IoT Device Virtualization for Efficient Resource Utilization in Smart City IoT Platform

Keigo Ogawa, Kenji Kanai, Kenichi Nakamura, Hidehiro Kanemitsu, Jiro Katto and Hidenori Nakazato
Waseda University, Waseda Research Institute for Science and Engineering
3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-0072, Japan
{k_ogawa, kanai, katto}@katto.comm.waseda.ac.jp

***Abstract*—To develop and interoperate smart city applications efficiently, smart city IoT platforms require efficient handling of various types of sensor devices, networking and computing resources, and different domain applications. To address this fact, in this paper, we introduce an IoT device virtualization that enables efficient utilization of computing resources. The proposal applies a micro-service sharing and dynamic resource scaling. In the performance validations, we implement an early prototype using Docker, Kubernetes, and Apache Kafka. Through the preliminary experiment, we confirm that the proposal can improve the application processing time by appropriately sharing and scaling micro services.**

Keywords—IoT platform; smart city; micro service; virtualization; resource management

I. Introduction

The Internet of Things (IoT) has become popular in academic and industry areas owing to evolutions of cloud computing and sensor devices. For IoT use cases, smart cities are one of the suitable candidates. Currently, smart city (or IoT) application developers individually install various sensor devices and arrange networking and computing resources to collect and analyze the sensor data. This results in excessive costs for smart city application providers and smart city owners.

To efficiently develop and interoperate smart city applications, recently, standardizations of IoT platforms, such as oneM2M [1] and FIWARE [2], are ongoing. The IoT platform (or IoT) requires an efficient handling of various types of sensor devices, networking and computing resources, and different domain applications, such as energy management, transportation management, and healthcare [3].

To address this requirement, the authors of this paper propose a research project named "Fed4IoT" [4], which is an acronym for federation of IoT and cloud infrastructures, to provide scalable and interoperable smart city applications. This project is a collaboration between EU and JP, and its objective is to deploy the proposed Fed4IoT platform to actual EU and JP smart cities and to improve the quality of smart city applications, such as city surveillance. Based on this motivation, the project primarily proposes two key technologies: IoT device virtualization and context-information sharing.

In this paper, to improve the processing (or response) time for smart city applications, we introduce the IoT device virtualization (one of the key technologies in Fed4IoT) that enables efficient utilization of computing resources. To validate the performance, we implement a prototype of the proposal using Docker and Kubernetes and confirm the advantage of the proposal through preliminary evaluations.

II. Related Work

Several researchers and application providers consider the utilization of edge/fog computing to provide IoT (or smart city) applications [5-7]. Because computing resources will be installed at physical proximity to end users, the edge/fog computing can provide low-latency and location-aware processing. Pre-processing and caching in the edge is one of realistic services of edge/fog computing. In addition, to provide further efficient processing, cooperation between edge/fog and cloud is mandatory [8].

To manage the various requirements of networking and computing resources, virtualization techniques, such as software-defined network (SDN), network function virtualization (NFV), and container and virtual machines (VM) are indispensable. In hardware virtualization, numerous researchers are attempting container-based and VM-based virtualizations, where Docker and OpenStack are typical examples. In particular, the container-based virtualization (Docker) can generate a container that includes application-specific environments such as libraries, and such a container does not affect other containers (i.e., isolation). In addition, it is well-known that the container has less overhead compared with the VM-based virtualization. This means that the container can scale a configuration of the virtualized hardware considerably easily and flexibly. Kubernetes is a well-known open-source container orchestration software to manage Docker containers. Recently, several researchers have developed IoT applications using Docker [9-12].

Based on these technologies, to improve the application response time (i.e., to reduce processing time), in this paper, we propose an IoT device virtualization technique that enables efficient utilization of computing resources using Docker and Kubernetes.

III. IoT Device Virtualization

A. Concept

A concept of IoT device virtualization is illustrated in Fig. 1. The primary objective of IoT device virtualization is to share IoT devices and service functions (or micro services) among different smart city applications. Although this concept is similar to FogFlow [13], FogFlow does not consider a device sharing among different applications and dynamic resource scaling. Unlike FogFlow, the proposal considers the micro-service sharing and dynamic resource scaling to improve the computing resource utilization and the application quality. As shown in the figure, the proposal adds two nodes (gateway nodes and function nodes) between the devices and applications. Gateway nodes play a role of pooling the raw data generated via sensors, and function nodes perform a (pre-)processing of the raw data, such as averaging sensor values or image processing

of a surveillance camera. As the platform opens common APIs, such as REST, to access the function nodes, using the APIs, the application providers are no longer concerned with connecting to the device itself, and the function nodes broker collecting and processing the sensor data. This fact enables the efficient sharing of IoT devices among different applications. In addition, applying the service (or network) function virtualization technique to the function nodes, these nodes can provide considerably flexible and scalable micro services to application providers. The detailed explanations are provided in the next subsection.

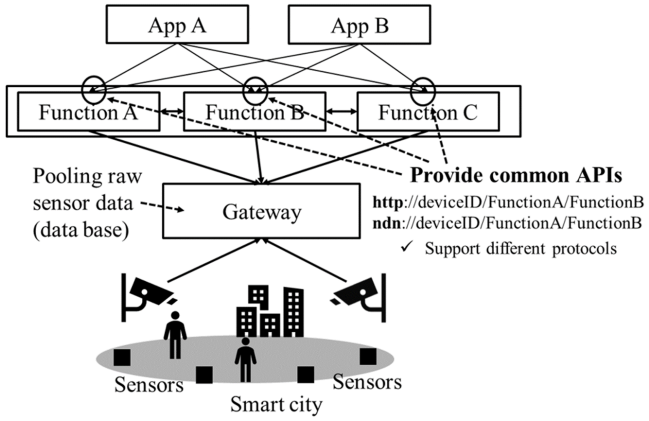

Fig. 1. Concept image of IoT device virtualization

B. Micro-service deployment, scaling, and sharing

A concept image of micro-service deployment, scaling, and sharing is illustrated in Fig. 2. As shown in the figure, the functions represent the micro services, such as obtaining the sensor data and performing human detection.

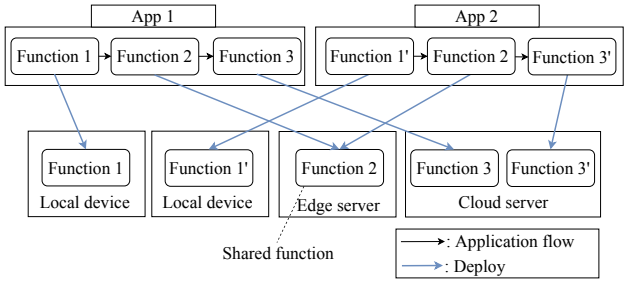

Fig. 2. Concept image of micro-service deployment, scaling, and sharing

First, in the micro-service deployment, to use the computing resources efficiently, each micro service (or function) should be deployed to the appropriate computing resource, such as a local device, edge/fog, and cloud. In addition, a deployment algorithm or policy is referenced from FogFlow and the previous research efforts on the resource allocation and optimization.

Second, in the micro-service scaling, to improve the resource utilization and application quality, the micro service should be scaled dynamically. This is because the micro services often run under dynamic changing environments and different application requirements. In addition, the micro services also have different resource requirements (data volume and processing load). Thus, the micro-service scaling should be controlled according to the application's daily (or historical) behaviors. We assume that such application behaviors can be predicted by monitoring the resource utilization and applying machine learning; however, this will be part of our future work.

Third, in the micro-service sharing, to further improve resource utilization and sharing of IoT devices, the micro-services should be shared among different (domain) applications. Because the micro services are deployed by the container, the micro services can be easily shared. In addition, because the container-based micro services are isolated, the micro services are easily and safely migrated to different computing nodes (local, edge/fog, and cloud). Such scheduling can also be optimized by monitoring the historical resource usages and applying machine learning as similar to the micro-service scaling.

C. Communication protocol

To exchange raw sensor data and intermediate results between the function nodes and the IoT devices, we apply a publish/subscribe (pub/sub)-based messaging model, such as Apache Kafka and MQTT as shown in Fig. 3. Because the function nodes will be shared or chained to each other, the communication paths will be changed dynamically. In the pub/sub-based messaging model, the communication paths can be easily managed by topics. When the function nodes are deployed, a broker node (or an orchestration node) specifies the basic publish topics and subscribe topics. By chaining the function nodes, the topic is named via simply lining the function name and separating the specific character: "http://DeviceID/FunctionA/FunctionB." This naming scheme can be quite similar to a naming scheme of Content Centric Networking (CCN) [14]; thus, we assume that CCN or related networking technologies can also be one of the suitable candidates with regard to the communication protocol.

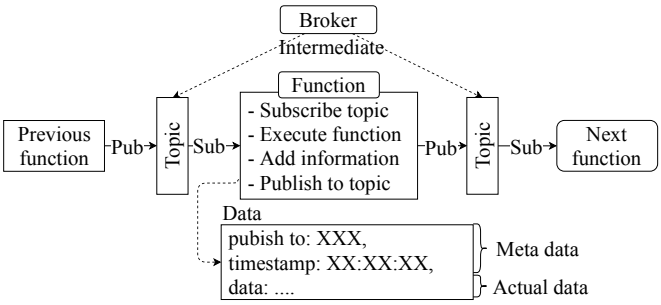

Fig. 3. Concept image of Pub/Sub-based function chaining

IV. Performance Evaluations

A. Experimental environment

To evaluate the concept of our proposal, we implement an early prototype using Docker, Kubernetes, and Apache Kafka. Using Docker, we implement the micro services, such as capturing images from cameras and performing object/human detection. Using Kubernetes, we deploy and scale the resources for the container-based micro services. Using Apache Kafka, we establish the connections between IoT devices and the function nodes and control the micro-service chaining. The detailed experimental environment is illustrated in Fig. 3.

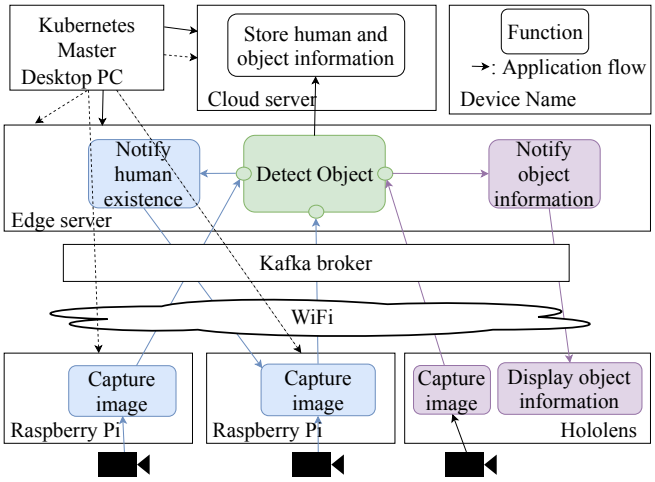

Fig. 4. Experimental environment of our implementation

As shown in the figure, we employ one desktop PC as a Kubernetes master (CPU: Intel Core i7-4770T 2.50 GHz, Memory: 16 GB), one cloud server as a Kubernetes node (CPU: Intel Xeon E5-2650 v3 2.30 GHx, Memory: 220 GB), one edge server as Kubernetes node (CPU: Intel Core i7-7700 3.60 GHz, Memory: 32 GB, GPU: NVIDIA GeForce GTX 1080Ti), two Raspberry Pi 3 B+ with USB cameras as IoT devices (camera A: resolution is 640×480, camera B: resolution is 1280×720). These nodes are placed in the laboratory, and the Kubernetes master is connected to the edge server via the Gigabit ethernet cable. In addition, the Kubernetes master creates a virtual network for Kubernetes nodes and manages (and scales) the Docker-based micro services. Furthermore, we employ HoloLens as a user device. HoloLens has an embedded camera and can transmit/receive image data. It should be noted that every IoT device node (Raspberry Pies and HoloLens) is connected to the Apache Kafka broker via the IEEE 802.11ac Wi-Fi network.

B. *Smart city application*

In the performance evaluations, we assume an image processing as a smart city application and employ two applications. In each application, micro services are containerized using Docker in advance.

1) Indoor surveillance using two fixed cameras

The first application is an indoor surveillance using two fixed cameras. This application uses two USB cameras A and B that have different video resolutions (as introduced in the previous subsection) and comprises of six steps as follows:

a) Obtain one image from camera A and transmit the image for every 350 ms.

b) Conduct object detection from the image of camera A, and send the detection result.

c) If a human is detected, send a message for activating camera B (i.e., notification of human existence)

d) After step c), wait for 2 s, obtain one image from camera B and transmit the image.

e) Perform human detection from the image of camera B and send the detection result.

f) Integrate and transmit the results to the cloud server.

It should be noted that the image is continuously generated from camera A and transmitted to the micro service that has a capability of object detection every 350 ms.

2) Object notification using HoloLens

The second application is object notification using one mobile camera, HoloLens, and comprises of four steps as follows:

a) Obtain an image from HoloLens and transmit the image every 1 s.

b) Conduct object detection from the image of HoloLens and then send back the results to HoloLens.

c) If objects are detected, then send back the results to HoloLens. (i.e., object notification)

d) Display the object information based on the received results.

It should be noted that, as differed from the previous application, the image is continuously generated from HoloLens and transmitted to the micro service that has a capability of object detection in every 1 s.

In these applications, the steps (1.a) and (1.d) implement two Docker containers that have a similar configuration but are deployed in different nodes. Further, the steps (1.b), (1.e), and (2.b) are implemented to the same Docker container, and this means that the same Docker container is shared among these steps. As shown in Fig. 4, a sharing Docker container named "Detect Object (provided by YOLOv3 [15])" is illustrated using green color, and the Docker containers used in the first and second applications are illustrated via blue and purple colors, respectively.

C. *Evaluation scenarios*

In the evaluation, we validate the performance of dynamic computing (CPU) resource scaling according to the application behaviors. Because the two applications share the micro service called "Detect Object", and a processing time for "Detect Object" significantly depends on allocated CPU resources and number of running cameras, the Kubernetes master will primarily scale this Docker container. In advance, we evaluate the performance of the processing time under different CPU resources and number of cameras, and confirm the results as shown in Table I. Experimental time is 300 s and the processing time demonstrates the average values. It should be noted that 1900 millicores represent 190% CPU usage. Based on the observation, we set an evaluation scenario as shown in Table II. We set a scenario where the CPU resource is always allocated 3000 millicores for comparison.

Table I: Average processing time under different cameras and CPU resources

Number of cameras	CPU (millicores)	Proc. time (s)
Camera A	1900	1.01
Camera A + B	4500	1.18
Camera A + B + Hololens	5500	1.43

Table II: Evaluation scenario

Time (min)	CPU (millicores)	Number of cameras
0 – 1	2000	Camera A
1 – 2	3000	Camera A + HoloLens
2 – 3	5000	Camera A + B
3 – 4	7000	Camera A + B + HoloLens

D. Results

Fig. 5 demonstrates the result of processing time as we set CPU resources always to 3000 millicores (i.e., comparison case: no efficient dynamic resource scaling case). As shown in the figure, the processing time is increased according to the increase in the number of devices. This is because the function resource is insufficient against the application requirements (as shown in Table I), and the images waiting for processing will be queued. From the viewpoint of resource provisioning, the resource is allocated excessively (i.e., over provision) during the first 120 s, and vice versa (i.e., under provision) during the next 120 s.

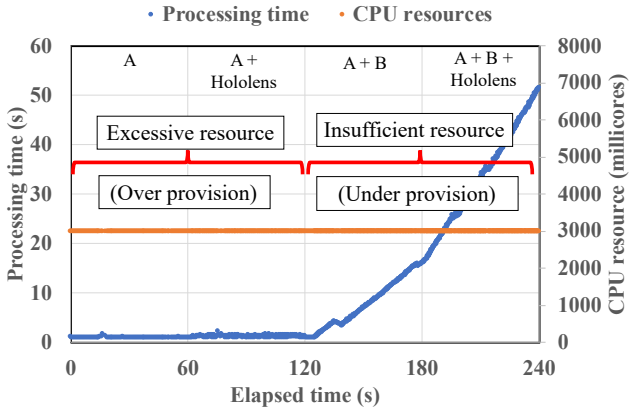

Fig. 5. Results of processing time with a constant resource allocation (comparison)

Further, Fig. 6 demonstrates the results of processing time in our proposal. From the figure, the results indicate that the processing time can be reduced as the CPU resource can be allocated appropriately. Thus, the results conclude that the proposal that enables utilization of efficient computing resource will potentially improve application quality.

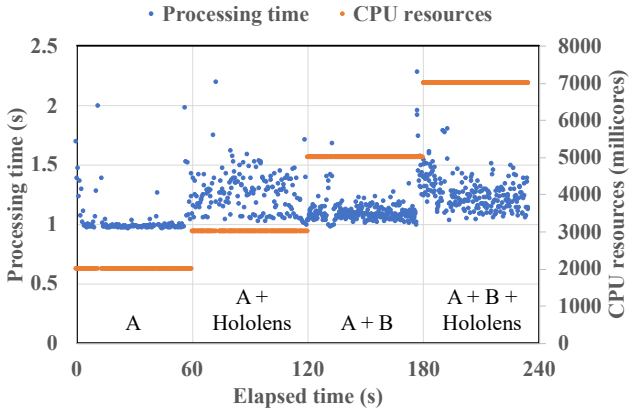

Fig. 6. Results of processing time with dynamic resource scaling (proposal)

V. Conclusions And Future Work

In this paper, we introduced an IoT device virtualization that efficiently utilizes computing resources to improve processing time for smart city applications. In the IoT device virtualization, the IoT devices and service functions (or micro services) are shared among different smart city applications. To improve the resource utilization and application quality, the proposal will scale the resources (or configurations) of micro services dynamically according to the application requirements and behaviors (e.g., historical resource usage). In the performance validations, we implemented an early prototype using Docker, Kubernetes, and Apache Kafka. Through the preliminary experiment, we confirmed that the proposal can potentially improve the application processing time by sharing and scaling micro services appropriately. In the future, we will evaluate the proposed performance in relatively large-scale environments. We will implement more realistic smart city applications and deploy the proposal to actual smart cities. In addition, we will propose prediction of the transition of resource utilization by applying machine learning.

Acknowledgments

This work was supported by the EU-JAPAN initiative by the EC Horizon 2020 Work Programme (2018-2020) Grant Agreement No. 814918 and Ministry of Internal Affairs and Communications "Federating IoT and cloud infrastructures to provide scalable and interoperable Smart Cities applications, by introducing novel IoT virtualization technologies (Fed4IoT)".

References

[1] oneM2M [online]: http://www.onem2m.org/

[2] FIWARE [online]: https://www.fiware.org/

[3] A. Banafa, "Three Major Challenges Facing IoT", IEEE IoT Newsletter, Mar. 2017.

[4] Fed4IoT [online]: https://fed4iot.org/

[5] F. C. Delicato, P. F. Pires, and T. Batista, "The Resource Management Challenge in IoT", Resource Management for Internet of Things, Springer Briefs in Computer Science, Springer, Cham, 2017.

[6] J. Pan and J. McElhannon, "Future Edge Cloud and Edge Computing for Internet of Things Applications", IEEE Internet of Things Journal, vol. 5, no. 1, pp. 439–449, Feb. 2018.

[7] C. Christian, P. Andrei, W. Gary, and G. Siobhan, "The Right Service at the Right Place: A Service Model for Smart Cities", IEEE Percom, pp. 1–10, 2018, Mar. 2018.

[8] C. Perena, Y. Qin, J. C. Estrella, S. R.-Marganiec, and A. V. Vasilakos, "Fog Computing for Sustainable Smart Cities: A Survey", ACM Computing Surveys, vol. 50, issue 3, no. 32, Oct. 2017.

[9] M. In, X. Wan, L. Xiao, Y. Chen, M. Xia, D. Wu, and H. Dai, "Learning-Based Privacy-Aware Offloading for Healthcare IoT with Energy Harvesting", IEEE Internet of Things Journal, Oct. 2018.

[10] K. Khanda, D. Salikhov, K. Gusmanov, M. Mazzara, and N. Mavridis, "Microservice-based IoT for Smart Buildings", International Conference on Advanced Information Networking and Applications Workshops, pp. 302–308, Taipei, 2017.

[11] R. Morabito, R. Petrolo, V. Loscri, and N. Mitton, "Enabling a lightweight Edge Gateway-as-a-Service for the Internet of Things", NOF 2016-7th International Conference on Network of the Future, Nov. 2016.

[12] J. Rufino, M. Alam, J. Ferreira, A. Rehman, and K. F. Tsang, "Orchestration of Containerized Microservices for IoT using Docker", 2017 IEEE International Conference on Industrial Technology, pp. 1532–1536, Toronto, 2017.

[13] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa, "FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities," IEEE Internet of Things Jounal, vol. 5, Issue 2, pp. 696–707, Apr. 2018.

[14] V. Jacobson, D. K. Smetters, J. D. Thornton, M. Plass, N. Briggs, and R. Braynard, "Networking named content", Commun. ACM, 55, 1, pp. 117–124, 2012.

[15] YOLO: Real-Time Object Detection [online]: https://pjreddie.com/darknet/yolo/