Hindawi
Mobile Information Systems
Volume 2018, Article ID 8618267, 9 pages
https://doi.org/10.1155/2018/8618267

Research Article

In-Network Data Processing in Software-Defined IoT with a Programmable Data Plane

Ki-Wook Kim,[1] **Sung-Gi Min**,[1] **and Youn-Hee Han**[2]

[1]*Department of Computer and Radio Communication Engineering, Korea University, Seoul, Republic of Korea*
[2]*Interdisciplinary Program in Creative Engineering, Korea University of Technology and Education, Cheonan, Republic of Korea*

Correspondence should be addressed to Sung-Gi Min; sgmin@korea.ac.kr

Received 15 December 2017; Accepted 13 February 2018; Published 1 April 2018

Academic Editor: Jeongyeup Paek

Making an SDN data plane flexible enough to satisfy the various requirements of heterogeneous IoT applications is very desirable in terms of software-defined IoT (SD-IoT) networking. Network devices with a programmable data plane provide an ability to dynamically add new packet- and data-processing procedures to IoT applications. The previously proposed solutions for the addition of the programmability feature to the SDN data plane provide extensibility for the packet-forwarding operations of new protocols, but IoT applications need a more flexible programmability for in-network data-processing operations (e.g., the sensing-data aggregation from thousands of sensor nodes). Moreover, some IoT models such as OMG DDS, oneM2M, and Eclipse SCADA use the publish-subscribe model that is difficult to represent using the operations of the existing message-centric data-plane models. We introduce a new in-network data-processing scheme for the SD-IoT data plane that defines an event-driven data-processing model that can express a variety of in-network data-processing cases in the SD-IoT environment. Also, the proposed model comprises a language for the programming of the data-processing procedures, while a flexible data-plane structure that can install and execute the programs at runtime is additionally presented. We demonstrate the flexibility of the proposed scheme by using sample programs in a number of example SD-IoT cases.

1. Introduction

In recent years, the software-defined Internet of Things (IoT), or SD-IoT, has become one of the main topics of IoT-related researches. It deals with several issues for the deployment and management of numerous IoT-infrastructure nodes by adopting the idea of software-defined networking (SDN) for which a centralized controller is employed, whereby flexible control and management capabilities are realized regarding the heterogeneous IoT-infrastructure nodes such as sensors, sensor gateways, network devices, and cloud servers. In the SD-IoT, the IoT applications in the cloud servers collect data from the sensor platforms that are then connected through the SD-IoT network.

The SD-IoT network basically provides a data-transferal service that is based on a message-centric model. In this model, the unit of information exchange is the message itself. The role of the network infrastructure is the ensuring of the delivery of the messages to their intended recipients, irrespective of the message contents. The focus of the current SDN research is this model, which can be applied for the data transferal of the SD-IoT network.

A number of the nodes of the SD-IoT network, however, are required for the implementation of the data processing in addition to the data-transferal functionalities of this network; here, sensor gateways for sensor platforms (Figure 1) or vehicular ad hoc network (VANET) nodes are typical examples. The overall capability of these nodes means they can process the collected data for the transmission of useful information to their servers, instead of sending whole raw sensing data. Most of the IoT applications use the data-centric publish-subscribe (DCPS) model for the nodal information exchange with the cloud servers. In this model, the publisher node supplies data whenever data are available to the remote subscribers that are interested in the data. The publishers and the subscribers interact with each other using

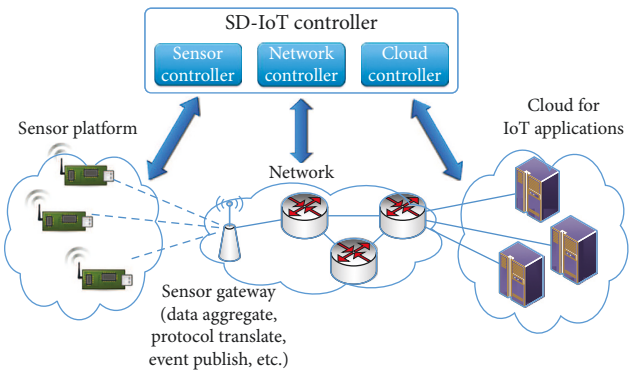

Figure 1: A SD-IoT architecture with a sensor gateway.

the standard interface. The subscriptions should be processed on the control/management plane, and upon their generation, the data are sent via the data plane.

To provide the IoT application that uses the DCPS model information-exchange method with the current message-centric SDN, the data-processing module must be located at the SDN controller; this is because the current SDN data plane is not sufficiently flexible to implement the data-processing module for the DCPS model. At present, these cases cannot be processed in the SD-IoT network as a result, so a separate middleware-application server emulates them. In [1], a Data Distribution Service (DDS) middleware that provides a DCPS abstraction to the applications is located over the SDN controller and is used by the IoT applications. While it presents a simple solution, its compatibility with the SDN concept is not precise because the data processing is addressed over the SDN control plane. According to the SDN concept, the data processing should occur on the data plane of each network node, and the control plane inserts the processing rules into the nodes.

For an incorporation of the two communication models that is consistent with the SDN concept, the SDN data plane should be upgraded to handle in-network data processing. The focus of the current message-centric SDN data plane is the matching of the incoming packets with the flow table entries and the forwarding of these packets; therefore, the table-based processing architecture presents a natural corresponding solution. The requisite complex operations of the in-network data processing, however, cannot be satisfied by this architecture.

The Event-driven Instruction-Based Packet Processing (E-IPP) scheme is presented in this paper as the SD-IoT data plane for the enhancement of the corresponding programmability. The event-driven processing model of the proposed scheme supports various predefined or user-defined events, so its extensibility is regarding programs that can be run on various events as well as the packet events. The scheme includes the event model, the E-IPP language, and the E-IPP Virtual Machine (E-IPP VM) structure. The language is used to program the application-specific procedure for the IoT data. The E-IPP VM is the data plane for the program that has been written in the E-IPP language, which is versatile enough to incorporate the DCPS model as well as the message-centric model; furthermore, it supports dynamic program loading at run-time.

2. Related Work

2.1. SDN in the IoT. A number of SDN-adoption studies that are regarding IoT networking have been completed. Subsequently, a segment of researchers favor the SDN-based control of every IoT feature as well as the network resources, calling it SD-IoT. Jararweh et al. suggested an SD-IoT framework [2] for which they introduced the concept of a software-defined system (SDSys). The SDSys hides all of the complexities of the management and control functionalities of the system resources from the end users. They also proposed a software-defined IoT control framework for which multiple SDSys types such as software-defined networks, software-defined storage systems, and software-defined security are integrated. Liu et al. suggested an SD-IoT architecture for an urban-sensing case [3] that consists of a data acquisition service for sensors, a data transmission service for network devices, and a data processing service for cloud servers. They also emphasized the role of data aggregation and compression, just as many studies have previously claimed in WSN [4–7]. This means that an intermediate node with data processing capability is needed in the SD-IoT network.

2.2. DCPS Model for the IoT. For the DCPS model, the concept of a "global data space," where the known-structure data values are exchanged, is built upon. Here, the applications that want to contribute information to this data space declare their intent to become a publisher. Similarly, the applications that want to access portions of this data space declare their intent to become subscribers. Each time a publisher wants to post new data into this global data space, it propagates the information to all of the interested subscribers. To handle the DCPS communications, each node is made aware of the contents of the incoming packets, meaningful information is generated from them, and the global data are finally updated. This model is used by some of the IoT network protocols that are present between the IoT devices and their client applications.

The DDS [8] that is standardized by the Object Management Group introduces one of the IoT network protocols with the DCPS model. This protocol provides platform/-language-independent mechanisms for the construction of distributed publish-subscribe systems with various quality of service (QoS) guarantees and reliability-control capabilities. The Message-Queuing Telemetry Transport (MQTT) protocol [9] is another IoT protocol of the DCPS model, and a broker is used to offload the burden of the handling of a large number of upstream server requests for IoT devices.

Some IoT applications require horizontal integrations wherein inter-protocol conversions occur. Recently, several attempts were made to address this issue including the oneM2M project that aims to provide a common IoT service platform by consolidating the currently isolated protocols [10]. oneM2M utilizes the representational state transfer (REST) system for the representation and management of

IoT devices, and common application protocols including the hypertext transfer protocol (HTTP), the constrained application protocol (CoAP), and the MQTT are used for the interworking with other systems besides oneM2M. The Eclipse IoT project provides another integrated IoT service platform [11]. The project is composed of many subprojects such as supervisory control and data acquisition (SCADA), Krikkit, and Ponte. Krikkit is a data-acquisition architecture that uses the DCPS model, while SCADA and Ponte provide a common communication mechanism and uniform open application programming interfaces (APIs) for programmers, thereby enabling conversions between various IoT protocols, such as the HTTP, CoAP, and MQTT.

2.3. Berkeley Packet Filter (BPF). In 1993, Steven McCanne and Van Jacobson introduced a novel way of filtering packets in the kernel, and it is called Berkeley Packet Filter (BPF) [12]. The BPF has been widely used for network applications such as libpcap and tcpdump. The BPF defines a virtual filter machine and the corresponding language. The language contains several instructions regarding the fetching of data from packets, the performing of arithmetic operations, and data comparison. A filter is defined using these instructions. The virtual machine executes the filter to decide whether an incoming packet is acceptable or not (Figure 2).

A number of the BPF extensions [13–15] have been proposed to improve the BPF speed and expressivity, but they do not extend the BPF functionality. Even though the BPF is excellent for packet matching, functional extensions are needed to make it suitable for versatile packet processing such as packet modification or generation. Jouet et al. [16] proposed the application of the BPF to the SDN data plane, but they used only the BPF to match the incoming packets.

The Instruction-based Packet Processing (IPP) scheme [17] has been proposed for the use of the BPF in the provision of the data-plane programmability in SDN. The IPP language supports various data-processing features as well as those of its packet processing, and its data plane supports the run-time installation of new programs.

3. E-IPP Scheme

The aim of the E-IPP scheme is the introduction of a programmable data plane for SD-IoT devices. It adopts many of the features of the IPP scheme, and the IPP has been extended so that it can be used as an event-driven processing model, thereby enabling in-network data processing as well as packet processing in various IoT situations. This scheme consists of an event model, a language to program the procedures for application-specific processing, and a virtual machine to install and run the program.

3.1. An E-IPP Event Model. An E-IPP event comprises *an identifier*, *a class*, *a subclass*, and *option data*. The *identifier* uniquely identifies the E-IPP event, the *class* categorizes the E-IPP event, and the *subclass* defines a unique event type within a specific class. The <class, subclass> pair identifies

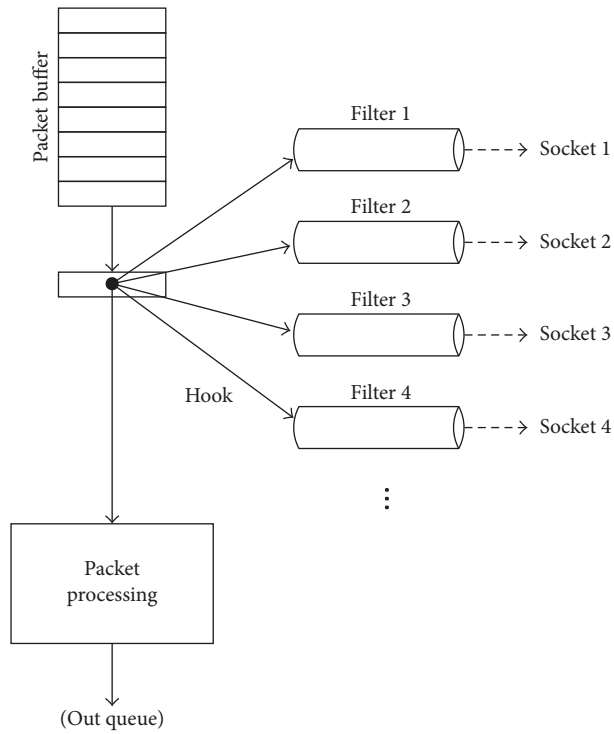

Figure 2: The concept of packet filtering in BPF.

a unique event type, and it indicates a specific event handler to deal with the event. The *option data* vary depending on the event type.

The two event categories are as follows: *system-defined* and *user-defined* classes. The system-defined class represents predefined event types such as the *init*, *packet-in*, and *timer* subclasses. The user-defined class includes application-specific event types that can be dynamically defined by IoT applications. To define a new event type, an IoT application assigns an unused event subclass that identifies the event type. Then, it registers the event subclass to the E-IPP VMs. The IoT applications may register their own event handlers, which includes their application-specific procedures, for both the system- or user-defined event types.

The E-IPP VM consists of several event-handling policies. Each event type is matched to one of the event-handling policies. A policy represents the number of event handlers that can be registered to an event type and the manner in which they are executed in the presence of more than one handler. The three policies are as follows: *singleton*, *sequential*, and *parallel*. The singleton policy is the representation of an event that carries an event-specific handler within itself. The init and timer event types represent the singleton policy. The sequential policy represents multiple event handlers that can be registered to an event type with their priorities, and the handlers will be executed according to their priority-based order. While the VM is executing the handlers, a proceeding handler may block the other handler executions of the lower priority. The packet-in event type is an example of the sequential-policy types. The parallel policy represents the independent execution of all of the registered

Table 1: Directives, new instructions, and predefined procedures.

Directive statement		*Description*
%import <exported symbol>		Import a external procedure
%addvar <var_name>		Declare an integer variable
%setarray <arr_name> <size>		Declare an array
%setinit <label>		Set the starting point of the initialization code
%defevent <ev_type> <size>		Define a new user-defined event type
Instruction	*Operands*	*Description*
out	port	Packet out to port
norm		Vender-dependent packet processing
wrt	offset	Write word A to packet offset
call	symbol	Call a external procedure
Ext procedure	*Scratch memory*	*Description*
reg_hnd	ev_type, label	Register an event handler
ureg_hnd	ev_type, label	Unregister an event handler
gen_ev	ev_type, data	Generate an event with given type and data
timer	expire, label	Register a timer event
npkt	size, null	Allocate a new packet buffer

handlers. The user-defined event types represent the parallel policy.

The events are generated from a variety of sources. The system-defined events are generated from the *loader*, the *input packet buffer*, and the *internal timer*, all of which are described later in this paper. The user-defined events are generated when an event generation code is executed, and the event-generation code may be placed in another event handler to concatenate the event handling.

3.2. E-IPP Language. The E-IPP code is written using the E-IPP language, which is a low-level language for the E-IPP VM. It consists of a series of *directives*, *instruction statements*, and comments. The directives direct the loader regarding the installation of the E-IPP code into the VM. The instruction statements are the symbolic machine codes of the E-IPP VM and are translated into executable codes by the loader.

The E-IPP language is derived from the BPF, especially the Linux Socket Filter (LSF) [18]. Accordingly, many of the BPF-language features have been inherited, such as the instruction-statement syntax, instruction set, scratch memory (addressable registers of a limited size), and BPF extensions (platform-dependent variables provided by the Linux kernel). The specialization of the original BPF, however, is regarding the packet matching, and it is not suitable for data processing and event handling. The E-IPP language extends the LSF to include the event-handling features. The directives and the predefined procedures that access the VM event-handling features are provided; the utilization of these allows for the defining of the new event types and event handlers that are registered to the VM. Further, new instructions are added to provide the modification and forwarding features of the packet to the LSF.

Table 1 show the directives, extended instructions, and library procedures that were used in the proposed examples. A user-defined event type can be defined by a *defevent* directive statement. The *defevent* directive statement consists of the event-type ID and the size of the option data, if any are present. The external libraries, which are imported by the *import* directive statement, provide a simple way to program complicated procedures with respect to a number of specific cases. The library procedures are called using the *call* instruction; its operand indicates a procedure using the symbol that is exported from the library. When a procedure is called, the scratch memory (a predefined array with the symbol "M" in the LSF) is used for the passing of the arguments. The *reg_hnd* procedure is used for the registeration of the event handler of a specific event type, and the event is generated by the *gen_ev* procedure. The generated event facilitates the scheduling of the registered event handlers of its type.

4. E-IPP VM

The structure of the E-IPP VM is derived from the IPP, but the behavior of each component changes on an event-driven basis. The E-IPP VM is composed of an *event scheduler*, an *execution engine*, a *loader*, *packet buffers*, and a *runtime storage* (Figure 3).

4.1. Event Scheduler. The event scheduler is a component of the VM that schedules the event handlers for the received events to the execution engine. The scheduler is composed of an event queue and an event-type table. When an event is generated, it is queued at the event queue. The event-type table stores the information for the event types and their event handlers. A table entry consists of an event-type field, an event-handling policy, and an ordered event-handler list.

When an event exits at the event queue, the event scheduler dequeues the event. The scheduler looks up the event-type table using the event type, and if a matching entry exists, it schedules the event handlers at the matching entry to the execution engine according to its policy. For the sequential policy, the scheduler links the event handlers of the list at the matching entry in the order of their priorities. Then, the scheduler places the ordered list into the wrapper handler that executes the handlers

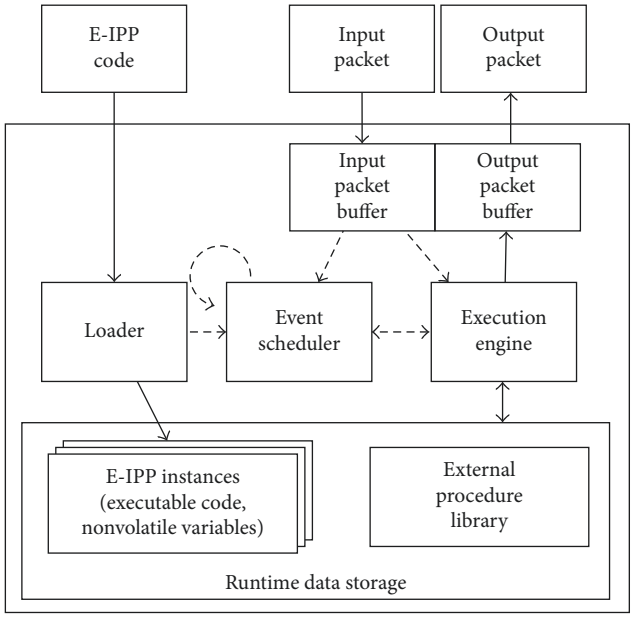

Figure 3: The E-IPP virtual machine architecture.

sequentially, and it may also terminate the execution prematurely. The wrapper is scheduled at the execution engine. In the case of the parallel policy, the scheduler schedules all of the event handlers in the list to the execution engine. In the case of the singleton policy, an event handler is not present in the event-type table entry; instead, the event itself carries its own handler identifier in its option data. The scheduler schedules the handler in the event to the execution engine.

4.2. The E-IPP Loader. The E-IPP loader installs the E-IPP code into the VM on behalf of the E-IPP application. When E-IPP code is received by the E-IPP loader, the loader compiles the code into the form that can be executed by the execution engine. The executable form of the E-IPP code is called the *E-IPP instance*. The E-IPP instance includes the runtime-data section for the nonvolatile data that are used to remember the state of the E-IPP instance. Then, the E-IPP loader stores the E-IPP instance at the runtime storage and generates an init event. The init event includes an initialization function that serves as the corresponding event handler. The initialization function usually registers new event types and event handlers to the scheduler.

4.3. Execution Engine. The execution engine runs the event handlers that are scheduled by the scheduler. The execution engine comprises an event-handler queue and a task queue. When an event handler is queued at its event-handler queue, the execution engine dequeues the event handler and assigns a task to it; then, the task is queued at its task queue. The execution engine runs the tasks of the task queue according to its scheduling policy. The default policy is first come, first served (FCFS).

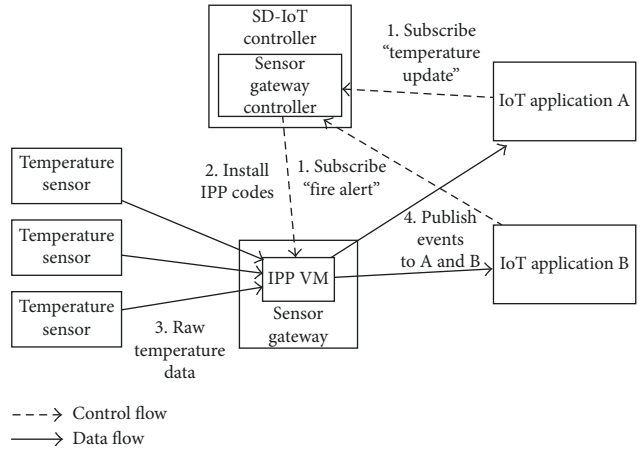

Figure 4: An example of SD-IoT applications.

4.4. Input and Output Packet Buffers and the Runtime Storage. The input and output buffers consist of their own packet queues and schedulers. When an incoming packet arrives at the E-IPP VM, the input-buffer scheduler receives it. It queues the packet into its packet queue and generates a packet-in event. The output buffer stores the outgoing packets, and the packets in the output buffer are transmitted outside of the VM by the output-buffer scheduler.

The runtime storage maintains the E-IPP instances and libraries for the external procedures. The libraries contain the predefined complicated procedures, and the procedures are called while the E-IPP instances are being executed.

5. An Example of SD-IoT Applications and Implemented E-IPP Codes

To explain the way in which IoT applications can utilize the E-IPP scheme, an example is now presented. For this example, it was assumed that three temperature sensors have been connected to a sensor gateway that is controlled by the SD-IoT controller. The sensors periodically send temperature data to the sensor gateway via the IEEE 802.15.4 protocol. Each packet that is sent from the sensors contains a *destPANId* and a *srcPANId* in the header and an action field and sensed data in the payload. The sensor gateway embeds the E-IPP VM as its data plane, and the E-IPP codes in the sensor gateway process the raw temperature data. The E-IPP codes send application-specific data to the IoT application that subscribes the application-specific data to the SD-IoT controller. Figure 4 shows the SD-IoT architecture that was used for the example.

The aim of the IoT application A is the attainment of the temperature data that are sent from the sensors, so it subscribed to the SD-IoT controller to receive the temperature-update events. The IoT application B is interested in the alert event that is regarding the exceeding of the temperature from one of the sensors beyond the set threshold of the application. When the IoT applications subscribe to the SD-IoT controller regarding their interests, the SD-IoT controller obtains the E-IPP codes for these interests and installs them in the E-IPP VM.

```
%import send_ip_msg     ; external procedure: send a event message
%defevent 0x000A 4      ; update_temperature event with 4byte data
%setarray event_data 4      ; sensor_id(2) + temperature(2)
%addvar sensor_id
%addvar temperature
%addvar destA
%setinit init

init:
   ld #0xC2800004     ; 192.128.0.4
   st destA
   ld PACKET_IN
   st M[0]
   ld PParser
   st M[1]
   call reg_hnd     ; adding "PParser" to "PACKET_IN" event
   ld #0x000A     ; update_temperature event
   st M[0]
   ld SendTemp
   st M[1]
   call reg_hnd     ; adding "SendTemp" to "update_temperature" event
   ret #0

; PACKET_IN event handler
PParser:
; check 802.15.4 MAC (destination_PANId==me(0x0001))
   ldh [3]
   jne #0x0001, end
   ldh [13]
   st sensor_id     ; store source PANId
; check the action field (action==update_temp(0x0010))
   ldh [29]
   jne #0x0010, end
   ldh [31]
   st temperature     ; store temperature
; make option data (sensor_id(2) + temperature(2))
   ldh sensor_id
   st event_data[0]
   ldh temperature
   st event_data[2]
; generate uptate_temperature event (args: event_type, data)
   ld #0x000A     ; event_type = update_temperature
   st M[0]
   ld event_data     ; data = event_data
   st M[1]
   call gen_ev
end:
   ret #0

; update_temperature event handler
SendTemp:
; prepare data to send event message
   ldh [0]
   st event_data[0]
   ldh [2]
   st event_data[2]
; call send_ip_msg (args: dest, event data size, event data)
   ld destA
   st M[0]
   ld #4
   st M[1]
   ld event_data
   st M[2]
   call send_ip_msg
   ret #0
```

Figure 5: E-IPP code for example application A: filtering packets and reporting to the server.

```
%addvar tThrshold
%addvar destB
%setinit init

init:
   ld #0xC2800005 ; 192.128.0.5
   st destB
   ld #0x0064
   st tThrshold
   ld #0x000A     ; update_temperature event
   st M[0]
   ld SendAlert
   st M[1]
   call reg_hnd     ; adding "SendAlert" to "update_temperature" event
   ret #0

; update_temperature event handler
SendAlert:
; check temperature in the event option data
   ldh [2]     ; load temperature
   jlt tThrshold, end     ; if (temperature < tThrshold) goto end
; call send_ip_msg (args: dest, event data size, event data)
   ld destB
   st M[0]
   ld #4
   st M[1]
   ld [0]     ; load event_data
   st M[2]
   call send_ip_msg
end:
   ret #0
```

Figure 6: E-IPP code for example application B: reporting to the server when certain conditions are met.

```
%addvar dest

; pkt sending procedure for publishing temperature events
; args: dest, event data size, event data
send_ip_msg:
   ld M[0]     ; load dest
   st dest
   ld M[1]     ; load event data size
   add #28     ; add UDP/IP header size
   add #4     ; add reporting time data size
   st M[0]
   npkt     ; allocate a new packet buffer
   ld #0x00000001
   wrt [12]     ;SrcIP=0x00000001
   ld dest
   wrt [16]     ;DestIP=dest

   ......

   ld cur_jiffies     ;reporting time
   wrt [38]
   ld M[2]     ;event_data (sensor ID and temperature)
   wrt [42]
   norm     ;send
   ret #0
```

Figure 7: E-IPP code for packet sending procedure.

Figures 5 and 6 show the simplified E-IPP codes that were installed for each of the subscriptions. In Figure 5, the first directive statement imports an external procedure named *send_ip_msg*. As shown in Figure 7, the arguments <*dest, event data size, event data*> are passed onto the procedure, and the procedure creates a new User Datagram Protocol (UDP)/Internet Protocol (IP) packet with the given event data, and the packet is then sent to the dest.

The other directive statements in Figure 5 define a new event type (update_temperature:0x000A), an array for the event option data, three variables (sensor_id, temperature, and destA), and the initialization code label (init). The initialization code sets the IP address of the application A server in the variable *destA*. Further, it registers the new event handlers, *PParser* and *SendTemp*, for the PACKET_IN and the update_temperature event types, respectively.

The PParser is the parser for the sensor packets. This parser uses the destPANId field to check whether the destination of the packet is the VM itself. The srcPANId is stored in the sensor_id. Next, the parser inspects the action field. If this field is *update temperature*, the following data are counted as the temperature data and are stored in the *temperature* variable. Then, it generates an *update_temperature* event. The sensor_id and the temperature are included in the event option data.

The SendTemp publishes the updated temperature to the application A server. It uses the imported procedure *send_ip_msg* to send a packet to the destA for which it shifts the option data from the update_temperature event to the sending-packet payload, thereby resulting in the sending of the sensor_id and the temperature.

The second E-IPP code (Figure 6) includes the procedures for the publishing of a fire alert to the application B. This code shares the update_temperature event that was generated by the first E-IPP code to reduce the packet-parsing load, and it checks the temperature data in the packet. If the temperature exceeds the threshold, it sends a fire alert to the application B server.

6. Comparison with Other Programmable Data Planes

This section presents the comparison of the proposed method with the other existing programmable data planes. The comparison subjects are the BPF, two methods that adopt the BPF (Jouet et al. and IPP, which are introduced in the Related Work section), and other existing programmable data planes that are introduced in the following subsection.

6.1. Existing Programmable Data Planes. A number of proposals have been presented to introduce programmability into the SDN data plane, such as P4 [19], open deeply programmable network node architecture (DPN) [20], ClickOS [21], and OpenState [22]. The P4 proposal suggests a propriety packet-processing language for SDN switch operations. The P4 language is targeted for the abstraction of parse-match-action pipeline operations in dedicated hardware. DPN and ClickOS are based on the Click modular router [23], and they use the Click module to program their data-plane operations. As the Click modular router does not support the run-time programmability, DPN and ClickOS also do not allow the addition of new actions in run-time. OpenState suggests a stateful data-plane model and it introduces an extended finite state machine (XFSM)

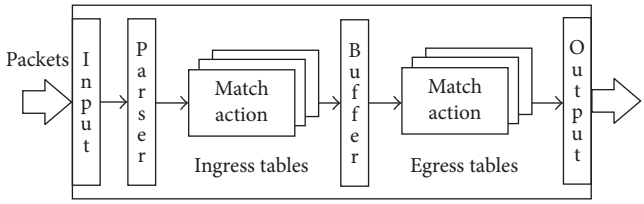

Figure 8: Abstracted forwarding model in P4.

for the data-plane programmability. The SDN controller defines the states and the events for flows, whereby the action for a state and an event are installed inside the network device. When an event occurs, the network device handles the event using the current flow state and the preconfigured actions. The data plane can be programmed to make a forwarding decision in consideration of the network situation without the controller intervention.

In spite of their programmable features, their languages are targeted to the table-based packet-processing models (e.g., forwarding model in Figure 8). They comprise ingress and egress tables, and the entire network programs are translated into packet-processing rules and are placed at one of these tables. The aim is the enabling of programmers so they can describe match-action tables that are dynamically populated by clearly articulated rules.

In the IoT environment, the table-based model is not flexible enough to cover all of the requirements of the IoT applications; for example, in the case where a sensor-management application needs to register a timer event on a sensor gateway node for a dead-sensor-indication procedure. Furthermore, in the presence of multiple sensor-status-monitoring clients, the gateway node generates an internal user-defined event for the dead-sensor indication to activate multiple notification procedures for each of the subscriber clients. It is obvious that these notification procedures are not packet-processing procedures, and the existing table-based packet-processing model is not capable in this case. The requisite processing model is sufficiently flexible to process these procedures as required.

6.2. Comparison of the Programmable Data Planes. Table 2 shows whether the programmable features of each of the data planes are those that are required by the various IoT applications. All of the scheme features allow for the arbitrary packet matching that is required by the IoT applications for the programming of new protocols. Among them, however, BPF and the scheme of Jouet et al. cannot modify and forward packets, because the corresponding researchers only considered packet-filtering cases.

DPN, ClickOS, and OpenState show weaknesses in terms of the data-processing features. These weaknesses of DPN and ClickOS are inherited from the simple click-module structure they use, and the focus of OpenState is the extension of the flow table; moreover, data-processing cases have not been considered. Alternatively, for P4, the programming flexibility for data processing has been considered, even though it adopts the table-based model. BPF and

Table 2: Comparison with other programmable data planes.

Features for IoT applications	P4	DPN	ClickOS	OpenState	BPF	Jouet et al.	IPP	E-IPP
Programmability for arbitrary packet matching	O	O	O	O	O	O	O	O
Programmability for packet modification and forwarding	O	O	O	O	X	X	O	O
Programmability for complicated data-processing operations	O	X	X	X	O	O	O	O
Support nonvolatile storage (for data aggregation)	O	O	O	O	Δ	Δ	O	O
Support timer event processing (for periodic reporting)	X	O	O	O	X	X	X	O
Support user-defined event processing (for publish-subscribe between multiple applications)	X	X	X	X	X	X	X	O
Runtime installation of new code	O	X	X	X	X	X	O	O

all of the other BPF-based methods comprise data-processing advantages, because they abstract the processor operation and consist of a number of the requisite instructions of data-processing operations.

Non-volatile storage is used for network status management or data aggregation. All of the methods use the non-volatile storage, because it is essential for basic operations such as the packet counter in network equipment, but the BPF-storage size for each filter is very small. This size was improved for IPP and E-IPP so that programmers can freely create a variable or an array within their code.

Timer-event processing is an important feature in many IoT-application cases, such as the periodical reporting of aggregated data or the finding of dead sensors. P4 does not comprise this feature, but DPN, ClickOS, and OpenState include the timer-event handling for packet retransmission and other purposes. BPF and the other existing BPF-inherited schemes do not comprise timer-related instructions, but the proposed scheme includes the timer-event-handling feature.

In situations where multiple IoT applications install their own code into the VM, user-defined events are necessary for the interworking between the various codes. This feature allows the handlers that are registered in the application code to receive the events that are generated from another application code, thereby allowing them to function as the subscriber code and the publisher code, respectively. This is the feature only the proposed scheme.

The run-time installation is the essential feature of dynamic provisioning, which is one of the advantages of the SD-IoT that is inherited from SDN. When an IoT application seeks a new service, the SD-IoT network needs to install the application code into the data plane at runtime so that it can be immediately reflected in the SD-IoT network. This feature is considered for P4, IPP, and E-IPP.

7. Conclusion

This paper contains the proposal of an SD-IoT data-plane scheme that is specialized for in-network data processing. It is based on a generic event-driven model, and it is suitable for the handling of the various requirements of in-network data processing as well as those of normal packet forwarding. It also includes a language for the defining of the user-defined procedures of the description of context-specific operations and their dynamic-installation interfaces. The presented examples show that the SD-IoT application can program the requisite data-aggregation and DCPS-agent functions of the SD-IoT gateway using the proposed scheme. These examples mean that the SD-IoT application can add in-network data processing to run-time services without the deployment of any additional network entities, and this is simply achieved by the implementation of several lines of code onto the data plane. The comparison with other programmable data planes shows that the features of the proposed scheme are suitable for various IoT applications.

Disclosure

This is an extended and revised version of a preliminary conference report that was presented at the 8th International Conference on Information and Communication Technology Convergence (ICTC) in 2017 [24].

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by the MIST (Ministry of Science and ICT), Korea, under the National Program for Excellence in SW (2015-0-00936) supervised by the IITP (Institute for Information and Communications Technology Promotion).

References

[1] A. Hakiri, P. Berthou, A. Gokhale, and S. Abdellatif, "Publish/subscribe-enabled software defined networking for efficient and scalable iot communications," *IEEE Communications Magazine*, vol. 53, no. 9, pp. 48–54, 2015.

[2] Y. Jararweh, M. Al-Ayyoub, A. Darabseh, E. Benkhelifa, M. Vouk, and A. Rindos, "Sdiot: a software defined based internet of things framework," *Journal of Ambient Intelligence and Humanized Computing*, vol. 6, no. 4, pp. 453–461, 2015.

[3] J. Liu, Y. Li, M. Chen, W. Dong, and D. Jin, "Software-defined internet of things for smart urban sensing," *IEEE Communications Magazine*, vol. 53, no. 9, pp. 55–63, 2015.

[4] S. Sasirekha and S. Swamynathan, "Cluster-chain mobile agent routing algorithm for efficient data aggregation in

wireless sensor network," *Journal of Communications and Networks*, vol. 19, no. 4, pp. 392–401, 2017.

[5] G. P. Gupta, M. Misra, and K. Garg, "An energy efficient distributed approach-based agent migration scheme for data aggregation in wireless sensor networks," *Journal of Information Processing Systems*, vol. 11, no. 1, p. 148, 2015.

[6] Z. Zhang, C. Jin, M. Li, and L. Zhu, "A perturbed compressed sensing protocol for crowd sensing," *Mobile Information Systems*, vol. 2016, Article ID 1763416, 9 pages, 2016.

[7] O. Adil Mahdi, A. W. Abdul Wahab, M. Y. I. Idris, A. Abu Znaid, Y. R. B. Al-Mayouf, and S. Khan, "Wdars: a weighted data aggregation routing strategy with minimum link cost in event-driven wsns," *Journal of Sensors*, vol. 2016, Article ID 3428730, 12 pages, 2016.

[8] G. Pardo-Castellote, "Omg data-distribution service: architectural overview," in *Proceedings of the 23rd International Conference on IEEE Distributed Computing Systems Workshops*, pp. 200–206, Providence, RI, USA, May 2003.

[9] A. Banks and R. Gupta, *Mqtt Version 3.1.1*, Vol. 29, OASIS Standard, Burlington, MA, USA, 2014.

[10] J. Swetina, G. Lu, P. Jacobs, F. Ennesser, and J. Song, "Toward a standardized common M2M service layer platform: Introduction to oneM2M," *IEEE Wireless Communications*, vol. 21, no. 3, pp. 20–26, 2014.

[11] T. E. Foundation, "Eclipse IoT Project," 2018, https://iot.eclipse.org/.

[12] S. McCanne and V. Jacobson, "The bsd packet filter: a new architecture for user-level packet capture," *USENIX Winter*, vol. 93, 1993.

[13] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis, "xPF: packet filtering for low-cost network monitoring," in *Proceedings of the IEEE Merging Optical and IP Technologies on High Performance Switching and Routing*, pp. 116–120, Kobe, Japan, May 2002.

[14] H. Bos, W. De Bruijn, M.-L. Cristea, T. Nguyen, and G. Portokalidis, "FFPF: fairly fast packet filters," in *OSDI*, vol. 4, p. 24, 2004.

[15] Z. Wu, M. Xie, and H. Wang, "Swift: a fast dynamic packet filter," in *NSDI*, vol. 8, pp. 279–292, 2008.

[16] S. Jouet, R. Cziva, and D. P. Pezaros, "Arbitrary packet matching in OpenFlow," in *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, pp. 1–6, Budapest, July 2015.

[17] K. Ki-wook, "Ipp project," 2016, http://wowook1.github.io/IPP-engine-for-ovs/.

[18] J. Schulist, D. Borkmann, and A. Starovoitov, "Linux socket filtering aka berkeley packet filter (bpf)," 2016, http://www.kernel.org/doc/Documentation/networking/filter.txt.

[19] P. Bosshart, D. Daly, G. Gibb et al., "P4: programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[20] A. Nakao, "Flare: open deeply programmable switch," in *Proceedings of the 16th GENI Engineering Conference*, Salt Lake City, Utah, March 2012.

[21] J. Martins, M. Ahmed, C. Raiciu et al., "ClickOs and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pp. 459–473, Seattle, WA, USA, April 2014.

[22] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: programming platform-independent stateful OpenFlow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[23] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[24] K.-W. Kim, S.-G. Min, and Y.-H. Han, "A programmable data plane to support in-network data processing in software-defined iot," in *Proceedings of the 2017 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 855–860, Jeju Island, South Korea, 2017.