management, data logging, security and privacy. While IoT will unquestionably bring high impact to every user's everyday-life and behavior, many challenging issues still need to be addressed.

The next four sections discuss some of the open issues in IoT which motivate us for our work. Section 1.5 lists the contributions in this thesis. Section 1.6 presents the structure of this thesis.

1.1 Description of Device's Capability

One major issues of achieving IoT is to describe a device's capability. A device's capability description specifies the interfaces in a universal format through which the device can be accessed. A machine interpretable capability description helps on machine-to-machine interactions. Automated configuration and management can be achieved against a device with a machine interpretable capability description in the IoT. However, due to the use of different vendor-specific interfaces on different devices, it is very difficult to universally formalize the capability description for every device.

A device can be either self-descriptive or external-descriptive. A device that is self-descriptive stores an on-board capability description and provides it upon request. In contrast, an external-descriptive device that provides vendor-specific interfaces without a formalized description. The formalized capability description is provided by a third party system, such as a middleware system, which sits between the device and the application domains and does translation work between standard capability description and the device's vendor specific interfaces. Previous solutions have been proposed based on these two directions to describe device's capability. In this thesis, we propose a way to define the capability description in a profile-based structure on the device's firmware, which is self-descriptive.

1.2 Devices Working Across Application Domains

Traditionally, embedded systems have been specialized to a specific application, such as a heart-rate monitor (HRM), a proximity tag for locating misplaced items, or a light switch in a smart home, etc. To us, a device in the IoT environment is more than just a single-purpose embedded system with direct or indirect connectivity to the Internet. It may have been designed and marketed for a single application, but it can potentially participate in applications that it was not originally designed for, as long as it can provide the service needed. For example, one particular application use case may need multiple devices to work together, in which case each of the devices was not specifically designed for this application.

In addition, thanks to advances in systems-on-chip (SoC) in terms of computing and communication capabilities, increasingly more devices with onboard SoC are designed with different hardware modules that are capable of fulfilling different application purposes.

We believe the above two aspects on "devices working across application domains" are the true power of IoT to be. Previous solutions of describing a device's capability mainly focus on hiding the vendor-specific interfaces provided by different devices with a universal interface. However, a common issues on the previous solutions is that devices described by the universal interface are designed to only serve for a specific application domain instead of multiple domains because functionalities in the devices are not described properly.

In this thesis, the capability profile structure we propose defines a device's capability based on the functionalities that the device provides, with which the first aspect of "devices working across application domains" can be realized. Moreover, we propose a centralized IoT middleware framework that allows tasks to be scheduled against one or more devices for a specific application domain, which fulfills the needs of the second aspect of "devices working across application domains".

1.3 Security and Privacy Enforcement

Various security techniques have been built in different NoT protocols used for embedded system. However, they bring a scalability issue that the security policies that can be applied on the communication channels are limited to the ones that are implemented by the NoT protocols. The authentication mechanisms that are used in different NoT protocols to protect privacy also suffer from the same situation. In many NoT protocols, access control is either nonexistent or relies on simple pairing with web-password verification over different NoT standards.

We believe that security and privacy should not be restricted by the implementation of NoT protocols. Developers should have the possibility to specify preferred security policies and authentication mechanisms to be applied for the devices designed by them. In this thesis, security and privacy are considered as two functionalities that a device can provide. To enable the scalability of security and privacy, the capability profile is used to specify the preference of security and privacy on devices so that different security policies and authentication mechanisms can be applied for different devices.

1.4 Low Power Requirement

Another major issue is low power consumption. Low power consumption is a mandatory feature in many IoT systems, especially those powered by battery or are wearable. Bluetooth Smart, also known as Bluetooth 4.0 Low Energy Technology (BLE), stands out as the lowest in average power consumption among many wireless standards [13]. A BLE node can last for one year on a CR2032 coin-cell battery. BLE also is built into many mobile devices, making it possible for users to interact with BLE nodes using their smartphone directly without infrastructure support. For these reasons, we believe BLE deserves high-priority support in IoT. The implementation of our work focus on BLE. However, the concept also applies to other low-power techniques that also have a profile-based application layer.

1.5 Contributions

The contribution of this work is fourfold. First, we give the design of the centralized IoT middleware framework, called rimware, which integrates the NoTs and application domains. Second, as part of the rimware definition, the profile-base capability structure on the device's firmware is given, through which the functionalities of a device can be described and accessed independently. Moreover, security and privacy are described and provided as parts of the capability description that helps firmware developers adopt different security and authentication mechanisms. Third, query and task scheduling against devices through rimware are enabled to allow "devices working across different application domains". Finally, we describe an implementation, named BlueRim, designed specifically for BLE devices, and we evaluate it using several real-world applications.

1.6 Disseration Structure

The rest of this dissertation is structured as follows. Chapter 2 introduces the background of protocols of NoT, and Chapter 3 reviews the background and related work of IoT. Chapter 4 presents the overview of our proposed framework, the “rimware”. Chapter 5 presents the concept of profile-based capability structure at the application layer of a device’s firmware. In the following chapter, we present a formalized way of querying and scheduling tasks against the NoT device through rimware. Implementation of BlueRim is given in Chapter 7. Evaluation is discussed in Chapter 8. Chapter 9 concludes this dissertation with directions for future research.

Chapter 2

Background: Protocols for Networks of Things

This chapter surveys common communication protocols for networks of things (NoT), which are inter-networked to form the Internet of Things. Our work exploits *profile-based* protocols, which standardize the packet format and semantics to enable inter-vendor compatibility. We also examine the security aspects of some of these protocols.

2.1 NoT Protocols

Many communication protocols are being used for NoTs. Table 2.1 compares some of the key features of these protocols.

Table 2.1: Comparison of common NoT Protocols

	BLE	ANT+	ZigBee	RF4CE	Wi-Fi	NFC
Topology	Broadcast, Star, Scan, P2P, No mesh	Broadcast, Mesh, Scan, P2P	Mesh, Star, Scan, P2P, no broadcast	Mesh, Star, Scan, P2P, no broadcast	Star, P2P.	P2P only
Cost (1-10ku)	$1.95	$3.33 + MCU	$3.20	$2.75	$3 + MCU	$1 + MCU
PCB size (mm^2)	20	125	306	305	60	100
MCU	Integrated	Low-end, sep.	Integrated	Integrated	High-end, sep.	High-end sep.
Need Regulator?	No	No	No	No	Yes ($1.50)	Yes ($0.33)
Energy per bit	153 nJ	710 nJ	185,900 nJ	(~ZigBee)	5.25 nJ	(reader side)
Peak Current	12.5 mA	17 mA	40 mA	40 mA	116 mA	50 mA
Coin battery life @120 B/s	191 days	52.64 days	(too high)	(to high)	(too high)	(too high)
Distance	100 m	30 m	100 m	100 m	150 m	5 cm
Coexistence	Freq. hopping (37)	Fixed channel (1/8)	Freq. agility (1/16)	Fixed	Active coexistence	None (short burst)
Throughput	305 Kbps	20 Kbps	100 Kbps	100 Kbps	6 Mbps (11b)	424 Kbps
Latency	2.5 ms	< ms	20 ms	20 ms	1.5 ms	1 second
Direct to Smartmobile	Yes	(few)	No	No	Yes	(few)

2.1.1 Bluetooth Low Energy

BLE, for Bluetooth Low Energy Technology, is a subset of the Bluetooth 4.0 standard. It uses the PHY and MAC layers of the ShockBurst protocol developed by Nordic Semiconductor, but it defines higher layers of the protocol stack to be integrated with “classic Bluetooth,” formally known as BR/EDR (Basic Rate / Extended Data Rate). One reason for BLE’s recent popularity is its extremely low energy consumption. This is a crucial feature in many IoT systems that must be miniature and portable but battery replacement is inconvenient. BLE represents one of the fastest growing protocols for NoTs. This section reviews the BLE stack in layers.

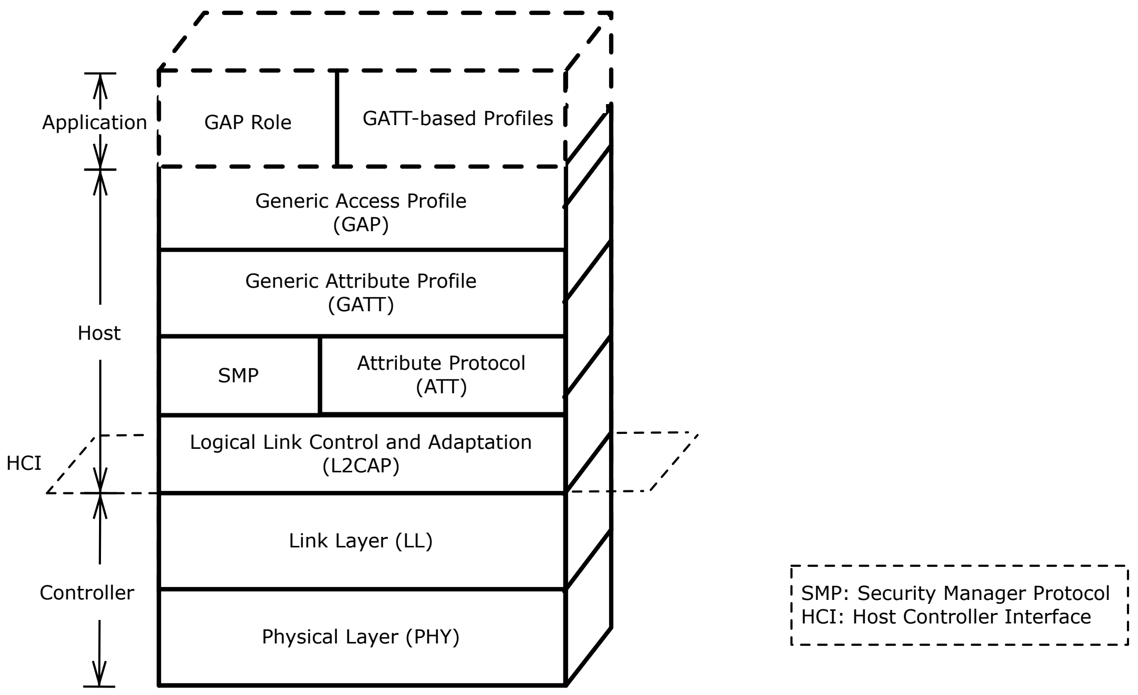

Application
GAP Role
GATT-based Profiles
Generic Access Profile (GAP)
Generic Attribute Profile (GATT)
Host
SMP
Attribute Protocol (ATT)
HCI
Logical Link Control and Adaptation (L2CAP)
Link Layer (LL)
Controller
Physical Layer (PHY)
SMP: Security Manager Protocol
HCI: Host Controller Interface

Figure 2.1: BLE Protocol Stack

Physical Layer (PHY)

The Physical Layer is responsible for transmitting and receiving bits over Radio Frequency (RF) channels. BLE uses the 2.4 GHz Industrial Scientific Medical band and has 40 RF channels with 2 MHz channel spacing. Of these 40 channels, three of them are advertising channels that are used for device discovery, connection establishment and broadcast transmission. The rest of the channels are data channels for bidirectional communication between connected devices. The Physical Layer has a data rate of 1 Mbps with a coverage range of tens of meters.

Link Layer (LL)

The Link Layer is a state machine that controls different states of BLE device. There are five states: Standby, Advertising, Scanning, Initiating, Connection, as shown in Fig. 2.2. When a device transmits data in advertising packets through the advertising channels, the device is in Advertising state. The device is referred to as an advertiser. The advertising packets may contains pure data if the

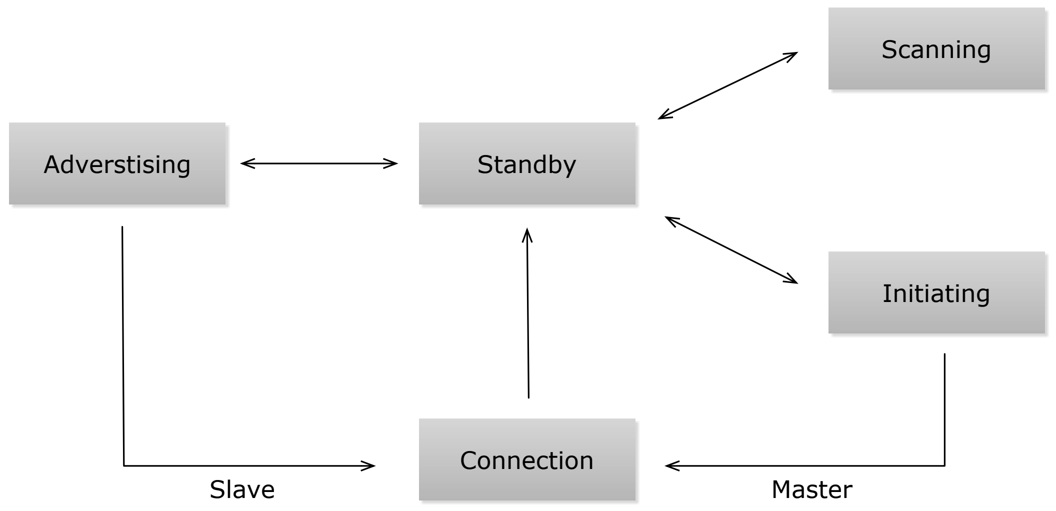


Figure 2.2: States in BLE Link Layer

advertiser only needs to broadcast the data without the intention to pair with other devices. On the other hand, the advertiser can also announce itself as a connectable device in the advertising packets through advertising channels. A device that only aims at receiving data through advertising channels without the commitment to pair is in Scanning mode and is referred to as an scanner. In contrast, a device that listens for connectable advertising packets to form a connection to another device is in Initiating mode and is called an initiator. When an initiator finds a connectable advertiser, it transmits a Connection Request message to the advertiser to create a point-to-point, bi-directional connection between the two devices. Once the connection is established, the two devices enter Connected mode in which they can transmit data packets through data channels to each other.

BLE defines two devices roles at the Link Layer for an established connection: the master and the slave, which correspond to the initiator and the advertiser, respectively, during the connection creation. In BLE, a master can have multiple simultaneous connections with different slaves, whereas a slave may belong to at most one master in BLE 4.0, although BLE 4.1 allows a slave to belong to more than one master.

Logical Link Control and Adaptation Protocol (L2CAP)

The L2CAP layer in BLE supports multiplexing for higher level protocols on top of a Link Layer connection. It is a simplified version compared to the one used in classic Bluetooth in that BLE's L2CAP only supports three higher layer protocols, which are ATT, SMP, and L2CAP LE signaling protocol, and it handles data packets in a best-effort (asynchronous) mode and does not offer retransmission and flow control mechanisms in classic Bluetooth. Because the maximum payload size of the L2CAP is 23 bytes, which is the data units of a single attribute in ATT, segmentation and reassembly are not used for BLE.

Attribute Protocol (ATT)

The ATT defines two roles: server, which stands for the device containing data, and client, which stands for the one accessing the server. The client or server role is independent of the slave or master role that is defined in Link Layer. The ATT allows a device playing the server role to expose a set of attributes to a device playing the client role by exchanging messages.

An *attribute* is a data structure that stores information managed by the GATT, which operates on top of the ATT. An attribute consists of three elements: a 16-bit handle, a 16-bit or 128-bit UUID that represents the attribute type, and a value of the attribute, as shown in Table 2.2. The ATT defines several types of operations that server and client can make use of to exchange information on attributes. The client can access the server's attributes by sending request-typed messages, which trigger response-typed messages sent from the server. The client can also send command-typed messages to write attribute values. For asynchronous communication, the server can also send unsolicited messages that contain attributes to notify the client whenever there is a value change on the attribute. The unsolicited messages can be of either *indication*-type, which requires a confirmation-typed message from client side, or *notification*-type, without requiring a confirmation.

Table 2.2: Structure of Attribute

Handle (2 octets)	Type (2 or 16 octets)	Value(0 to 512 octets)

Security Manager Protocol (SMP)

BLE supports two security modes, *LE Security Mode 1* and *LE Security Mode 2* over a connection from two BLE devices. These two security modes apply security functionality over Link Layer and over ATT layer, respectively. Each security mode may work on different levels with different requirements on encryption and *pairing*. A summary of security mode and level is given in Table 2.3. Encryption on Link Layer in *LE Security Mode 1* is performed using 128-bit AES-CCM algorithm [53] with Long-Term Key (LTK) used as input for the encryption key. If *LE Security Mode 2* is used, which means the connection is not encrypted, data signing is used for transferring authenticated data and Connection Signature Resolving Key (CSRK) is shared between two devices for signing the data. To distribute the above keys, a short-term encrypted session needs to be established and the Short-Term Key (STK) needs to be generated for the session.

To generate the STK, pairing has to be performed between two devices. There are three methods to peform pairing: Out of Band, Passkey Entry and Just Works. The first two are authenticated pairing and prevent Man-In-The-Middle attacks [21], whereas the third method is unauthenticated. The Security Manager Protocol is the message protocol used for distributing specific keys for pairing and transport on top of a fixed L2CAP channel. The actual security mode and level to be used is determined by GAP.

A vulnerability that currently exists in BLE is that except for Out-of-Band pairing mode, which is cumbersome to implement in practice, none of the other pairing methods is protected against passive eavesdropping during the pairing process. In these pairing mode, an adversary who obtains the pairing messages can determine the LTK, the CSRK or the IRK [46].

Note that, instead of solving the existing vulnerability against passive eavesdropping that currently

Table 2.3: Security Modes and Levels in BLE

		Pairing	Encryption	Layer
LE Security Mode 1	Level 1	No	No	Link Layer
	Level 2	Unauthenticated	Yes	
	Level 3	Authenticated	Yes	
LE Security Mode 2	Level 1	Unauthenticated	No	ATT
	Level 2	Authenticated	No	

exists in the link layer of BLE, our interest in security over BLE is to enable the scalability of security on BLE based on the developer's preference by making use of the GATT-based profile in the application layer. A developer can describe the preferred security policy to be applied on the communication channel for the BLE device in the device's GATT-based profile. Our work is based on the assumption that there is no passive eavesdropping in the experimental environment.

Furthermore, the term "authentication" as used in BLE security is the authentication in the sense of cryptography, which refers to "verification that the two devices have the same secret key (for encryption or for data signing)" [18]. However, this device-level authentication does not solve the general sense of authentication, i.e., the privacy issue, which is "the process of verification of the identity of the remote device" [20]. This two security-related issues, as part of the motivation of our work, will be discussed further in Chapter 4.

Generic Attribute Profile (GATT)

The GATT defines a profile-based service framework using the ATT. The framework defines formats of services and characteristics as well as the discovery procedure from one device to another. A *characteristic* is a value with properties and configuration information about how the value may be accessed and information about how the value should be displayed and represented. The characteristics's value and its properties and configuration information are stored as attributes. There may be other attributes used as the descriptors for the characteristic to specify information related to the characteristic's value, such as measurement units. A *service* is a group of one or more characteristics, whose identification is also stored in attributes. In our work, the BLE's GATT profile hierarchy is used to create the profile that describes an NoT device's capability. Through the profile, each of the functionalities in a device's can be accessed and utilized independently. The definition of BLE's GATT profile hierarchy is discussed in details in Section 2.2.

Generic Access Profile (GAP)

The BLE GAP is the base profile that describes the behaviors and methods for device discovery, connection establishment, security and authentication, association models and service discovery. GAP define four device roles for BLE: Broadcaster, Observer, Central, and Peripheral:

- Broadcaster role broadcasts advertisement data through adverting channels without establishing connection with other devices.
- Observer role receives data transmitted by the Broadcaster.
- Central role scans for devices that advertise themselves to be connectable and initiates and manages multiple connections with peripherals.
- Peripheral role broadcasts the device itself as a connectable device and waits for a request from a Central to establish a link-layer connection.

Each GAP role specifies the requirement for the underlying controller. For example, Central and Peripheral roles requires the device's controller to support master and slave roles, respectively. A BLE device may support multiple roles if supported by the underlying controller.

GATT-based Application Profiles

Application profiles can be created by following the GATT profile hierarchy to contain a collection of services for the specific application. The purpose of creating application profiles is to enable application interoperability through the same application domain.

2.1.2 ANT+

ANT+ is a profile-based protocol on top of ANT, which is a popular lightweight wireless protocol based on the same ShockBurst PHY that is also used by BLE. ANT+ has been popular with sports and fitness applications such as heart-rate monitors, bike speedometers, and treadmills, although they are being taken over by BLE. ANT is very simple, low-power protocol that basically just broadcasts without pairing. The response time is therefore very fast, but at the same time it offers no privacy.

2.1.3 ZigBee

The ZigBee [9] Alliance is an association of companies working together to develop standards (and products) for reliable, cost-effective, low-power wireless networking. ZigBee technology has been embedded in a wide range of products and applications across consumer, commercial, industrial and government markets. ZigBee builds upon the IEEE 802.15.4 [30] standard, which defines the physical and MAC layers for low-cost, low-rate personal-area networks. ZigBee defines the network-layer specifications for star, tree and peer-to-peer network topologies and provides a framework for application programming in the application layer.

ZigBee and BLE are similar in that they both use profile concept to define communication interface of different devices for different application use cases. However, the term "profile" mean different things. BLE uses GATT-based profile structure in its application layer for the discovery of services and exchanges data from one device to another through the profile's characteristics. Whereas, ZigBee's profile does not specify a communication structure. Instead, it specifies initial settings, the communication sequence and message format for different types of devices that are involved in the particular application use cases.

2.2 BLE's GATT Profile Hierarchy

BLE defines a profile-based application layer structure, whose hierarchy is defined by GATT, which is discussed in Section 2.1.1. As shown in Fig. 2.3, the GATT profile hierarchy consists of three types of components:

- Service
- Characteristic
- Descriptor

The top level of the hierarchy is a profile consists of one or more services. According to the definition of BLE specification, each service represents a particular function or feature of a BLE device or portions of the device. A service consists of a list of characteristics. The service may also contain references to other services. A characteristic carries a single value and a number of descriptors that describe the meanings of characteristic's value such as measurement unit, valid range, description in human readable text, etc. Every service, characteristic, and descriptor is identified by a Universally unique identifier (UUID), with a length of either 16-bit or 128-bit long.

2.2.1 Properties of Characteristic

Properties of a characteristic specify the accessibility types on a characteristic's value as shown in Table 2.4. The 'Broadcast' property is set when the characteristic value is permited to be broadcasted. The 'Read' and 'Write' properties enable a characteristic value to be accessed by read operations or write operations, respectively. The 'Write Without Response' property enables the

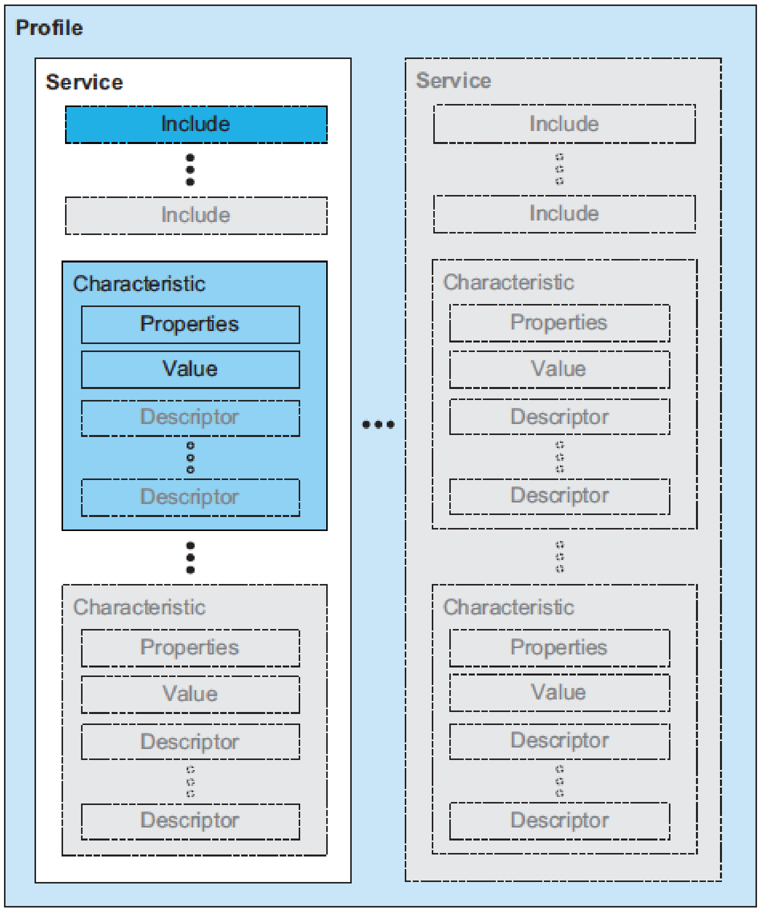


Figure 2.3: GATT Profile Hierarchy

Table 2.4: Characteristic properties

Properties	Hex Value
Broadcast	0x01
Read	0x02
Write Without Response	0x04
Write	0x08
Notify	0x10
Indicate	0x20
Authenticated Signed Writes	0x40
Extended Properties	0x80

Table 2.5: Descriptor Types of Characteristic

Descriptors	UUID
Characteristic Extended Properties	0x2900
Characteristic User Description	0x2901
Client Characteristic Configuration	0x2902
Server Characteristic Configuration	0x2903
Characteristic Presentation Format	0x2904
Characteristic Aggregate Format	0x2905

same accessibility on a characteristic value as the ‘Write’ property does, but in a different response manner. With ‘Write’ property a response message is sent back after every write operation against the characteristic is received and processed to indicate whether or not the operation succeeds, whereas with ‘Write Without Response’ property no response is sent. The ‘Notify’ property and ‘Indicate’ property enable a characteristic’s value to be sent out to other party with push notification when there is an update on the characteristic’s value. The difference between ‘Notify’ property and ‘Indicate’ property is that characteristics with ‘Indicate’ property require confirmation message from the party where the notification is sent to, whereas characteristics with ‘Notify’ property do not. The ‘Authentication Signed Writes’ enables write operations against a characteristic value along with authentication signatures. The ‘Extended Properties’ property indicates that extended properties are defined and is set for this characteristic in the ‘Extended Properties Descriptor’ of the characteristic. Multiple properties can be assigned to one characteristic.

2.2.2 Descriptors of Characteristic

Different types of descriptors [17] are defined in BLE to describe a characteristic’s value as shown in Table 2.5.

- ‘Characteristic Extended Properties’ descriptor: This descriptor that associates with the ‘Extended Properties’ property discussed in Section 5.1.5 serves for setting extended accessibility properties to the characteristic.

- ‘Characteristic User Description’ descriptor: This is a human-readable textual string for a description of the characteristic.

- ‘Server Characteristic Configuration’ descriptor: This descriptor is used to enable ‘Broadcast’ property of characteristic.

- ‘Client Characteristic Configuration’ descriptor: This descriptor is used to enable ‘Notify’ property of characteristic.

- ‘Characteristic Presentation Format’ descriptor: This descriptor stores the representation information about a characteristic value such as format, exponent, unit, name space, description, etc.

- ‘Characteristic Aggregate Format’ descriptor: This descriptor stores the handles of ‘Characteristic Presentation Format’ descriptors for each sub-value when a characteristic’s value is used to store multiple sub-values.

The GATT-based application profile is used to specify a BLE device’s capability, which is discussed in Chapter 7.

Chapter 3

Background and Related Work: IoT

This chapter introduces the concept of middleware and its relate work in IoT. The cloud technology and the challenges to IoT brought by cloud are also discussed.

3.1 Middleware

Originally, middleware is a concept of NoT. Because of the existence of different protocols and data formats provided by different types of devices, middleware components are employed to provide a universal interface to these devices or the NoTs formed by them to ease the difficulty of application development with these devices. By hiding vendor-specific details from application developers, middleware components allow Remote Procedure Call (RPC) servers to support function calls over different languages and architectures. This trend of NoT middleware development were motivated by increased technological possibilities of embedded system devices, e.g., more memory, larger storage capability, and better processors. These possibilities allow different types of middleware to run on devices. Therefore, NoT middleware is often deployed on devices as part of the firmware that runs on top of the OS layer as show in Fig. 3.1. Developers are allowed to write applications

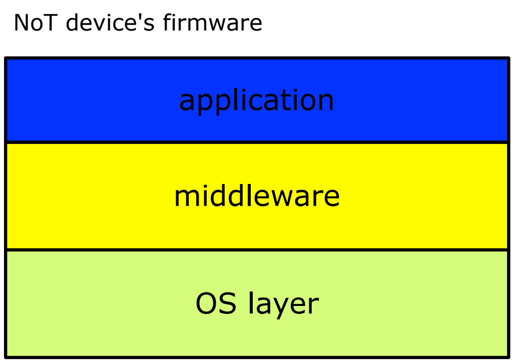


Figure 3.1: NoT device's firmware structure with NoT middleware

in a standard way with the universal interface provided by the middleware.

Under the original definition, NoT middleware often can be categorized as follows:

- Query-based: This type of middleware views the NoT as a distributed database. Each device typically has a local database to store sensed data and a query engine to receive and process queries, which is usually in a modified form of standard SQL. Besides executing and returning the results of any query, query-based middleware is responsible for routing the query to the correct device by maintaining a network path. Examples of this type of middleware include TinyDB [39], SNEE [27], and KSpot [11].

- Deployment-based: This type of middleware installs a virtual machine programming environment that hides details of the underlying operating system and hardware. Thus, software developers can write programs in a common language and deploy it to any device that runs the virtual machine. Deployment-based middleware enables reusable programs to be run or updated across different devices. Examples of deployment-based middleware includes Maté [35], MagnetOS [14], and Wukong [45].

- Communication-based: This type of middleware discovers the nature of many NoTs, that is, communication often occurs because of events that happen in the monitored environment. For example, the detection of facing down of a motion sensing device on a baby may result in transmission of that signal to all interested applications. The event-driven communica-

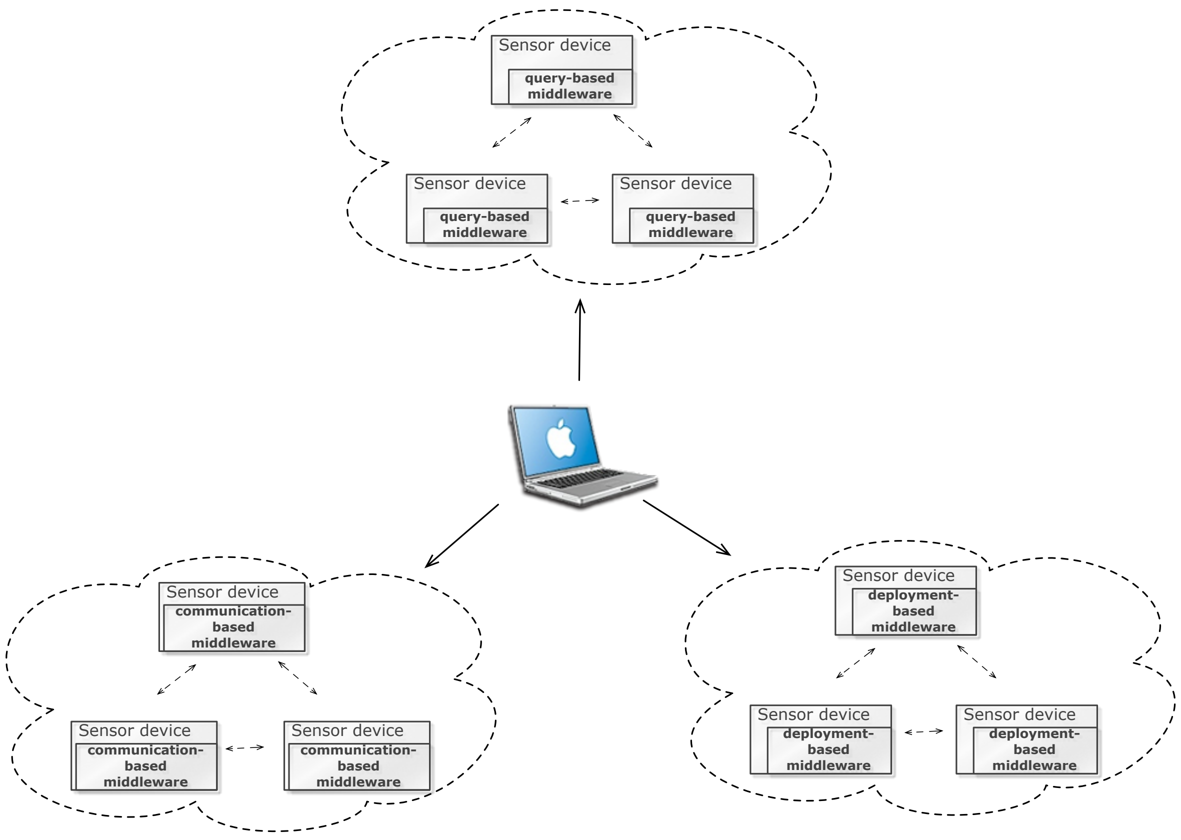


Figure 3.2: Classification of middleware of NoT

tion has led to publish/subscribe mechanism for NoT middleware. When a device detects an event, the event is "published" to all the applications that "subscribed" this event by transmitting the resulting measurements as messages. Additionally, applications can "subscribe" to receive messages either triggered by a special event or with a certain time interval. Examples of communication-based middleware include Mires [50], TinyMQ [48] and WMOS [36].

The NoT middleware categorization is shown in Fig. 3.2.

However, the trend of NoT middleware development has its limits. First, it is only suitable for devices that are powerful in terms of CPU, memory and storage capability. However, many current devices are still limited by their resource to run, e.g., a virtual machine. Second, it is designed only for devices that work for a specific application because the application, as part of the firmware, also

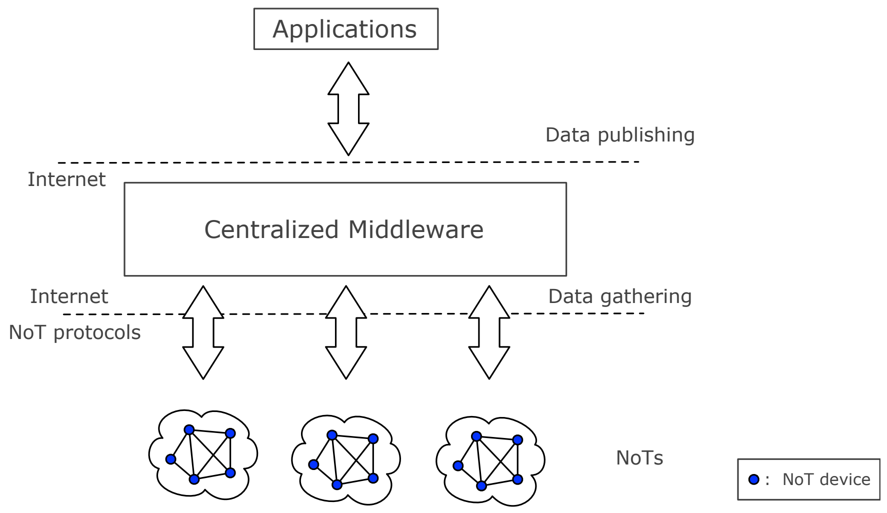


Figure 3.3: Position of the centrailized middleware for IoT with respect to NoTs and applications

runs on devices. In practice, many real-world applications provide their services based on only the device's measurements without requiring NoT middleware. Devices in an NoT function primarily as simple data sources. Data collected from existing NoTs are often served for more than just one single application. Therefore, the ideas underlying the approaches for NoT middleware have not been widely adopted in practice.

Moreover, in recent years, data-centric applications designed for IoT have started collecting data from multiple existing NoTs. Consequently, middleware for IoT that focuses on data gathering from existing NoTs and data publishing for the use by third party applications has been popularized. Middleware for IoT provides a universal interface for developers to access any NoT device. Unlike middleware for NoTs, middleware for IoT often resides outside the NoT devices and manages the IoT in a centralized manner as shown in Fig. 3.3. Under each NoT, a gateway is often used to bridge between different NoT protocols and the Internet. Because this centralized design considers NoT devices as data providers and separates the application layer from the firmware stack, software developers can design applications such as remote monitoring and management without worrying about flashing new firmware.

In recent years, the Service Oriented Architecture (SOA) approach is often adopted for the centralized IoT middleware. The adoption of SOA allows the middleware to present services provided by NoT devices as web services. It also allows applications to fulfill complex tasks by composing services provided by different device. Examples of IoT middleware include SANY [15], PULSENet [26], and SenseWeb [28].

The design of IoT middleware focuses on several important aspects. First, because devices are accessed through middleware in a centralized manner, IoT middleware needs to abstract the capability descriptions of the devices that are managed by it in a standard way and present them as the universal interface. We will discuss this in the next section. Second, IoT middleware needs to take care of the management issues such state monitoring, data collecting and storage, security and privacy, etc. Our work is based on the centralized IoT middleware design. We expand the design by introducing a specification of self-descriptive device capability description as well as by involving the concept and functionality from the cloud.

3.2 Device Capability Abstraction

The purpose of device capability abstraction is to provide a universal interface to access different devices. However, because of different hardware specifications, it is difficult for a device to be described in a universal manner. Previous solutions used on device capability abstraction can be summarized into two categories that are opposite to each other: external-descriptive and self-descriptive.

3.2.1 External-descriptive Solutions

The external-descriptive solutions require no formal description provided from the device side. Instead, an agent, such as middleware component, provides the universal interface to access any device by hiding the vendor-specific interface. The external-descriptive solutions are often used by

SOA-based middleware, in which case device's description information is often exposed as web services. For instance, SenseWeb [28] is a centralized middleware framework from Microsoft Research. In SenseWeb, devices can be addressed through a *sense gateway*, which provides a uniform interface for the rest of SenseWeb, hiding any vendor-specific aspects. The *sense gateway* communicates with applications and other components in SenseWeb through a a common Web Service Application Programming Interface (WS-API). A similar solution that uses a smart gateway to provide the universal interface to access devices is discussed in [24].

OGC's Sensor Web Enablement (SWE) initiative [19] is one of the most commonly used standards to make sensor devices discoverable and accessible over the Internet through web service interfaces. SWE's powerful modeling language, SensorML, aims at modeling any type of sensor systems and provides information such as discovery process, sensor's capabilities in XML and wrapped as Sensor Observation Service (SOS), the web service standard from OGC for publishing on the web. Although SWE means to make sensor devices known and discoverable through SensorML, we often see in real practice that sensor devices need assistance from an intermediate application on translating their capabilities into SensorML and registering them as SOS to join in an SWE-based system due to the fact that capabilities of sensor devices are often described by vendor-specific interfaces instead of by SensorML.

However, external-descriptive solutions have their limits. Because external-descriptive solutions translates vendor-specific interface over NoT protocols into a universal interface over the Internet. The translation often happens on the gateway, which serves as the bridge between NoT protocols and the Internet. However, it leads to an embarrassing situation that the gateway not only is required to have connectivity on NoT protocols and Internet, but also must have knowledge on the device's capability. The gateway has to be re-programmed to gain "knowledge" every time when there is a device with new interfaces trying to connect to the gateway. It is difficult to maintain such a huge knowledge base on each local gateway and may also bring security problems, especially in a highly mobilized IoT where different types of NoT devices roam frequently and connect to different local gateways at different places.

3.2.2 Self-Descriptive Solutions

In contrast, with self-descriptive solutions a device provides the capability description by itself so that it can be accessed directly. In this type of solutions, capability description is often stored on the device and can be obtained through message exchange by following a particular communication protocol. The advantage of self-descriptive solutions compared to external-descriptive solutions is that a device is able to share its capability description with others directly without the assistance from middleware. Our work belongs the second category. IEEE 1451 [34] provides device's description at the transducer level. It embeds the TEDS documents to provide description of hardware modules in a device, e.g. manufacturer data and calibration data, etc. Constrained Application Protocol (CoAP) [52] is a RESTful application layer protocol design that minimizes the complexity of mapping with HTTP. Device capabilities can be exposed as a list of CoAP sources. CoAP benefits from low header overhead and parsing complexity.

3.3 Task Composition

In IoT, it is often the case that an application purpose must be fulfilled by the collaboration of services provided by multiple devices. In that case, techniques of composing tasks against multiple devices are required. We first review the techniques on task composition used in NoT middleware. Because IoT middleware often adopts SOA principles, we then review the composition techniques used in web service.

3.3.1 Task Composition of NoT

TinyDB [39] uses an SQL-like query format to compose a task. To process a composed task, issued queries are first routed to the correct devices from root device and system will maintain routing path of concerning devices. After receiving the results from all concerning devices the root device will aggregate the sub-results and return the final result.

WuKong project [45] defines profiles that are independent of the transport protocol such as ZigBee, Z-Wave, or Wi-Fi. Profile enables WuKong-compliant devices (called Wu-devices) to interoperate and task mapping from a flow-based program. In case of a device crash, the mapper can re-task it to substitute devices (i.e., with matching profile).

SHARE [37, 38] uses TinyOS-based task coordination design. Tasks in SHARE are composed and transmitted in XML. It defines event semantics checking and conversion based on signal type system (STS) that captures both values and service triggering. Based on the compatibility of event semantics, redundant computations in uncoordinated tasks are removed from runtime. The task optimization in our work discussed in Section 6.3 is highly inspired by the STS.

3.3.2 Composition of Web Services

A complex task may require multiple web services to collaborate with each other in which situation service composition is required. There are two main techniques for service composition: flow-based composition and AI-based composition.

A *workflow-based* method for web service composition usually generates either a static workflow or a dynamic workflow. In a static workflow, a complex service is modeled as a graph that defines the execution sequence in the process. The graph is manually created and can be updated dynamically. Eflow [22] is one example of static workflow composition. In a dynamic workflow method, it assumes that the availability of services may change frequently to fit a highly dynamic environment. CSDL [23] is one example of dynamic workflow composition. Static and dynamic techniques may be combined for service composition such as Polymorphic Process Model (PPM) [47]. The dynamic workflow of PPM includes a set of service-based processes, where a service is modeled as a state machine. The state machine is used to specify the states of a service and the state transitions. In the composition process, dynamic service composition is supported by reasoning based on the state machine.

Because it is insufficient to achieve the goal of interoperation simply by integrating business processes across enterprise boundaries using standard messages and protocols alone, *AI-based* composition is introduced, which aims to declaratively specify the prerequisites, consequences, and data flow of a web service. AI-based composition languages such as BPEL4WS [12], Web Services Flow Language [7], XLANG [8], and Web Service Choreography Interface [5] have been offered as solutions.

Besides declarative languages, ontology is also used to assist web service composition. The Semantic Web Service (SWS) framework [41] was introduced as a solution by combining traditional web services and ontology. The core of a semantic web service is adding semantic markups to a traditional web service. Ontology is used to describe the properties and semantics of web services in an unambiguous form so that the user can locate, select, employ, compose, and monitor the web. It is based on a service description ontology that includes three sub-ontologies: service profile, service grounding, and a process model. SWS framework usually includes the vision of:

- Automatic Web Service Discovery: Given a service request, the Semantic Web Service system can find a service provider that offers a service that can fulfill the requirements such as input/output data, conditions, and restrictions. Note that the term “discover” means finding one particular service, not a combination of multiple services.
- Automatic Web Service Execution: In the traditional web service framework, after a service is discovered it has to be manually invoked. A Semantic Web Service, however, allows a

computer program or agent to automatically execute it.

- Automatic Web Service Composition and Interoperation: Traditionally, if no single existing service can be identified to satisfy a service request, several services have to be manually found and a workflow has to be manually built to fulfill the requirement. The Semantic Web Service system is targeted to automatically select and compose appropriate web services to satisfy the service request.

Typical solutions for semantic web service composition include OWL-S [40], METEOR-S [43], SWSL [4], and WSDL-S [6].

3.4 Cloud and IoT Middleware

A *cloud* is a general term for a pool of automatically managed computing resources that are accessible over the Internet. The resources can include CPU cycles, storage, and the services offered using these resources. The main goal of the cloud structure is to provide on-demand, prompt access to these services while being able to scale with usage, from few users to millions or billions of users. The concept of a *cloud stack* may be useful for describing the different levels of cloud services. They are IaaS, PaaS, and SaaS.

- Infrastructure as a Service (IaaS) refers to the hardware and software infrastructure on which cloud services are actually run. The hardware includes the server machines, storage systems, the networks (to the Internet as well as high-speed networks between servers on a rack, for example). The software includes the operating system that runs on these machines and layers of software that integrate them. Examples include Amazon Web Services, Rackspace, and OpenStack.

- Platform as a Service (PaaS) refers to the tools and services for facilitating the develop-

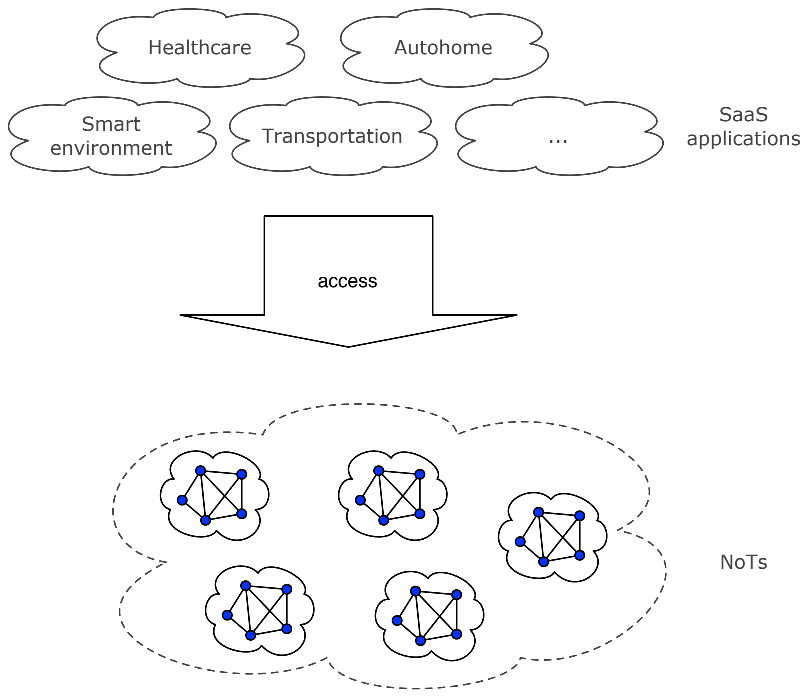


Figure 3.4: Application domains formed by clouds

ment and deployment of applications as cloud services. PaaS provides computing platform resources including operating system, programming language runtime environment, web server, and database engine. In a PaaS, users may choose the platform components they need to build their online/offline application. PaaS manages the underlying computing and storage resources automatically to fulfill the needs of applications running on it in terms of availability and scalability. Examples of PaaS include Heroku, Google App Engine, Microsoft Azure.

- Software as a Service (SaaS) refers to the application-level services for the end user. Examples include web mail, google map, google doc, facebook.

In recent years, integration of IoT and cloud becomes the popular research trend. Functionalities of cloud benefits the implementation of IoT middleware in several aspects. For instance, the worldwide Internet accessibility allows devices to be accessed from anywhere without downtime. In addition, the large storage ability allows data collected from devices to be archived at centralized middleware instead of storing at local NoTs. Moreover, the powerful computation capability of cloud enables IoT middleware to process thousands of queries against NoT and return results immediately. The implementation of our work is a SaaS application that is built on a PaaS runtime platform, thus having all above benefits from functionalities of cloud.

Besides the powerful functionality, cloud technology also brings challenges to IoT. From the application aspect, there exists more than just one cloud. A SaaS application is considered as a cloud and the application forms its own application domain. Applications that serve for different application purposes create different application domains as shown in Fig. 3.4. In cloud-based architecture, it is often the case that different SaaS applications share the same group of users. For instance, an application under Facebook domain share the same group of users with Facebook. Different applications that share the same group of users are considered to be under the same application domain. In this thesis, we assume applications that are under the same application domain serve for the same application purpose, for instance, healthcare, autohome, smart environment, etc., as shown in Fig. 3.4. The formation of different cloud-based application domains generates the needs of one device working for different application domains. It also generates the needs of multiple devices collaborating with each other to work for any application domain. Many works [32, 44] have been proposed to create cloud-based IoT middleware. However, these works mainly focus on establishing an IoT middleware structure with cloud functionality that supports one specific application domain and do not consider the issue of one device working across application domains. Many data centric implementations, such as [42], support for multiple application domains, but they are concerned with only sharing archived data among different application domains and do not support runtime tasks collaborating from multiple devices. Other proposed works such as [29] support runtime monitoring on devices from different application domains are through a subscribe/publish mechanism. However, because this type of framework often lacks definitions of device capability descriptions, it can only support read-only operation such as state monitoring and data collecting but does not support remote device manipulation. Our work in this thesis aims at providing a solution to respond to the challenges brought by the cloud concept and to address the issues in the existing solutions.

Chapter 4

Rimware: System Overview

We envision an enhanced architecture for IoT by introducing our proposed *rimware*, a middleware layer that spans NoT device's firmware, gateway, and centralized components. The term rimware is coined from the analogy that the proposed middleware resides in a rim that wraps the physical space (i.e., NoTs) on the inside and application domains on the outside, as shown in Fig. 4.1. The halo on the edge between rimware and NoTs denotes that rimware also plays a role in the firmware of NoT devices on the NoTs side. *Rimware* aims at supporting remote device access from different application domains and enablement of collaboration on multiple devices for tasks across application domains. This chapter presents the overview of the *rimware* structure.

4.1 Overview of Proposed Firmware-Gateway-Central System

An overview of a firmware-gateway-central system running rimware is shown in Fig. 4.2. Different from classic centralized IoT middleware, which often consists of the gateway and centralized components, rimware is composed of three parts: profile-structured firmware, plug-in-style gateway, and the centralized components of rimware. On the firmware sides, the capability of any NoT

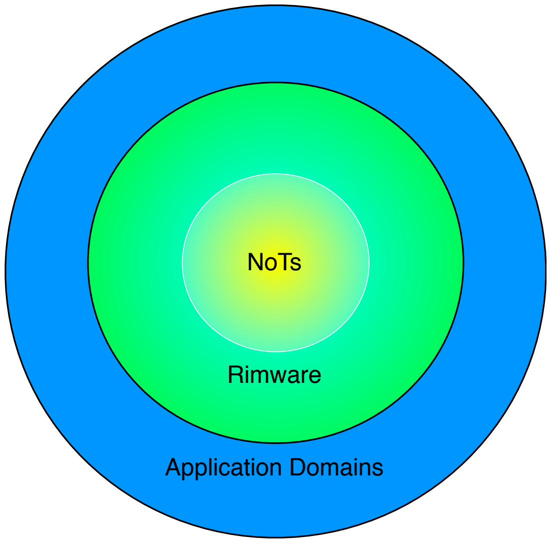


Figure 4.1: Relationship of rimware, NoTs and application Domains

device is abstracted into a profile structure. When the device connects to a gateway or a rimware-enabled smartphone, the gateway or smartphone obtains the capability profile from the device over NoT protocols and passes it to the centralized components of rimware over the Internet. The centralized components then applies the security policy and privacy mechanism on the communication channel against the NoT device according to the description in its capability profile. The information in the capability profile is translated and presented to the application domains in the format of web APIs. A user with the the correct privilege from any application domain can access the device's functionalities remotely either through its web APIs or by self-composed queries. A user can also compose and issue a task to run on any device through the centralized components of rimware. Moreover, tasks that involve multiple devices can also be scheduled through the centralized components of rimware. The next three sections present the principle parts of rimware.

4.2 Firmware

Rimware requires a profile-based interface to abstract an NoT device's capability on the firmware side. A device's capability is expressed by a capability profile. The capability profile consists

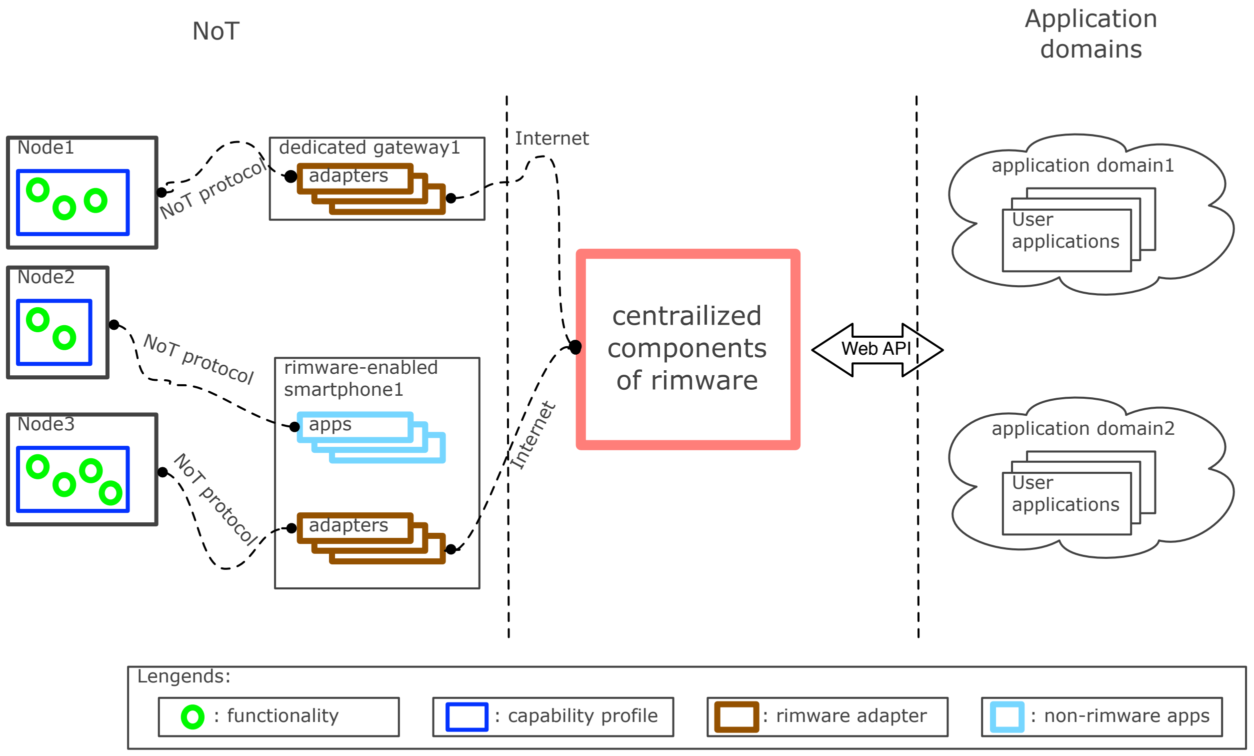


Figure 4.2: Overview of rimware

of one or more functionalities. For instance, a heart rate monitoring device may have two functionalities: monitoring the current heart-rate at the specified sampling rate and providing the device's hardware information such as manufacture's name, serial number, etc. The capability profile may also include security policies and privacy mechanisms to be applied after the communication channel with the device is established. For implementation, the application layer of the rimware-compatible device's firmware is built for abstracting the device's capability in a profile structure.

During discovery stage, the NoT device exposes its capability profile to the other party (e.g., the gateway). The other party accesses the device's functionalities through its capability profile. Profile-structured design on the firmware brings several advantages over the classic centralized IoT middleware. First, it releases the burden from the gateway in the classic centralized IoT middleware that uses external-descriptive solutions as discussed in Section 3.2.1. With the uniform profile structure, the gateway does not need to be re-programmed when a new device with a different version of firmware connects to it. Instead, the gateway acts as the conduit when the centralized components of rimware access the the capability profile of any NoT device. Second, because the profile structure is a self-descriptive solution for device capability abstraction as mentioned in Section 3.2.2, it enables the possibility of sharing capability information between devices without the interference of a gateway. For example, it is possible for a non-rimware app as shown in Fig. 4.2 to communicate with an NoT device directly as long as it understands the capability profile of the device.

The capability profile structure is the fundamental concept of our work. The construction of capability profile is discussed in detail in Chapter 5.

4.3 Gateway

The gateway in rimware serves the purpose of bridging the NoT devices with the centralized components of rimware. This means it must have connectivity of the NoT protocols downstream and connectivity of the Internet upstream. The gateway runs adapter processes in a plug-in-style architecture to bridge the two sides. As shown in Fig. 4.2, the gateway program can be set up either on a dedicated local device or a rimware-enabled smartphone. A gateway can establish communication channels with multiple devices by instantiating multiple adapter instances.

4.3.1 NoT Device Adapter

An *adapter* on the gateway is a running process that acts as the interfacing process between an NoT device and the centralized components of rimware. When a gateway starts running, it proactively discovers and connects with those nearby NoT devices that are advertising themselves as connectable devices over the NoT protocol. For every connected device, the gateway instantiates an adapter on the gateway and uses the adapter. The adapter is used for exchanging message with the device through the NoT protocol. Once the device is connected, the adapter sends messages to the device and asks for the structure of device capability profile. After obtaining the device capability profile structure, the adapter passes it to the centralized components of rimware over the Internet.

The gateway adapter acts as the conduit. Messages containing different types of operations including read, write, subscribe, and unsubscribe from the centralized components can be sent to any NoT device to access its capability profile through an adapter.

4.4 Centralized Components of Rimware

The centralized components of rimware sit between gateways and application domains, and communicate with the two sides over the Internet. The centralized components serve several functionalities:

- Storing the capability profile of every NoT device and translating it into web APIs
- Applying the required security policies and privacy on the communication channels of the NoT devices
- Monitoring the state of every NoT device
- Storing data collected from NoT devices
- Processing composed tasks from users against any NoT device(s)

Note that rimware does not take care of access control against users from application domains. Instead, we assume the accessibility checking against users should be done at the application domains side before any request is issued to the rimware. The reason is that, to maintain the accessibility mappings between NoT devices and users from different application domains and for every user, the rimware needs to first maintain a mapping between the user's identity in rimware and the identity in the user's application domain. Due to the different identification mechanisms used in different application domains, it is possible that two users from different application domains have the same the identity in their own domains. Therefore, it is difficult to do access control against users from the rimware side. Since the applications from different domains are built to work with rimware, it is reasonable for them to do the accessibility checking. From the rimware side, when rimware receives a request from the application domains side, it processes the request without caring about which application the request comes from or which user of the application issues the request.

As shown in Fig. 4.3, the centralized components of rimware are:

- Knowledge base
- Task scheduler
- Data store

The next three subsections discuss each of the centralized components.

4.4.1 Knowledge Base

The knowledge base serves for storing and translating the device's capability profile, monitoring the online/offline state of every NoT device, applying the required security policies and performing the authentication checking. The knowledge base is built by a managing process called the *KB manger*.

For every device, the KB manager maintains in the knowledge base a unique KB record that is generated when the device joins rimware for the first time. The KB record is identified by the device's on-board ID, for instance, the MAC address. The KB record contains several kinds of information as follows:

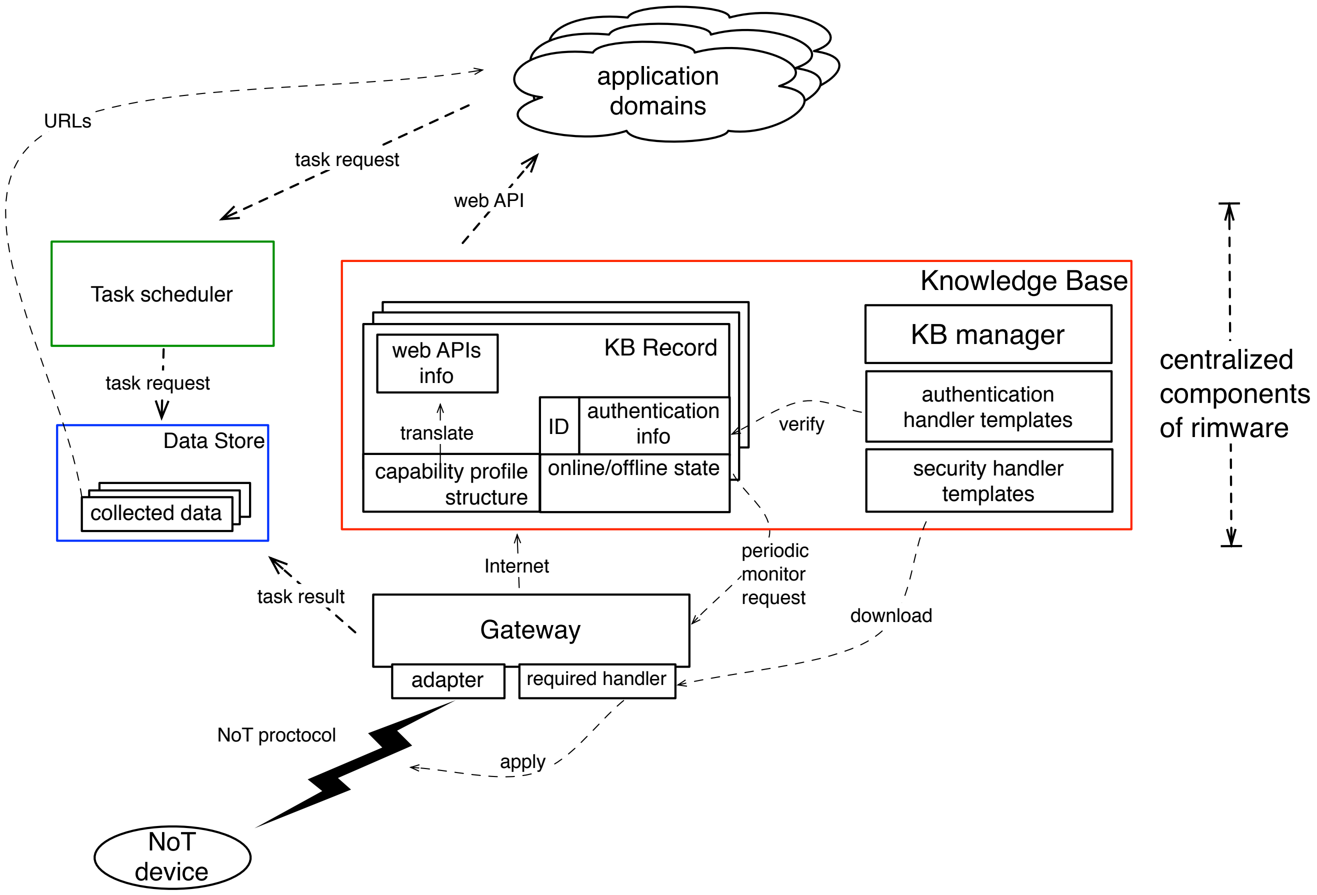


Figure 4.3: Structure of centralized components of rimware and the interaction with gateway and application domains

- Device's unique ID
- The device capability profile structure that contains the description of the profile
- Web APIs translated from the device capability profile structure
- Online/offline state
- Authentication information depending on whether or not there is a required privacy mechanism to be applied against the device.

The KB manager periodically sends messages to all gateways to track online/offline state of every NoT device and updates the latest state in every device's KB record.

4.4.2 Adaptive Application-level Security Enforcement with Knowledge Base

The knowledge base solves the scalability issue on the security enforcement of NoT protocols by using a plug-in-style design. In most NoT protocols, security policies that can be applied on the NoT communication channels are often limited to the ones that are implemented by the NoT protocols. Developers are not allowed to choose security policies other than the ones specified the NoT protocols. The knowledge base solves this issue by maintaining a list of pre-defined templates of different types of security handlers. These templates are identified by their type IDs. The *security handler* is a piece of code that applies a specific security policy on the communication channel between any NoT device and the corresponding gateway. Additional templates that contain new security policies can be contributed by developers when they need to apply them on their devices.

When a connection is established between a gateway and an NoT device, the gateway asks for the device capability profile structure and passes it to the knowledge base. The KB manager in the knowledge base scans the capability profile structure to see if it contains information about required security policy. If such information exists, the KB manager sends messages to the NoT device through the gateway asking for detailed information including the security type and configuration parameters. The KB manager then finds the matched template of security handler from the template list. By using the template, the KB manager initializes an instance of the security handler with the configuration parameters and asks the corresponding gateway to download the instance of security handler in order to apply the security policy on the communication channel between the gateway and the NoT device. After the security policy is applied, security handler on the gateway handles the encryption of outgoing messages and decryption of incoming messages as long as the communication between the gateway and the NoT device remains active. The instantiation of security handler is shown in Fig. 4.3.

Moreover, the firmware developers can set up restrictions on the communication channels during the time the preferred security policy is not applied. For instance, the firmware can be setup to deny all the accesses to the content of the capability profile except that one that specifies the preferred security policy before the security policy is applied.

Although the security support at the link layer of NoT protocols may be still needed for establishing a secured connection before an NoT device's capability profile can be accessed, in the application layer, the plug-in-style design of security handlers enables the rimware to adaptively apply the preferred security policy on the communication channel between the gateway and the device according to the security policy specification in the device capability profile The templates of security handlers in the knowledge base, combined with the capability profile on the firmware, enables the scalability of the security enforcement for NoT protocols. We name this technique *adaptive application-level security enforcement*.

4.4.3 Device-initiated Adaptive Privacy Protection with Knowledge Base

For privacy protection, the knowledge base also uses a plug-in-style design on authentication as it does on security enforcement. In rimware, NoT devices are considered as data sources. Preventing the data sources from being abused by a fraudulent party is important. When an NoT device that was previously in rimware is temporarily disconnected from the local gateway for some reason, such as to roam to another place, it is possible that a fraud party detects the device and tries to connect to it before a rimware gateway finds the device. Authentication used in many NoT protocols that rely on simple pairing with web-password verification does not solve this issue. For this reason, different from the general sense of authentication, which is to "verify of the identity of the remote device" [20], in rimware the authentication is performed reversely. In rimware, the authentication is initiated from NoT device side. The purpose of the authentication is to check whether the other party is trusted, i.e., whether it is a gateway of rimware, in order to protect the device's privacy. The knowledge base maintains a template list of authentication handlers as shown in Fig. 4.3. The *authentication handler* is a piece of code that performs the authentication checking against the gateway with the authentication mechanism specified in the NoT device's capability profile. Additional templates that contain new authentication mechanisms can be contributed by developers when they apply them on their devices. When an NoT device joins rimware through a gateway, the KB manager in the knowledge base finds the authentication requirement from its capability profile and initializes an instance of authentication handler of the corresponding type. Different from security handler, authentication handler only handles the one-time authentication when an NoT device joins the rimware. Therefore, instead of being kept at the gateway after initialization, the authentication handler remains in the knowledge base and destroy itself after performing the authentication with the device.

As one example, if the authentication is based on a pre-defined password, i.e., both the device and the authentication handler know the password beforehand, then the handler sends the correct password to the device through gateway for verification. Depending on the type of authentication, the authentication information may be initialized and stored both in the device's KB record and on the device side.

As another example, assume the authentication is based on access token matching mechanism, which represents the credentials for the rimware's privileges to fully or partially access the device. When the device joins the rimware for the first time, the authentication handler either generates a random token or receives the token from the device, and ensures that the token information is stored both in the device's KB record on the rimware side and in the device's capability profile on the device side. Assuming the device disconnects from the rimware and connects back from another gateway, a newly initialized authentication handler first checks whether or not an access token is already stored in the device's KB record. If there is one, then the handler sends the token to the device for access control verification.

The plug-in-style design of authentication handlers enables the rimware to adaptively perform the authentication checking from a device to a gateway according to the preferred authentication mechanism specified in the device's capability profile, thus enabling the scalability of the privacy protection. We name this technique as *device-initiated adaptive privacy protection*.

4.4.4 Task Scheduler

The task scheduler receives task requests from application domains and schedules the tasks to run on the desired NoT devices. Tasks that are scheduled by the task scheduler can be categorized into two types in terms of their time duration: direct-access tasks and long-term tasks.

Direct-access tasks are those that access one or more devices in sequence or in parallel through the web APIs or by composed queries. A scheduled task returns immediately after the result is obtained from the NoT device. For example, a direct-access task can be scheduled against a heart rate monitoring device to read the current heart rate. If the task succeeds, the task returns with the current heart rate. If the task fails, the task returns with an error message obtained from the device's side.

As the second type, *long-term tasks* serve the purpose of collecting data by subscribing to the data generated by a particular functionality of any device, such as collecting heart rate readings constantly from the heart rate monitoring device. Multiple types of data can subscribed and collected by a long term task. Because a long terms task constantly collects the data, it cannot return a complete result instantly after the task is issued. Instead, the temporary result is stored in the data store and two URLs are returned as result for each of the subscribed data. One of the URLs is for accessing the result stored in the data store, and the other one is the API to receive the push notification for the latest result. Details on scheduling tasks is discussed in Chapter 6.

4.4.5 Data Store

Each entry stores the result of a subscribed data generated by a particular functionality of an NoT device in a long-term task. For every entry in the data store, an observing processing is initialized to receive the latest data and to store the data at the data entry whenever there is a push notification received from the subscribed data on the corresponding functionality of the device.

Chapter 5

Device Capability Profile

In rimware, different functionalities from one NoT device may be used by different application domains. Rimware defines a formalized profile-based interface on the NoT's firmware to abstract its capability. Every functionality of the device can be utilized individually through the device capability profile. In addition, security and privacy prerequisites for the communication between an NoT device and the rimware can also be specified using the capability profile. This chapter presents the device capability profile model.

5.1 Device Capability Profile Model

In recent years, more and more NoT devices are designed to serve for not only one but multiple application purposes. For instance, a light switch controlling device may have some other on-board modules for detecting the ambient environment, such as temperature or luminance, with which the device can do more than just turning on or off light. The purpose of modeling a device is to describe an NoT device's capability in a formal way such that different functionalities from the device can be utilized by different application domains. Our work attempts to model the functional-level

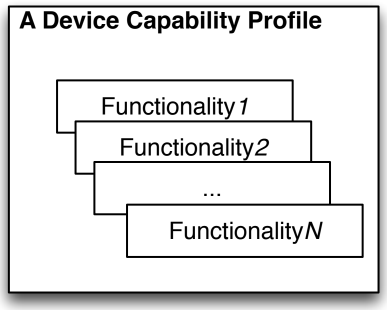


Figure 5.1: An example of a device capability profile model

capability by explaining how a device would be used in real practice.

5.1.1 Functionality

As shown in Fig. 5.1, in the device capability profile model, an NoT device's capability is composed of a list of functionalities. A functionality is a set of actions or features provided by an NoT device that serves for the same purpose. From the hardware aspect, a functionality usually represents the functions provided by a particular hardware module. Each functionality in a device's capability is able to serve individually without depending on other functionalities. If some actions or features in a device have dependencies between one another, each of them is considered as part of the same functionality.

For example, as shown in Fig. 5.2, an infant monitor device provides four different functionalities:

- "Device Information" is the functionality that provides general information about the device such as manufacturer, serial number, software revision, etc.,
- "Body Temperature" is the functionality provided by the on-board temperature-sensing module that constantly monitors the current body temperature reading of the infant,
- "CO_2" is the functionality provided by the on-board CO_2 sensing module that constantly detects the CO_2 density around the infant's face, and

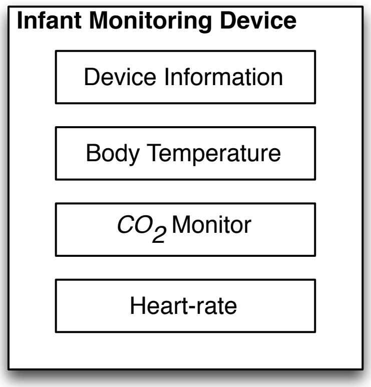


Figure 5.2: Functionalities of an infant monitoring device

- "Heart-rate" is the functionality that constantly measures the infant's heart rate readings.

5.1.2 Attribute

In a functionality, there are one or more attributes. An example of a functionality model is given in Fig. 5.3. An attribute represents a particular action or feature in a functionality. In the model of the device capability profile, every attribute is composed of a single value and additional descriptions. The value represents the attribute's current behavior. The additional descriptions provide information about the attribute's value, such as the format, unit, human-readable description, etc. Attributes in the same functionality may be associated with one another. Attributes are categorized into five types in terms of their access permissions. Five predicates with different semantics are defined to denote different types of attributes that may be contained in a device's functionality as follows:

- *fact*
- *property*
- *stream*
- *security*

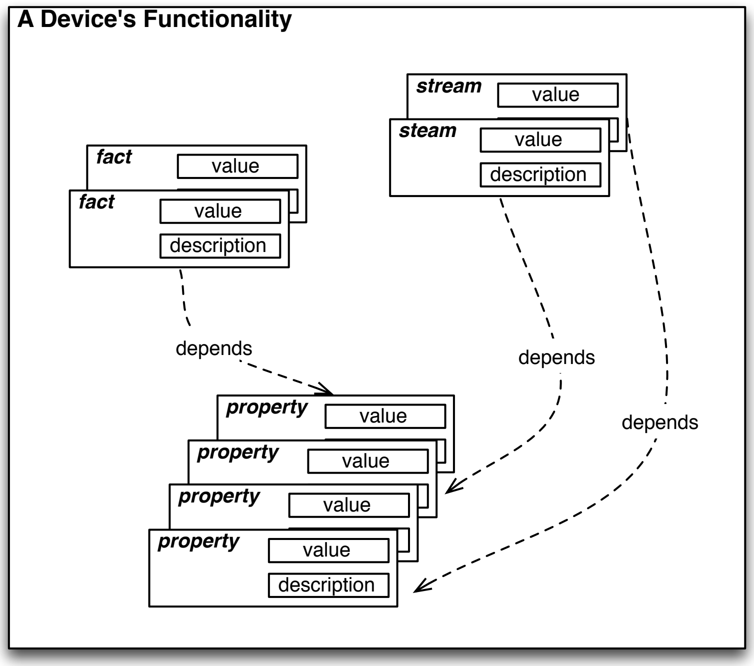


Figure 5.3: Example of a device's functionality model

- *authentication*

The *fact*-typed attribute and *stream*-typed attribute represent a functionality's behavior. In a functionality that contains multiple *fact*-typed attributes or *stream*-typed attributes, each of the attributes represents a part of the functionality's behavior. The *property*-typed attribute in a functionality, on the other hand, affects the behavior of the functionality or any *fact*-typed or *stream*-typed attribute in the functionality. In some particular cases, the *property*-typed attribute may also represent a functionality's behavior. The *security*-typed and *authentication*-typed attributes are two special types of attributes that do not exist in general functionalities. They are only used in functionalities that specify the required security policies and authentication mechanisms to be applied.

The next three subsections discuss each of the first three attribute types. The rest two attribute types is discussed in Section 5.4.1.

5.1.3 *fact*-typed Attribute

A *fact*-typed attribute defines the data that is provided by a functionality in an NoT device's capability. The "fact" term, which is borrowed from Data warehouse [31] research field, is often used to represent the data reported at raw level. A functionality can have multiple *fact*-typed attributes. Examples of *fact*-typed attribute of a device includes raw readings, device specification, device's state information, etc. For example, in an NoT device, the device's hardware details, such as the manufacture's name, hardware revision, model number, etc., are considered *fact*-typed attributes from the functionality of the device that provides the information of the device's specification. Because of the immutable characteristic, *fact*-typed attribute permits only read operations.

In some particular cases, it is possible that a functionality of a device may have no *fact*-typed attribute. For instance, suppose in a light switch controlling device there is a functionality implemented to take inputs to control the light's on/off state as well as to provide the readings on the current state of the light. The light's on/off state then can be accessed by both read and write operations, and therefore it is not a *fact*-typed attribute, but a *property*-typed attribute that is discussed in Section 5.1.5.

5.1.4 *stream*-typed Attribute

The *stream*-typed attribute describes a data stream formed by a sequence of same type of data generated constantly by an NoT device's functionality. A functionality can have multiple *stream*-typed attributes. The *stream*-typed attribute is often generated either periodically or based on some specific conditions. For example, suppose there is a functionality provided by a heart rate monitoring device monitors the heart rate readings on a human body at a certain sampling rate and outputs them as a sequence of data, and generates a lead-off signal when a lead is off the body which prevents the device from receiving heart rate readings. The consistent sequence of heart rate readings is a *stream*-typed attribute in the functionality. The lead-off detection signal is another *stream*-typed attribute in the functionality.

The *stream*-typed attribute also permits only read operations as it has the same immutable characteristic as the *fact*-typed attribute does. However, when a *stream*-typed attribute is accessed by read operations, each operation returns only the latest result instead of returning a constant data sequence. To receive all the data from a *stream*-typed attribute, the publish/subscribe [50] mechanism is often used, with which a notification message containing the latest result is sent from the attribute to the subscribers every time when a new result is generated.

The time interval or the condition that affects a *stream*-typed attribute's behavior, if adjustable, is often exposed as a *property*-typed attribute in the device's functionality, which is discussed in Section 5.1.5.

5.1.5 *property*-typed Attribute

The *property*-typed attribute specifies the adjustable setting that affects the behavior of a functionality. A functionality can have multiple *property*-typed attributes. A *property*-typed attribute must at least be accessible by write operations and also possibly by read operations or even in a publish/subscribe mechanism. For instance, in the example of the heart rate monitoring device in Section 5.1.4, the current sampling rate may also be considered as valuable information in some use cases. Therefore, the *property*-typed attribute that is used to control the sampling rate may be set to be accessible by both read and write operations.

If in a functionality of an NoT device, there is a *property*-typed attribute that is specifically used to control the behavior of any *fact*-typed attribute or *stream*-typed attirbute, we say the *fact*-typed attribute or *stream*-typed attribute *depends on* the *property*-typed attribute. For instance, the adjustable sampling rate parameter in the example of the heart rate monitoring device in Section 5.1.4 can be exposed as a *property*-typed attribute that the sequence of heart rate readings depends on.

Dependency relationship can only be established from *fact*-typed or *stream*-typed attributes that represent the behaviors of functionalities to *property*-typed attributes that affect the behaviors of functionalities. It is possible that a *fact*-typed or *stream*-typed attribute depends on multiple *property*-typed attributes. If two *property*-typed attributes in a functionality are mutually dependent, then at least one of them must be able to represent the behavior of another functionality. For example, considering the light switch controlling device in Section 5.1.3, the device is equipped with an ambient-light sensing module and programmed to be able to automatically turn on the light when the ambient luminance is below a specific threshold, then the threshold settings can be exposed as another *property*-typed attribute in its functionality that serves for controlling the light switch. However, the behavior of the previous *property*-typed attribute, the light's on/off state, depends on the threshold value, which is not allowed. For this reason, the two *property*-typed attributes are separated into three independent functionalities. The first controls the light switch directly by taking inputs of the on/off state. The second controls the light's on/off state automatically according to an input threshold. In the second functionality, the light's on/off state should be specified as an attribute with *fact*-typed, because in this functionality, the light's state should not be controlled directly.

The *property*-typed attributes can be further classified into three categories in terms of different types of actions they base upon to affect the behavior of a functionality or the behavior of other types of attributes:

- Conditional actions based: The *property*-typed attributes in this category affect the behavior of a functionality or an attribute when a particular event with a setup condition is triggered. For instance, in the example of automatic light switch controlling based on ambient luminance mentioned above, the attribute that is used to store an input of the threshold value for automatic light switch controlling belongs to this category.
- Instant actions based: The *property*-typed attributes in this category controls the enablement/disablement of a functionality or an attribute through an instant action. For example,

the attribute that takes an input to control the light's on/off state in the example of the light switch controlling device in Section 5.1.3 simulates the instant action of turning on or off a light through a light switch, thus belonging to this category.

- Periodic actions based: The *property*-typed attributes in this category affect the behavior of a functionality or an attribute periodically, where the attributes specify the frequency. For instance, the attribute for storing the input of sampling rate in the example of the heart rate monitoring device mentioned above belongs to this category.

Note that the *property*-typed attribute is not designed to represent the behavior of a functionality. However, a *property*-typed attribute can represent the behavior of a functionality in some particular cases, when it represents the behavior of itself. In other words, no other attributes depend on the *property*-typed attribute in the functionality. For instance, the on/off state information in the light switch controlling device in the example in Section 5.1.3 is a *property*-typed attribute that represents the behavior of the functionality as well as the behavior of itself since there is no other *fact*-typed or *stream*-typed attribute in the functionality. This special situation happens only in functionalities that represent the behaviors of actuator modules.

5.2 Syntax Expression of Device Capability Profile

In the expression of a device capability profile model, the capital letter C is used to denote a device's capability. The capital letter F is used to denote a functionality inside a device's capability. Different types of attributes are presented by the three predicates: *fact*, *stream*, and *property*. The names of the device capability profile, functionalities, and attributes are represented as subscripts. In the parenthesis after each functionality, attributes that belong to the functionality are separated by commas. In the parenthesis after each attribute, the names of *property*-typed attributes, separated by commas, are given to denote that the attribute's output depends on the behavior of the

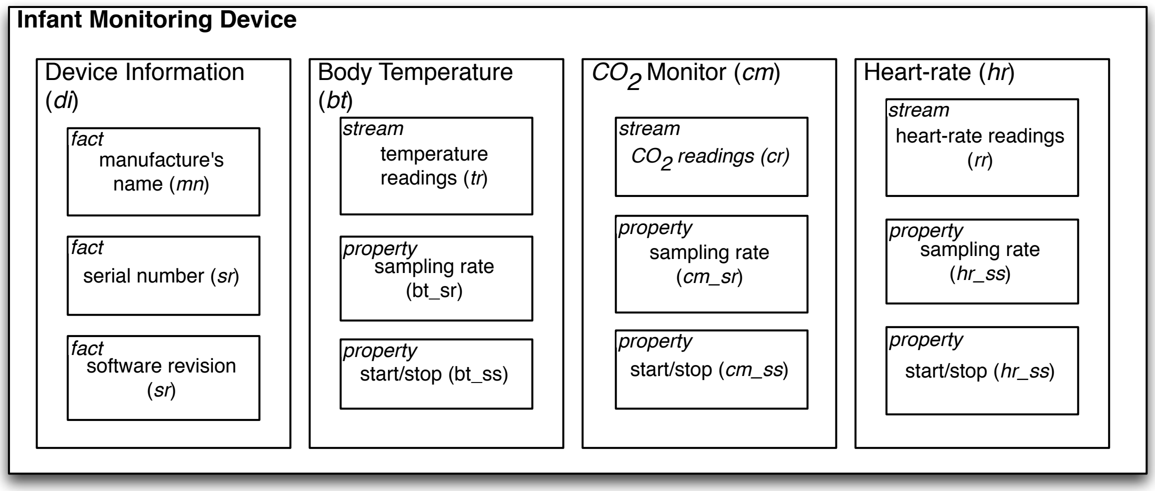


Figure 5.4: A complete device capability profile of the infant monitoring device

property-typed attributes.

We revisit the example shown in Fig. 5.2 and complete the device capability profile by adding attributes into each of its functionalities as shown in Fig. 5.4. In each attribute, the type information is given in the upper-left. The short names of each functionality and attribute are given in the parentheses, respectively. Supposing that the infant monitoring device's name is *imd*, its device capability profile can be expressed as follows:

$$\begin{aligned}
C_{imd}(F_{di}(fact_{mn}, fact_{sn}, fact_{sr}),\\
F_{bt}(stream_{tr}(bt\_sr, bt\_ss), property_{bt\_sr}, property_{bt\_ss}),\\
F_{cm}(stream_{cr}(cm\_sr, cm\_ss), property_{cm\_sr}, property_{cm\_ss}),\\
F_{hr}(stream_{rr}(hr\_sr, hr\_ss), property_{hr\_sr}, property_{hr\_ss}))
\end{aligned}, \tag{5.1}$$

where each of the components in the device capability profile is identified by its short name.

Table 5.1: Required access permissions of different types of attributes

	Access Permissions		
Attribute	**Read**	**Write**	**Subscribe**
fact	yes	no	no
stream	optional	no	yes
property	optional	yes	optional
security	implementation dependent		
authentication	implementation dependent		

5.3 Device Capability Profile in Firmware

In an NoT device's firmware, the device capability profile work as an application on top of the the device's operating system environment. Every device should have only one capability profile. The next two subsections discuss the specification of each of components as well as the access control of device capability profile.

5.3.1 Functionality, Attribute, and Attribute's Description in Firmware

In firmware, every functionality that contains one or more attributes should be addressable with an ID or handle. Each attribute in a functionality is composed of a value, an ID or handle to access the value and one or more descriptions that describe the attribute as well as its value. Each attribute in a functionality should be assigned with the proper access permission as shown in Table 5.1. The value of each attribute should be accessible individually through message exchange on secured and authenticated communication channel. An observer function should be associated with each of the *stream*-typed attributes to enable the publish/subscribe data push mechanism and should be able to be activated or deactivated by an input "subscribe" signal or "unsubscribe" signal. When an observer is activated, it notifies the subscriber whenever there is an update on the value of the associated *stream*-typed attribute.

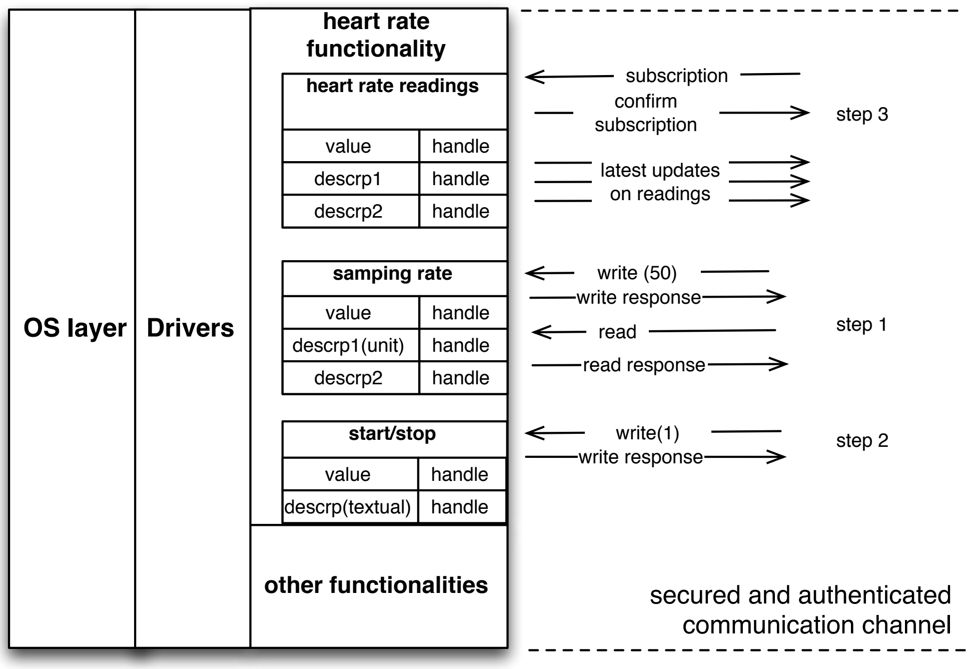


Figure 5.5: An example of infant monitoring device interacting on the communication channel with detailed description on heart-rate monitor functionality

Descriptions of attributes should be categorized into different types in terms of the different aspects they describes such as information about presentation format including the value's type, unit, exponent, information about the valid range of the value, dependency relationship with other attributes, a textual description of the attribute in natural language, etc. Each type of the descriptions should be stored in order in the attribute or identified by its type ID if multiple descriptions with the same type in an attribute are allowed. For a *property*-typed attribute, a description dedicated to describing the action types as discussed in Section 5.1.5 should be specified. For *fact*-typed and *steam*-typed attributes, a description dedicated to describing the dependencies as discussed in Section 5.1.5 should be specified.

Fig. 5.5 shows an example of the infant monitoring device mentioned in Section 5.2 interacting on the communication channel with details given on its heart rate functionality. In this example, descriptions of attributes are obtained first to understand the meanings of the values for every attribute. A integer value of 50 is sent to the "sampling rate" attribute to setup the sampling rate to

50 Hz. A integer value of 1 indicating a start signal is sent to the “start/stop” attribute to start the heart rate monitoring. A subscription message is sent to the “heart rate readings” attribute to ask for notifications whenever there is an update on heart rate reading. The communication between the heart rate monitoring device and the other party is carried out over a secured and authenticated channel. Section 5.4.1 discusses how the device capability profile is configured for establishing a secured and authenticated communication channel.

5.3.2 Access Control on Device Capability Profile

Access control should be established at different levels against functionalities, attributes, and attribute descriptions. Upon the establishment of a connection to an NoT device, the device capability profile structure, including the hierarchy of the profile, the IDs or remote handles of each functionality, attribute, and attribute’s description should be obtainable by the other party without restrictions. However, a device capability profile should be designed not to expose any content of attributes in any of the functionalities except the ones that stores the device’s information, preferred security policy, and authentication mechanism before the communication channel is secured and authenticated.

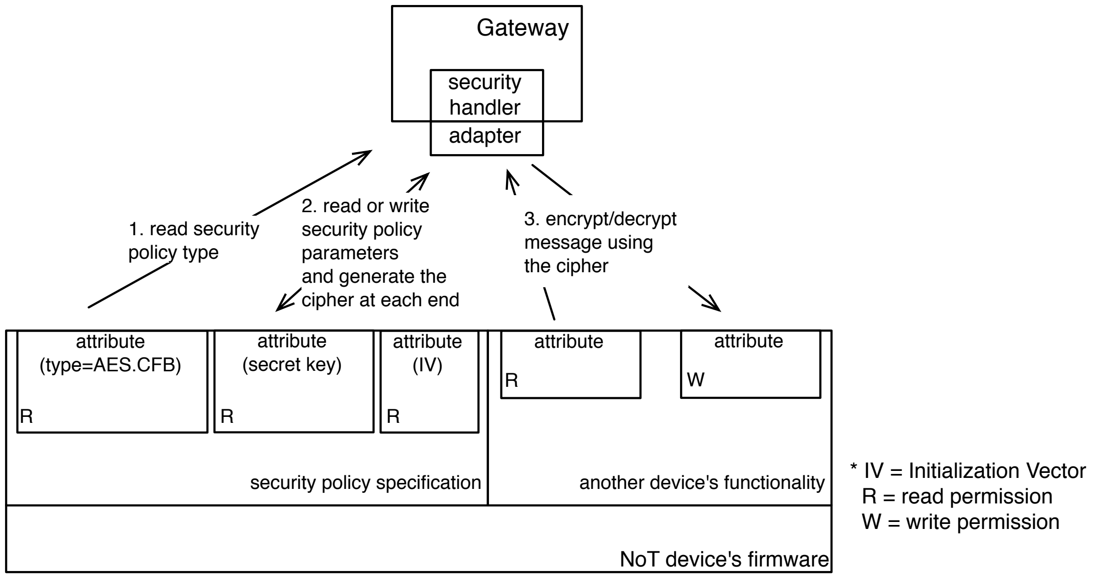
5.4 Security and Privacy Specification using Device Capability Profile

As discussed in Sections 4.4.2 and 4.4.3 the rimware solves the scalability issues for both the security enforcement and the privacy protection by using the *adaptive application-level security enforcement* technique and the *device-initiated adaptive privacy protection* technique, respectively. To apply these two techniques, the device’s capability profile is configured to contain the specifications about the preferred security policy and authentication mechanism. In the device’s capability model, security and privacy are also considered as two functionalities that a device is capable of. Therefore, their specifications are specified in two separate functionalities. Different from a regular functionality that represent's a part of the behavior of a device, these two special functionalities contain only *security*-typed attributes and *authentication*-typed attributes, respectively. A *security*-typed attribute or *authentication*-typed attribute can have any type of access permissions, including Read, Write, Read&Write, depending on the specification requirement. Each of the two functionalities is assigned a pre-defined ID or handle to indicate that the functionality is used for the specification of security policy or authentication mechanism. The next two subsections discuss the structure inside each of the functionalities.

5.4.1 Security Description in Device Capability Profile

In the functionality for security policy specification, an attribute that stores the type information of the preferred security policy, with read access permissions, always exists. The type information is used by the centralized side of the rimware to find the template of the security handler with the corresponding type so that an instance of the security handler can be initialized and utilized on the gateway side as discussed in Section 4.4.2. Other parameter values that are used to apply the security policy are stored in other attributes. Two examples are given as follows to illustrate the specification of security policies using attributes with different access permissions in the security functionality.

The example as shown in Fig. 5.6 illustrates the configuration of a device capability profile for security policy specification based on a symmetric key cryptography algorithm as well as the the process of applying of the security policy . On the firmware side, an attribute with read permissions is used to store the type information in the security policy specification functionality. This attribute states that the Advanced Encryption Standard (AES) with Cipher FeedBack (CFB) mode [25] should be used for security, which is basically sharing a same secret and an initialization vec-



Gateway
security handler
adapter
1. read security policy type
2. read or write security policy parameters and generate the cipher at each end
3. encrypt/decrypt message using the cipher
attribute (type=AES.CFB)
R
attribute (secret key)
R
attribute (IV)
R
attribute
R
attribute
W
security policy specification
another device's functionality
NoT device's firmware
* IV = Initialization Vector
R = read permission
W = write permission

Figure 5.6: An example of applying a security policy based on a symmetric key cryptography algorithm

tor to generate the cipher for encryption or decryption at each end. Another two attributes with read permissions are used to store the necessary parameters, the secret key and the initialization vector in the functionality. The values of the parameters are randomly generated by the device. On the gateway side, as discussed in Section 4.4.1, the security handler with the corresponding type generates a cipher by acquiring the necessary parameter values by accessing the device capability profile. On the devices side, upon sending the parameter values to the gateway side, the device generates a cipher with the same parameters, and removes the access restrictions on other functionalities so that the gateway can send encrypted messages to access those functionalities.

Another example of profile configuration on security policy specification is given in Fig. 5.7 for applying the security policy that utilizes the RSA public-key cryptography algorithm implemented in Public-Key Cryptography Standards (PKCS) with Optimal Asymmetric Encryption Padding (OAEP) scheme [10]. To send and receive encrypted message, both sides need to generate a pair of public and private keys, and share the public key generated by their own to the other side for encryption and keep the private key for decryption. For that reason, the attribute that is used to stored the public key to obtain is set with write permission to obtain the public from the gateway

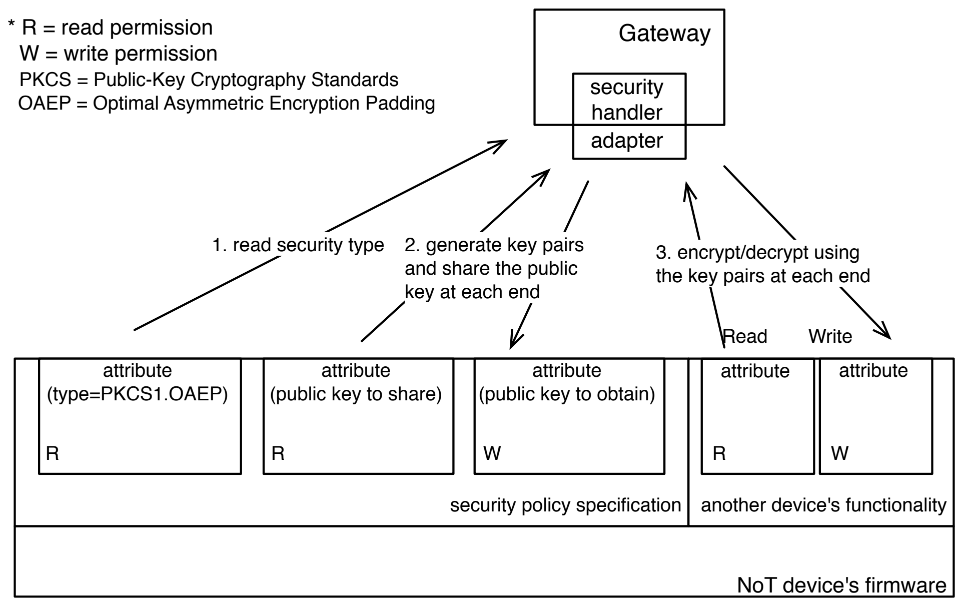

* R = read permission
W = write permission
PKCS = Public-Key Cryptography Standards
OAEP = Optimal Asymmetric Encryption Padding
Gateway
security handler
adapter
1. read security type
2. generate key pairs and share the public key at each end
3. encrypt/decrypt using the key pairs at each end
Read
Write
attribute (type=PKCS1.OAEP)
R
attribute (public key to share)
R
attribute (public key to obtain)
W
attribute
R
attribute
W
security policy specification
another device's functionality
NoT device's firmware

Figure 5.7: An example of applying a security policy based on a public-key cryptography algorithm

side.

Note that, in the case that an authentication checking is required by the device side, other functionalities should not be accessible until the authentication checking succeeds. The next subsections discuss privacy specification in the device capability profile that describes the mechanisms to be used for authentication checking.

5.4.2 Privacy Description in Device Capability Profile

The functionality for authentication mechanism specification follows a similar structure as the one used in the security functionality. An attribute with read permissions is used to store the type information of the preferred authentication mechanism. The information is used by the centralized side of the rimware to initialize the authentication handler with the corresponding type as discussed in Section 4.4.3. The parameters that need to be exchanged between the NoT device and the authentication handler are stored in other attributes with the corresponding access permissions.

Fig. 5.8 illustrates the capability profile configurations on two different NoT devices for two au-

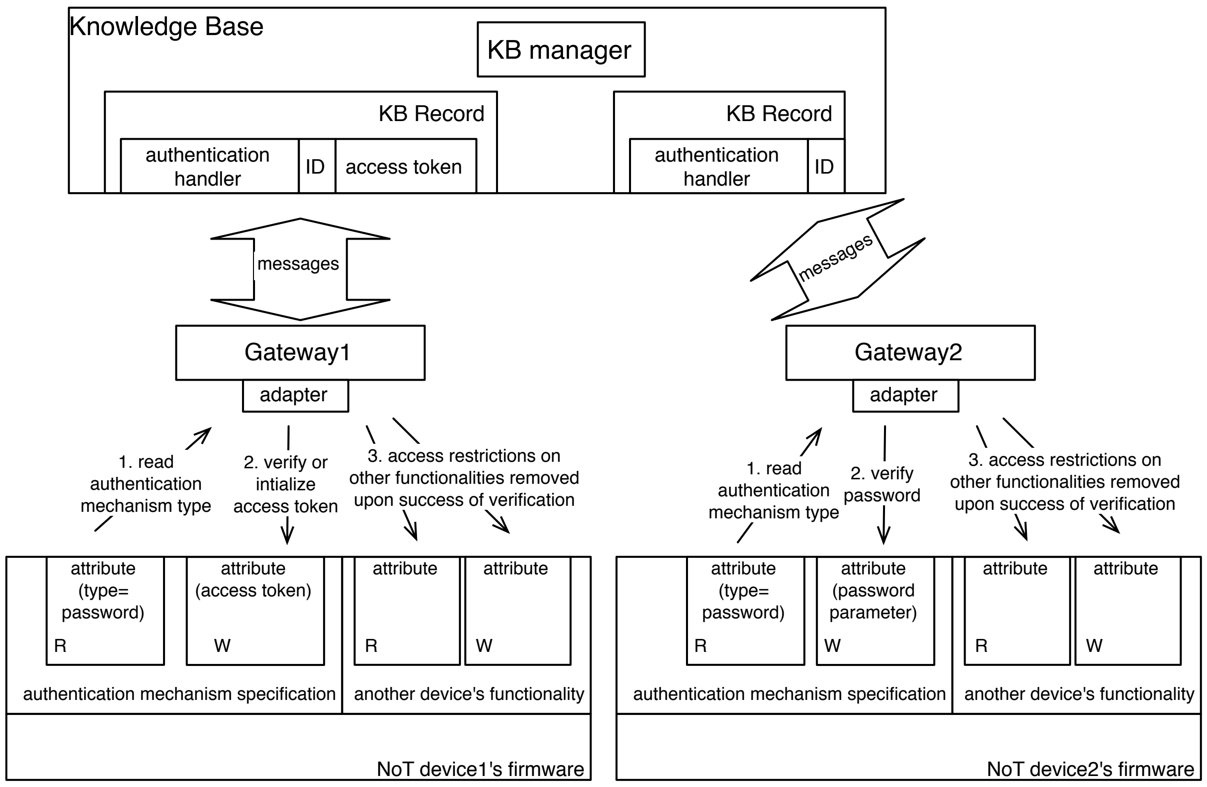

Knowledge Base
KB manager
KB Record
authentication handler
ID
access token
KB Record
authentication handler
ID
messages
messages
Gateway1
adapter
Gateway2
adapter
1. read authentication mechanism type
2. verify or intialize access token
3. access restrictions on other functionalities removed upon success of verification
1. read authentication mechanism type
2. verify password
3. access restrictions on other functionalities removed upon success of verification
attribute (type= password)
R
attribute (access token)
W
attribute
R
attribute
W
authentication mechanism specification
another device's functionality
NoT device1's firmware
attribute (type= password)
R
attribute (password parameter)
W
attribute
R
attribute
W
authentication mechanism specification
another device's functionality
NoT device2's firmware

Figure 5.8: An example of authentication checking based on two different mechanisms

thentication mechanisms that are discussed in Section 4.4.3 for the protection of the devices' privacy. The device on the right uses an authentication mechanism is based on a pre-defined password, i.e., both the device and the authentication handler know the password beforehand. The device on the left performs authentication checking based on an access token that is randomly generated by the authentication handler at the first time when the device joins rimware.

Chapter 6

Query and Task Scheduling

As mentioned in Chapter 4, in the rimware environment an NoT device can be accessed either through the web APIs that are translated from the device capability profile or by composed queries from application domains. In addition, different types of tasks can be scheduled to allow the functionalities from one device to work for different tasks as well as to allow the functionalities from different devices working together for one task. This chapter discusses how the queries and tasks are composed in rimware. Optimization on task scheduling is also discussed.

6.1 Access on Attributes

In rimware, the utilization of an NoT device's functionality is done by accessing the attributes of the functionality. Four types of access can be made against an attribute in terms of different access permissions as follows:

- Read access for an access with read permission,
- Write access for an access with write permissions,

Table 6.1: Mappings between access type and standard HTTP methods

Access Types	standard HTTP Methods
read	GET
write	PUT
subscribe	POST
unsubscribe	DELETE

- Subscribe access for a subscription against any of the *stream*-typed attribute that is discussed in Section 5.3
- Unsubscribe access for stopping the generation of notifications from a *stream*-typed attribute

Attributes can be accessed in two different formats from application domains. One is to access through the web APIs that are translated from the device capability profile as discussed in Section 4.4.1. The other is to access by composed queries. The next subsections discusses each of the formats. To access an NoT device's attribute, the device's KB record that contains information such as the device's ID, the capability profile structure information, web APIs, etc., has to be obtained first. Section 6.1.3 discusses how the information in an NoT device's KB record is searched.

6.1.1 Web APIs-based method

In the knowledge base (Section 4.4.1), for every NoT device, every attribute in a functionality of the device is translated into a web API. A format of the API is shown as follows:

```
https://device_id/functionality_id/attribute_id
```

The web APIs follow the REST architectural style to make use of standard HTTP methods for different types of access.

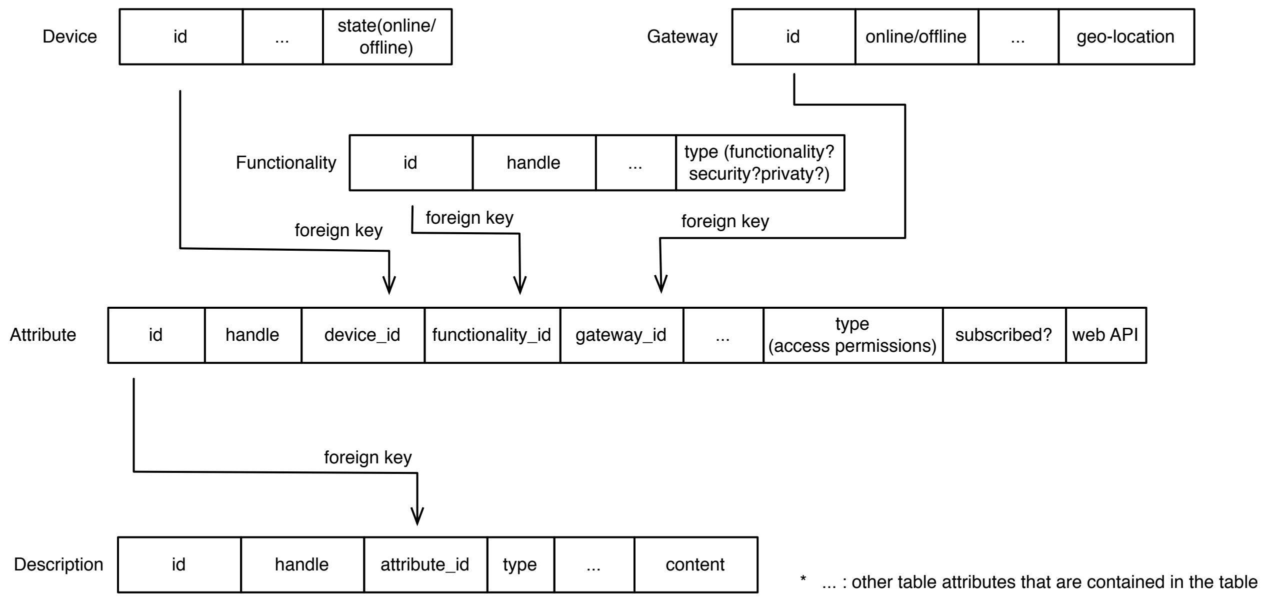


Figure 6.1: The table structure used for KB records maintenance in a relational database

6.1.2 Query-based method

A query-based access against a device's attribute is made through a universal web API. The syntax of a query statement can be expressed as follows, beginning with keyword *Query*:

$$Query(access\_type,\ device\_id,\ functionality\_id,\ attribute\_id,\ [attribute\_value]) \tag{6.1}$$

, where *access_type* is one of the four access types mentioned above, the "device_id", "functionality_id", "attribute_id" specify the identification of the device and functionality that the attribute belongs to as well as its own identification, "attribute_value" parameter that only exists in query with write permissions specify the value to be sent to the attribute. When a query is processed, a feedback message is returned indicating whether or not the query succeeds

Table 6.2: An example of searching for an online device base on various conditions

```
select Device.id
from  Attribute, Device, Gateway, Functionality
where Device.id = Attribute.device_id and
      Gateway.id = Attribute.gateway_id and
      Functionality.id = Attribute.functionality_id and
      Gateway.geo-location = ``Irvine, CA'' and
      Attribute.handle = ``FFEC'' and
      Functionality.handle = ``FFE0'' and
      Device.state = ``online''
```

6.1.3 Search for Device's KB Record

In rimware, the KB records are maintained using a relational database. Information about a device's state, functionalities, attributes, descriptions of attributes and the device's corresponding gateway is store in database tables. Fig. 6.1 provides the table structure that is used for storing KB records in a relational database. The "handle" column in table "Functionality", "Attribute", and "Description" stores the handle address or an ID that is used to identify the functionality, attribute, or description in the device.

Standard SQL [49] queries are used to find the information of any device with a specified condition in the database. Table 6.2 shows an example for searching the identification list of the devices from the database where every device in the list is located in Irvine, CA, and has a functionality with a handle address of 0xFFE0, and there is an attribute in the functionality with a handle address of 0xFFEC.

6.2 Task Scheduling

In addition to access through web APIs and queries against an NoT device, rimware allows tasks to be scheduled against one of multiple NoT devices. A task can be considered as a superset of individual accesses on attributes that are issued in different orders. We use query-base access to denote a subset of a task in this section. If in a task there is only one individual access on a single attribute against a device, the task is issued as a query as shown in Section 6.1.2. In most cases, a task consists of more than one attribute access.

A task can be scheduled in sequence, in parallel, or mixed. Two tasks that run in sequence are connected using the logical operator **before** as follows:

query-statement **before** *query-statement*

Two tasks that run in parallel are connected using the logical operator **and** as follows:

query-statement **and** *query-statement*

The syntax of a general task statement that may contain sub-tasks in sequence, in parallel, or mixed can be expressed using the logical operator **before**.

task ::- *query-statement* [**before** | **and** *query-statement* | *sub-task*]
sub-task ::- *query-statement* [**before** | **and** *query-statement* | *sub-task*]

There are two types of tasks that rimware can schedule in terms of their time duration: direct access task and long term task as discussed in Section 4.4.4. In a direct access task, each of the queries must be either read or write access or unsubscribe access. When the task finishes, a sequence of the result of with the same order of the queries in the task returns. In a long term task, one or more subscribe accesses must be issued. When a subscribe access is found, the rimware opens an entry in the data store that is discussed in Section 4.4.5 for receiving and storing the incoming data stream sent from the corresponding *stream*-typed attribute. On the result list, for each subscribe access, a tuple that contains two URLs returns. One of the URLs is to access the entry in the data store to obtain the complete result. The other one is the API for receiving push notifications for the latest data results. Note that, in direct access tasks, subscribe accesses can also be issued. However, no entry for the incoming data stream is opened in the data store, which means the device's corresponding gateway will ingore the notification messages sent from the device on the subscribed attribute(s).

6.3 Task Optimization

The rimware support multiple tasks or sub-tasks to run on an NoT device simultaneously. However, in some cases, when a long-term task is scheduled to run against a functionality of an NoT device on a particular *stream*-typed attribute, another task has already been scheduled and running against the same functionality on the same attribute. However, inputs for the *property*-type attributes that affect the behavior of the *stream*-typed attribute may not be the same. This special situation leads to a problem that if there is no optimization on the scheduling, the latter task needs to wait till the the former finishes completely. Therefore, in those particular cases an optimization is needed.

Note that optimization only works for long-term tasks, where different inputs on the same *property*-typed attribute a *stream*-typed attribute depends on may lead to different results stored in the data entries of the data store.

As discussed in Section 5.1.5, a *property*-typed attribute may affect another attribute or the entire functionality based on different actions. For a *stream*-typed attribute, a *property*-type attribute that it depends on can affect its behavior based on either periodic actions or conditional actions. The rimware takes care of optimization on periodic action based *property*-type attribute. For example, the *stream*-typed attribute that generates the heart-rate readings in the heart rate monitoring device example discussed in Section 5.1.4 depends on the "sampling rate," which is a periodic action-based *property*-typed attribute. The next section discusses how the optimization is applied.

6.3.1 Optimization on Tasks Affected by Periodic Actions

When two tasks that are scheduled to subscribe the same *stream*-typed attribute provides two different inputs to the periodic action based *property*-typed attribute that the *stream*-typed attribute depends on, three types of optimization may be used as follow. Note that, in all three types of the optimization, we assume the input value for specifying the frequency of the periodic actions in the *property*-typed attribute is always integer-based. The input value against a *property*-typed attribute from the the latter task is denoted as c. The input value against *property*-typed attribute from the the former task, that is the same as the attribute's value of in the device capability profile, is denoted as p.

- If $c < p$ and p is a multiple of c, then a downsampling service is injected against the result for the latter task.
- If $c < p$ but p is not a multiple of c, then an interpolation service is injected against the result for the latter task to provide a lossy result.
- If $c > p$, which means the latter task requires a higher frequency and thus should receive more results than the former within a certain amount of time, then c is sent to the *property*-typed attribute in the device capability profile to take effect. According to the relationship of the two input values, a downsampling service or interpolation service is injected to the result of the former task. If there are other tasks running to observe the result from the same *stream*-typed attribute, then the downsampling services or interpolation services are injected to their observation process accordingly.

Fig. 6.2 shows the workflow of the optimization process on *stream*-typed attributes that are affected by periodic action-based *property*-typed attributes. Note that, as highlighted in in Fig. 6.2, the *property*-typed attribute may not have read permissions so that the current value of the attribute is not retrievable from the device capability profile. In that case, the optimization process finds the

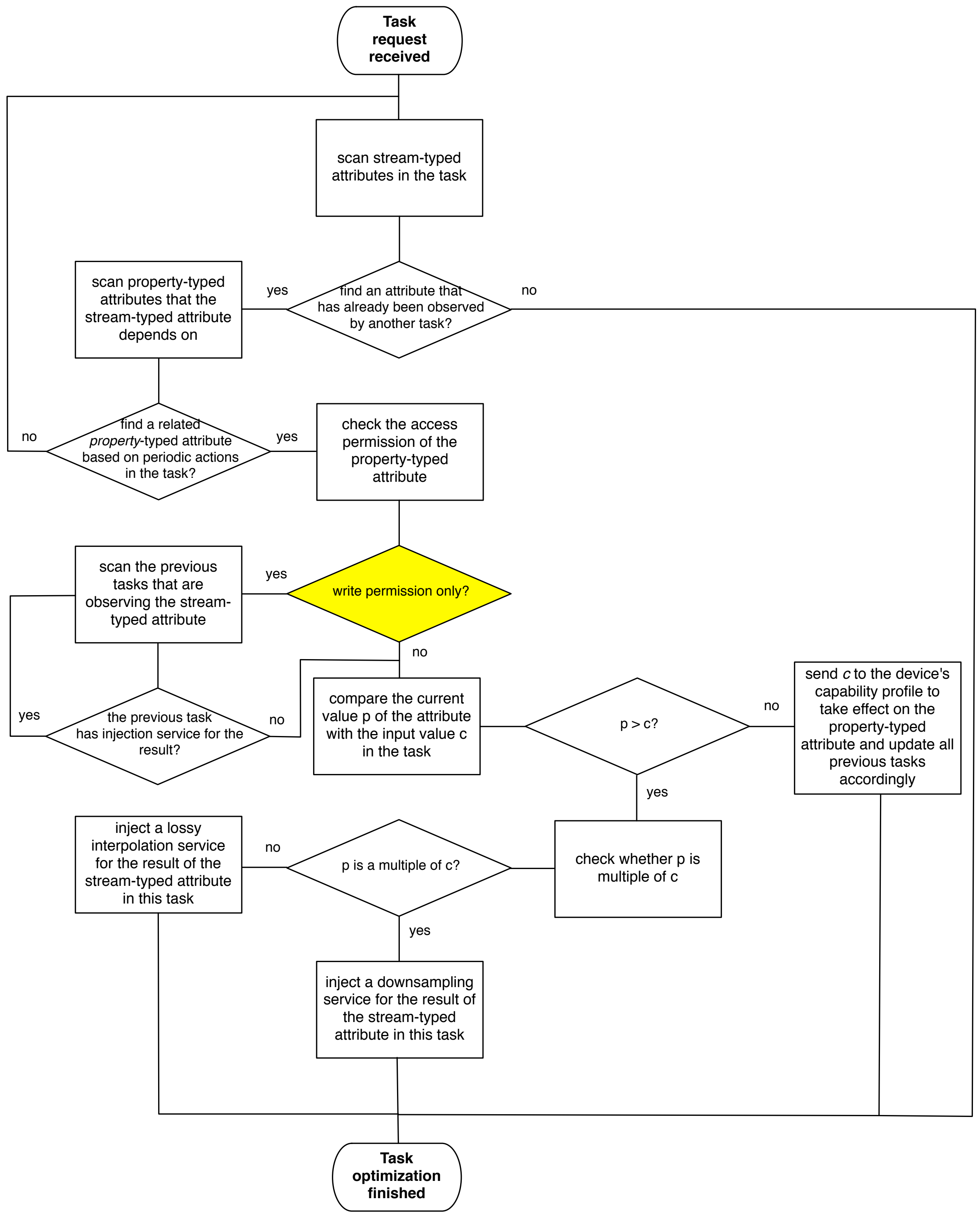

Task request received
scan stream-typed attributes in the task
find an attribute that has already been observed by another task?
yes
no
scan property-typed attributes that the stream-typed attribute depends on
find a related *property*-typed attribute based on periodic actions in the task?
no
yes
check the access permission of the property-typed attribute
write permission only?
yes
no
scan the previous tasks that are observing the stream-typed attribute
the previous task has injection service for the result?
yes
no
compare the current value p of the attribute with the input value c in the task
p > c?
no
yes
send *c* to the device's capability profile to take effect on the property-typed attribute and update all previous tasks accordingly
check whether p is multiple of c
p is a multiple of c?
no
yes
inject a lossy interpolation service for the result of the stream-typed attribute in this task
inject a downsampling service for the result of the stream-typed attribute in this task
Task optimization finished

Figure 6.2: A workflow of the optimization on tasks affects by periodic actions

running task that specifies the input value of the attribute and does not have injection services for its result to obtain the current value of the attribute.

Chapter 7

Implementation: BlueRim

BlueRim is an implementation of rimware that is specifically designed for NoT devices utilizing the Bluetooth Low Energy (BLE) protocol. This chapter discusses the implementation details of each module of rimware in BlueRim, including the capability profile construction in firmware, gateway and centralized components.

7.1 BLE Device and Emulator

The BLE's GATT profile hierarchy is utilized to construct a profile to represent a device's capability at the application layer in the device's firmware. The next section describes the mappings between the GATT-based profile and the capability profile model. The platforms that are used to implement the GATT-based capability profile includes the EcoBT board [1], which utilizes the BLE stack from Texas Instrument (TI), and a MacBook working as an emulator of a BLE device that utilized the BLE framework from Apple. Sections 7.1.2 and 7.1.3 discuss the two respective platforms.

Table 7.1: Mappings of components between a device capability profile and a GATT-based profile

Components in Capability Profile	**Components in a GATT-based Profile**
Functionality	Service
Attribute	Characteristic
Description of attribute	Descriptor

Table 7.2: Mappings between attribute types and characteristic properties

Attribute Types	**Characteristic Properties**
fact	Read
property	Write
stream	Notify

7.1.1 GATT-based Capability Profile

As discussed in Section 2.2, BLE's application layer defines a profile-based structure whose hierarchy is defined in GATT. In BlueRim, an NoT device's capability is presented using a GATT-based application profile, where components in the device capability profile discussed in Section 5.1 including functionalities, attributes and descriptions are described by the components defined by the GATT profile hierarchy. Table 7.1 shows the mappings between components in a device capability profile and the ones in a GATT-based profile, where each functionality is represented by a service, each attribute is represented by a characteristic and each description is represented by a descriptor.

Note that, although in a GATT-based profile, one service can depend from another service as discussed in Section 2.1.1, in the capability profile model, functionalities are independent from each other. For this reason, a GATT-base profile that is used to specify a BLE device's capability is not allowed to have dependencies between services.

Different types of attributes in a device capability profile discussed in Section 5.1.2 are represented by characteristics with different access permissions, i.e., the properties of characteristic discussed in Section 2.2. Table 7.2 shows the mappings between attribute types defined in a device's capability profile and the characteristic properties.

Table 7.3: The structure of 'Characteristic Presentation Format' descriptor used to describe an attribute of a BLE device's capability

Names	**Format**	**Possible Values**
Format	8-bit unsigned integer	1: Boolean; 2: unsigned 2-bit integer; 3: unsigned 4-bit integer, etc.
Exponent based on 10	8-bit signed integer	any 8-bit signed integer
Unit	16-bit unsigned integer	1: ^oF; 2: ^oC; 3: lumen, etc.
Type	8-bit unsigned integer	1: *fact*; 2: *stream*; 3: *property* based on conditional actions; 4: *property* based on instant actions; 5: *property* based on periodic actions;
Dependency	16-bit unsigned integer	UUID of a *property*-typed attribute

Different types of descriptors are used to store the descriptions of attributes. In a GATT-base profile, one characteristic that is used to represent an attribute may be associated with multiple descriptors. The 'Characteristic User Description' descriptors are used to provide natural language-based descriptions of the attribute. The 'Characteristic Presentation Format' descriptors store presentation information of the attribute's value, including the format, exponent, unit, attribute type and other attributes that it depends on. For any *fact*-typed or *stream*-typed attribute discussed in Section 5.1.2, multiple 'Characteristic Presentation Format' descriptors are specified, each of which stores the UUID of one *property*-typed attribute that the attribute depends on. The 'Characteristic Presentation Format' descriptors are specified in a structured format as shown in Table 7.3.

Pre-defined UUIDs are used to identify the services for storing the security and privacy specification and the characteristics that are used to store the security type, authentication type, and other parameters.

7.1.2 EcoBT

The EcoBT board [33], as shown in Fig. 7.1, is a wireless sensor board based on BLE. It is centered around TI's CC2540 System-on-Chip (SoC) in a miniature size. As shown in Fig. 7.2, the software framework provided by TI are separated into several layers.

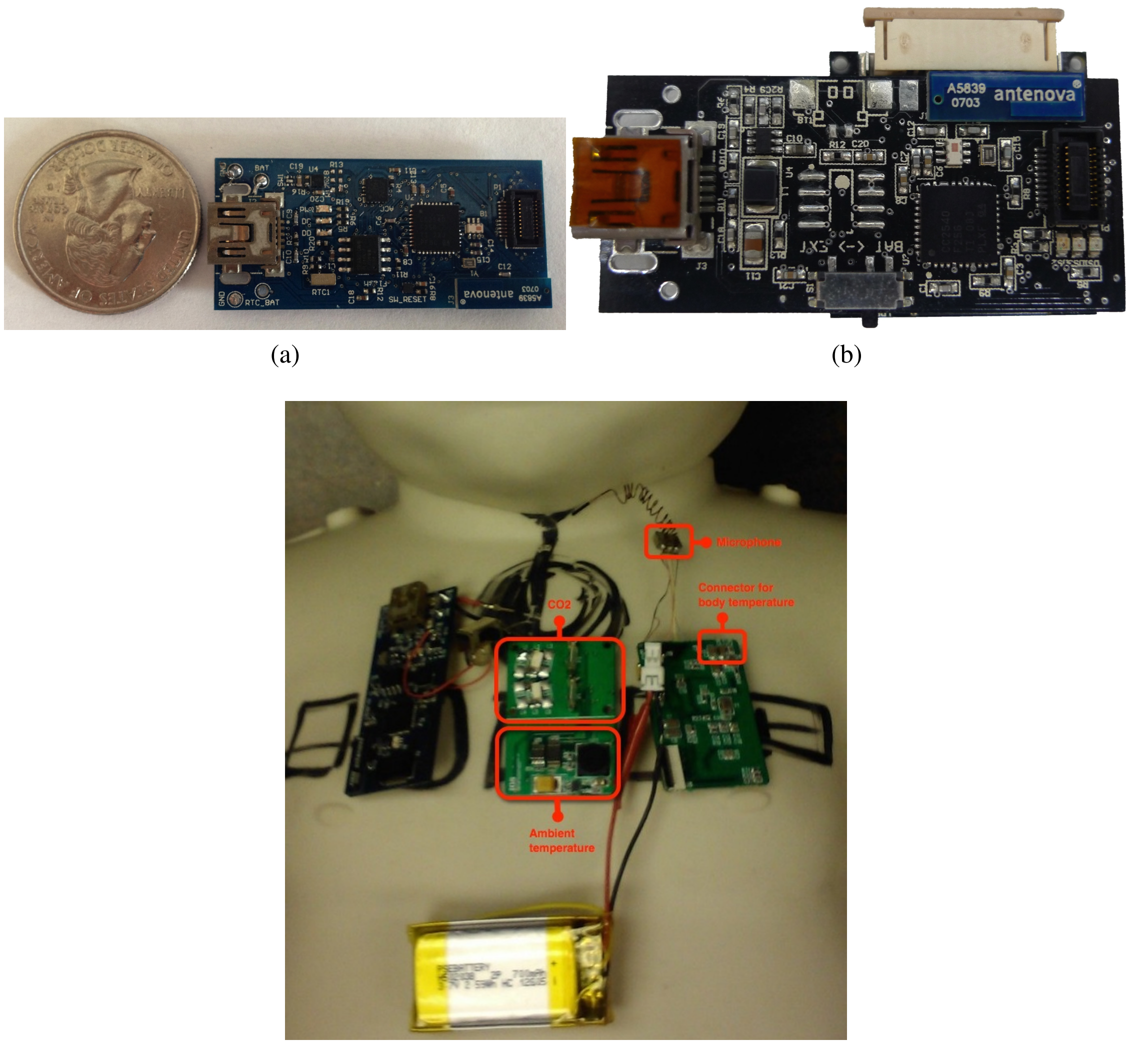


Figure 7.1: (a) EcoBT Platform for ECG recording and infant monitoring, (b) EcoBT with ECG module, and (c) EcoBT with modules for Infant Monitoring

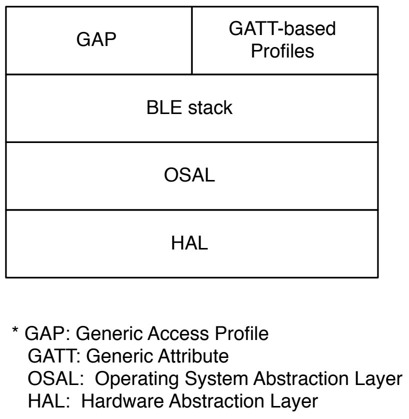

GAP
GATT-based Profiles
BLE stack
OSAL
HAL
* GAP: Generic Access Profile
GATT: Generic Attribute
OSAL: Operating System Abstraction Layer
HAL: Hardware Abstraction Layer

Figure 7.2: Software framework of EcoBT

The Hardware Abstraction Layer (HAL) communicates with the underlying physical hardware through SPI or UART and provides the interface of abstraction of the physical hardware to the upper layers.

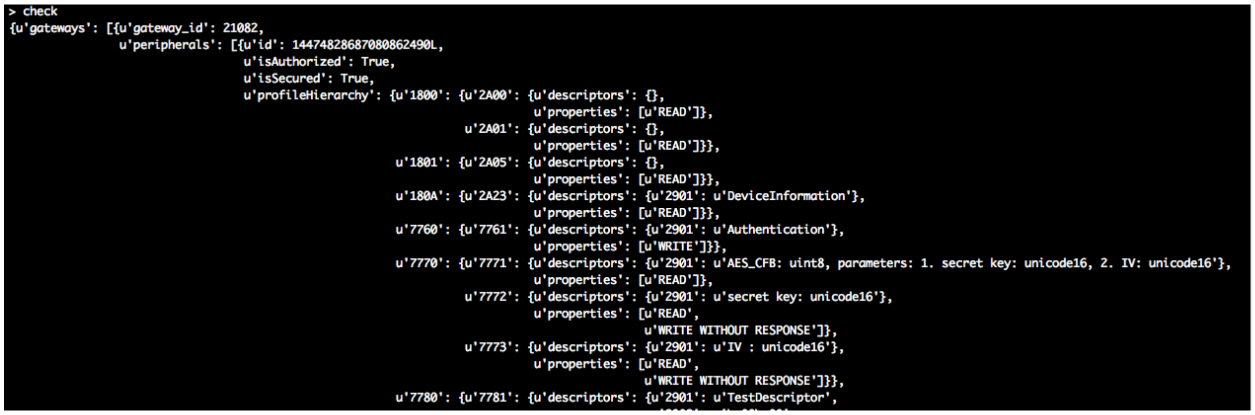
The Operating System Abstraction Layer (OSAL), which is built on top of HAL, is a simple runtime-support layer for the CC2540 SoC to run the BLE stack. Different from operating systems in the traditional sense, OSAL is a task-based non-preemptive control loop triggered by task events. There can be up to 256 tasks, each of which has an 8-bit ID. Each task can supports up to 16 types of events, where a 16-bit event flag expresses the set of events to be handled. One special type of event supported by OSAL is the messaging event type for inter-task communication. Any task can emit and receive events. OSAL can dispatch the event immediately or schedule it after some timer delay. The rest of the events are free for each task to define to whatever it means. A task could communicate with each other either by either setting events or sending messages, where a message uses dynamically allocated memory for the data to send, while an event per se has no associated data. The meaning of other events are defined by the owner task. OSAL runs a task loop that dispatches tasks from the highest priority task down to the lowest and then sleeps, if enabled, when no more task needs to be dispatched. The OSAL mechanism is required by TI's BLE protocol stack and defines the overall firmware architecture. The stack implements the full BLE protocol [16], with which GAP-related settings and the GATT-based profile can be specified in the application layer.

7.1.3 BLE Device Emulator by MacBook

A MacBook Pro (15-inch, 2009 model) with Mac OS X v10.9 is used as the emulator of BLE device that is specifies to provide different functionalities. The emulator, which emulates the Peripheral role discussed in Section 2.1.1, is built with the CoreBluetooth framework, a BLE protocol implementation for Mac OS X by Apple. The framework follows an event-driven architecture and uses a delegate process to receive callbacks from all the received events. All types of events including the device's state change and the receipt of incoming access request are dispatched to the delegate to take the specified actions in their corresponding callback functions. Besides the emulator, the CoreBluetooth framework is also used to build the MacBook-based gateway in BlueRim which is discussed in the next section.

7.2 Gateway

The gateway program discussed in Section 4.3 works as the Central role discussed in Section 2.1.1 in BLE protocol. The gateway is a lightweight program which serves only for the application of security policy and the message exchange between the device side and the centralized components of BlueRim. The gateway is built on a a 2009 Model MacBook Pro with Mac OS X v10.9 with both BLE and Internet connectivity. The gateway program is written in Python with the support of the CoreBluetooth framework for Mac OS X. The PyObjC library is used to bridge between the Python and Objective-C programming languages due to the fact that CoreBluetooth framework only provide interface to Objective-C.



```
> check
{u'gateways': [{u'gateway_id': 21082,
               u'peripherals': [{u'id': 14474828687080862490L,
                                 u'isAuthorized': True,
                                 u'isSecured': True,
                                 u'profileHierarchy': {u'1800': {u'2A00': {u'descriptors': {},
                                                                           u'properties': [u'READ']},
                                                                 u'2A01': {u'descriptors': {},
                                                                           u'properties': [u'READ']}},
                                                       u'1801': {u'2A05': {u'descriptors': {},
                                                                           u'properties': [u'READ']}},
                                                       u'180A': {u'2A23': {u'descriptors': {u'2901': u'DeviceInformation'},
                                                                           u'properties': [u'READ']}},
                                                       u'7760': {u'7761': {u'descriptors': {u'2901': u'Authentication'},
                                                                           u'properties': [u'WRITE']}},
                                                       u'7770': {u'7771': {u'descriptors': {u'2901': u'AES_CFB: uint8, parameters: 1. secret key: unicode16, 2. IV: unicode16'},
                                                                           u'properties': [u'READ']},
                                                                 u'7772': {u'descriptors': {u'2901': u'secret key: unicode16'},
                                                                           u'properties': [u'READ',
                                                                                           u'WRITE WITHOUT RESPONSE']},
                                                                 u'7773': {u'descriptors': {u'2901': u'IV : unicode16'},
                                                                           u'properties': [u'READ',
                                                                                           u'WRITE WITHOUT RESPONSE']}},
                                                       u'7780': {u'7781': {u'descriptors': {u'2901': u'TestDescriptor',
```

Figure 7.3: BlueRim Interactive Shell Interface

7.3 Cloud-base Centralized Components

The centralized component discussed in Section 4.4 in BlueRim is programmed in Python and deployed in a privately maintained Openshift environment. Openshift is PaaS cloud solution powered by RedHat that supports the creation of an application deployment environment with a customized specification on runtime environment, database support, integration service, etc. The cloud-based implementation on the centralized components enables the computation and storage scalability of the knowledge base and the data store. In the knowledge base, KB records are stored in a MySQL relational database. Tornado web framework [2] is used to create the web APIs of every device capability profile is exposed in REST architectural style as discussed in Section 6.1.1. Queries and tasks can be made through a shell-based environment to access the devices under BlueRim. Fig. 7.3 provides the screenshot of the interactive shell environment, where commands can be issued to check the entire hierarchy information under BlueRim including gateways, connected peripherals and capability profiles of peripherals. In the data store, two URLs are generated for every entry that is used to store the result of a long-term task. One is to access the result stored in data entry. The other one is to receive push notifications sent to the observing process of the data entry from the subscribed device. The latter is implemented using the WebSocket protocol [3].

Chapter 8

Evaluation and Case Studies

This chapter presents the evaluation of BlueRim, the implementation of rimware specifically designed for BLE devices, and several case studies that BlueRim is used in practice.

8.1 Evaluation

BlueRim is evaluated in three aspects: code size, responsive time, and power consumption. These three factors are critical to the performance of a BLE device when it is working for any application domain(s). While BlueRim brings benefits to the application domains, we discover from the evaluation that the impact on the performance of a BLE device is infinitesimal and can be negligible. The next three sections discuss the evaluation on each of the aspects, respectively. Note that, the strength of the specified security policy, such as whether or not it is resistant to certain types of attacks, is not evaluated because the strength of the security policy depends on the cryptography used in the security policy.

Table 8.1: Code sizes of each component in BlueRim

Component	Functionality	Bytes
Device with the demo firmware	Security functionality	5406
	Privacy functionality	1596
	Overall	58038
Gateway	Adapter	17718
	Overall	62053
Centralized components	Overall	65011

8.1.1 Code Size

In BlueRim, only the code size of the firmware side are varied from one to another depending on the functionalities that the devices provide. The code size of the other components in BlueRim remain constant. The firmware side is evaluated against a demo EcoBT device that specifies the security functionality, privacy functionality, and a simple functionality that output a static value upon the request with read permission. The device capability profile model organizes the BLE's GATT-based profile using existing GATT profile hierarchy without adding additional logic. Therefore, except the security and privacy functionalities, it has no difference on the code size to the structure of the other regular functionalities from a device with a regular GATT-based application profile. In the device's security functionality, the AES symmetric block cipher encryption in mode CFB, as discussed in 5.4.1, is specified for the *adaptive application-level security enforcement* using a 16-byte shared key. The authentication mechanism based on randomly generated access token, as discussed in Section 5.4.2, is specified in the device's privacy functionality. The code size of each component in BlueRim including firmware, gateway, and centralized components is given in Table 8.1. Given that the BLE micro-controller (CC2540) has 256 KB of code flash and 8 KB of SRAM, the memory overhead is marginal in the firmware side. The gateway, which is evaluated against the Macbook-based gateway, is a lightweight program. Upon on the connection with every BLE device, an adapter with only a size of 17718 bytes is generated to bridge the communication between the device and the centralized components of BlueRim.

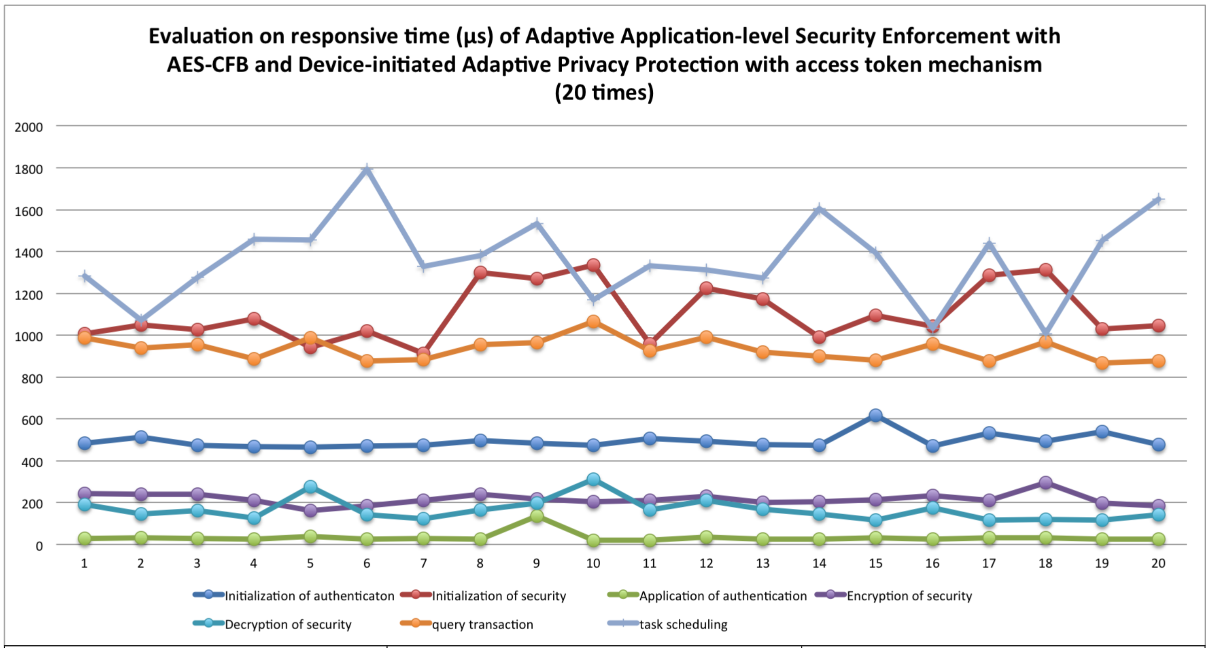
8.1.2 Responsive Time

BlueRim does not change the way that a BLE's GATT-based application profile is accessed. Therefore, it makes no difference between accessing a BlueRim-compatible device and accessing a regular BLE device on the responsive time, if the BlueRim-compatible does not specify the security and privacy functionalities. If the security and privacy functionalities are specified for the *adaptive application-level security enforcement* and *device-initiated adaptive privacy protection*, the impact on the responsive time that BlueRim brings to a BLE device depends on the initialization and the application of the preferred security policy and authentication mechanism that are specified in the security and privacy functionalities. The impact on responsive time is evaluated against the demo device used for the evaluation of code size in Section 8.1.1. Table 8.2 shows the average time on the initialization and the application of both *adaptive application-level security enforcement* and *device-initiated adaptive privacy protection*. Note that, the application of *device-initiated adaptive privacy protection* is a one time operation, whereas for *adaptive application-level security enforcement* the message encryption occurs on every outgoing message and the message decryption occurs on every incoming message. Although encryption and decryption are carried out on every transaction of the message exchange, the impact of which on the overall performance is negligible.

8.1.3 Power Consumption

Power consumption is the most important performance factor especially for BLE that is designed for wireless communication with low energy consumption. As mentioned in Section 8.1.2, BlueRim does not change the communication way between the BLE device and the other party. Therefore, there is no extra power consumption on a BlueRim-compatible device if the security and authentication functionalities are not specified. Similar to the impact on the responsive time, when these two functionalities are specified, the impact on power consumption occurs during the time of the initialization and the application of the preferred security policy and authentication mechanism

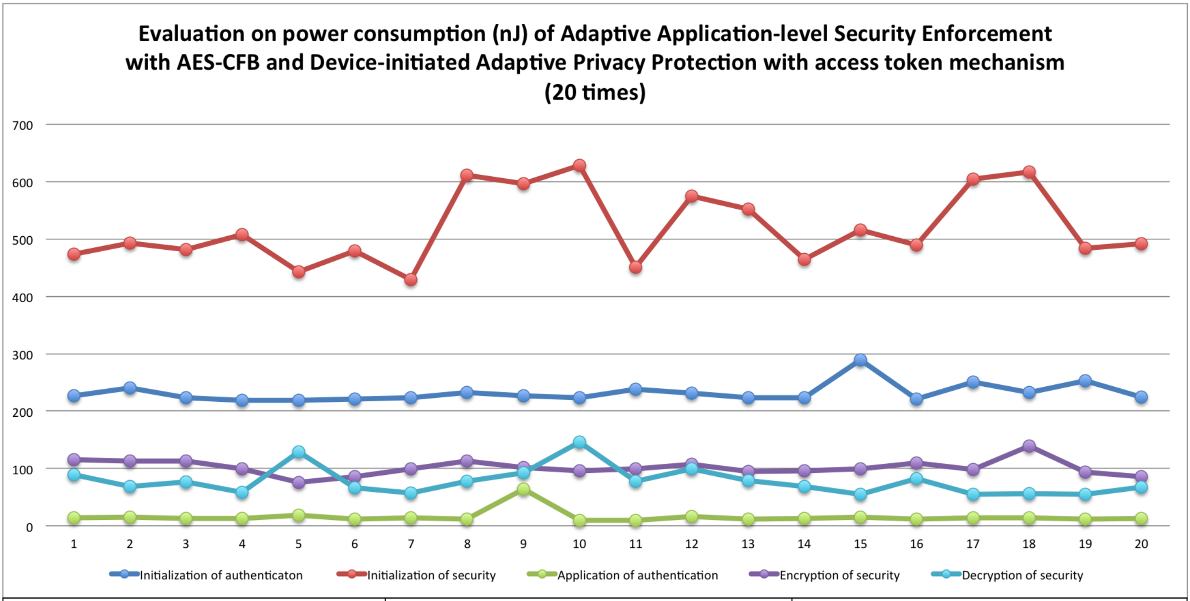
Table 8.2: Impact on responsive time from security and privacy functionalities in BlueRim for the demo device



Service	**Average of initialization** (µs)	**Average of application** (µs)	
		Encryption	**Decryption**
adaptive application-level security enforcement	1105	216	165
device-initiated adaptive privacy protection	494	33	

Type	**Response Time**
Query transaction	933
Task scheduling	1363

Table 8.3: Impact on power consumption from security and privacy functionalities in BlueRim for the demo device



Service	**Average of initialization**(nJ)	**Average of application** (nJ)	
		Encryption	**Decryption**
adaptive application-level security enforcement	519	102	78
device-initiated adaptive privacy protection	232	16	

that are specified in the security and privacy functionalities. Table 8.3 shows impact on average power consumption on the initialization and the application of *adaptive application-level security enforcement* and *device-initiated adaptive privacy protection*, which is negligible on the overall performance of a BLE device.

8.2 Case Studies

The case study of BlueRim is carried out by three different devices with the corresponding applications that are built for making use of functionalities on different types of BLE devices through BlueRim. The first one is the electrocardiography (ECG) collecting device. An ECG analysis application remotely collects an individual's ECG recordings from the ECG collecting device through

BlueRim, which demonstrates the BlueRim's ability of tasking on one functionality from one device and the task optimization. The second one is the infant monitoring device that monitors an infant's health state by the corresponding application through BlueRim, which demonstrates the task scheduling against different functionalities from one device by an application domain. The third one is the water pipe monitoring application that monitoring the vibrations of the water pipes at different locations, which demonstrates the utilization of functionalities from multiple devices

The next three sections discuss the three applications, respectively.

8.3 ECG Recorder

The ECG recorder is the EcoBT device discussed in Section 7.1.2 equipped with a ECG recording module. A ten-electrode strip which connects with the ECG recording module is attached to the human body with a proper placement to collect the ECG data. The ECG recording module is modeled as a functionality provided by the ECG recorder in the capability profile. The corresponding application takes control of the ECG recorder and utilizes the ECG recording functionality remotely. For each recording, a long term task is scheduled to ECG recorder through BlueRim by the ECG recording application and the result is retrieved from the data store in BlueRim. Fig. 8.1 shows a screenshot of ECG recording application that displays a 10-second ECG recording result.

8.4 Infant Monitor

The infant monitoring device is a EcoBT device equipped with multiple hardware modules for collecting different types of physical data, including ambient infant's body position with an accelerometer, CO_2 density with a CO_2 sensor, breathing sound with a microphone, body temperature with temperature sensor, from an infant. Each of the modules is modeled as one functionality

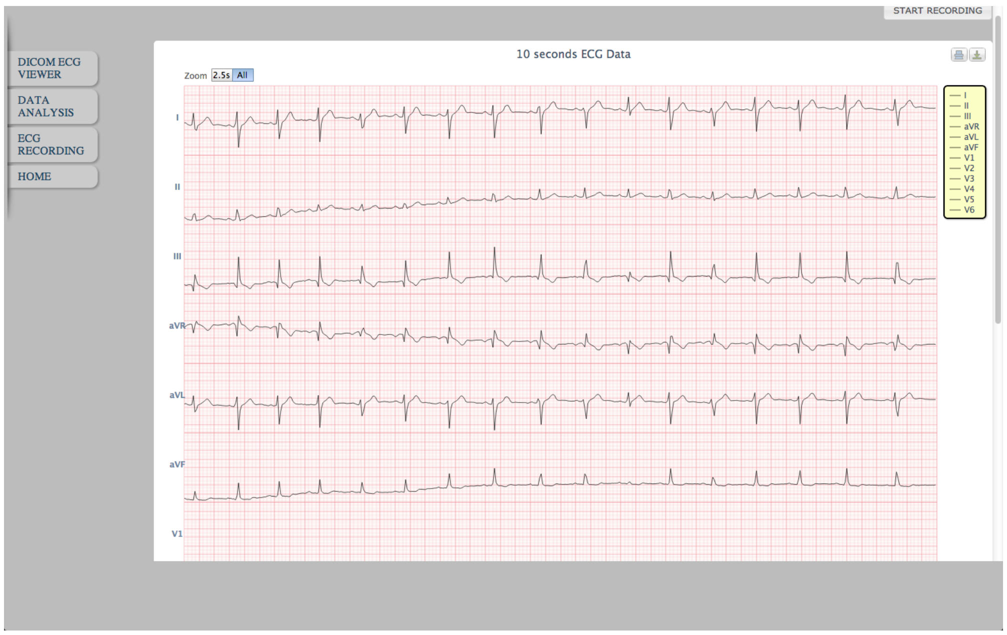


Figure 8.1: An screenshot of the ECG recorder applicaition

in the device's capability profile which can work independently from other functionalities. From the application side, for a real-time remote monitoring, a task is scheduled to subscribe the data from all the functionalities on the infant monitoring device. Fig. 8.2 shows a screenshot of the application that is proceeding the real-time monitoring. In addition, instead of monitoring the overall state of an infant, data collection task may be scheduled against any functionality individually to collect data for users with different interests on the infant. Tasks that are scheduled against the same device are taken care of by the task scheduler and optimization is proceeded if possible as discussed in Section 6.3.

8.5 Water Pipe Monitor

The water pipe monitoring device performs noninvasive monitoring on the exterior of the water pipes by measuring its vibration at a certain frequency. As shown in Fig. 8.3, several devices are deployed at different location of UCI campus. The application issues a task against the BlueRim to schedule data collection on all of the devices and retrieve the data according to the user specifi-

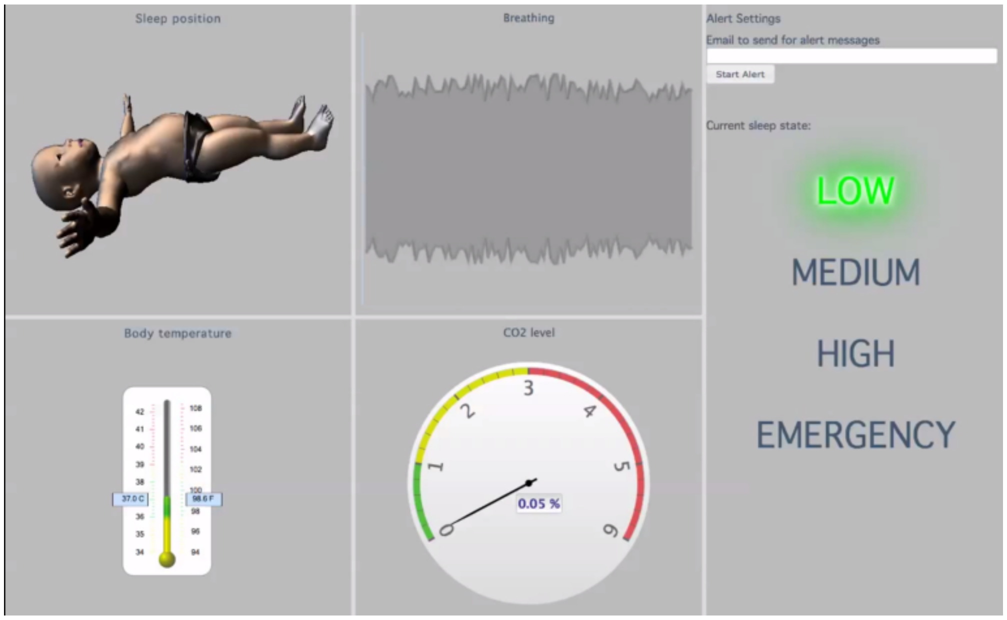


Figure 8.2: An screenshot of the infant monitoring applicaition

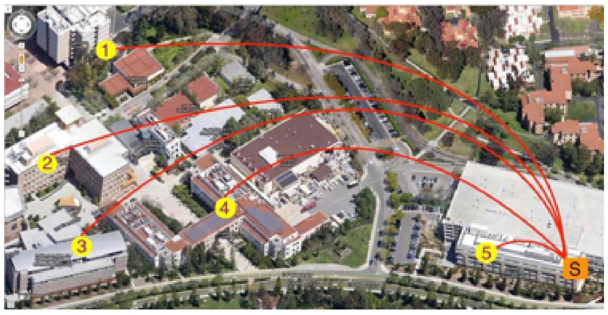


Figure 8.3: Locations of deployed devices

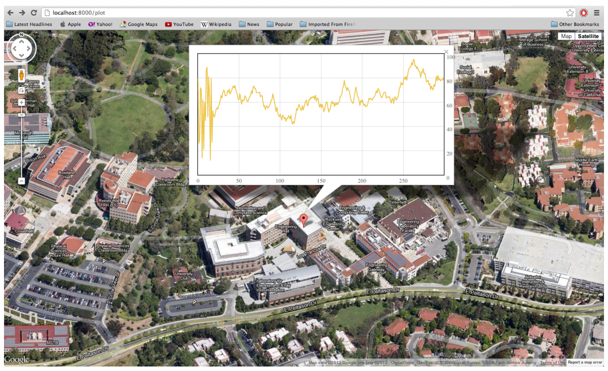


Figure 8.4: The collected result of a specific location

cation, for example, from a certain location as shown in Fig. 8.4.

Chapter 9

Conclusions and Future Work

We have introduced the concept of describing an NoT device's capability using a self-descriptive profile-based structure as well as exposing the device capability profile as the access interface. Together with the device capability profile, a centralized managed IoT middleware layer, rimware, is introduced for the integration between IoT and application domains to form a very powerful cyber-physical system. The *adaptive application-level security enforcement* and the *device-initiated adaptive privacy protection* techniques are introduced to ensure security and privacy for NoT devices in the highly mobilized rimware environment. We believe that the true power of IoT is the ability of allowing NoT devices to work across application domains. Rimware unleashes the potential of NoT devices by exposing the device capability profile as web APIs for M2M interactions. It also provides the formalized way to schedule tasks on devices, with which functionalities from multiple devices can be utilized by different application domains. The implementation of rimware, BlueRim, which is specifically designed for BLE devices, take advantages of BLE's very long battery life on the device side and the cloud functionality on the centralized side. The effectiveness of the fundamental features of rimware have been validated in several real-world applications with different access patterns while retaining their ability to consume very low power. We believe that our approach represents an important technology in taking IoT closer to realizing the full potentials.

Several future works needs to be done to perfect the role of rimware as the middleware of IoT including:

- Applying and experimenting the device capability profile concept on other NoT protocols, such as ZigBee and ANT+, by building a profile-based application layer in the firmware as well as deploying gateway program on the platforms with the connectivity of the corresponding protocols.
- Utilizing more on the descriptions of attributes in the device capability profile to encode more meta information, such as quality of service, to provide a more powerful device search and task scheduling.
- Enabling collaboration from multiple devices without involvement of the gateway and the centralized components of rimware using the self-descriptive device capability profile.
- Providing automatic recovery mechanism on the failure of tasks.
- Because of the lightweight characteristic of the gateway, a mobile version gateway may be developed and deployed on the smartphone device.

Bibliography

[1] Emebedded Platforms Lab. `http://epl.cs.nthu.edu.tw/`.

[2] Python Tornado Web Framework. `http://www.tornadoweb.org/`.

[3] RFC 6455: The WebSocket Protocol. `http://tools.ietf.org/html/rfc6455`.

[4] Semantic Web Services Language. `http://www.daml.org/services/swsl/`.

[5] Web Service Choreography Interface. `http://www.w3.org/TR/wsci/`.

[6] Web Service Semantics - WSDL-S. `http://www.w3.org/Submission/WSDL-S/`.

[7] Web Services Flow Language. `http://www.ebpml.org/wsfl.htm`.

[8] XLANG. `http://www.ebpml.org/xlang.htm`.

[9] ZigBee Document 053474r06, Version 1.0, ZigBee Specification. *ZigBee Alliance*, 2004.

[10] PKCS #1 v2.2: RSA Cryptography Standard. RSA Laboratories, 2012.

[11] P. Andreou, D. Zeinalipour-Yazti, M. Vassiliadou, P. Chrysanthis, and G. Samaras. Kspot: Effectively monitoring the k most important events in a wireless sensor network. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 1503–1506, March 2009.

[12] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services (BPEL4WS) 1.1, May 2003.

[13] S. Ashok and R. V. Krishnaiah. Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology. *International Journal*, 2013.

[14] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(2):1–5, Apr. 2002.

[15] T. Bleier, B. Bozic, R. Bumerl-Lexa, A. da Costa, and e. a. Costes, S. SANY: an open service architecture for sensor networks. *The SANY Consortium*, 2009.

[16] S. Bluetooth. Bluetooth: Bluetooth Core Specification v4.1. 3 December 2013.

[17] Bluetooth SIG. GATT Descriptors. `https://developer.bluetooth.org/gatt/descriptors/Pages/DescriptorsHomePage.aspx`.

[18] Bluetooth SIG. GATT Specification. `https://developer.bluetooth.org/TechnologyOverview/Pages/v4.aspx`.

[19] M. Botts, G. Percivall, C. Reed, and J. Davidson. OGC® Sensor Web Enablement: Overview and High Level Architecture. *GeoSensor Networks*, pages 175–190, 2008.

[20] W. E. Burr, D. F. Dodson, E. M. Newton, R. A. Perlner, W. T. Polk, S. Gupta, and E. A. Nabbus. *Electronic Authentication Guideline*. NIST Special Publication 800-63-1. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Dec. 2011.

[21] F. Callegati, W. Cerroni, and M. Ramilli. Man-in-the-Middle Attack to the HTTPS Protocol. *Security & Privacy, IEEE*, 7(1):78–81, Feb. 2009.

[22] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and dynamic service composition in eflow. In B. Wangler and L. Bergman, editors, *Advanced Information Systems Engineering*, volume 1789 of *Lecture Notes in Computer Science*, pages 13–31. Springer Berlin Heidelberg, 2000.

[23] F. Casati, M. Sayal, and M.-C. Shan. Developing e-services for composing e-services. In K. Dittrich, A. Geppert, and M. Norrie, editors, *Advanced Information Systems Engineering*, volume 2068 of *Lecture Notes in Computer Science*, pages 171–186. Springer Berlin Heidelberg, 2001.

[24] S. K. Datta and C. Bonnet. Smart M2M gateway based architecture for M2M device and Endpoint management. In *ITHINGS 2014, IEEE International Conference on Internet of Things 2014, September 1-3, 2014, Taipei, Taiwan*, Taipei, TAIWAN, PROVINCE OF CHINA, 09 2014.

[25] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. NIST Special Publication 800-38A. Jan. 2001.

[26] S. M. Fairgrieve, J. A. Makuch, and S. R. Falke. PULSENet^{TM}: an implementation of sensor web standards. *International Symposium on Collaborative Technologies and Systems, 2009 CTS'09*, pages 64–75, 2009.

[27] I. Galpin, C. Brenninkmeijer, F. Jabeen, A. Fernandes, and N. Paton. An architecture for query optimization in sensor networks. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1439–1441, April 2008.

[28] W. I. Grosky, A. Kansal, S. Nath, J. Liu, and F. Zhao. SenseWeb: An Infrastructure for Shared Sensing. *MultiMedia, IEEE*, 14(4):8–13, 2007.

[29] M. M. Hassan, B. Song, and E.-N. Huh. A framework of sensor-cloud integration opportunities and challenges. In *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*, ICUIMC '09, pages 618–626, New York, NY, USA, 2009. ACM.

[30] IEEE. Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Personal Area Networks (WPAN), 2003.

[31] S. Ikeda, P. C.-Y. Sheu, and J. P. Tsai. A Model for Object Relational OLAP. *International Journal on Artificial Intelligence Tools*, pages 551–595, 2010.

[32] W. Kurschl and W. Beer. Combining cloud computing and wireless sensor networks. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '09, pages 512–518, New York, NY, USA, 2009. ACM.

[33] T. K. Lai, A. Wang, C.-M. Chang, H.-M. Tseng, K. Huang, J.-P. Li, W.-C. Shih, and P. H. Chou. Demonstration Abstract: An 8×8 mm^2 Bluetooth Low Energy Motion-Sensing Wireless Sensor Platform. In *The 12th ACM/IEEE Conference on Information Processing in Sensor Networks, Demo Session, Berlin, April 2014.*

[34] K. Lee. IEEE 1451: A Standard in Support of Smart Transducer Networking. *Proceedings of the 17th IEEE*, 2:525–528, 2000.

[35] P. Levis and D. Culler. MatÉ: A tiny virtual machine for sensor networks. *SIGARCH Comput. Archit. News*, 30(5):85–95, Oct. 2002.

[36] L. S. Lifang Zhai, Chunyuan Li. Research on the Message-Oriented Middleware for Wireless Sensor Networks. *Journal of Computers*, 6(5), May 2011.

[37] J. Liu, E. Cheong, and F. Zhao. Semantics-based optimization across uncoordinated tasks in networked embedded systems. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 273–281. ACM Request Permissions, Sept. 2005.

[38] J. Liu and F. Zhao. Towards Semantic Services for Sensor-rich Information Systems. In *Broadband Networks, 2005. BroadNets 2005. 2nd International Conference on*, pages 967–974, 2005.

[39] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, Mar. 2005.

[40] D. Martin, M. Burstein, and G. Denker. OWL-S 1.2 draft release. `http://www.ai.sri.com/daml/services/owls/1.2/`, 2006.

[41] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, Mar. 2001.

[42] N. Mitton, S. Papavassiliou, A. Puliafito, and K. S. Trivedi. Combining Cloud and sensors in a smart city environment. *EURASIP Journal on Wireless Communications and Networking*, 2012(1):1–10, 2012.

[43] A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma. Meteor-s web service annotation framework. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 553–562, New York, NY, USA, 2004. ACM.

[44] V. Rajesh, J. M. Gnanasekar, R. S. Ponmagal, and P. Anbalagan. Integration of wireless sensor network with cloud. In *Recent Trends in Information, Telecommunication and Computing (ITC), 2010 International Conference on*, pages 321–323, 2010.

[45] N. Reijers, K.-J. Lin, Y.-C. Wang, C.-S. Shih, and J. Y. Hsu. Design of an intelligent middleware for flexible sensor configuration in m2m systems. SENSORNETS, 2013.

[46] M. Ryan. Bluetooth: With Low Energy Comes Low Security. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, pages 4–4, Berkeley, CA, USA, 2013. USENIX Association.

[47] H. Schuster, D. Georgakopoulos, A. Cichocki, and D. Baker. Modeling and composing service-based and reference process-based multi-enterprise processes. In B. Wangler and L. Bergman, editors, *Advanced Information Systems Engineering*, volume 1789 of *Lecture Notes in Computer Science*, pages 247–263. Springer Berlin Heidelberg, 2000.

[48] K. Shi, Z. Deng, and X. Qin. TinyMQ: A Content-based Publish/Subscribe Middleware for Wireless Sensor Networks. In *SENSORCOMM 2011, The Fifth International Conference on Sensor Technologies and Applications*, pages 12–17, 2011.

[49] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. Connect, learn, succeed. McGraw-Hill Education, 2010.

[50] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner. Mires: a publish/subscribe middleware for sensor networks. *Personal and Ubiquitous Computing*, 10(1), Dec. 2005.

[51] O. Vermesan, P. Friess, P. Guillemin, S. Gusmeroli, H. Sundmaeker, A. Bassi, I. S. Jubert, M. Mazura, M. Harrison, and M. Eisenhauer. Internet of Things Strategic Research Roadmap. *Chapter 2 in Internet of Things: Global Technological and Societal Trends*, pages 9–52, 2011.

[52] Villaverde, B C and Pesch, D and De Paz Alberola, R and Fedor, S and Boubekeur, M. Constrained Application Protocol for Low Power Embedded Networks: A Survey. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 702–707, 2012.

[53] J. Zheng, M. J. Lee, and M. Anshel. Toward Secure Low Rate Wireless Personal Area Networks. *Mobile Computing, IEEE Transactions on*, 5(10):1361–1373, Oct. 2006.