Master's Thesis

A Novel Service-Oriented Platform for the Internet of Things

서비스 기반의 사물인터넷 플랫폼

FEBURARY 2018

DEPARTMENT OF COMPUTER SCIENCE

COLLEGE OF ENGINEERING

SEOUL NATIONAL UNIVERSITY

Hyun Jae Lee

서비스 기반의 사물인터넷 플랫폼

A Novel Service-Oriented Platform for the Internet of Things

지도교수 하 순 회

이 논문을 공학석사학위논문으로 제출함

2017 년 11 월

서울대학교 대학원

공과대학 컴퓨터공학부

이 현 재

이현재의 석사학위논문을 인준함

2017 년 11 월

위 원 장 김 지 홍 (인)

부 위 원 장 하 순 회 (인)

위 원 유 승 주 (인)

Abstract

As Internet of Things (IoT) has received substantial attention in industry and academia recently, many IoT devices and IoT platforms have been proposed and being developed. In this paper we propose a novel IoT platform, called SoPIoT that is different from existent IoT platforms in several aspects. Since a device is abstracted with a set of services it provides, any computing resources can be easily integrated to the platform. In addition to the general use case of IoT where the smart devices provide useful services cooperatively and autonomously without the intervention of the user, SoPIoT allows a user to define a composite service dynamically at run-time by a script language program. To SoPIoT, the IoT system looks like a distributed system that consists of many computing resources, running multiple applications currently where an application corresponds to a composite service. The central middleware maps and schedules the services to the computing resources. The scalability of SoPIoT is achieved by forming the hierarchy of middlewares. The viability of the proposed IoT platform is confirmed by building a smart office test-bed. Experimental results show that a central middleware can support more than 1,000 devices.

Keywords : Internet of Things; Service Oriented Architecture; Hierarchical Middleware; Composite Service; Service Mapping and Scheduling

Student Number : 2016-21229

Contents

List of Figures

List of Tables

1. Introduction

Over the last few years, Internet of Things (IoT) has made significant progress and received substantial attention in industry and academia. As it has gained much attention, many IoT devices as well as IoT platforms have been developed [18, 20, 21, 26]. Early IoT devices introduced into the market such as Nest [17] and Amazon Alexa [1] have demonstrated how IoT can change our daily life; smart things sense the environment and react according to our intention but without our intervention [19]. The concept of IoT that is initially concerned about machine-to-machine (M2M) communications is recently extended to Internet of Everything (IoE) that interconnects not only physical devices but also data and processes. Artificial intelligence techniques based on the big data collected from the IoT devices will be used to orchestrate all devices adaptively to our needs and situation [7]. In the fourth industrial revolution era that puts its basis on IoE, it is anticipated that selling a software-centric service, not a hardware product will emerge as a new business model. This vision of service-based business model is known as Internet of Services (IoS).

While there is a consensus on how IoT will be evolved in the future in terms of functionality and applications, there is no agreement on the IoT platform to realize the future. In fact, there exist many IoT platforms proposed and being developed worldwide that have different features and limitations. They can be classified into two types in terms of communication structure between devices. Some platforms such as Iotivity [13] and AllJoyn [2] allow direct information exchange between devices, denoted as D2D (device-to-device) structure. Although the D2D structure has its advantages of easy deployment and scalability, it is hard to manage the IoT system as a

whole and vulnerable to a security attack. Another type of platforms such as Xively [24] and ThingSpeak [23] is web-based meaning that devices are connected to a web server that mediates the communication between devices. Since the web server can monitor and manage all devices and their communications, it allows us to apply state-of-the-art techniques for big data analysis and artificial intelligence to realize ambient intelligence. The web-based platform, however, requires the high overhead of supporting web protocol to the devices. In addition, since communication latency is likely to vary dynamically depending on the network condition, real-time performance cannot be guaranteed in general.

In this paper, we propose a novel IoT platform with a hierarchical communication structure. At the bottom layer, a local middleware mediates all communications between devices geographically close to the middleware. While it keeps the benefits of the web-based structure, it overcomes the drawbacks by using a lighter communication protocol, Message Queuing Telemetry Transport (MQTT), than HTTP and local network for communication between the devices and the middleware. Forming the hierarchy of the middlewares, it achieves the scalability of the system, which resembles the cellular network of a telecommunication system. For example, an IoT system for a smart building can be constructed with two levels of hierarchy. Each room becomes a smart office that consists of a local middleware and IoT devices attached to the middleware. The building has a central middleware that connects all local middlewares and will be connected to an upper layer middleware for external connection.

Another key feature of the proposed IoT platform is that an IoT device is abstracted as a set of services it provides. Thus the proposed IoT platform is named as SoPIoT

(Service-oriented Platform for IoT). By abstracting the devices with services, SoPIoT can easily support smart devices that are originally designed to support a different IoT platform, which is not true for the existent IoT platforms in general. In case there are multiple devices that provide an identical service, they are not distinguishable in terms of the functionality. If a user requests a service, the IoT system can use any device to serve the request. On the other hand, the middleware distinguishes them by their non-functional attributes such as energy, location, and cost and chooses the right device to serve, which raises a technical challenge how to map the requested services to the devices.

Note that any computing resource can be attached to SoPIoT if it is abstracted as a set of services it provides. A cloud that provides many different services is regarded as a virtual device in SoPIoT. Thus SoPIoT will become the IoE platform by connecting all computing resources that include not only smart devices but also clouds and all kinds of embedded systems once they are abstracted as services and the associated virtual devices. Since all devices, real or virtual, are regarded as computing resources that provide designated services, the whole IoT system becomes a huge scale distributed systems with heterogeneous processing elements, managed by the hierarchical middlewares.

In addition to the general use case where the smart devices provide useful services autonomously without the intervention of the user, SoPIoT supports another use case of an IoT system, which makes it distinguished from the existent IoT platforms. Programmability of integrated services is a critical issue when it comes to IoT framework that allows a user to define a sophisticated request dynamically to the IoT system [25]. We call this capability as programmability of the IoT platform. A user

request may include a set of services that can be conditionally triggered or repeated with a given termination condition. To support this capability, a script language is defined in SoPIoT. The language is regarded as a very high-level programming language that a casual user who is ignorant of computers may learn easily. To generalize this idea, we define a composite service that consists of a set of services, programmed by the script language. If a composite service is frequently asked, it can be defined in a library to be reused. A composite service can be defined statically when the IoT system is deployed or dynamically by the user at run-time. Each service a smart device provides is called an elementary service to differentiate it with composite services.

Since there may exist many composite services defined and running concurrently, mapping and scheduling of services to devices are necessary. It is a research issue to find a mapping and schedule to accommodate more composite services in a given IoT system, which remains as a future work. In this paper, we focus on the service-oriented architecture of the proposed IoT platform and the performance of the local middleware to examine how many devices and services can be handled.

Our key contributions can be summarized as follows.

- We present a novel service-oriented IoT platform, SoPIoT, where a device is abstracted with a set of services. Since service abstraction hides the hardware component that provides the services, any computing resources can be easily integrated into the platform to make IoE a reality.
- SoPIoT allows a user to define a composite service dynamically at run-time. To this end, a high-level script language is designed to easily express various service requests from users such as event-driven, periodic event, conditional statement, and

iteration. Such programmability of the IoT platform is a unique characteristic of SoPIoT to make the IoT system human-centric. In the future, we envision that a user can order a request verbally and the IoT platform serves the request if the verbal request can be translated into the script language automatically using advanced machine learning techniques.

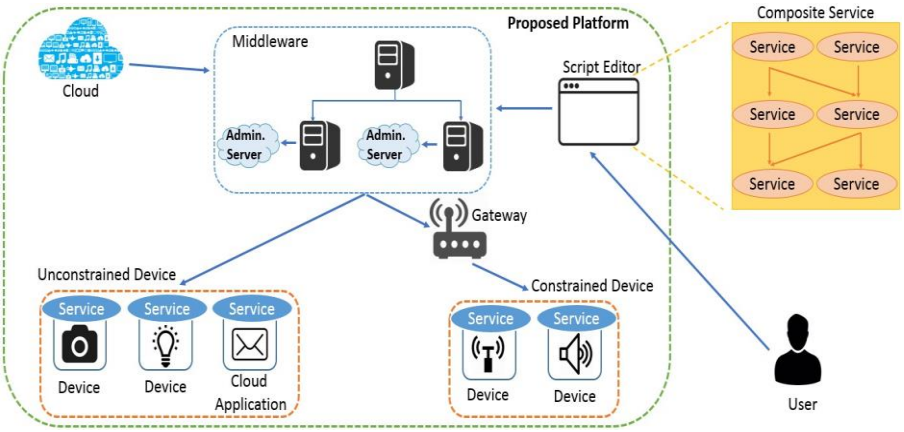
- In SoPIoT, a local middleware mediates all communications between devices, disallowing device-to-device communication in principle. Such a centralized scheme makes it easy to orchestrate the IoT system easily and intelligently. We believe that it also makes easy to make the IoT system secure by protecting the communication channel between the device and the middleware and the middleware itself. To prevent a centralized middleware from becoming the performance bottleneck as the system size grows, SoPIoT uses a middleware hierarchy to make the IoT platform scalable.
- SoPIoT views the IoT system as a kind of distributed systems that consists of computing devices with a restricted set of services and applications are represented as composite services. Then the role of middleware hierarchy is to map and schedule the applications to the computing devices. Since there can be timing constraints and resource constraints, real-time scheduling with resource constraints will be a challenging problem to solve in the future.
- We have built a testbed based on the proposed IoT platform with various kinds of devices. Experiments examine the capacity of the middleware in terms of the number of devices and the number of services it can accommodate. Constructing the middleware hierarchy is left as a future work.

The remainder of the paper is organized as follows. In the next section, the overall structure of the proposed IoT platform is overviewed. In section 3, the service

abstraction technique of devices, which is followed by section 4 that explains how to define a composite service with a newly proposed script language. The roles of a central middleware are explained in section 5 that include device management and service scheduling. After we present a testbed of the proposed IoT platform in section 6, experimental results on the performance and capacity of a central middleware are discussed in section 7. After section 8 reviews the related work on IoT platforms, section 9 concludes the paper.

2. Overall Structure of the Proposed IoT Platform

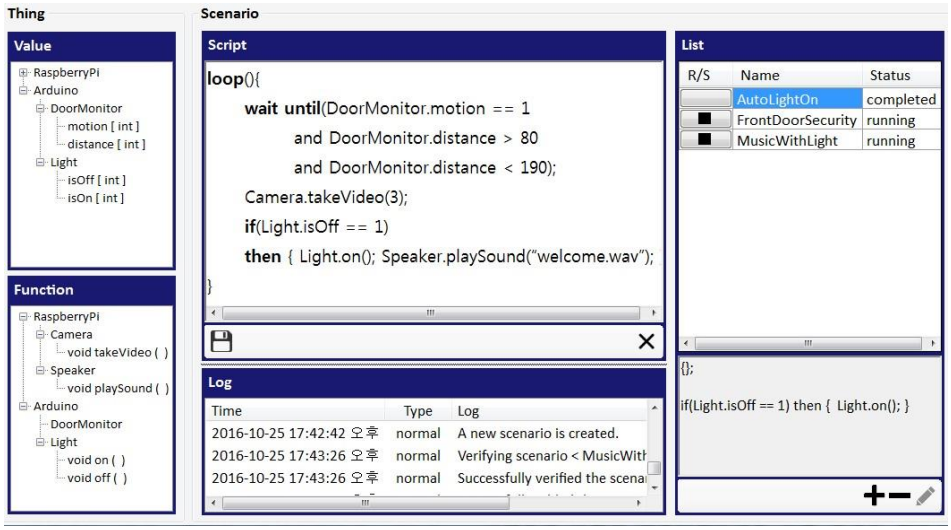
The overall structure of the proposed IoT platform is shown in Figure 1. A device is registered to the IoT platform with the services it provides, which characterizes the platform as service oriented. Sophisticated services can be created based only on its functionality of devices or cloud application [14]. Some devices, called unconstrained devices, are powerful enough to load an operating system and supports a network layer of TCP/IP stack for Wi-Fi connection. For those devices, MQTT is used in SoPIoT as an internet application protocol [16]. MQTT is an ISO standard publish-subscribe-based messaging protocol on top of the TCP/IP protocol. As shown in the figure, a cloud application can be registered as an unconstrained virtual device that provides a service.



[Figure 1] Overall structure of the SoPIoT platform

On the other hand, there will be numerous small devices such as sensors and transducers that have limited resources and weak computation power to support the

Wi-Fi protocol. They are usually battery-operated and use a lightweight communication protocol such as ZigBee, Bluetooth, and so on. Those constrained devices use Message Queuing Telemetry Transport for Sensor Networks (MQTT-SN) [11] which is a variation of the MQTT protocol aimed at embedded devices on non-TCP/IP networks. Constrained devices are connected to the middleware via a gateway that translates other protocols into MQTT.



[Figure 2] : A script editor implemented in SoPIoT

The IoT connects real world smart devices that embed intelligence in the system to process device specific information. It detects context based on that information and makes an autonomous decision [10] [19]. To support this feature, some existent IoT platforms allow the users to define a new service using the simple Event-Condition-Action (ECA) paradigm of programming where an action is triggered by a particular event that satisfies the given condition. It requires to recognize and control specific devices to receive events and request actions. On the other hand, in SoPIoT a user can

specify a composite service with a script programming language at a high level without the knowledge of the physical devices involved. Thus the proposed platform includes a client that provides a script editor a user can use. Figure 2 shows a captured screen of a script editor used in our experiments. The services and values that are registered to the platform are listed on the left side. The center window is used to write a script program to define a composite service with this information in a script language. Each registered composite service and its status are shown on the right side. The user can stop or restart a composite service just by changing its status. The script language itself will be explained in section 4.

At the center of the platform, there is a hierarchy of middlewares. A middleware at the bottom layer plays the role of the central manager of all devices, physical or virtual, registered to the IoT platform through the middleware. When a new device enters into the IoT system, it first finds a middleware, called home middleware, close to the device with the best network connectivity. The middleware maintains a database that stores the information of the registered devices and their services. It periodically checks the status of each device. As mentioned earlier, SoPIoT considers the IoT system as a distributed computing system that consists of varying number of heterogeneous computing devices that perform a predefined set of services. Services are the tasks that run on the distributed system. A composite service represented as a script program is translated to a service graph that defines the execution dependency of tasks. Since the same service can be served by multiple devices, it is necessary to map and schedule the services to the devices. Thus the middleware becomes the run-time system of the IoT platform. Note that a requested service may involve multiple devices that are distant geographically. Then multiple middlewares in the middleware hierarchy should

cooperate to serve the request. The details of middleware functionality are described in section 5.

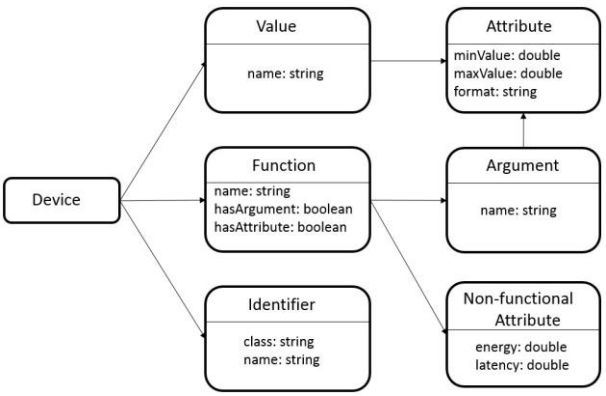
A middleware mediates all transactions between devices and sends them to the associated local administration server. The administration server has multiple purposes. First, it collects the big data from IoT devices, which can be analyzed to make the IoT system more intelligent. Second, it provides a graphical user interface to display the dynamic variation of values and services in the IoT system for monitoring and run-time management. Lastly, it also plays the role of the logging server.

The last component of the SoPIoT platform is the clouds computing environment that is accessed by the administration server or by the middleware. By letting all administration servers send IoT big data to a specific cloud, it will get benefits from big data analysis and advanced machine learning techniques performed in the cloud.

SoPIoT is designed to utilize the clouds applications as much as possible. As explained above, a cloud application can be registered to the platform as a form of the virtual device. In addition, there may exist a set of services that are hidden from the user but visible to the middleware only. For instance, we are planning to use a cloud application that recognizes the human voice into a textual form that will be translated into a script program that SoPIoT understands.

3. Service Abstraction of Devices

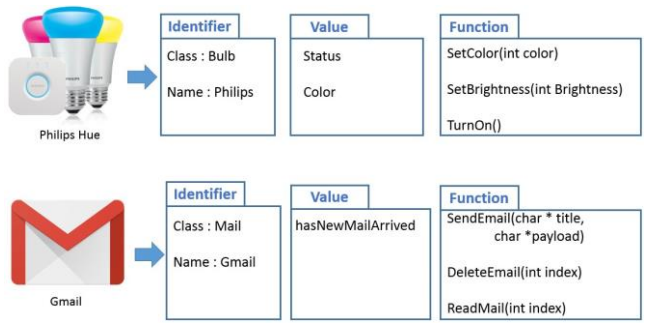
The IoT is an ecosystem where multiple entities participate to build a system and cooperate with each other. Integration of a new device to the system should be friction-free and seamlessly to enable interplay with existent entities immediately [5]. Otherwise, it will not be possible to add new devices at run-time once integration is done. In SoPIoT, a device is registered with its abstract model through a consistent interface mechanism between the device and the associated middleware, without affecting the other devices at run-time. Since the abstract model hides the hardware peculiarities of the device, it is easy to accommodate a variety of IoT devices, even third party devices that are originally developed for other IoT platforms.



[Figure 3] : The abstract model of a device in SoPIoT

The abstract model of a device consists of three parts as shown in Figure 3, identifier, values, and service functions. The identifier consists of its class and the name of the device. Values represent the values that will be reported to the middleware on request. A sensor data that a device reads from the attached sensor is a typical example. In fact,

the value part is syntactic sugar, replacing service functions to read the values. Functions represent the services that the device can provide. A function may have input arguments and a return value. Also, it may have optional non-functional attributes such as latency and energy consumption to perform the service. If they are given, they will be considered in the mapping and scheduling of services in the middleware. Note that numeric variables and function arguments are associated with an allowable range to check the validity of the information.



[Figure 4] : Two examples of device abstraction in SoPIoT

Based on abstract model, we have developed a library and a template that can be used to develop a device that conform to our abstract model. It is written in C++, which is known as a suitable language for embedded systems. In our library, it automatically connects to network, send ping message and sensor's value periodically. Furthermore, it handles actuation that is requested from middleware then return its result to notify whether actuation is done normally. Developing a device on top of the library can be done in two steps which is quite simple. First, developers specify property of device such as its value and function. All that is left is only to write callback functions of variable and function. Any third party vendors are able to develop their device easily

by provided library. They do not have to take care of network issues nor underlying complexity of devices.

Figure 4 shows two example devices that are abstracted with their identifiers, values, and functions. A specific commercial product, Philips Hue, is abstracted as a smart bulb that provides three functions, setColor, setBrightness, and turnOn, to set the color and brightness of the bulb, and to turn on and bulb, respectively. The second device is a virtual device that is associated with a cloud mail service, providing three functions to send, receive, and delete a mail.

A device may provide multiple services. If it can perform multiple services concurrently, it is divided into multiple virtual devices to make each virtual device serve one service at a time. It means that a device can perform a single function among the listed functions in its abstract model. It also implies that we associate a separate virtual device to each cloud application since cloud applications can be run concurrently in general.

```
1:  procedure RESOURCE MANAGER
2:      if broker address is unknown then
3:          perform auto discovery;
4:      initialize connection to broker;
5:      register device to middleware;
6:
7:      while termination request is not received do
8:          send alive message periodically;
9:          read value periodically;
10:         if value is valid then
11:             publish value to middleware;
12:
13:     unregister from middleware;
```

[Figure 5] : Template of resource manager

```
1: procedure FUNCTION MANAGER
2:     while termination request is not received do
3:         receive request from middleware;
4:
5:         if request is valid then
6:             execute requested function;
7:             publish result of function to middleware;
```

Figure 6: Template of function manager

[Figure 6] : Template of function manager

To be integrated into the SoPIoT platform, a device should install a device module for which the template and the common library are prepared in SoPIoT. It consists of two managers running concurrently, resource manager and function manager. Templates of the resource manager and the function manager are sketched in Figure 5 and Figure 6, respectively. At the initialization phase, the resource manager finds an MQTT broker through which it is connected to middleware. After connection to a broker is made, it performs registration with its abstract model. After initialization, it repeats the main loop until the termination condition is met, where it sends the alive message and the registered values periodically. Note that the value is published to the middleware only when it is valid. The function manager waits until it receives a service request from the middleware. After receiving a request, it executes the request function and publishes the result back to the middleware.

4. Composite Services

To support easy development of composite service by a user that has no knowledge on conventional computer programming, a high-level script language is devised. The syntax of the script language is presented in Figure 8. Basically, a composite service is defined as a list of services where a service is represented in the following form: class.function or name.function. There are three kinds of control constructs that can be expressed by the script language. A conditional execution of services is expressed by an if-else construct and an iterative execution of services by a loop construct. We repeat the loop execution repeatedly with a given period. Note that we can specify the loop termination condition. Moreover, we can block the loop execution until a certain condition is met, which is expressed by the wait until statement. The services are executed sequentially as listed in the script program. The middleware waits until it receives the result back from the device once it requests a service.

```
1  loop(Date.time > 07:00) {
2  A:      wait until(Camera.motion == 1);
3  B:      Camera.takePhoto();
4  C:      Vision.checkValidity(Camera.photo);
5
6  D:      if(Vision.isValid == 1)
7  E:              Bulb.turnOn();
8          else
9  F:              Speaker.alarm();
10 }
```

[Figure 7] : Code example of a composite service

Figure 7 shows an example composite service that could be defined when a user leaves a house. The composite service is repeatedly performed after 7 AM. It first requests a camera device to detect any motion. If it detects motion, the camera takes a picture of the moving object, probably a person. Then, the middleware sends the picture to a cloud to check if the person is included in the pre-verified group of people who has permission to enter. The cloud application to check the validity of the photo is abstracted with a virtual device, classified as Vision in this example. If the photo turns out to be valid, the middleware turns on the bulb. Otherwise, it requests the speaker to sound the alarm.

```
Statementlist  →  Statement
               |  Statementlist; Statement
Statement      →  if( Conditionlist ) Statement
               |  if( Conditionlist ) Statement else Statement
               |  loop( LoopCondition ) Statement
               |  wait until( LoopCondition ) Statement
               |  SERVICE
LoopCondition  →  /* void */
               |  PeriodTime , Conditionlist
               |  PeriodTime
               |  Conditionlist
Conditionlist  →  Condition
               |  not Condition
               |  Conditionlist or Conditionlist
               |  Conditionlist and Conditionlist
Condition      →  Expression operator Expression
Expression     →  SERVICE
               |  ℕ
               |  ℝ
               |  STRING_LITERAL
```

[Figure 8] : Syntax definition of the SoPIoT script language

The script language description of a composite service is translated into a service graph, which is defined as follows.

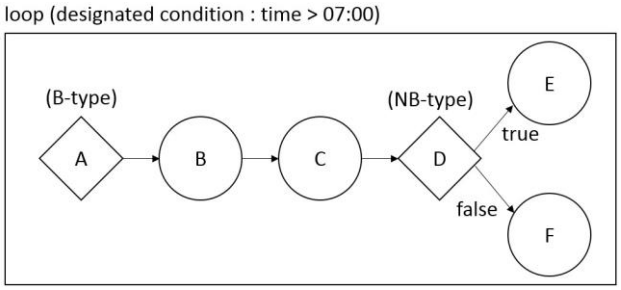
Definition 1 (Composite Service Graph): Composite Service graph G is defined as tuple (S,C,L,E) where S denotes a finite set of service nodes, C a finite set of condition nodes, L a finite set of loop nodes, and E a finite set of edges describing execution flow between nodes. A node $s \in S$ in the service graph represents an elementary service to be executed in a device.

Definition 2 (Condition Node): Condition node $c \in C$ is defined as a tuple (type, true_port, false_port) where true_port and false_port represent two output port to trigger the next node to execute according to the result of condition. The type of condition node can be either Blocking (B-type) or Non-blocking (NB-type). The B-type node blocks the execution of service graph until the specified condition is satisfied while the NB-type condition node continues execution without blocking. The false_port is omitted if the condition node is B-type.

The B-type condition node is associated with the wait until statement and the NB-type with the if-else statement. The condition node is executed in the middleware that reads the values in the condition statement from the corresponding devices and evaluates the condition.

Definition 3 (Loop Node): Loop node L is defined as a tuple (SG, period, designated_condition) where period represents the period of loop iteration and designated_condition represents the condition to remain in the loop. If the condition is not satisfied, loop is terminated. Loop node represents as a super node that contains a subgraph of G, SG. It should be noted that both period and designated_condition can be omitted. Loop node without both of them refers to an infinite loop that iterates

endlessly unless user stops the composite service. On the other hand, if there only exists period, it is an infinite loop that is periodically executed. Lastly, if there exists designated_condition only, it is similar to the conventional while construct that that iterates as long as the condition is met.



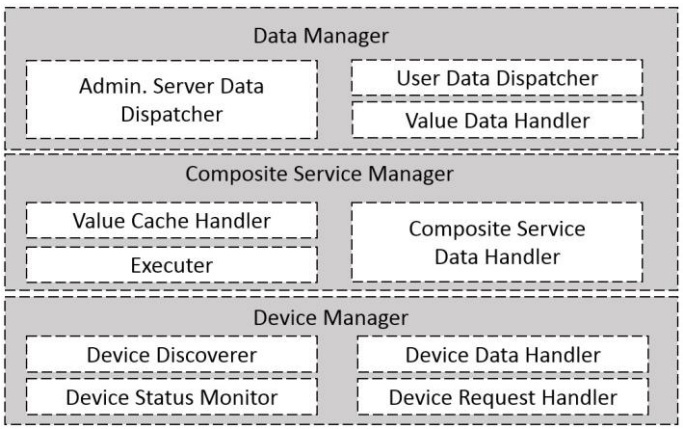
[Figure 9] : Service graph representation of the composite service in Figure 7

The composite service defined by a script program in Figure 7 is translated into a service graph shown in Figure 9. At the top level, it has a single loop node with a designated condition. The internal graph of the loop node consists of 4 service nodes and 2 condition nodes: the first condition node is B-type and the second condition node is NB-type.

5. Central Middleware

As the heart of the IoT platform, the middleware supports the following functional requirements: device management, interoperability, autonomous integration, programmability, data management, and scalability [3] [8]. The SoPIoT middleware supports these requirements by abstracting devices as services for the first three requirements, mapping service requirements to devices for programmability, and forming a hierarchical structure for scalability. Our middleware is written in C and runs on linux based systems which makes it possible to run on embedded devices such as Raspberry Pi or Odroid. It has been tested that our middleware runs well on Raspberry Pi 3 without any compatibility issues.

5.1. Structure of a Local Middleware



[Figure 10] : Component-level structure of a local middleware

We first explain the structure of a local middleware that manages the devices and services directly. A local middleware consists of three managers as shown in Figure 10: device manager, composite service manager, data manager. The figure shows the main handlers that compose each manager and their names imply what functions they perform.

• **Device Manager**

The device manager is in charge of device discovery to add any device to the system autonomously. For unconstrained devices that use TCP/IP layer, it uses zero-configuration networking (zeroconf) [27] that automatically connects a device to the system without any manual operator intervention or special configuration servers. On the other hand, auto-discovery of unconstrained devices is naturally accomplished by the MQTT-SN protocol in which the device searches a gateway via multicast communication that a light-weight protocol such as Zigbee, Bluetooth, and Z-wave usually supports. It continuously monitors the status of each device that changes dynamically. It makes the IoT platform open with the plug-and-play feature, meaning that it enables any device to join and leave the system seamlessly after the system is deployed. The device manager maintains the device database that stores the abstraction model of each device and the device status. It also serves the requests that a device issues.

• **Composite Service Manager**

The composite service manager is in charge of handling composite services that a user requests. A new composite service is requested with a script program; the composite service manager translates the script program into a service graph and checks if it is valid. How to schedule and execute the composite service will be

explained in the next subsection. It checks the status of the composite service. A user may run, pause, resume, or delete the service. In case a device involved in the service is disconnected at run-time, the manager detects the abnormal disconnection, disables the composite service, and then let the user know of the situation. It may restart the composite service automatically if the disabled device is re-connected. This way the middleware automatically reacts to the system status change, claiming it as self-adaptive. Each received value is stored in a value cache to be used when processing the service graph as an argument of a function. For example, if a user wants to send an email with an attachment of a taken photo, the photo has to be cached in the middleware.

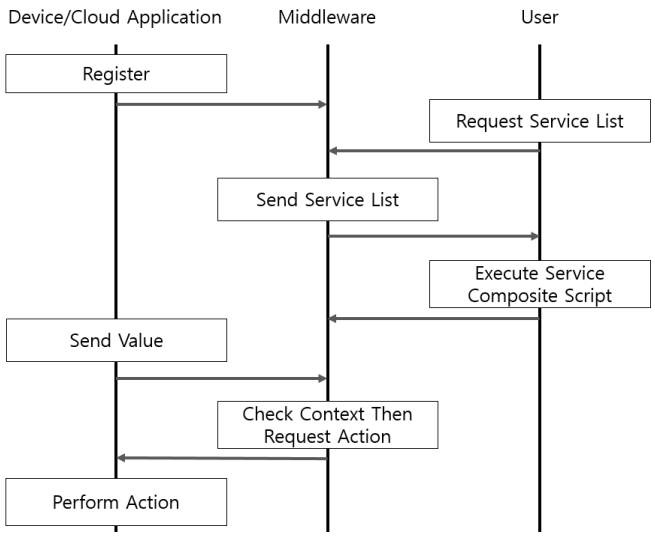
• **Data Manager**

The data manager manages the values and the current state of the middleware. When a user or a script editor of Figure 2 requests, it delivers the current system data, available services and devices. The data manager performs check-pointing of the system, in order to recover the system once an unexpected failure of the system occurs to due to diverse reasons. It also communicates with the local administration server. It sends all communicating data between devices to the local server and serves the request from the server for the IoT system monitoring and management.

5.2. Execution of Composite Service on Middleware

In SoPIoT, middleware is in charge of executing composite services those are requested by user in run-time. In addition, it is also responsible for providing programmable environment of distributed IoT system to user. Overall work flow of

developing composite service is shown in figure 11. First, device or cloud application abstracts its functionality into services then register to middleware. Abstraction model of each device is stored in database by device manager. Since user should know about current service list to write a composite service, script editor automatically requests of service lists as it is connected to middleware. User data dispatcher module in device manager sends current service list and composite service list as shown in figure 2. With these services, user can write a composite service with proposed script language. Written script is parsed by composite service manager into service graph as described in section 4. Each composite service is run as an independent thread in Linux to avoid latency issue when handled in a single thread.



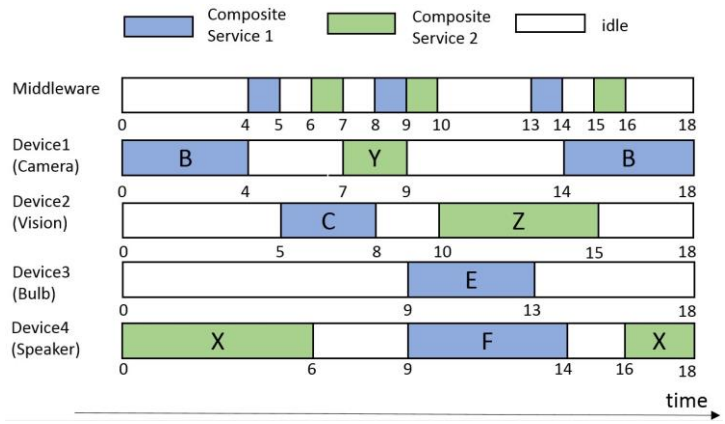
[Figure 11] : Flow chart of example composite service

In case of condition node, executer module in composite service reads value then check condition. If condition is met, it requests action to device. When handling blocking type of condition, executer module holds execution until the specified

condition is met. Value of device is read from value cache handler which saves value into cache when value is received from the device. In addition, when value of device is handled as an argument, it is also read from value cache handler then passed to device. Value cache can handle various type of data including binary data such as picture or music.

5.3. Service Mapping and Scheduling

The proposed IoT platform raises an interesting scheduling problem for the middleware. Since many composite services can be run in parallel and device status may change dynamically, it is necessary to map and schedule the elementary services to devices dynamically at run-time. Based on non-functional properties of the abstraction models of devices, the middleware selects the devices and schedules the elementary services to the selected devices.



[Figure 12] : A scheduling example of two composite services

Service Node	X	F,Z	B,E	C	Y	Middleware (Condition Node)
Execution Time	6	5	4	3	2	1

[Table 1] Execution time of nodes

The schedule can be represented as a Gantt chart where the horizontal axis represents time, and the vertical axis represents the devices and the middleware. Figure 12 shows a scheduling example of two composite services of which one is the service graph of Figure 9. Service graphs are distinguished by the color of the rectangles whose lengths mean the execution times of service nodes, which are given in Table 2. Note that a condition node is executed in the middleware and assumed to take 1 time unit. It is assumed that communication time between a device and the middleware is negligible compared with the execution time or included in the execution time.

Let us close look at the schedule of composite service 1 that corresponds to the example of Figure 9. To consider the worst-case schedule, two nodes E and F are all scheduled even though only one node will be executed. Even though there is a non-deterministic blocking delay due to the B-type condition node, A, it is assumed that there is no blocking delay to repeat the loop. Note that the earliest time the next invocation of service node B is 14 when node E is executed in the first iteration of the service graph. From the schedule, we can estimate the worst-case response time of the composite service and compute the utilization of each selected device.

Since the device status or the execution time of the service may change dynamically, the schedule should be updated dynamically at run-time accordingly. The middleware constructs a new schedule when a new composite service enters to check if the service

graph is schedulable or not. Suppose that composite service 2 enters and it requires the service of three devices that are currently serving composite service 2. Then, we overlay the schedule of composite service 2 with that of composite service 1 as shown in Figure 12. The schedule diagram can be understood as the resource reservation table. If there is no valid schedule for a new composite service, the middleware rejects the request and notifies the user.

There are several possible objectives of scheduling. One is to accommodate as many composite services as possible. Another is to lengthen the live times of composite services if some battery-operated devices have limited energy budget. In case there are multiple levels of service criticality, mixed criticality scheduling should be performed.

Thus a variety of scheduling problems can be induced from the middleware depending on the system characteristics. For a given set of requirements and assumptions, finding an optimal schedule is an open problem to research in the future. Theoretical analysis of schedulability is also left as a future research topic.

5.4. Hierarchical Structure of Middleware

Since the local middleware mediates all communications between devices and performs run-time scheduling of composite services, there will be a limit on the number of devices and the number of services it can accommodate. To make the IoT platform scalable, a hierarchical structure of middleware is proposed in SoPIoT. It is quite natural to build a hierarchy since a large-scale IoT system is likely to be partitioned into geographical regions.

The upper layer middleware handles all communication requirements between devices that are registered to different middlewares. If a local middleware finds that a new composite service requires a service that is not existent in the local region, it may ask the upper middleware to find a device in the other region. Hence the upper middleware should collect the device and service information from the local middlewares. The upper middleware monitors and manages the local middlewares. More researches should be performed on the management of hierarchical middlewares and scheduling over multiple middlewares in the future.

6. Test-bed: Smart Office

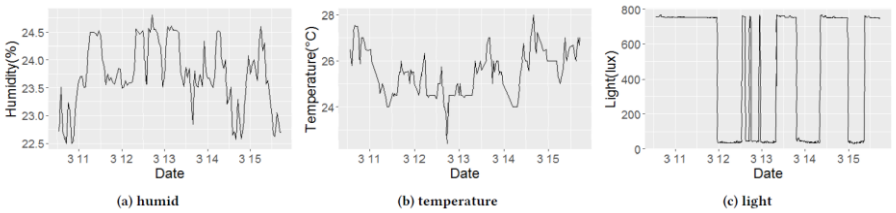
A test-bed of the SoPIoT platform has been built to investigate the viability of the proposed idea. In this section, we describe a real implementation of a smart office. For a proof-of-concept experiment, we have deployed three unconstrained devices with Raspberry Pi 3 and three constrained devices with Arduino which are both open source hardware platforms. A Gateway that translates the MQTT-SN protocol into the MQTT protocol is implemented with a Raspberry Pi 3 as well. In addition, we have integrated seven virtual devices that correspond to the following cloud applications: Weather, Gmail, Microsoft vision service, text-to-speech, date and time, a menu of the day, and SMS. Table 2 shows three representative examples of the deployed services, one in each type of device. Note that a device usually provides multiple services because a Raspberry Pi 3 or an Arduino board is equipped with more than one sensors and actuators.

Name	Device Type	Service Type	Service Name
Camera	Unconstrained Device	function	takePhoto
		function	takeVideo
		value	photo
		value	video
		value	motion
Monitor	Constrained Device	value	brightness
		value	dust

		value	humid
		value	temperature
MSVision	Cloud Application	function	evaluateEmotion
		function	getDescription
		value	emotion
		value	description

[Table 2] Representative examples of the deployed services

We have implemented several composite services that make the office life convenient and comfortable. Some examples are listed here. (1) When someone enters into the office, take a photo of the entering person with a camera located at the front door, evaluate his/her emotion by Microsoft Vision API, and play a consoling music if emotion turns out to be anger. (2) In case a meeting is called, turn on the monitor automatically, turn on the air conditioner if the temperature is higher than a given threshold, and record the meeting with a camera. If the meeting is over, finish recording and turn off the monitor. (3) Crawl the menu of the day from a website, convert it into an audio file by the text-to-speech conversion service of a cloud, and play the converted audio with a speaker at the designated lunch and dinner time. (4) Obtain the outside weather information every ten minutes since the office has no window. If it rains, send an email to students to notify of the weather condition. (5) If the motion is detected when all lights of the office are turned off after 9 PM, turn on the lamp and take a photo to send it with SMS and an E-mail.



[Figure 13]: A scheduling example of two composite services

Figure 13 shows the data on the humidity, temperature and light intensity, collected during 5 days from the deployed smart lab system. It is observed that humidity is high at daytime when students are present in the office since a humidifier is turned on. A similar pattern is observed with temperature as well. Lastly, light intensity shows a regular pattern that it reaches to saturation when the light is turned on and drops when the light is turned off. A slightly irregular pattern is observed when the light is turned off due to the light of the hallway. If we send those data to a cloud for big data analysis, we can devise a more intelligent control of the IoT system in the future

7. Experiments

Since the central middleware can be the bottleneck of the IoT platform, it is important to evaluate the performance of the local middleware. Experiments are conducted to examine how many devices can be handled by a local middleware.

7.1. Experiment Setup

Since there is no difference between a real device and a virtual device from the middleware point of view, we use a server that creates as many virtual devices as we want. We have two more servers, one for the middleware and the other for the local administration server where the MQTT broker is also run. Mosquitto [15] is used for the MQTT broker, which is an open source message broker that implement the MQTT protocol. Technical specification of each servers are as follows.

- Server 1(Middleware) : AMD Phenom(TM) II X6 1090T Processo 3.20GHz CPU, 16GB RAM, 512GB SSD and Ubuntu 16.04
- Server 2(Administration local server and broker) : Intel Core(TM) i7-3770 3.40GHz CPU, 16GB RAM, 256GB SSD and Ubuntu 14.04
- Server 3(Virtual things) : Intel(R) Core(TM) i7-2600 3.40GHz, 4GB RAM, 128GB SSD and Ubuntu 16.04

Two different kinds of databases are used. The middleware uses Sqlite that is server-less and can be easily embedded into the end program. We use an in-memory database for the temporary information that does not have to be preserved on a sudden system

failure. In the administration local server, SoPIoT uses PostgreSQL which is widely used as an object-relational database.

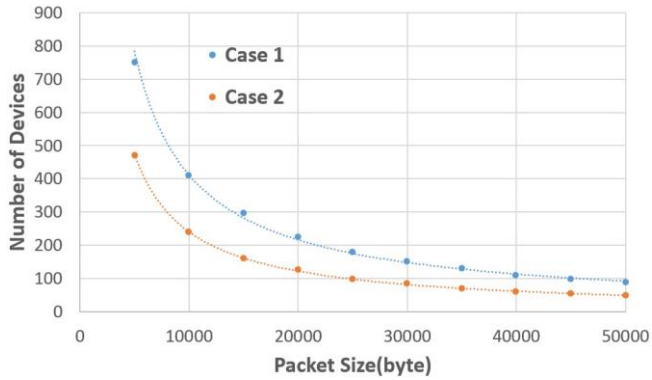
To assess exact performance of the platform, experiment has been conducted under following conditions.

1. Middleware and virtual things are connected through Wi-Fi to mimic IoT network environment where most of the things are connected to middleware through wireless connection. On the other hand, middleware and administration server are connected through wired connection which will be common use case.
2. Because of the device registering overhead, when measuring performance of middleware, it was accessed 10 seconds after virtual things register to middleware.
3. The performance of the systems are monitored for one minute then averaged.
4. Each virtual thing sends one value per second to middleware. We have used string value to vary the size of the value easily.

7.2. Results

The first set of experiments is made to examine how many devices a local middleware supports with the varying number of packet sizes when each device sends a packet every second. We vary the size of the packet from 5 Kbyte to 50 Kbyte. It is observed that the maximum number of devices that can be handled by middleware without saturation decreases as the size of packet increase in both cases, which is

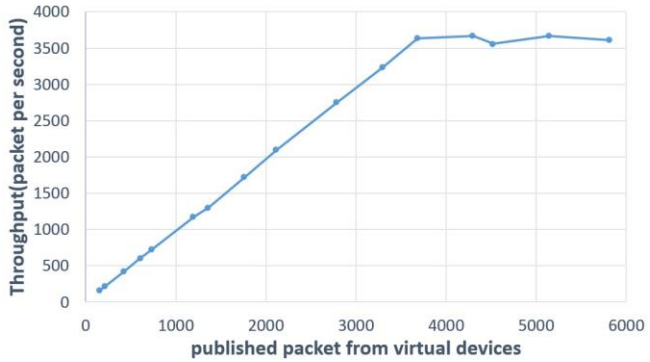
displayed in Figure 14. In fact, the number of devices is inversely proportional to the packet size. It is because that the network bandwidth becomes the bottleneck. To confirm this observation, we have conducted this experiment twice with different network conditions, case 1 and case 2. In case 1, where network capacity is relatively high, the middleware could support 410 devices when the packet size is 10 Kbytes and 225 devices when it is 20 Kbytes. The effective network bandwidth of case 1 is about 4.5MB/s. On the other hand, the effective network bandwidth of case 2 is about 2.5MB/s. Note that this experiment is designed to stress the network with heavy communication requirements. It is expected that the communication volume from a real device is usually less than 1 KB.



[Figure 14] : Experiment result of devices that can be handled by middleware with packet size

In the second set of experiment, we fix the packet size to 500 bytes and monitor the throughput of the middleware as the number of virtual devices is increased. By using the small packet size, we could prevent the network from becoming the performance bottleneck. As displayed in Figure 15, the throughput of middleware increases linearly

as the total number of published packets from the virtual devices increases. Eventually, the throughput becomes saturated when the total number of packets reaches at 3600. Through more detailed examination, it is found that Mosquitto [15], the MQTT broker, becomes the bottleneck.



[Figure 15] : Experiment result of middleware throughput with number of packets from virtual devices

These two sets of experiments reveal that the capacity of the local IoT system is determined by either the MQTT broker or the network bandwidth, not by the local middleware itself. The local middleware could support a few thousand devices if the average packet publication rate from each device is less than 2KB/sec, which will be the usual case in practice.

Even though we use a Wi-Fi network between computers, we could not emulate the network contention due to simultaneous publication from multiple devices. Since the packet issue rate will be infrequent, network contention will not degrade the performance significantly, we expect. In summary, we conclude that the local

middleware will be able to support at least 1,000 devices without being the performance bottleneck.

8. Related Works

While there have been numerous IoT platforms proposed and being developed, we select some representative IoT platforms to compare their characteristics with the proposed SoPIoT platform. Table 3 summarizes the comparison result in terms of programmability, openness to third party vendor, support for constrained devices that can not support TCP/IP layer, framework structure, and so on.

A popular approach is to make the IoT platform web-based, which Xively [24], IBM Watson[28], ThingWorx[29], Carriots [4] and ThingSpeak [23] belong to. A common drawback of the web-based platform is that they do not support constrained devices and require the significant overhead of HTTP protocol to the devices. They are distinguished by their services to the user. They provide different representational state transfer APIs (RESTful APIs) to the device developers. Xively, IBM Watson, Thingworx and ThingSpeak allow a user to define composite services by the ECA paradigm of programming. On the other hand, Carriots offers a software development kit (SDK) written in Groovy [6] to allow application developers to define composite services in addition to the ECA-style of programming. Since Groovy is a rich programming language, sophisticated composite services can be expressed in Carriots, including inter-operation between devices and cloud applications such as Dropbox and Twitter. Web-based platforms usually facilitate diverse data management services to display and analyze the data to user's request. Even though IFTTT [12] is a free web-based service, it can be used as a web-based IoT platform. It connects pre-existing IoT devices and cloud application such as Hue, Alexa, G-mail, etc. in a ready-to-use form. A user can define a composite service with an applet which is based on the ECA

programming paradigm. Unlike the other web-based platforms, it does not support data management services.

Framework	Program-mability	Open to third party	Support for constrained device	Framework Structure	Web-based	Data Management
Xively [24]	ECA-based	No	No	Centralized‘	Yes	Yes
ThingSpeak [23]	ECA-based	No	No	Centralized	Yes	Yes
IBM Watson [28]	ECA-based	Yes	Yes	Centralized	Yes	Yes
ThingWorx [29]	ECA-based	Yes	Yes	Centralized	Yes	Yes
Carriots [4]	Yes	No	No	Centralized	Yes	Yes
IFTTT [12]	ECA-based	No	No	Centralized	Yes	No
Iotivity [13]	No	Yes	Yes	D2D	No	Yes
AllJoyn [2]	No	Yes	Yes	D2D	No	Yes
ScriptIoT [9]	No	Yes	No	Centralized	No	No
SOCRADES [22]	ECA-based	No	No	Centralized	Yes	Yes
OpenHAB [30]	ECA-based	No	No	Centralized	No	Yes

SoPIoT	Yes	Yes	Yes	Centralized	No	Yes

[Table 3] Comparison of IoT Platforms

A group of IoT platforms allows Device-to-device (D2D) communication. Both IoTivity [13] and AllJoyn [2] are open source software frameworks that support device-to-device connectivity, discovery and data transmission. IoTivity is based on the resource based RESTful architecture where devices are abstracted as resources and resource management operations such as create, read, update, and delete (CRUD) are provided. AllJoyn [2] uses a client-server model to let compatible devices and applications find each other, communicate and collaborate across heterogeneous devices. Although D2D communication has its advantage of easy and serverless deployment, it is hard to control or monitor communications that are distributed. To be more specific, it is demanding to manage resources in the system and assure security without centralized middleware.

There also have been some efforts to develop service-oriented IoT platforms that support programmability as well. ScriptIoT [9] is a service-oriented script framework that offers both polling and event-driven mechanism for composite service specification. It provides executable client APIs based on a bash shell script which is frequently used among Unix system. However, their APIs support limited functions, i.e. fetch data, register script and request action.

Since ScriptIoT does not provide the status of the connected services, an application programmer has to find out service lists manually to program composite services. Service-Oriented Cross-layer Infrastructure for Distributed smart Embedded devices (SOCRADES) [22] aims at enabling automatic device integrations by dynamically

discovering and integrating a device as services. It allows development of composite services in a scenario-specific way. However, it does not support a development environment for devices nor support constrained device that is not able to have a TCP layer. The Open Home Automation Bus(OpenHAB)[30] aims at supporting various home automation technologies and devices. It allows development of composite services based on rules and scripts which are triggered by specific events. They are written in a script language with a Java-like syntax. Although it can connect pre-existing IoT devices in a ready-to-use form, it needs manual integration of services, which makes it difficult for third parties to participate in.

9. Conclusion and Future Work

In this paper a novel service-oriented IoT platform, called SoPIoT, is presented where each device is abstracted into a set of services that it provides. Thanks to the abstraction model, a cloud application can be integrated to the platform as a service that is provided by a virtual device. A key feature of SoPIoT lies in its programmability by defining a script language that is easy to run for a user that has no knowledge on computer programming. A user can request a composite service at run-time by writing a script program. Another key feature is to form a hierarchy of middlewares. At the bottom layer, a local middleware mediates all communications between devices, playing the role of the central manager of the attached devices. Through experiments, it is confirmed that the local middleware can handle more than a thousand of devices without being the performance bottleneck if the average packet publication rate of each device is below 2KB/sec. By forming the hierarchy of middlewares, the proposed IoT platform achieves scalability of the system which is a crucial requirement of IoT. The proposed platform has been demonstrated with a test-bed smart office in which several useful composite services have been developed.

SoPIoT views the IoT system as a kind of distributed systems that consists of computing devices with a restricted set of services and applications are represented as composite services. Then the role of middleware hierarchy is to map and schedule the applications to the computing devices under several constraints and requirements with respect to performance and resources. It remains as future work to find an optimal mapping and scheduling scheme of services to the devices at run-time. Expanding the

test-bed from smart office to smart building will be done in the future, which will reveal many challenges associated with the hierarchical management of resources.

References

[1] Alexa. 2017. http://alexa.amazon.com. (2017). Accessed: 2017-03-10.

[2] AllJoyn. 2017. https://allseenalliance.org/. (2017). Accessed: 2017-03-14.

[3] Soma Bandyopadhyay, Munmun Sengupta, Souvik Maiti, and Subhajit Dutta. 2011. Role of middleware for internet of things: A study. International Journal of Computer Science and Engineering Survey 2, 3 (2011), 94–105.

[4] Carriots. 2017. https://www.carriots.com/. (2017). Accessed: 2017-03-10.

[5] Dimitrios Georgakopoulos, Prem Prakash Jayaraman, Miranda Zhang, and Rajiv Ranjan. 2015. Discovery-Driven Service Oriented IoT Architecture. In Collaboration and Internet Computing (CIC), 2015 IEEE Conference on. IEEE, 142–149.

[6] Groovy. 2017. http://groovy-lang.org/. (2017). Accessed: 2017-03-10.

[7] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. Future generation computer systems 29, 7 (2013), 1645–1660.

[8] Sumi Helal. 2005. Programming pervasive spaces. IEEE Pervasive Computing 4, 1 (2005), 84–87.

[9] Han-Chuan Hsieh, Kai-Di Chang, Ling-Feng Wang, Jiann-Liang Chen, and HanChieh Chao. 2016. ScriptIoT: A Script Framework for and Internet-of-Things Applications. IEEE Internet of Things Journal 3, 4 (2016), 628–636.

[10] Yinghui Huang and Guanyu Li. 2010. Descriptive models for Internet of Things. In Intelligent Control and Information Processing (ICICIP), 2010 International Conference on. IEEE, 483–486.

[11] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. 2008. MQTT-SN publish/subscribe protocol for Wireless Sensor Networks. In Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on. IEEE, 791–798.

[12] IFTTT. 2017. https://ifttt.com/. (2017). Accessed: 2017-03-10.

[13] Iotivity. 2017. https://www.iotivity.org/. (2017). Accessed: 2017-03-14.

[14] François Jammes and Harm Smit. 2005. Service-oriented paradigms in industrial automation. IEEE Transactions on Industrial Informatics 1, 1 (2005), 62–70.

[15] Mosquitto. 2017. https://mosquitto.org/. (2017).

[16] MQTT. 2014. http://mqtt.org/. (2014). Accessed: 2017-03-10.

[17] Nest. 2017. https://nest.com/. (2017). Accessed: 2017-03-14.

[18] Charith Perera, Prem Prakash Jayaraman, Arkady Zaslavsky, Dimitrios Georgakopoulos, and Peter Christen. 2014. Mosden: An internet of things middleware for resource constrained mobile devices. In System Sciences (HICSS), 2014 47th Hawaii International Conference on. IEEE, 1053–1062.

[19] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. 2014. Context aware computing for the internet of things: A survey. IEEE Communications Surveys & Tutorials 16, 1 (2014), 414–454.

[20] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Clarke. 2016. Middleware for internet of things: a survey. IEEE Internet of Things Journal 3, 1 (2016), 70–95.

[21] Vasile-Marian Scuturici, Sabina Surdu, Yann Gripay, and Jean-Marc Petit. 2012. UbiWare: Web-based dynamic data & service management platform for AmI. In Proceedings of the Posters and Demo Track. ACM, 11.

[22] Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, Luciana Moreira Sá De Souza, and Vlad Trifa. 2009. SOA-based integration of the internet of things in enterprise services. In Web Services, 2009. ICWS 2009. IEEE International Conference on. IEEE, 968–975.

[23] ThingSpeak. 2017. https://thingspeak.com/. (2017). Accessed: 2017-03-10.

[24] Xively. 2017. https://www.xively.com/. (2017). Accessed: 2017-03-10.

[25] Yi Xu and Abdelsalam Helal. 2016. Scalable Cloud–Sensor Architecture for the Internet of Things. IEEE Internet of Things Journal 3, 3 (2016), 285–298.

[26] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. 2014. Internet of things for smart cities. IEEE Internet of Things journal 1, 1 (2014), 22–32.

[27] Zeroconf. 2004. http://www.zeroconf.org. (2004). Accessed: 2017-04-07.

[28] Watson. 2017. https://www.ibm.com/watson/. (2017). Accessed: 2017-08-19.

[29] Thingworx. 2017. https://www.thingworx.com/. (2017). Accessed: 2017-08-19.

[30] Florian Heimgaertner, Stefan Hettich, Oliver Kohlbacher, and Michael Menth. 2017. Scaling Home Automation to Public Buildings: A Distributed Multiuser Setup for OpenHAB2. (2017).