Digital Communications and Networks 10 (2024) 461–471

Contents lists available at ScienceDirect

Digital Communications and Networks

journal homepage: www.keaipublishing.com/dcan

Interworking between Modbus and internet of things platform for industrial services

Sherzod Elamanov [a,b], Hyeonseo Son [a], Bob Flynn [c], Seong Ki Yoo [d], Naqqash Dilshad [a], JaeSeung Song [a,*]

[a] Department of Convergence Engineering for Intelligent Drone, Sejong University, South Korea
[b] SyncTechno Inc, South Korea
[c] Exacta Global Smart Solutions, USA
[d] Coventry University, UK

ARTICLE INFO

Keywords:
Internet of things
Interoperability
Interworking
Modbus
oneM2M

ABSTRACT

In the era of rapid development of Internet of Things (IoT), numerous machine-to-machine technologies have been applied to the industrial domain. Due to the divergence of IoT solutions, the industry is faced with a need to apply various technologies for automation and control. This fact leads to a demand for an establishing interworking mechanism which would allow smooth interoperability between heterogeneous devices. One of the major protocols widely used today in industrial electronic devices is Modbus. However, data generated by Modbus devices cannot be understood by IoT applications using different protocols, so it should be applied in a couple with an IoT service layer platform. oneM2M, a global IoT standard, can play the role of interconnecting various protocols, as it provides flexible tools suitable for building an interworking framework for industrial services. Therefore, in this paper, we propose an interworking architecture between devices working on the Modbus protocol and an IoT platform implemented based on oneM2M standards. In the proposed architecture, we introduce the way to model Modbus data as oneM2M resources, rules to map them to each other, procedures required to establish interoperable communication, and optimization methods for this architecture. We analyze our solution and provide an evaluation by implementing it based on a solar power management use case. The results demonstrate that our model is feasible and can be applied to real case scenarios.

1. Introduction

With the growth of the Internet of Things (IoT), many organizations are working on the development of new and existing IoT standards, platforms, and communication protocols that would satisfy requirements of emerging Industry 4.0 [1]. The goal of Industry 4.0 is to reach higher levels of efficiency and productivity by applying digitalization and optimization of operational processes. Industry 4.0 presumes broad usage of digital technologies to support factory automation, real-time control and management [2–5]. Therefore, a decent digital infrastructure based on industrial IoT networks should be built. However, the range of existing IoT solutions is so large that the variety of IoT technologies leads to the divergence of the IoT market and increases the complexity of establishing interworking among them [6]. Moreover, the number of diverse connected devices is projected to increase, thus making it more complicated for an IoT service layer platform to establish interworking with them.

In addition, in recent years, various R&D studies have been conducted to provide more intelligent and secure IoT services through interworking with IoT platforms with new technologies such as blockchain, edge computing, and artificial intelligence. Some research results show that the performance and reliability of the IoT service can be improved through interworking between these technologies and the IoT platform, which indicates that the convergence of the IoT platform and interworking with other technologies is essential for the success of IoT technologies [7–10].

Devices in the industrial domain often operate on different protocols that are not supported by a conventional IoT system. One example of such protocols used to interact with connected devices is Modbus. Modbus has been widely used in industrial distributed applications [11]. The fact that

* Corresponding author.
E-mail addresses: elamanov@synctechno.com (S. Elamanov), hyeonseo0128@sju.ac.kr (H. Son), bob.flynn@exactagss.com (B. Flynn), ad3869@coventry.ac.uk (S.K. Yoo), dilshadnaqqash@sju.ac.kr (N. Dilshad), jssong@sejong.ac.kr (J. Song).

https://doi.org/10.1016/j.dcan.2022.09.013
Received 19 March 2021; Received in revised form 5 August 2022; Accepted 19 September 2022
Available online 6 October 2022

it is lightweight and simple makes it easy for industry organizations and device manufacturers to adopt. However, as the number of connected devices operating on Modbus and other protocols increases, there is a need among industry organizations to connect them within an IoT network. Therefore, there is a high need for integrating Modbus devices to IoT service platforms that can provide device management, network services, data storage, etc.

In this paper, we propose a solution for interoperability between electronic devices operating over Modbus protocol and an IoT service platform. More specifically, a Modbus device connected to an IoT service platform shall be available for reading data from it and writing data to it through the IoT platform without exposing internal Modbus protocol related information to the IoT application. This interworking stack is highly requested in numerous industrial applications where different devices and sensors should be connected to a single IoT network to provide real-time monitoring and management.

For our solution, we used a method in which interworking of Modbus devices with an IoT platform is established via abstracting the Modbus devices on the IoT service layer. We introduce an interworking proxy enitiy (IPE), which plays the role of the linking entity between Modbus devices and the IoT service layer. This architecture can be applied in industrial applications where user needs to have an access to Modbus devices via an IoT network.

To sum up, the main contributions of this research work are as follows:

- It introduces a standardized interworking mechanism using IPE based on an abstraction of Modbus devices and mapping them to an IoT service platform resources and interfaces. Our approach enhances the existing interworking mechanism in oneM2M global IoT standard specifications to allow it to support flexible protocol interworking: by modelling Modbus device and its data into a common device template in oneM2M, it is possible to expose Modbus device information to various IoT applications.
- It shows an implementation of our approach based on a solar power management system deployed in the university laboratory and an evaluation of this approach that demonstrates the feasibility of the interworking. The results showed that the proposed mechanism improved the performance of the retrieval time by 33% under the test conditions.

This paper discusses technologies that are going to consider for interworking in Section 2. Section III provides information on general interworking architecture and use case configuration. Section IV describes detailed interworking procedures to enable read and write function. In Section V, we describe the implementation of the proposed architecture and provide our evaluation results. Section VI concludes the papers and highlights the main points of our proposed architecture.

2. Background

In this section, the brief description of technologies we are going to use and motivation to use them for our approach will be provided. In particular, along with the Modbus protocol, we selected oneM2M for the IoT service platform and Smart Device Template (SDT) as a tool for device abstraction.

2.1. Modbus protocol

Modbus is a communications protocol derived from the Master/Slave architecture originally developed by Modicon (now Schneider Electric) [11]. Modbus is considered as the most commonly used protocol in industrial domain applications [12].

Modbus has several variants depending on communications medium. Modbus in Remote Terminal Unit (RTU) mode and Modbus over Transmission Control Protocol (TCP) are the most commonly used variants today. Modbus RTU is used to transfer data over wired serial interfaces such as RS-232 or RS-485, and Modbus TCP is designed for use in Ethernet and the Internet. In this paper, we consider the RTU variant as it is simple and can be generalized into the TCP variant as well.

In the Modbus protocol configuration, the Modbus Master typically is a software component running on a computer and serves as a host to access Modbus Slaves. Data exchange is always initiated by the Master, meaning only a Master can send requests and a slave responds to it. According to the Modbus protocol, one Master can handle up to 247 Slaves connected to it. The internal data structure of a Slave device is made of registers of different size and access rights. Table 1 shows a summary of register types used in Modbus.

The frame format is comprised of Slave ID (1 byte), function code (1 byte), data (n x 1 byte), and Cyclic redundancy check (CRC) (2 bytes). Slave ID is a unique numerical address that identifies a Slave device. Function code tells the Slave device what action should be performed, for example, read or write. Data field contains extra information depending on the function code. For example, for function code `03`, which is read multiple holding registers, the data field should contain the address of starting register and number of registers to be read. CRC is a checksum field used to validate the integrity of the request message.

2.2. OneM2M global IoT standards

OneM2M is a global initiative that develops standard specifications for a service layer platform to enable IoT in various domains such as industry, smart cities, and home automation. The reference architecture of oneM2M depicted in Fig. 1 is designed as a 3-layer model comprising application layer, common services layer, and network services layer [6, 13]. Each layer holds its own type of entities which are responsible to provide corresponding services [14]. For example, Application Entity (AE) is responsible for application logic, Common Service Entity (CSE) provides M2M services, and Network service Entity (NSE) establishes connectivity between underlying networks and CSEs.

In oneM2M the hierarchical structure of entities is realized through the use resources. Examples of common resources that are used in oneM2M are `CSE-Base`, `AE`, `FlexContainer`, `Subscription` and others. The CSE-Base represents a CSE and it is the parent node of all resources that reside in the CSE. The `AE` resource type represents AE entity. The `flexContainer` resource is a customizable container for storing information that allows the definition of a schema with custom-Attributes that are not opaque for the storage of custom data models. A `flexContainer` resource with defined resource schema is called a `flexContainer` resource specialization.

AEs can use services provided by CSE over `Mca` which supports Create, Read, Update, and Delete (CRUD) operations and subscription-based notifications. If AE subscribes to resource, it receives notifications from CSE whenever the resource is changed (e.g., update and delete operations)

The variety of resources in oneM2M and possible functionalities they provide allow us to build a flexible IoT platform that can support interworking with different technologies, including Modbus. Furthermore, as a horizontal service layer, oneM2M facilitates adding new interworking technologies to an IoT platform. All these properties of oneM2M are very important when it comes to building an IoT solution for industrial operations.

Table 1
Modbus register types.

Register name	Type	Size
Coil	Read-write	1 bit
Discrete input	Read-only	1 bit
Holding register	Read-write	16 bits
Input register	Read-only	16 bits

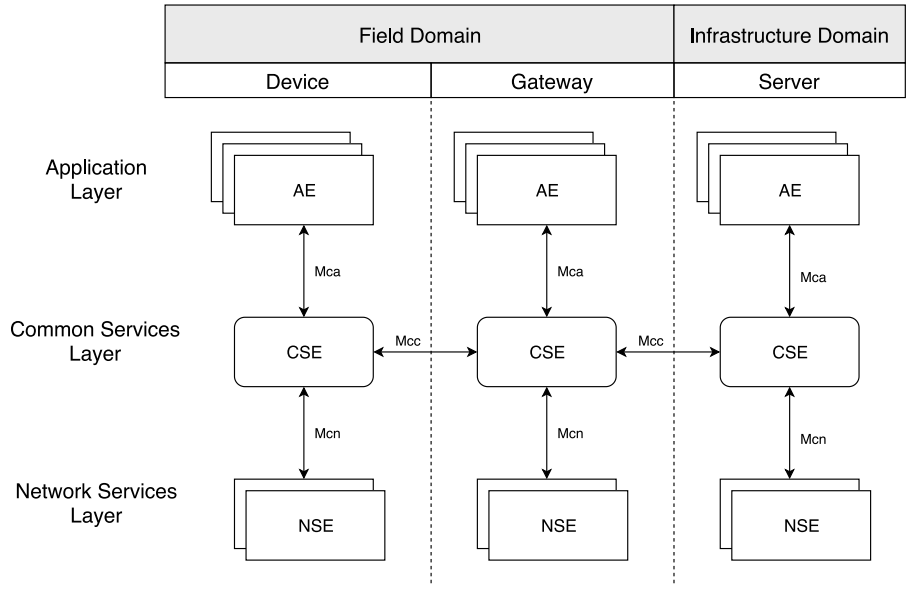

Fig. 1. OneM2M reference architecture.

2.3. Smart device template

The Smart Device Template (SDT) introduced by the Home Gateway Initiative is a modelling template that is intended to provide a standard way to design the capabilities of connected devices [15]. SDT helps to abstract functionalities of devices, such as data points, actions, and events, from the devices' underlying network technologies. By providing a convenient unified Application Programming Interface (API), SDT simplifies interworking between an IoT platform and heterogeneous protocols by using UML diagrams. Fig. 2 shows an overview of how SDT components relate to each other. The SDT architecture is based on separating device services into ModuleClasses, where each ModuleClass can have various DataPoints, Actions, Events, and Properties.

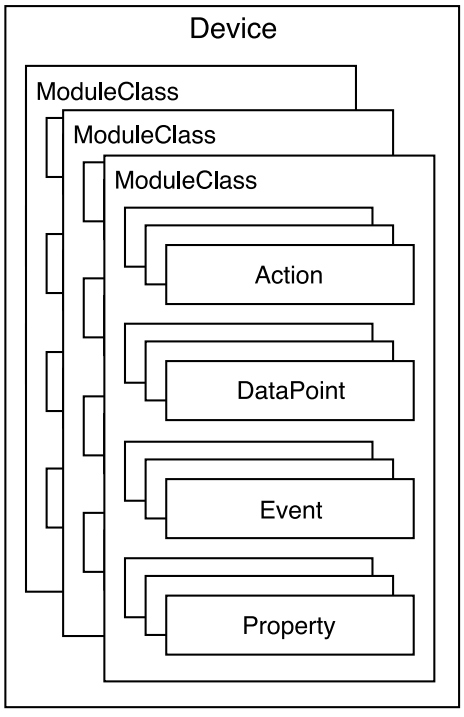

Fig. 2. Device abstraction using SDT components.

SDT, with its modular component structure, can be easily applied to describe Modbus devices and their services. This kind of modelling of Modbus devices helps to separate the definition of functional services, which are used to define the API and internal structure of an IoT platform, from the protocol-specific information (e.g., register types, addresses).

2.4. Motivation for interworking

Because of the rapid growth of the IoT market, there exist many different protocols and various types of connected devices. As a result of the evolution of technologies at some points, IoT systems can face a situation when existing technologies and devices become outdated and should be replaced by new ones, or a new feature based on a new technology need to be added. In such situation, it becomes complicated to manage an IoT system as it grows and needs to be scaled horizontally [14,16]. Moreover, typically, the development of an IoT application involves several stakeholders system, such as different organizations, device manufacturers, government; this increases integration complexity as all activities shall be controlled and coordinated among the stakeholders. Therefore, the cost and effort of adding new technologies to an existing system can be very high. For these reasons and in order to avoid the additional burden of developing interworking mechanisms to establish smooth interoperability between different platforms and protocols in the IoT, the proper integration procedure shall be researched and conformed across all stakeholders.

OneM2M has proved itself as a decent service layer IoT standard and is being used vastly in the industry domain [17] as well, whereas Modbus became a de facto standard protocol to communicate with industrial electronic devices [12]. Therefore, in order to integrate oneM2M applications and devices working on Modbus protocol, these two technologies need to be interoperable. For example, consider a dashboard application that accesses Modbus device data (e.g., temperature sensor data) through oneM2M service layer. The application is defined at a functional level using SDT, which models device functionalities (e.g., read data, turn on/off device) without regard for the technology of the devices. Initial deployment may use all Modbus devices or multiple technologies like ZigBee [18] and Modbus. The development of the dashboard application is not complicated by the need to communicate with each device technology. This allows the dashboard application to work only with functional information provided by the oneM2M service layer API, whereas technical aspects of device technologies are handled by the service layer

and the interworking mechanism.

The challenge of creating a communication flow between oneM2M and Modbus is a Modbus device does not provide any semantic information for its registers, and the same register address can be interpreted differently in different Modbus devices. For this reason, Modbus devices should be abstracted and mapped to oneM2M resources without exposing protocol details to oneM2M applications. In order to provide interworking, the so-called Interworking Proxy Entity (IPE) [19] can be used as a medium entity between oneM2M platform and Modbus devices.

In the following section, we explain the role of the IPE and how it is used in our architecture to achieve interworking between an IoT platform and Modbus devices.

3. Interworking architecture

In this section, we describe an overall interworking architecture design, including the key entities involved and communication mechanisms between them.

In our proposed architecture, we consider an industrial environment in which there is one Modbus master application and all Modbus devices communicate with it. This implies that all the communications between the devices and an IoT platform occur over a single gateway IPE. However, in the actual environment, there could exist several gateways composing a bigger IoT network, and the architecture can be scaled horizontally by adding additional IPEs. In the case of multiple IPE deployments, all the IPEs shall communicate with a single oneM2M IoT cloud platform, so the management of all connected devices remain user-centric.

Fig. 3 shows the high-level architecture for the interworking between oneM2M based platform and devices working through Modbus protocol. The architecture consists of Field and Infrastructure domain entities. The entities in the field domain are those that are deployed on-site and infrastructure domain entities are deployed in the cloud.

The MN-CSE is a Registrar CSE for the Modbus IPE and manages all the data related to connected Modbus devices and provides the connectivity with the Infrastructure domain entities. The Modbus IPE is positioned in the middle between the Modbus devices and the MN-CSE. It consists of three main parts: the Modbus master, the ADN-AE, and the Data Cache. The Modbus master provides an interface to access the Modbus devices (slaves according to master-slave model). The ADN-AE is a oneM2M entity that is used to communicate with the oneM2M system and manage the protocol translation mechanisms.The Data cache is an optional entity that is applied to speed up the communication between the Modbus devices and the oneM2M system. The Data Cache replicates the data of Modbus devices, so the oneM2M applications can fetch data from it in a shorter period of time.

In the infrastructure domain, the IN-CSE is a cloud-based oneM2M IoT platform which provides services for IoT applications to consume the data generated by the Modbus devices through `Mca`. `Mca` supports such protocols as HTTP and MQTT. In this paper, a dashboard web application is used as an application to interact with the IoT system. The dashboard application can support functionalities like visualizing the current state of devices, generating data, and sending some control commands to the devices as well.

The architecture supports two generic functionalities. First, monitoring of Modbus devices through a dashboard application. In this case, IPE reads the data from Modbus devices, uploads the data to the oneM2M server, and the data is displayed on the dashboard (AE) application. Second, writing data to a Modbus device from the dashboard (AE) application through IoT network. If this is the case, the IN-CSE is responsible for registering commands from the dashboard application and sending them to the IPE, whereas IPE performs the write requests to the device.

The authors of the paper have defined mapping rules for representing Modbus model as oneM2M resources and how to convert Modbus requests into oneM2M-compliant requests, and vice versa. Particularly, mapping rules define how to represent oneM2M resources based on Modbus register types and describe the exact steps of executing interworking procedures, including defining possible interworking scenarios, how to interpret incoming oneM2M and Modbus messages, and how to convert messages using mapping rules.

4. Interworking technologies

In this section, based on the high-level architecture and the use cases to be presented, we will describe detailed interworking mechanisms between oneM2M and devices communicating using the Modbus protocol.

4.1. Use case configuration

For the use case setup, we are going to consider a solar power management system. The purpose of this system is to monitor and manage solar power generation and consumption with the help of a solar charge controller device. Solar panels, a battery, and a load (a light bulb) are connected to the controller, which can manage the charging and discharging of the battery, and provide real-time data and some statistical information. The controller acts as a Modbus slave device and communicates with the Modbus master application running as a part of the IPE.

In this paper, we consider two scenarios to demonstrate interoperable communication:

1. The IPE continuously reads real-time data from the solar charge controller by sending Modbus messages, transforming the responses into oneM2M compliant resources, and uploading the data to the added MN-CSE. The MN-CSE then sends data to the dashboard application, where it is displayed to a user.
2. The dashboard ADN-AE application sends control commands to update some values in the solar charge controller. The MN-CSE sends

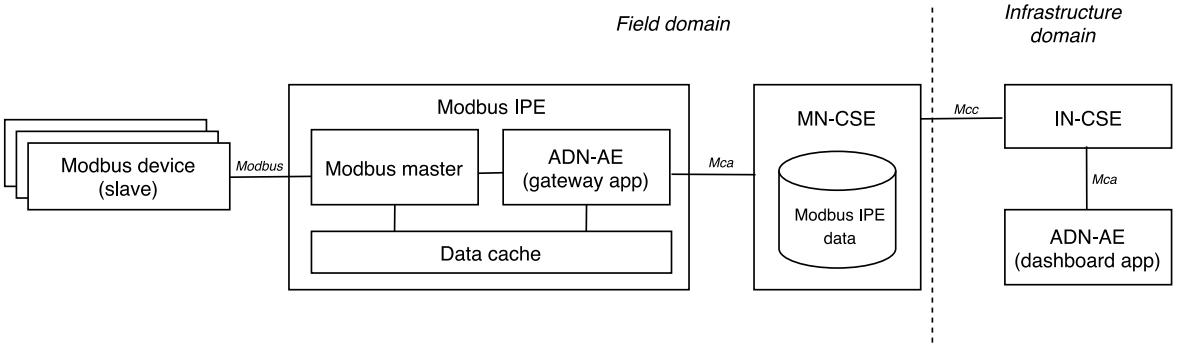

Fig. 3. High-level interworking architecture between oneM2M and Modbus.

this request to the IPE, which transforms the oneM2M message into a Modbus message and sends it to the device.

4.2. Mapping of modbus devices to oneM2M resources

As was mentioned in Section 2, SDT is used as an intermediary tool to map the Modbus data model to oneM2M resources. The mapping procedure consists of 2 steps: map the Modbus device data model to the SDT schema and then map the derived SDT schema to oneM2M resources. The second step, mapping SDT schema to oneM2M resources, is defined in the oneM2M specification [15]. This subsection gives details for the first step: explains how to derive SDT schemas from the Modbus device data model. We use a solar charge controller device from the use case to demonstrate how a Modbus device is mapped into SDT schemas.

First, SDT `ModuleClasses` are defined according to the existing capabilities of the device. By analyzing the solar charger controller device functionalities, the authors come up with the following `ModuleClasses` that best describe the capabilities of the device: `battery`, `powerGeneration`, `powerConsumtion`. The data variables in Modbus devices are stored in registers of different types and can occupy several registers depending on the variable data type. In the proposed mapping rule, the data variables in Modbus devices are mapped to SDT DataPoints, and the attributes of DataPoints (`DataType`, `writable` and `readable`) are assigned based on the register type and length (number of registers occupied). The rules for performing the mapping are shown in Table 2. For example, assuming the solar charge controller has "battery level" register, which is a holding register with a length of 2 that stores integer value. Then, this register is mapped into a DataPoint with `DataType=xs:integer`, `readable=True`, and `writable=False`. The SDT `read` Action is defined so that AEs could make explicit Read requests targeting either a Modbus device or the Data Cache.

Once the SDT schemas are composed, they are mapped into oneM2M resources. According to the oneM2M specifications [15], a `flexContainer` resource is used as a base resource for mapping the SDT schemas. Thus, according to the rules of mapping SDT schemas into oneM2M resources, ModuleClass component is mapped to the `flexContainer` resource specialization, and its DataPoints are mapped to the `flexContainer` customAttributes. Device components are mapped to a `flexContainer` resource specialization with its Module components mapped into child `flexContainer` resource specialization. An example of mapping of SDT Device and ModuleClass components is shown in Fig. 4 solarChargeController device is mapped to a `flexContainer` resource specialization named deviceSolarChargeController, and Modules are mapped to corresponding child `flexContainer` resource specializations.

4.3. dataCacheProperties and nodnProperties

In addition to the above described mapping, the resources representing device entities are extended with 2 additional customAttributes - `dataCacheProperties` and `nodnProperties`.

`DataCacheProperties` is a customAttribute added to the `flexContainer` resource specialization derived from SDT Device. This attribute is used by IoT application to configure the optional Data Cache unit, if present. `dataCacheProperties` can be used to configure the following Data Cache properties: status (on/off), refresh rate - how frequently IPE updates the Data Cache to replicate data from Modbus devices, monitoring attributes - the list of Data Points that need to be regularly updated.

`NodnProperties` (non-oneM2M Device Node Properties) customAttribute is added to the `flexContainer` resource specialization representing an SDT ModuleClass with the purpose of supporting Modbus and oneM2M mapping consistency. Since the data structure of the Data Cache is capable of keeping this information internally, `nodnProperties` is only applied when the Data Cache is not applied. The `nodnProperties` attribute is is used by the IPE to identify the mapping relationship between the Modbus registers supported by the device and oneM2M resources. The IPE sets the value of the `nodnProperties` attribute when the `flexContainer` resource is created. The `nodnProperties` stores one-to-one mapping in serialized string format (e.g., JSON) between each DataPoint and a Modbus register from which it is created. For Modbus interworking, the `nodnProperties` attribute contains slave id, register type, address, and length attributes for each DataPoint. For example, a valid content of the `nodnProperties` attribute for a Modbus battery is shown in Fig. 5.

4.4. Interworking proxy entity

The IPE is a key entity for keeping interoperability between oneM2M and Modbus. There are several oneM2M specifications describing interworking with IoT protocols [20, 21]. The responsibilities of the Modbus

Table 3
Register type to function code mapping for Modbus read requests.

Register type	Function code
Coil	0x01
Discrete Input	0x02
Holding register	0x03
Input register	0x04

Table 4
Register type and length to function code mapping for Modbus write requests.

Register type	Length >1	Function code
Coil	false	0x05
Coil	true	0x0F
Holding register	false	0x06
Holding register	true	0x10

Table 2
Mapping between Modbus register types and SDT Data points.

Modbus variable		Data Points		
Modbus register type	Length	DataType	Readable	Writable
Coil (1 bit, Read-Write)	1 (1 bit)	xs:boolean	True	True
Discrete Input (1 bit, Read-Only)	1 (1 bit)	xs:boolean	True	False
Holding Register (16 bit, Read-Write)	2 (4 bytes)	xs:integer/xs:float	True	True
Input Register (16 bit, Read-Only)	2 (4 bytes)	xs:integer/xs:float	True	False
Holding Register (16 bit, Read-Write)	1 (2 bytes)	xs:integer	True	True
Input Register (16 bit, Read-Only)	1 (2 bytes)	xs:integer	True	False
Holding Register (16 bit, Read-Write)	4 (8 bytes)	xs:double	True	True
Input Register (16 bit, Read-Only)	4 (8 bytes)	xs:double	True	False

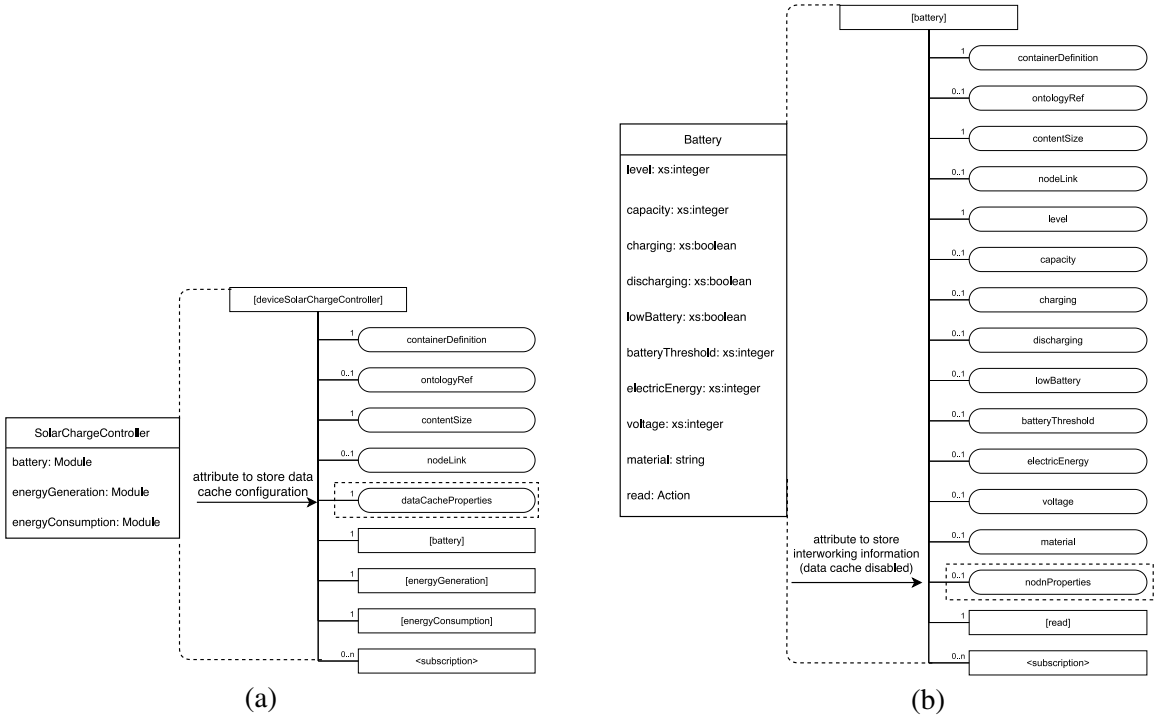

Fig. 4. (a) Mapping of the SDT solarChargeController device to oneM2M `flexContainer` resource specialization named deviceSolarChargeController (b) Mapping of the SDT Battery module to oneM2M `flexContainer` resource specialization named Battery.

IPE can be summarized as following:

- Device registration. The IPE discovers or is provisioned with Modbus devices and builds mapping schemas to transform Modbus devices' registers information to oneM2M resources and creates resources in CSE over `Mca`.
- Message translation. The IPE performs translation of Modbus messages to oneM2M messages and conversely from oneM2M to Modbus messages.
- Interacting with devices. The IPE sends read or write requests to the Modbus devices according to the action received from the dashboard application.

As mentioned in Section 3, the interworking model can use the optional Data Cache, which uses an IPE for setting interworking with Modbus devices. The Data Cache is an optional element that, if present, may be used to speed up the access to oneM2M resources by the Modbus IPE. The dynamic adding/deletion of oneM2M exposed resources must also be reflected in the Data cache and subsequently in Modbus devices. The entity-relationship diagram of the Data cache schema is shown in Fig. 6. The schema composed of 3 entities (Device, ModuleClass, and DataPoint) inherits its structure from the SDT components used in the mapping procedure. However, the attributes of entities in the DataCache are different from those of SDT entities. The attributes in the DataCache unify both Modbus data and oneM2M resources information to maintain resource mapping.

```
{"level": {
    "slaveID": 1,
    "registerType": "inputRegister",
    "address": "23",
    "length": 2
 },
 "capacity": {
    "slaveID": 1
    "registerType": "inputRegister",
    "address": "25",
    "length": 2
 },
 "charging": {...},
}
```

Fig. 5. Example content of `nodnProperties` in JSON format.

4.5. Requests optimization

In many cases, the IoT application may require reading multiple variables from a Modbus device. As a result, it may lead to a higher response time. For the faster data access to Modbus devices, the IPE implements an algorithm to make Modbus requests more efficient. The Modbus protocol supports the reading or writing of several registers if they are allocated in contiguous register memory. This protocol feature can be used to optimize the requests by grouping the Modbus registers with adjacent memory addresses into batches. Then, each batch of registers can be accessed in a single Modbus request. For example, IPE needs to update 4 Holding registers representing 4 different DataPoints with data addresses `0x0100`, `0x0101`, `0x0200`, `0x0201`. Then, the algorithm creates 2 groups (`0x0100`, `0x0101`) and (`0x0200`, `0x0201`) and assigns for each of them a function code `0x10` (update multiple Holding registers).

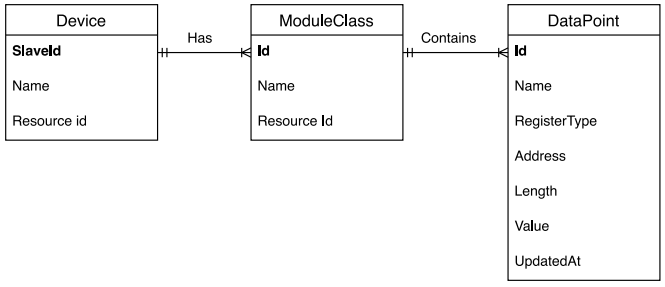

Fig. 6. Entity-relationship diagram of data cache.

Algorithm 1 Algorithm for creating Modbus execution plan

Input: *slaveId* - Modbus slave Id, *op* - operation (read or write), $DP_1 \ldots DP_N$ - list of data points to be read or written, *getFcode* - function to identify appropriate Modbus function code

Output: *execPlanList* - array of objects, where each object represents a separate request for a batch

Initialisation :
batchNumber← 0
execPlanList← []
N← *length*(*DP*)
Create batches for execution:
for *i* = 0 to *N* **do**
  if (*execPlanList*[*batchNumber*] = None) **then**
    execPlanList[*batchNumber*] ← (*slaveId*, DP_i.*address*,DP_i.*type*, *length* ← 0, *data* ← [])
  end if
  execPlanList[*batchNumber*].*length* += DP_i.*length*
  if *op* = "write" **then**
    execPlanList[*batchNumber*].*push*(DP_i.*value*)
  end if
  if $i \neq N-1$ **then**
    if $DP_i.address + DP_i.length \neq DP_{i+1}.address$ **then**
      batchNumber ← *batchNumber* + 1
    end if
  end if
end for
Assign function code for batches:
M← *length*(*execPlanList*)
for *i* = 0 to *M* **do**
  $execPlanList_i.address$←
  getFcode(*op*, $execPlanList_i.type$, $execPlanList_i.length$)
end for
for *i* = 0 to *M* **do**
  execPlanListi.fcode←
  getFcode(*op*, *execPlanListi.type*, *execPlanListilength*)
end for
return *execPlanList*

The steps to organizing the requested DataPoints into batches are shown in Algorithm 1. The algorithm outputs an execution plan of Modbus requests with assigned function code, starting address, and length based on the operation to be performed (read or write) and the list of involved DataPoints. Table 3 and Table 4 show the mapping rules applied by the algorithm to identify the appropriate function code for Read and Write operations respectively. It is assumed that the DataPoints are sorted by their address in increasing order when they are fed into the algorithm.

5. Interworking procedures

5.1. Continuous monitoring scenario

Consider a scenario where the dashboard application shows (near) real-time data from a Modbus device. The IPE needs to continuously monitor the device and upload the data from the device to the CSE hosting server. The dashboard application ADN-AE should be subscribed to the `flexContainer` device representation in the CSE to receive notification of changes to the device.

Let us consider that the CSE has a `flexContainer` resource registered for a ModuleClass of a Modbus device. The steps described in Fig. 7 show how to achieve interworking for monitoring of DataPoints of the `flexContainer` resource. For this scenario, we assume the ADN-AE has initially subscribed to the `flexContainer` resource. First, if the Data Cache is not intended to be applied, the Modbus IPE sends a retrieve `flexContainer` resource request to the Hosting CSE to get the `nodnProperties` attribute required for constructing the Modbus message. The CSE verifies the access privileges and responds to the retrieve request with the resource representation that includes `nodnProperties` attribute (Step 2). The Modbus IPE uses information stored in `nodnProperties` and constructs an execution plan for Modbus requests (Step 3). After the Modbus messages are constructed, the IPE sends these messages to the Modbus device (Step 4). If the messages are valid, the Modbus device responds with data for each received read request (Step 5). The Modbus IPE processes the Modbus response messages by mapping its content to a oneM2M message and sends a request to the CSE to update the `flexContainer` resource with the read latest values (Step 6). The CSE updates the `flexContainer` resource internally (Step 7) and responds to the IPE with a successful update message (Step 8). The update triggers a notification to the dashboard (ADN-AE) application (Step 9).

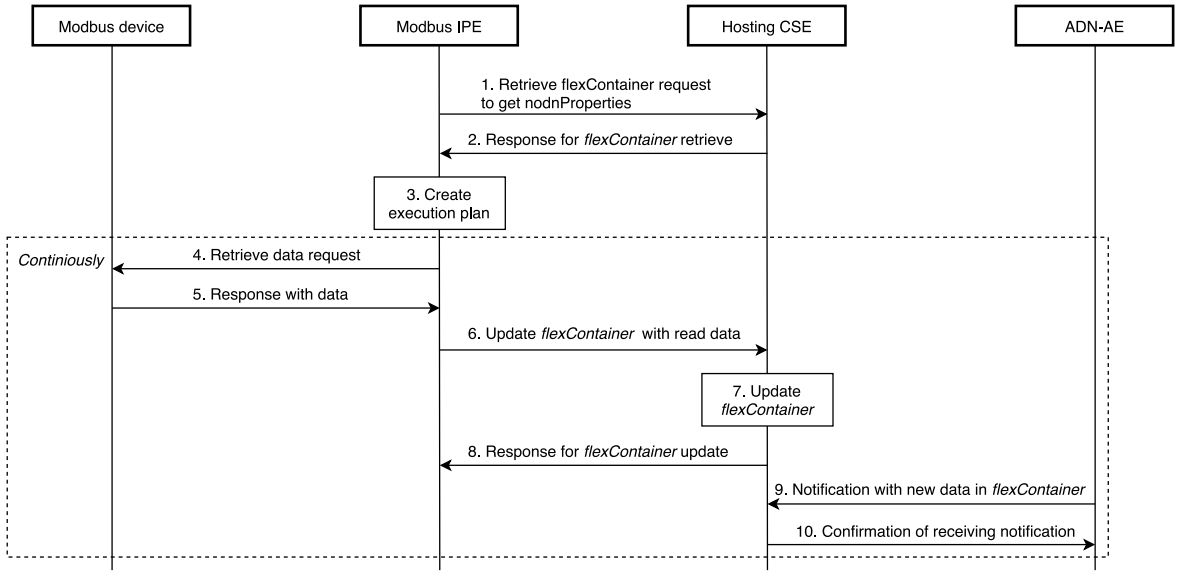

Fig. 7. Modbus device monitoring call flow.

Finally, the dashboard application updates its UI accordingly and responds with a confirmation message for receiving the notification (Step 10).

The above routine is very simple and efficient at the same time. As Modbus and oneM2M resources are related with one-to-one mapping, the translation from one to another can be performed quickly by the IPE with minimum computational resources spent, which is a very important factor for lightweight gateway setups.

5.2. Data writing scenario

Consider a scenario where it is required to write some data to a Modbus device through a dashboard (AE) application, for example, to trigger some actuator function or set some configuration parameter in the device. In this case, the Modbus IPE subscribes to the `flexContainer` resource by blocking update configuration. The blocking type of subscription ensures that the resources in the CSE and the corresponding data in Modbus device registers are synchronized. This prevents cases when the resources in the CSE are updated successfully, but there is an error updating a Modbus device, resulting in the data in two places being different.

The steps described in Fig. 8 show how to implement this scenario. First, the dashboard (AE) application sends a request to the CSE to update customAttribute(s) of the `flexContainer` resource, which corresponds to the register that should be updated in the device (e.g., batteryThreshold attribute of Battery). The hosting CSE sends a notification for the received update request to the Modbus IPE (notification includes `nodnProperties`) and temporarily blocks the updated `flexContainer` resource for any Update or Delete operations (Step 2). The Modbus IPE uses information stored in the `nodnProperties` attribute (if the Data Cache is not applied) or fetches the interworking information from the Data Cache to construct the execution plan from Modbus Write requests (Step 3). The Modus IPE sends the message(s) to the Modbus device (Step 4). The Modbus device updates the register(s) in the request(s), actuates according to the new value of the register, and responds with the written data to the Modbus IPE (Step 5). If the IPE supports the Data Cache, it is updated according to the updates made in the Modbus device to keep data consistency (Step 6). If there was no error, the IPE responds to the notification request from the hosting CSE with a successful update message (Step 7). If the device was updated successfully, the hosting CSE updates the `flexContainer` resource internally, otherwise discards the changes. The resource is unlocked for Update operations (Step 8). The hosting CSE responds to the dashboard AE application with the result of the Update request (Step 9).

5.3. Data cache access

When an IoT application does not require a continuous monitoring functionality but needs to make occasional Read requests to retrieve current data from Modbus devices, it shall perform the procedures defined in Fig. 9. For this procedure, ADN-AE - for ModuleClass must be subscribed for `flexContainer` resource and the Modbus IPE - for the child `read` resource. In this procedure, the Modbus IPE can use the Data Cache for faster data access. The IPE periodically reads data from the device to keep the Data Cache synchronized with the device data. The Data Cache refresh rate and the DataPoints monitored can be configured through the `dataCacheProperties` attribute of Device `flexContainer`.

The first step is to send a request by Updating (Read) resource and indicating the desired DataPoints in the body of the request. The Hosting CSE forwards this request through a notification to the Modbus IPE (Step 3). The IPE retrieves the request from its Data Cache (Step 5) and updates the `flexContainer` resource for ModuleClass (Step 7). The Hosting CSE updates the `flexContainer` resource and sends a notification to the ADN-AE with the requested DataPoints (Step 9). The ADN-AE sends a response message to the CSE to confirm receiving the notification (Step 10).

6. Implementation and evaluation

In this section, we are going to evaluate and verify the feasibility of the proposed architecture by implementing and testing it based on the use case presented in the previous section.

6.1. Experiments

Table 5 shows the summary of hardware specifications used in the solar power management system for demonstrating interworking. oneM2M platform was executed on a computer with an Intel Core-i7-8700 3.20 GHz processor, 16-GB of RAM, and a 64 bit Ubuntu 18.04 operating system. As an implementation of CSE, we used OM2M product, an open-source project developed by the Eclipse foundation [22]. CSE is configured to run over HTTP for CRUD requests and MQTT for notifications.

For the solar charge controller, we used an EPSolar VS4548BN model, which provides a Modbus RTU interface. IPE is executed on Raspberry Pi 3, which acts as a gateway for Modbus devices to be available for the IoT platform. As a power input, we used a solar panel with maximum power of 12W. The solar charge controller is connected to the Raspberry Pi 3

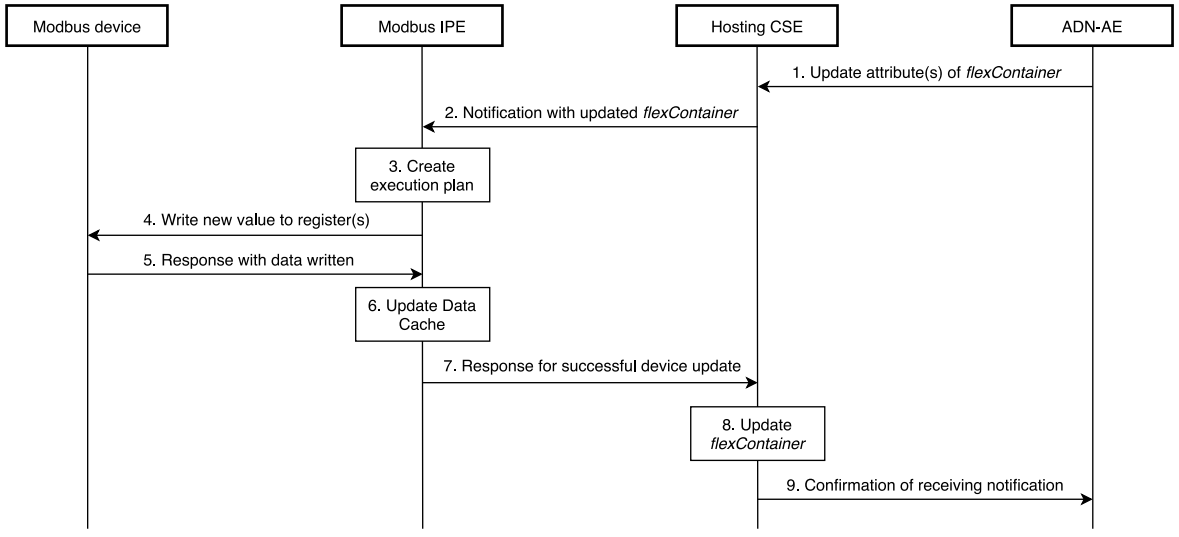

Fig. 8. Writing to the Modbus device call flow.

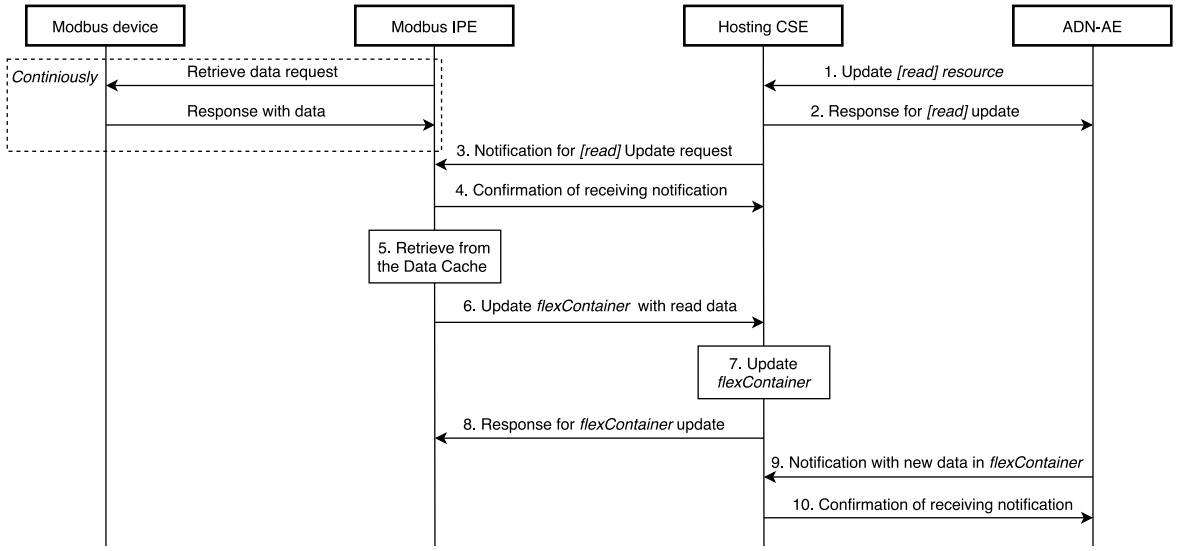

Fig. 9. Reading from the IPE Data Cache associated with a Modbus device.

Table 5
Use case configuration list.

EPSolar VS4548BN for solar charge controller	
Max input voltage	30 V
Rated battery current	45 A
Raspberry Pi 3 for IPE gateway	
CPU	4x ARM Cortex-A53, 1.2 GHz
RAM	1 GB LPDDR2
OS	Raspbian Buster
Computer for oneM2M CSE	
CPU	Intel Core i7 8700, 3.2 GHz
RAM	16 GB DDR4
OS	Ubuntu 18.04
Computer for dashboard AE application	
CPU	Intel Core i5 8500, 2.8 GHz
RAM	8 GB DDR4
OS	Ubuntu 18.04

gateway over USB RS485 serial interface.

In this experiment, our goal is to demonstrate the feasibility of interworking between the IoT platform and Modbus devices in terms of the following criteria:

- Data integrity. The data that is read from a Modbus device by the IPE should always be represented in the dashboard (AE) application.
- Latency. The elapsed time between the time when the IPE reads from a Modbus device and the time when it is finally delivered to the dashboard AE application should be reasonably low to keep real-time support.
- High load support. The architecture should be able to process requests at high frequency and maintain latency within a reasonable amount of time.
- Execution plan algorithm. Check if the algorithms allows to make Modbus requests in a shorter period of time compared when requests each DataPoint is accessed separately.
- Data Cache. Check that the Data Cache works properly and allows to reduce travel time for explicit Read requests.

For data integrity and latency, we were continuously reading the real-time power of solar power generation and uploading the data to the oneM2M server according to the procedure described in Section 5.1. We took measurements every 10 s from 13:30 to 15:40 during one day. As a result, we collected 960 data points of the solar power generation level. Once each data measurement is collected by the IPE, the IPE performs mapping to oneM2M message and sends it to the CSE. When the CSE updates its resources, it triggers a notification for the new data to the dashboard AE application, where the application registers the received time and measures elapsed time. Fig. 10a shows a sample of measured data points and the time when it was registered at the IPE and the dashboard AE application within a 90 s time interval. The figure justifies that all data points were successfully mapped and sent from the IPE to the dashboard and the elapsed time is not significant for this use case. The average latency was 244 ms (standard deviation = 51 ms) where on average 104 ms (standard deviation = 39 ms) was spent on serial communication between the solar charge controller and IPE. Fig. 10b shows the result of the load test. For rates up to 60 measurements per minute, the system showed similar result. At the rate of 80 and 100 measurements/min, the average time gradually increased, but the system still performed in acceptable amount of time, less than 300 ms.

To evaluate the efficiency of execution planning algorithm, 30 variables (19 input registers, 7 holding registers, 2 discrete inputs, and 2 coils) were selected to be retrieved from the Modbus device, and the total time of retrieving all variables was measured. The algorithm outputs an execution plan consisting of 18 requests based on the selected variables. Fig. 10c shows that the average time to retrieve 30 variables using the algorithm execution plan is 1.95 s compared to 2.88 s when each variable was retrieved one by one, i.e., 30 requests in total. This experiment showed that the algorithm increases the performance of the retrieval time by 33% under the test conditions.

The Data Cache was implemented using SQLite in-memory database. When Data Cache was storing information on all variables of the tested Modbus device (about 110 variables), the retrieve time for one DataPoint was made in less than 1 μ s. This implies that the time to retrieve from the Data Cache is really negligible compared to the time spent on retrieving the same variable from the Modbus device (104 ms on average).

6.2. Insights, possible applications and standards

The experiment results showed that our interworking solution is feasible in practice and has reasonable performance results considering that we used a device working on the serial Modbus RTU variant. The interworking times stayed steady and low, including during load test. Supposedly, faster times could be obtained on devices that use the Modbus TCP variant. The execution planning in couple with the Data Cache can increase the performance of the IPE significantly in certain cases.

The Modbus protocol has been implemented in a wide range of

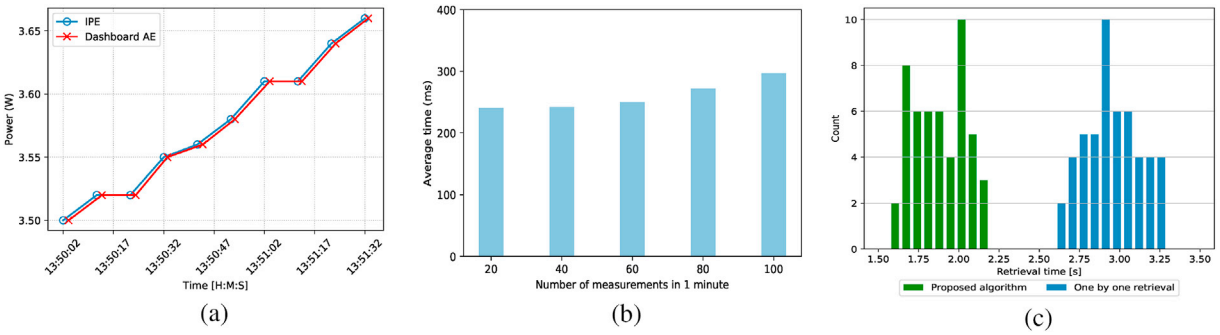

Fig. 10. Measured solar generation power at an instance of time collected by IPE and sent to Dashboard AE over oneM2M IoT platform: (a) 2 h period, (b) 1 min 30 s period. (c) Average latency under different measurement frequency.

industrial devices, from sensors to multi-purpose process automation controllers. Therefore, the proposed interworking solution can be generalized and used in various industrial IoT applications where the Modbus protocol is used together with other industrial IoT protocols.***SCADA systems***: Supervisory Control And Data Acquisition (SCADA) system [23] is a hardware-software complex for collecting real-time information from remote objects for processing, analysis and possible management. The main goal of the SCADA systems is to provide complete information about a technological process and the means for managing it. However, SCADA systems are outdated due to the technological advancement, and our proposed IoT platform supporting the interworking feature can replace them and provide extra features.

The basic components of an SCADA system are a supervisory system, a Human-Machine Interface (HMI), and Remote Terminal Units (RTU). These components can be represented by the units presented in the interworking architecture. The supervisory system gathers data from the field connected devices, sends control commands to them, and interacts with HMI software. The oneM2M CSE can perform all the supervisory system functions and, as a cloud platform, it can be utilized by other applications (oneM2M AEs) for advanced data analysis (e.g., complex event processing) or reports generation. The HMI provides a schematic graphical representation of the deployed system and allows operating personnel to control the connected devices. In our case, the HMI software can be implemented using oneM2M AE that can use modern IoT protocols, such as MQTT and CoAP. The presented Modbus IPE can perform the role of RTU for communicating with connected Modbus devices. Both PLCs (programmable) and RTUs (non-programmable) are connected to sensors and actuators and used for transmitting data to the supervisory system. The presented Modbus IPE can perform the role of PLC/RTU which connects Modbus devices.

Global standards: We have reported the proposed Modbus interworking mechanism to oneM2M that agreed on its standard specification. Initially, the proposed mechanism was studied in oneM2M as its technical report, TR-0043, "Study on Modbus Interworking", to see the feasibility of the interworking mechanism [20]. Then, the standard group agreed to proceed the development of Technical Specification (TS) for the proposed mechanism. The latest release of oneM2M, i.e., Release 4, contains TS-0040, "Modbus Interworking", which presents the standardized interworking mechanism using IPE described in this paper [24].

ITU-T standards are referenced by many countries, government states and corporations. For IoT technologies, it is vital to build common and easy access standards that benefit the widest community of users worldwide, as in the case of mobile communications. Therefore, oneM2M and ITU-T collaborate with each other to develop great standards and achieve deployment on a wider scale with conforming interoperability between different IoT technologies. At present, the relevant group for IoT standardization in ITU-T is Study Group 20 (ITU-T SG20). Over the past year, oneM2M and ITU-T SG20 have been working on transposing from published oneM2M standards, including 18 technical specifications and six technical reports into ITU-T SG20 under the Y.4500 series (see https://www.itu.int/rec/T-REC-Y/en). At the moment, both organizations collaborate to get oneM2M Rel-2 and 3 are transposed by ITU-T SG20. As the proposed technical report and specification are developed as part of oneM2M Rel-4, we expect the proposed mechanism to be transposed by ITU-T SG20 in the coming years.

Scalability: OneM2M standards have been widely used in IoT and vertical areas, e.g., smart cities, smart factories, smart homes and smart buildings. For example, the South Korean Government invested in three large scale Smart City trials based on the oneM2M standard. All three trials generated outstanding results and showcased the feasibility of using IoT technologies in Smart City situations. The proposed interworking solution allows various Modbus devices to be connected to the oneM2M IoT platform. This means that IoT services utilizing Modbus devices can be delivered to service consumers in various industrial domains by the consistent oneM2M interface. Therefore, we believe that the proposed solution is scalable because it allows IoT applications to manage various IoT devices using different network protocols, such as ZigBee, Modbus, Wi-Fi, and LoRa, via the standardized `Mca` interface.

7. Conclusions

In this paper, we presented an interworking model between Modbus protocol and IoT service layer platform that can be applied in the industrial domain. The oneM2M standard based IoT platform has been selected for our interworking architecture as it is being developed by many organizations worldwide and applied in different fields. First, we described the interworking architecture where we have used an oneM2M IPE as an intermediate unit between Modbus devices and the IoT platform, which provides continuous interworking between them. Furthermore, we have used SDT to abstract Modbus devices and register them in oneM2M server in a structured and convenient representation. We have defined the mapping rules to convert Modbus register data into oneM2M resources and the detailed procedures how to interact with Modbus devices through a oneM2M dashboard application. To improve the performance of the architecture, we introduced the algorithm that optimizes the requests execution plan and the Data Cache that decreases response time for retrieving requests.

For the validation of our model, we have provided an implementation based on the solar power management system. Our evaluation results have shown the feasibility of using oneM2M and the IPE for integration of Modbus devices to the oneM2M IoT platform.

For future work, we intend to enhance the IPE capabilities, including dynamic device registration and semantics support using ontologies. We also consider adding support of the industrial protocols such as Profibus and CIP, so that a wider range of devices could be connected to IoT network though a single IPE.

Acknowledgements

This research was conducted with the support of the Korea Research Foundation with the funding of the Ministry of Science and Information and Communication Technology (No.2018-0-88457, development of translucent solar cells and Internet of Things technology for Solar Signage).

References

[1] N. Wu, Z. Li, K. Barkaoui, X. Li, T. Murata, M. Zhou, Iot-based smart and complex systems: a guest editorial report, IEEE/CAA.J. Autom. Sin. 5 (1) (2018) 69–73.
[2] Y. Lu, Industry 4.0: a survey on technologies, applications and open research issues, J.Ind.Inf.Integration 6 (2017) 1–10.
[3] M. Liyanage, P. Porambage, A.Y. Ding, A. Kalla, Driving forces for multi-access edge computing (mec) iot integration in 5g, ICT Express 7 (2) (2021) 127–137.
[4] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, M. Hoffmann, Industry 4.0, Business & Information Systems Engineering 6 (4) (2014) 239–242.
[5] S. Park, J. Park, J. Oh, Design and implementation of trusted sensing framework for iot environment, J. Commun. Network. 23 (1) (2021) 43–52.
[6] J. Swetina, G. Lu, P. Jacobs, F. Ennesser, J. Song, Toward a standardized common m2m service layer platform: introduction to oneM2M, IEEE Wireless Commun. 21 (3) (2014) 20–26.
[7] B. Cao, Y. Li, L. Zhang, L. Zhang, S. Mumtaz, Z. Zhou, M. Peng, When internet of things meets blockchain: challenges in distributed consensus, IEEE Network 33 (6) (2019) 133–139.
[8] L. Zhang, B. Cao, Y. Li, M. Peng, G. Feng, A multi-stage stochastic programming-based offloading policy for fog enabled iot-ehealth, IEEE J. Sel. Area. Commun. 39 (2) (2021) 411–425.
[9] B. Cao, Y. Li, L. Zhang, L. Zhang, S. Mumtaz, Z. Zhou, M. Peng, When internet of things meets blockchain: challenges in distributed consensus, IEEE Network 33 (6) (2019) 133–139.
[10] Y. Li, B. Cao, M. Peng, L. Zhang, L. Zhang, D. Feng, J. Yu, Direct acyclic graph-based ledger for internet of things: performance and security analysis, IEEE/ACM Trans. Netw. 28 (4) (2020) 1643–1656.
[11] A. Polianytsia, O. Starkova, K. Herasymenko, Survey of the iot data transmission protocols, in: 2017 4th International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S T), 2017, pp. 369–371.
[12] D. Peng, H. Zhang, L. Yang, H. Li, Design and realization of modbus protocol based on embedded linux system, in: 2008 International Conference on Embedded Software and Systems Symposia, 2008, pp. 275–280.
[13] H. Park, H. Kim, H. Joo, J. Song, Recent advancements in the internet-of-things related standards: a onem2m perspective, ICT Express 2 (3) (2016) 126–129.
[14] J. Kim, J. Yun, S. Choi, D.N. Seed, G. Lu, M. Bauer, A. Al-Hezmi, K. Campowsky, J. Song, Standard-based iot platforms interworking: implementation, experiences, and lessons learned, IEEE Commun. Mag. 54 (7) (2016) 48–54.
[15] OneM2M, TS-0023: Home appliances information model and mapping, oneM2M Tech. specification V3.9.0, 2021. http://refhub.elsevier.com/S2352-8648(22)00188-2/sref15. (Accessed 8 November 2019).
[16] Y. Mehmood, F. Ahmad, I. Yaqoob, A. Adnane, M. Imran, S. Guizani, Internet-of-things-based smart cities: recent advances and challenges, IEEE Commun. Mag. 55 (9) (2017) 16–24.
[17] Z. Fan, R.J. Haines, P. Kulkarni, M2m communications for e-health and smart grid: an industry and standard perspective, IEEE Wireless Commun. 21 (1) (2014) 62–69.
[18] S. Safaric, K. Malaric, Zigbee wireless standard, in: Proceedings ELMAR 2006, 2006, pp. 259–262.
[19] J. Yun, S.-C. Choi, N.-M. Sung, J. Kim, Demo, Towards global interworking of iot systems – oneM2M interworking proxy entities, in: Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, SenSys '15, ACM, New York, NY, USA, 2015, pp. 473–474.
[20] OneM2M, TR-0043: Modbus Interworking, oneM2M Tech. rep. V0.2.0, 2019. http://refhub.elsevier.com/S2352-8648(22)00188-2/sref20. (Accessed 15 October 2020).
[21] OneM2M, TS-0014: LWM2M Interworking, oneM2M Tech. specification V3.2.0, 2020. http://refhub.elsevier.com/S2352-8648(22)00188-2/sref21. (Accessed 13 March 2020).
[22] Eclipse OM2M. https://www.eclipse.org/om2m/(2015–2020). (Accessed 13 December 2020).
[23] A. Daneels, W. Sater, What is scada?, in: Proceedings of the International Conference on Accelerator and Large Experimental Physics Control Systems Italy, 1999, pp. 339–343.
[24] OneM2M, TS-0040: Modbus interworking, oneM2M Tech. specification V0.1.0, 2020. http://refhub.elsevier.com/S2352-8648(22)00188-2/sref24. (Accessed 8 May 2021).