Virtualized Sensor System: an Access Unification and Software-defined Sensors

Naoki Aoyama
Graduate School of Engineering,
Nagoya Institute of Technology,
Nagoya-shi, 466-8555 Japan
Email: naoki@moss.elcom.nitech.ac.jp

Yonghwan Kim
Graduate School of Engineering,
Nagoya Institute of Technology,
Nagoya-shi, 466-8555 Japan
Email: kim@nitech.ac.jp

Yoshiaki Katayama
Graduate School of Engineering,
Nagoya Institute of Technology,
Nagoya-shi, 466-8555 Japan
Email: katayama@nitech.ac.jp

***Abstract*—As the growth and the spread of Internet of Things (IoT), various context-aware applications, which determine their behaviors based on the recognized context, are widely studied and developed. To develop such applications, an application programmer has to understand every sensor's specification, e.g., access method (i.e., how to get a current value), for handling all the necessary sensors. Moreover, even some sensors are the same types, e.g., temperature sensors, they may have different units of values, hence, the unification of each sensor's unit may be required. These make some inexperienced programmers hard to develop context-aware applications. In this paper, we present a novel middleware named *Virtualized Sensor System (VSS)*, which provides some features for application programmers as follows: (i) provides unified methods to access every sensor in the system without any knowledge of their specifications, and (ii) can create new *software-defined sensors* by composing of some hardware sensors. To help to create a new *software-defined sensor*, we present a new markup language, *Virtual Sensor Markup Language (VSML)*. Our proposed system *VSS* and markup language *VSML* can make the developments of the context-aware applications using various sensors easy even if an application programmer is inexperienced in them.**

Keywords–Sensor Middleware; Sensor Virtualization; Software-defined Sensor; Support for Application Developments; Internet of Things.

I. INTRODUCTION

Background and contribution of this study, and related works are introduced in this section.

A. Background and Contribution

As the growth and the spread of Internet of Things (IoT) [1]-[3], various context-aware applications, which determine its behavior based on the recognized context, are widely studied and developed [4]. Context-aware applications can provide more flexible services to users based on their current context (e.g., their locations, activity history, or their environments), hence, more context-aware applications, especially using various sensors, are expected to emerge. However, the emergence of new various sensors may make many programmers difficult to develop applications, because they should understand every sensor's specification to develop a context-aware application using these new sensors. This implies that to make sensors easy to use regardless of the specifications of new sensors is an important issue. And as context-aware applications become more multifaceted, some specific sensors which do not exist but may be useful for some limited applications can be demanded. For example, there can be application programmers who want to use a water vapor capacity sensor which can be calculated using one thermometer (i.e., temperature sensor) and one hygrometer (i.e., humidity sensor) and there also can be some other programmers who want to use a new motion sensor consisting of several motion sensors which are referred in a particular order. There are several problems for application programmers to develop applications using various sensors. Here, we mainly focus on the following three issues: (a) how to acquire data from sensors of various specifications, (b) how to refer to past data of sensors, and (c) how to use data from sensors which do not actually exist but can be defined using some other sensors. To solve the three issues mentioned above, in this paper, we present a new middleware named *Virtualized Sensor System (VSS)* which provides three helpful features for sensor users as follows: (a) a function for unifying access methods and data formats, (b) a database for managing all the past data from sensors, and (c) a function for creating software-defined sensors. From these features of our proposed system, sensor users can easily access all sensors using unified methods, even their past data, regardless of the actual existence of them (see Figure 1).

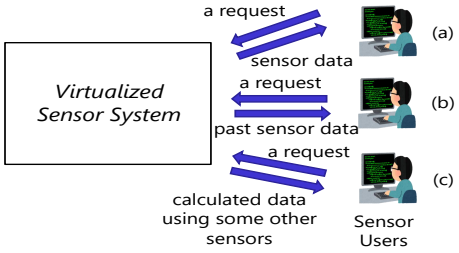

Figure 1. Proposed system in this paper

B. Related Works

A system for unifying interfaces of IoT devices using SQL query is proposed [6]. In [6], Unified IoT Device Query (UDQ) system allows application programmers to use an unified interface for using sensors. UDQ system is capable of unifying interfaces of only hardware sensors. However, to use some calculated data using some hardware sensors, an application programmer has to implement some methods for its calculation and accession for all the necessary sensors.

A system to manage network-connected sensors by applying Simple Network Management Protocol (SNMP) technology is also proposed [7]. The data generated by network-connected sensors which are managed by SNMP are stored in virtual database called Management Information Base (MIB).

 ISBN: 978-1-61208-692-7

SNMP agents have MIBs, and they output the data in response to requirements from SNMP managers. SNMP managers are able to access MIBs that SNMP agents have by using commands defined in SNMP. However, only hardware sensors can be managed in the system proposed in [7], and the system does not allow users to define new sensors' data which are calculated by data that hardware sensors generate.

A middleware named *Global Sensor Networks (GSN)* [8] for processing sensor data in sensor networks is proposed. *GSN* enables users of *GSN* to define new sensors through an XML-based language. A new sensor defined by a user of *GSN* is called a virtual sensor. Application programmers can acquire data from new sensor networks consisting of defined virtual sensors in the same way. However, the XML-based language which is used in *GSN* makes a user of *GSN* difficult to define a virtual sensor which is referred in a particular order because a user of *GSN* cannot specify conditions which determine the order directly. *VSS* provides a way to express a sensor which is referred in a particular order to users of *VSS*, and users of *VSS* can define such sensors easier than users of *GSN* (i.e., users of *VSS* can direcly define such sensors). For example, a new temperature sensor that notifies to users when the following event occurs can be defined by users of *VSS* easier: an event that sensor's data generated by a hardware temperature sensor drops below 20 degrees only after sensor's data generated by the same sensor becomes greater than 20 degrees.

A network simulator for Wireless Sensor Network (WSN) named *Configurable Multi-Layer WSN (CML-WSN)* is proposed in [9]. Each node in WSN has a capability for sensing and ad-hoc network is constructed based on each node's communication range. One of the main problems of ad-hoc networks is that there is no infrastructure, so the routes change dynamically. This may cause a decrease in quality of services, much power consumption, or security problems. As a result, communication problems, e.g., packet loss, can be occurred. To address the problems, *CML-WSN* allows users to create network topologies, configure devices, inject packets, and change network settings. However, different points between *CML-WSN* and *VSS* are the features which are provided to sensor users, and *VSS* provides methods to users of *VSS* that make developments of applications easy.

C. Organization of this paper

The rest of the paper is structed as follows. We introduce some preliminaries to help to understand our work in Section II. Section III presents our proposed system *VSS* and its prototype system is presented in Section IV. A conclusion and future works are given in Section V.

II. PRELIMINARIES

Preliminaries to help to understand our work are introduced in this section.

A. Sensors

In this paper, we call a device generating any digital data which can be processed by any computational entities a sensor, this means that sensors considered in this paper have wider meaning than typical sensors. For example, we can consider OpenWeatherMap [5], which is a web-based service providing weather information, as a sensor even no hardware sensor exists. We call any devices or services generating any digital data an *actual sensor*.

(a) Pull (b) Push

Figure 2. Pull and Push data acquisition

B. Two acquisition types: Pull and Push

Pull: A pull acquisition is an acquisition method to acquire sensor data immediately whenever it is requested.

Push: A push acquisition is an acquisition method to acquire sensor data by notification from sensors when one or more preset conditions by the sensor user are satisfied.

These imply that a pull and a push acquisition can be expressed as an active and a passive acquisition respectively. These two acquisition types are depicted in Figure 2.

C. Software-defined Sensor

In this paper, we call a sensor which is created by composing of *actual sensors* a *software-defined sensor*. For example, a water vapor capacity sensor which can be calculated using one temperature sensor and one humidity sensor, and a motion sensor consisting of several motion sensors, which are referred in a particular order are both *software-defined sensors*. Unified methods provided to sensor users as an API enable sensor users to use *actual sensors* and *software-defined sensors* in the same way. We call a user who defines a *software-defined sensor* in order to provide its data to sensor users a system manager. We present an XML-based language named *Virtual Sensor Markup Language (VSML)* which enables system managers to define new *software-defined sensors*. We call a file written in *VSML* a virtual sensor definition file. We assume that there are processes which interpret virtual sensor definition files and create requested *software-defined sensors* at all times.

III. PROPOSED SYSTEM: *VSS*

A design and an implementation of proposed system *VSS* are introduced in this section.

A. Overview of VSS

An overview of *VSS* is shown in Figure 3. *VSS* has four main layers in order to solve requests from sensor users illustrated in Figure 1. Firstly, sensor users can use virtual sensors in *VSS* using two acquisitions (Pull and Push) regardless of the specifications of the sensors due to *Unifying Data Acquisition Layer*. Secondly, *VSS* enables sensor users to use unified data formats due to *Unifying Data Format Layer*. Thirdly, *VSS* enables sensor users to use all the past data due to *Data Management Layer*. Finally, *VSS* enables sensor users to use *software-defined sensors* defined by system managers due to *Creating Software-Defined Sensors (S.D.S) Layer*.

To help to understand the features of our proposed system, we give an example scenario here. Assume that there are two different temperature sensors denoted by t_1 and t_2 respectively and two different humidity sensors denoted by h_1 and h_2. And we also assume a sensor user who considers to implement a context-aware application using these all sensors and two additional special sensors as follows: (a) a sensor outputs the

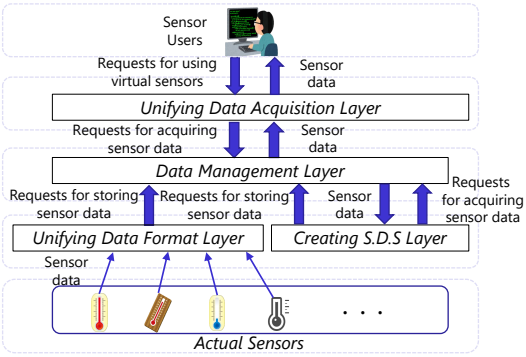

Figure 3. Architecture of our proposed system *VSS*

average value of two temperature values which are generated by two different temperature sensors a year ago respectively, and (b) a sensor of the same function as the sensor (a) using two different humidity sensors. Let t_3 (resp. h_3) be the former one (resp. the latter one). Note that the temperature sensors t_1 and t_2 are *actual sensors* but t_3 is a *software-defined sensor* in this case. If the sensor user arranges all these 6 sensors, the user must understand the specifications of 4 *actual sensors*, and has to maintain database for storing sensor data from at least 1 year ago. The sensor user easily uses all sensors introduced above using the API provided by *VSS* even the user never knows these sensors' specifications. Particularly, *Unifying Data Format Layer* unifies all sensors' data formats and units. Note that the data format and the unit depend on the specification of each sensor. Thus, the sensor user does not need to know the specifications of 4 *actual sensors*. *Data Management Layer* maintains all sensors' data, hence the sensor user can access the sensor data a year ago. *Creating S.D.S Layer* allows users to create any *software-defined sensors* freely, so the sensor user can create two *software-defined sensors* in this case. Finally, *Unifying Data Format Layer* provides many access methods for virtual sensors in the system as an API. This layer allows the sensor user to acquire sensors' data easily. These all layers are illustrated in Figure 3. As the example scenario introduced above, our proposed system *VSS* offers many helpful layers for sensor users who want to develop applications (especially, context-aware applications) using various sensors. The detail of each layer will be explained in the next subsection.

B. Layers which construct VSS

VSS consists of four layers as shown in Figure 3. Each layer in Figure 3 consists of one or more modules, and each module in any layers is a function, which consists of the following three components: (a) input data, (b) steps of an execution, and (c) output data. Note that any implementations of each layer in Figure 3 is encapsulated from any modules in another layer.

1) Unifying Data Format Layer: Modules in *Unifying Data Format Layer* receives sensor data from *actual sensors*, and converts the format of the sensor data into the predetermined unified format.

2) Data Management Layer: Data Management Layer manages all sensor data received from drivers in *Unifying Data Format Layer*. Each sensor data received in *Data Management Layer* is stored in its local database with its received time as its timestamp. *Data Management Layer* realizes the following two acquisition types: (a) returns sensor data which is maintained in its local database when it requested, and (b) returns sensor data which is maintained in its local database when its received data is changed.

3) Unifying Data Acquisition Layer: Modules in *Unifying Data Acquisition Layer* receive requests from sensor users and realize the two types of acquisition (Pull and Push).

Pull: A module for pull acquisitions requests the most recent sensor data before the specific time which is requested by a sensor user to *Data Management Layer*, and forwards it to the sensor user.

Push: A module for push acquisitions receives a sensor's name (consisting of a sensor's type, a unit of output data, and a datetime), a condition, and an event handler in advance, and when this layer detects satisfactions of conditions required by sensor users, the module notifies sensor users by calling event handlers specified by them.

4) Creating S.D.S Layer: Creating S.D.S Layer creates *software-defined sensors* defined by virtual sensor definition files, and these sensors can be used as *actual sensors*. *Creating S.D.S Layer* maintains virtual sensor definition files and interpret them. *Creating S.D.S Layer* bring sensor data which are necessary for calculations from *Data Management Layer*, calculate data using them, and send calculated sensor data back to *Data Management Layer*. System managers can define two types of *software-defined sensors*, combination-type and sequence-type, using *VSML*. When a system manager sends a virtual sensor definition file to *VSS*, *Creating S.D.S Layer* creates a *software-defined sensor* based on a type which is written in the file. Two types of specifications for system managers to define *software-defined sensors* using *VSML* are presented in the following paragraphs.

Combination type: Modules for combination-type *software-defined sensors* have three roles. Firstly, these modules bring sensor data which are necessary for calculations from *Data Management Layer*. Secondly, these modules calculate data using current sensors' values. Finally, these modules send calculated data to *Data Management Layer*. System managers define a combination-type *software-defined sensor* by specifying the following three items: (a) a description of one or more method names of pull acquisition (i.e., which sensor data are used for calculations), (b) a formula for calculations (i.e., how to calculate data specified in (a)), (c) a description of metadata consisting of the following three items: (i) a sensor's type (e.g., temperature), (ii) a unit of output data, and (iii) a data type which implies that defined sensor data consists of only a datetime, or consists of a value and a datetime.

Sequence type: Modules for sequence-type *software-defined sensors* have three roles. Firstly, these modules maintain each sensor's state in its local storage. Secondly, these modules transit the state if necessary referring received sensor data in its chronological order. Finally, this state transition determines the output data and its output timing, and these modules send the determined output data by the state transition to *Data Management Layer*. In this paper, a state transition table is used for maintaining a history of sensor data to create sequence-type *software-defined sensors*. System managers define a sequence-type *software-defined sensor* by specifying the following three items: (a) a description of one or more method names of push acquisition (i.e., which conditions are used for state

 ISBN: 978-1-61208-692-7

transitions), (b) a description of a state transition table (i.e., how to determine output data using conditions specified in (a)), (c) a description of metadata consisting of the following three items: (i) a sensor's type (e.g., temperature), (ii) a unit of output data, and (iii) a data type which implies that defined sensor data consists of only a datetime, or consists of a value and a datetime.

C. Implementation of VSS

In this subsection, we explain the implementation of each module in each layer, presented in the previous subsection, in our proposed system *VSS*.

1) Unifying Data Format Layer: *Unifying Data Format Layer* maintains sensor drivers. Each driver is corresponding to each *actual sensor* in one-to-one. When a driver receives sensor data from its corresponding *actual sensor*, the driver converts the format of the sensor data into the predetermined unified format. Each driver sends sensor data and its corresponding metadata to *Data Management Layer*. A metadata consists of a sensor's name, a sensor's type, a unit of output data, and a data type which implies that the output data may change or not. A metadata of each *actual sensor* is generated and sent by its corresponding driver.

2) Data Management Layer: *Data Management Layer* maintains a local database for storing sensor data and metadata. It stores all the sensor data received from *Unifying Data Format Layer* and *Creating S.D.S Layer* to its local database with its timestamp (i.e., receipt time of sensor data), and it sends corresponding sensor data to a module in *Unified Data Acquisition Layer* or *Creating S.D.S Layer* when it requested.

3) Unifying Data Acquisition Layer: *Unifying Data Acquisition Layer* consists of two modules for the two acquisition types introduced in Section III.B. A module for pull acquisitions provides a method for using pull acquisition illustrated in Section III.B.3) to sensor users. A module for push acquisitions consists of two components, a module for judging conditions, and an Event-Condition-Action (ECA) rule database. The ECA rule is a rule for an event-driven action. An ECA rule is generally represented by the following syntax: On *Event* If *Condition* Do *Action*, which means that when the *Event* occurs, if a system satisfies the *Condition*, then the system executes the *Action*. A module for judging conditions receives ECA rules specified by sensor users. An ECA rule received by a sensor user consists of the following three components: (a) a sensor's name for an *Event*, (b) an expression which includes the sensor's name specified in the *Event* for a *Condition*, and (c) an event handler which is called when the *Condition* is satisfied for an *Action*. For example, when a sensor user requests to call an event handler when the output value from one sensor T becomes greater than 20, a module for judging conditions meets the requirement if the user assigns $T > 20$ as a *Condition*. An algorithm for push acquisitions is executed as the following steps.

(Step1) When a sensor user sends a sensor's name, a condition, and an event handler to the module for judging conditions, a module for judging conditions registers the received sensor's name as an Event, the received condition as a Condition, and the received event handler as an Action. This set of an Event, a Condition, and an Action is registered as an ECA rule. The module for judging conditions stores every ECA rule to its database. (Step2) Whenever each sensor's name registered as an Event outputs a data with its timestamp, the module for judging conditions acquires it from *Data Management Layer*. (Step3) When the module for judging conditions acquires new data from a sensor registered as an event, it checks the conditions referring the corresponding ECA rules in ECA rule database. If the conditions are satisfied, the module for judging conditions proceeds the next step, Step4. (Step4) The module for judging conditions determines each event handler corresponding the ECA rules, and call it.

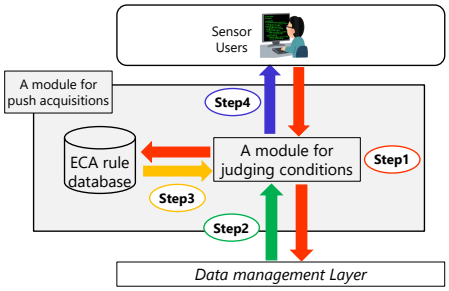

Figure 4. A module for push acquisitions

Diagrams which illustrate how steps of proposed algorithms proceed are depicted in Figure 4, 5, and 6. One or more arrows of the same color as the frame color of the ellipse surrounding each step ID illustrated inside it represent the corresponding dataflow described in an algorithm. A diagram for verifying the different steps of the above algorithm is illustrated in Figure 4.

4) Creating S.D.S Layer: *Creating S.D.S Layer* consists of the following four components: (a) virtual sensor definition files, (b) a module for interpreting virtual sensor definition files, (c) a module for combination-type *software-defined sensors*, and (d) a module for sequence-type *software-defined sensors*. Specifications of the module (a) are presented in subsection III.B, and the module (b) can be implemented with an XML parser, e.g., Document Object Model (DOM), which is available as an API. And two implementations which realize two modules, (c) and (d), are presented in the following paragraphs.

Combination type: A module for combination-type *software-defined sensors* consists of the following two components: (a) a module for interpreting virtual sensor definition files, and (b) a module for exchanging data with *Data Management Layer* and calculating data using values which are got from *Data Management Layer* based on those definition files. The module (b) consists of for corresponding one combination-type *software-defined sensor*. For example, we assume that two sensors A and B exist in the system. In this case, a system manager can create a new sensor C as a *software-defined sensor* which returns the average value of the output values of two sensors A and B. The formula to calculate the output value of C is maintained by the above module (b). An algorithm for a combination-type *software-defined sensor* is executed as the following steps.

(Step1) A system manager sends a virtual sensor definition file (shortly, definition file) to *VSS*. The definition file includes (a) a list of methods for acquiring the values of the sensors which are necessary for calculating a new value, (b) a formula for a calculation, and (c) metadata of the required sensors. These are stored in the module for calculations. (Step2) The module for calculations gathers all the sensor values using

 ISBN: 978-1-61208-692-7

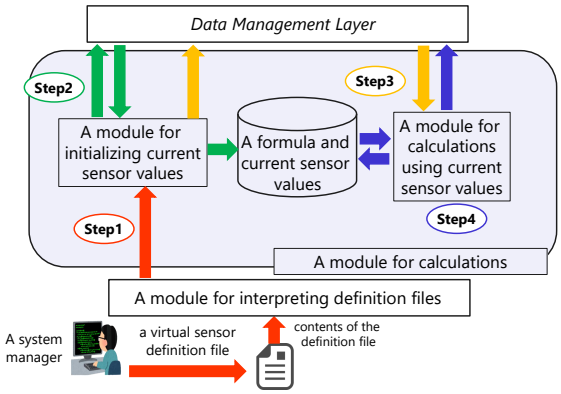

Figure 5. A module for calculations

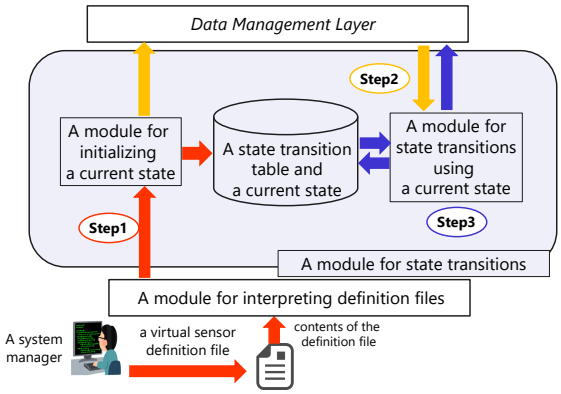

Figure 6. A module for state transitions

methods in the definition file, and calculate the value using the gathered values referring the formula in the definition files. This value is stored in this module temporarily. (Step3) The module for calculations sends a request to send the corresponding sensor values whenever they are updated to *Data Management Layer*. This implies that a push acquisition from *Data Management Layer* is realized by this request. Whenever the module for calculations receives a new sensor data, the module for calculations enqueues it to its local maintaining queue. (Step4) If the local queue is not empty, the module for calculations dequeues one sensor value and calculates a new value again referring the stored formula. This new value is compared with the value stored temporarily in Step3, and the newly calculated value is sent to *Data Management Layer* as a new updated value if it is different with the temporary value. Otherwise, the new value is discarded. This process will be repeated until its local queue becomes empty.

A diagram for verifying the different steps of the above algorithm is illustrated in Figure 5.

Sequence type: A module for sequence-type *software-defined sensors* consists of the following two components: (a) a module for interpreting virtual sensor definition files, and (b) a module for exchanging data with *Data Management Layer* and transiting its current state based on those definition files. The module (b) consists of each module for corresponding one sequence-type *software-defined sensor* defined by system managers. For example, when a system manager defines a *software-defined sensor* T which sends sensor data to *Data Management Layer* based on state transitions shown in TABLE I, a module for state transitions maintains a state transition table and a current state of the table. The value ϵ in TABLE I represents that no data is sent to *Data Management Layer* even if the condition is satisfied.

TABLE I. AN EXAMPLE OF STATE TRANSITION TABLE

Source	Destination	Condition	Output
0	1	$T > 20$	ϵ
1	0	$T < 20$	"return20"
1	2	$T > 21$	ϵ
2	1	$T < 21$	"return21"

Note that every state transition in a module for state transitions implies that the module for state transitions transits the current state which is represented with a source to the state which is represented with a destination defined by a system manager. An algorithm for a sequence-type *software-defined sensor* is executed as the following steps.

(Step1) A system manager sends a virtual sensor definition file (shortly, definition file) to *VSS*. The definition file includes (a) a list of methods for acquiring the conditions of the sensors which are necessary for transiting a current state, (b) a state transition table consisting of the following four items: (i) a state which represents a source, (ii) a state which represents a destination, (iii) sensor data which is sent to *Data Management Layer* when the transition from the source to the destination occurs, and (iv) a method name of a push acquisition, (c) an initial state name of the state transition table specified in (a), and (d) metadata of the required sensors. These are stored in the module for state transitions. The initial state is set as a current state of the state transition table. (Step2) The module for state transitions sends a request to send the corresponding conditions whenever they are satisfied to *Data Management Layer*. This implies that a push acquisition from *Data Management Layer* is realized by this request. Whenever the module for state transitions receives a new sensor data, the module for state transitions enqueues the condition which is satisfied in this case to its local maintaining queue. (Step3) If the local queue is not empty, the module for state transitions dequeues one condition. The module for state transitions updates the current state with the state which represents destination referring the stored state transition table and the current state. If the state transition occurs by the condition and data which is specified by a system manager is a string of letters except for a special symbol ϵ referring the stored state transition table, the module for state transitions sends the data to *Data Management Layer*. This process will be repeated until its local queue becomes empty.

A diagram for verifying the different steps of the above algorithm is illustrated in Figure 6.

IV. Prototype System

We implement the prototype system of our proposed system to verify its operation.

A. Environment of the prototype system

The prototype system has three kinds of sensors: a temperature sensor (BD1020HFV), a temperature and humidity sensor (DHT11), and three motion sensors (Grove-PIR Motion Sensor) as shown in Figure 7.

In our prototype system, all the layers of *VSS* are implemented using Java, and we use the file system of Windows 10, NTFS, for the database of *Data Management Layer*. WatchService [10], which is provided by a basic library of

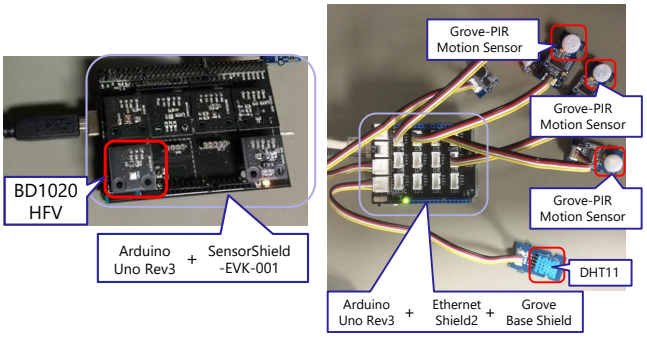

Figure 7. Hardware sensors used in the prototype system

Java, is used for detecting changes of sensor data in *Data Management Layer*. The environment of the prototype system is shown in TABLE II.

TABLE II. EXPERIMENTAL ENVIRONMENT

CPU	Intel Core i5 6500 3.20GHz
OS	Windows 10
RAM	DDR4 8 GB
JRE	1.8.0_172-b11

As an example of the *software-defined sensor*, we introduce a new *software-defined sensor* which varies its output value by the sequence of the output of three different motion sensors (see Figure 8).

B. Latency for push acquisitions

We confirm the average latency of push acquisitions, and as a result, we found that the average latency is 3.65ms in the case of prototype system. We expect to estimate latencies which are confirmed in other environments of implementations of *VSS*.

V. Conclusion and Future Works

In this paper, we proposed *VSS* which provides unified methods to access any sensor regardless of their specifications, and can create new *software-defined sensors* by composing of one or more *actual sensors*.

To verify the operations of our proposed system *VSS*, we implemented the prototype system, and we found our system is operated well and push acquisitions have practically low latency in the case of our prototype system. However, some problems can be considered if our system is implemented in the distributed manner, e.g., a system consisting of two or more computational entities which are connected by networks. Especially, there are several important issues which have to be discussed in an implementation of *Data Management Layer*. For example, we may consider the following issues: (a) a fault-tolerance; a system guarantees its availability even if some nodes in the system fail, and (b) load balancing; to ensure a reliability of the system, a load of each node should be balanced as possible. Finally, we also consider that how to implement a sequence-type *software-defined sensor* can consistently process the message from every sensor in the actual order in which it is sent.

```
<VirtualSensor name="multimotion" type="sequence">
  <states>
    <statename>0</statename>
    <statename>1</statename>
    <statename>2</statename>
  </states>
  <initialstate>
    <statename>0</statename>
  </initialstate>
  <functions>
    <function>
      <source>0</source>
      <condition>watchMotionUpdate("MOTIONa",
          currenttime, "change")</condition>
      <output>a</output>
      <destination>1</destination>
    </function>
    <function>
      <source>1</source>
      <condition>watchMotionUpdate("MOTIONb",
          currenttime, "change")</condition>
      <output>ab</output>
      <destination>2</destination>
    </function>
    <function>
      <source>2</source>
      <condition>watchMotionUpdate("MOTIONc",
          currenttime, "change")</condition>
      <output>abc</output>
      <destination>0</destination>
    </function>
  </functions>
  <description>
    <sensortype>motion</sensortype>
    <datatype>string</datatype>
    <unit>null</unit>
  </description>
</VirtualSensor>
```

Figure 8. A definition of a new motion sensor

References

[1] S. Madakam, R. Ramaswamy, and S. Tripathi, "Internet of Things (IoT): A Literature Review," Journal of Computer and Communications, volume 3, pp. 164-173, 2015.

[2] S. Pote, "Internet of Things Applications, Challenges and New Technologies," International Conference on Advances in Computer Technology and Management (ICACTM), At Pimpri Chinchwad, Pune, pp. 45-51, Feb 2018.

[3] Z. K. A. Mohammeda, and E. S. A. Ahmedb, "Internet of Things Applications, Challenges and Related Future Technologies," World Scientific News, Vol. 67, pp. 126-148, 2017.

[4] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," International Journal of Ad Hoc and Ubiquitous Computing, Vol. 2, Issue 4, pp. 263-277, June 2007.

[5] OpenWeatherMap, URL: https://openweathermap.org/ [retrieved: January, 2019].

[6] S. Kinoshita, Y. Kuga, and O. Nakamura, "Unifying Interfaces of IoT Devices Using SQL Query (in Japanese)," Information Processing Society of Japan Multimedia, Distributed, Cooperative, and Mobile(DICOMO) Symposium, pp. 1036-1041, 2016.

[7] H. Hui-Ping, X. Shi-De, and M. Xiang-Yin, "Applying SNMP Technology to Manage the Sensors in Internet of Things," The Open Cybernetics & Systemics Journal, pp. 1019-1024, 2015.

[8] K. Aberer, M. Hauswirth, and A. Salehi, "The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks," Swiss Federal Institute of Technology, Lausanne (EPFL), Tech. Rep., pp. 1-21, 2006.

[9] C. Del-Valle-Soto, F. Lezama, J. Rodriguez, C. Mex-Perera, and E. M. de Cote, "CML-WSN: A configurable multi-layer wireless sensor network simulator," Applications for Future Internet: International Summit, pp. 91-102, AFI 2016, Puebla, Mexico, May 25-28, 2016.

[10] WatchService, URL:https://docs.oracle.com/javase/7/docs/api/java/nio/file/WatchService.html [retrieved: January, 2019].

 ISBN: 978-1-61208-692-7