PAPER Special Section on Fusion of Network Virtualization/Softwarization and Artificial Intelligence towards Beyond-5G Innovative IoT Services

Empirical Study of Low-Latency Network Model with Orchestrator in MEC**

Krittin INTHARAWIJITR[†*a)], ***Nonmember***, **Katsuyoshi IIDA**[††b)], ***Senior Member***, **Hiroyuki KOGA**[†††c)], ***Member***, ***and*** **Katsunori YAMAOKA**[†d)], ***Senior Member***

SUMMARY The Internet of Things (IoT) with its support for cyber-physical systems (CPS) will provide many latency-sensitive services that require very fast responses from network services. Mobile edge computing (MEC), one of the distributed computing models, is a promising component of the low-latency network architecture. In network architectures with MEC, mobile devices will offload heavy computing tasks to edge servers. There exist numbers of researches about low-latency network architecture with MEC. However, none of the existing researches simultaneously satisfy the followings: (1) guarantee the latency of computing tasks and (2) implement a real system. In this paper, we designed and implemented an MEC based network architecture that guarantees the latency of offloading tasks. More specifically, we first estimate the total latency including computing and communication ones at the centralized node called orchestrator. If the estimated value exceeds the latency requirement, the task will be rejected. We then evaluated its performance in terms of the blocking probability of the tasks. To analyze the results, we compared the performance between obtained from experiments and simulations. Based on the comparisons, we clarified that the computing latency estimation accuracy is a significant factor for this system.

key words: *mobile edge computing, low-latency network, prototype, and experiment*

1. Introduction

Cyber-physical systems (CPSs) are composed of Internet of Things (IoT) devices in the physical world and computation units in cyberspace in 5G and Beyond-5G. One specific type of services is about latency-sensitive applications, such as virtual/augmented reality (VR/AR), on-line gaming, and remotely controlled robots [2]–[4]. They require a guarantee of latency and resources from the networks to satisfy their quality requirements. However, they usually take up significant cloud computing computational resources. Communications with the cloud may have a long latency because cloud servers are sometimes far from the CPS system. According to [5], 4G cellular networks cannot support low-latency CPS services, so new technology should be developed for them.

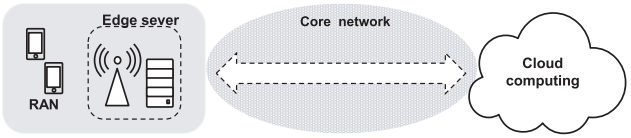


Fig. 1 Mobile edge computing.

Many studies have proposed low-latency network architectures in which cloud capabilities are distributed to areas close to users. For instance, fog computing and cloudlets [6] can be used to create a virtual server at the edge of a network for supporting latency.

The European Telecommunications Standards Institute (ETSI) has standardized a technology called *mobile edge computing* (MEC) [6], [7], a.k.a. multi-access edge computing, which provides local services by placing servers at the base stations of cellular networks (Fig. 1). The servers allocate resources to mobile devices over the radio access network (RAN), and their domain encompasses the cloud computing. When mobile applications require more resources or global services, they can normally access the cloud computing through the core networks. A report [8] showed that the concept of MEC can satisfy the requirements of IoT and CPS, especially those of network latency and mobility. However, MEC [9], [10] has been questioned from various aspects such as energy efficiency, network traffic, and low-latency network services.

To enable low-latency network services in MEC, there exist some research and development efforts. For example, mathematical models of MEC systems have been constructed, and they are evaluated by simulation [11]–[13]. In particular, we have proposed an MEC system and its model that guarantees the latency of the offloading task and evaluated the system by simulations [14]–[16]. Our simulation results showed that how our system can guarantee the latency of the computing tasks, namely, they revealed the characteristics of performance metrics and provided basic guides to select the parameters such as the controller interval.

However, simulations are not enough for actual devel-

Manuscript received April 12, 2020.
Manuscript revised July 16, 2020.
Manuscript publicized September 1, 2020.

[†]The authors are with the Dept. of Information and Communications Engineering, Tokyo Institute of Technology, Tokyo, 152-8552 Japan.
[††]The author is with the Information Initiative Center, Hokkaido University, Sapporo-shi, 060-0811 Japan.
[†††]The author is with the Dept. of Information and Media Engineering, University of Kitakyushu, Kitaskyushu-shi, 808-0135 Japan.
[*]Presently, with the Rakuten Mobile, Inc.
[**]Earlier version of this paper has been presented in [1].
a) E-mail: krittin.in@gmail.com
b) E-mail: iida@iic.hokudai.ac.jp
c) E-mail: h.koga@kitakyu-u.ac.jp
d) E-mail: yamaoka@ict.e.titech.ac.jp
DOI: 10.1587/transcom.2020NVP0005

opments, because they are based on ideal assumptions. Un expected issues arise in real networks. For example, the operating system of the edge server may create unexpected latencies. Due to the complexity of real environments, a simulation cannot accurately represent a whole system. Thus, we can say that an MEC implementation requires an evaluation on a real system to confirm that it operates as intended.

Here, we tackle the next step of MEC research, i.e., testing it in a real network environment. First, we designed a testbed based on our previous study. Then, we developed it in a real network environment. The communication of each entity in MEC was established through the core network. The computations of each process were executed by a real server. In terms of the latency guarantees, we evaluated our proposal by comparing it with simulations alone and analyzed the results of the simulation and the implementation.

This paper is organized as follows. Section 2 explains the low-latency network architecture and related work. Section 3 summarizes our previous study. Section 4 describes the logic and design of the system. Section 5 analyzes numerical results. Section 6 concludes the paper.

1.1 Contribution

Even though this work is extended from the previous work that is a simulation study, this study reveals aspects that simulation studies could not reveal. It describes the design and implementation of MEC architecture guaranteeing the latency performance. And we compare the performance between the simulation results and the numerical results obtained from the real testbed of MEC.

- As a proof-of-concept system, a design of an MEC implementation, which is based on the system in [15], [16], is proposed. Specifically, based on the standardized ETSI MEC architecture, we extracted necessity components required by guaranteeing the latency. We also implemented a method of communication latency measurements and designed and implemented an extended version of the equation about computing latency estimation.
- We evaluated the performance of MEC for specific server capacity and network conditions. They could be a guide for future development of MEC when the system requires higher performance.
- The presented analysis reveals an issue of MEC that could not be seen in the simulation. Namely, we showed a part of the answer for a question about the different tendency between the experimental and simulation results. Through the analysis of the computational latency distribution of the experiments and simulations, we clarified that the computing latency estimation accuracy is a significant factor causing this difference.

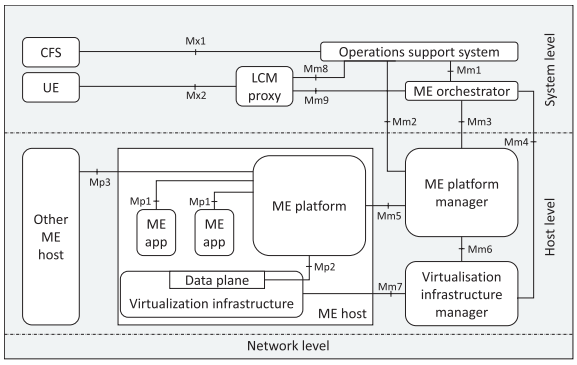


Fig. 2 MEC architecture [7].

2. Low-Latency Network Architecture

According to [3], [4], CPS requires a low-latency network architecture to achieve its objective. MEC is a technology that can provide low-latency network services and distributed computing resources. It will be implemented in 5G and in Beyond-5G.

MEC is being developed into the real world, and ETSI has standardized its architecture [7]. The framework proposed by ETSI has three levels: system, host, and network. The system level is for managing the MEC host level and contact with users and third-party operations. The host level contains an MEC host over the virtual infrastructure to provide services. The network level maintains connections with external entities.

The reference architecture of ETSI takes into account the system and host levels of the framework. Figure 2 illustrates the entities of each level. Generally speaking, the mobile edge system level uses a mobile edge (ME) orchestrator to manage the host level. It monitors resources in mobile edge hosts and commands a host manager to start or end services. User equipment (UE) applications connect to the orchestrator through a proxy in order to request a server, a.k.a. an edge host.

The mobile edge host level consists of an ME platform manager (MEPM), virtualization infrastructure manager (VIM), and ME host. The MEPM mainly manages the ME host according to service rules from the orchestrator. The ME host operates an ME platform (MEP) providing computational resources to the ME applications running on the virtualization infrastructure (VI). The MEP is controlled by MEPM, while the VI is separately controlled by a VI manager.

Many studies have aimed at making MEC more practical on the basis of the ETSI architecture. For instance, the Open Edge computing project [17] is developing many MEC functions as open source software, but it has not completed all of them yet.

The paper [18] discussed how to decide on the locations of edge servers by considering statistics on the user population. It showed how important the location of the

edge servers is in 5G development.

The paper [19] described a testbed for MEC with an orchestrator to test mobility and handover between two edges. The authors of [8] also examined migration between edges, focusing on IP mobility.

The authors in [20] proposed a design for an MEC implementation integrated with the Network Function Virtualization (NFV) architecture, because they share certain common entities and functions. In particular, the orchestrator is an important entity in both NFV and MEC. The paper [21] proposed developing orchestrators by using containers, but the system was controlled by cloud computing like a master-slave system. The authors of [13] also emphasized information prediction in MEC, though they supported only a low level concept.

Two surveys [22], [23] describe many possible projects from different aspects. MEC design has focused the implementation of each individual entity. Many studies focused on only the communication latency to provide services. Some others, e.g., [12], considered computation latency as resource allocation.

A number of studies have considered latency in both communication and computation. For example, the paper [24] evaluated the response time of content caching in MEC. The paper [25] calculated the latency of both communication and computation for developing scheduling strategies that use MEC in manufacturing. ENORM in [26] observed latencies of gaming in fog computing. The authors of [27] proposed a mathematical model of task offloading in mobile cloud computing, they besides proposed an online algorithm to guarantee the latency of the workloads stochastically. In [28], the authors formulated the Dynamic Task Offloading and Scheduling problem in MEC, and they also proposed a solving method of the problem, which can guarantees the latency.

To illustrate the existing researches in the literature, we depict an illustration of the existing research categories in Fig. 3. The left-hand side category is about implementation researches such as [24]–[26], and the right-hand side one is a group of researches about guaranteeing the latency such as [27], [28].

In the left-hand side category, each of the researches has implemented a system including edge computing and their key performance metric is the latency. In the right-hand side category, the authors have proposed theoretical models about guaranteeing the latency.

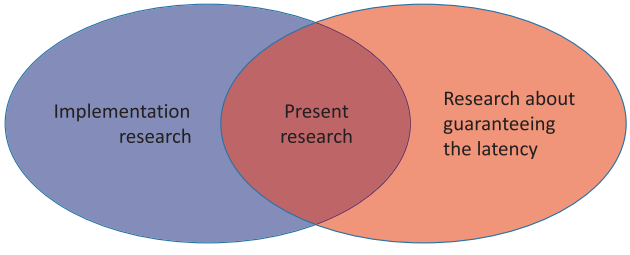


Fig. 3 Illustration of the existing research categories.

According to our literature survey, we cannot find the researches included in the intersection. In this study, we implemented a system, which aims to guarantee the latency performance including computing and communication ones, based on the edge computing. In other words, the present study is classified into the intersection of two categories. Below, we describe our previous simulation study [16] and an experimental study using an actual implementation and present a complete analysis of MEC.

3. Simulation Study

In our previous study, we created an MEC model and tested it in a simulation [14]. In the current study, we made the model more practical by adding a controller to the system in [15], [16].

Here, we briefly describe the model's design from the previous paper, as we will use it as a basis for the implementation design in the next section. The system model is illustrated in Fig. 4. A source node $S_i, 1 \le i \le N$, that represents a group of nearby mobile users generates a workload, w_k, at a rate λ in accordance with a Poisson process and sends it to any edge node $E_j, 1 \le j \le M$; the mobile edge host processes an accepted workload by processor sharing (PS) [29]. Note that a server that uses PS equally shares its resources (CPUs, memories, etc.) with all current workloads without a buffer. The PS concept utilizes the server to maximal effectiveness by using the least amount of resources.

The communication latency was calculated by hop count and hop delay, and the computing latency was estimated by PS using the current number of processes. Note that the actual computational latency would become known after the edge node finished the workload because of PS.

We modeled the MEC architecture by examining whether the total computing and communication latency exceeded the allowable latency. The system determined policies to select the target edge host for each workload. We considered three polices: *random, lowest latency, and minimum unfinished workload*.

We determined the objective function of the model by finding the minimum blocking probability (defined as the fraction of rejected workloads). A workload that exceeds the allowable latency is rejected by the system. Accepted workloads are executed at an edge host selected by the defined policy.

We considered two system procedures regarding the system constraints. One was a *strict system* that protected

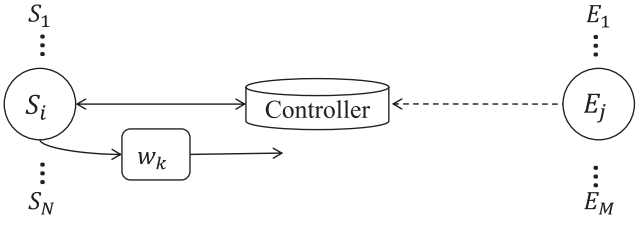


Fig. 4 System model with a controller.

Table 1 Comparison between strict and permissive systems.

Name of system	Workload types to be checked	Checking load	Error
Strict system	Existing and new workloads	High	None
Permissive system	New workloads only	Low	Exists

services from unexpected longer latencies; this procedure takes into account that a new workload might affect the computing latency of existing workloads when PS is used. The system checked a latency of a new workload or an existing workload. The strict system placed a burden on the server, so we defined a *permissive system* that relaxed the constraint of checking the latency of every workload at the chance of causing decision errors instead. These two systems are summarized in Table 1. In the table, "Checking load" means the computing load to decide workload acceptance when a new workload request is arrived. "Error" means the possibility of decision error, i.e., accepted workloads of which the latency requirements were not satisfied.

The previous studies [14]–[16] used three evaluation metrics. The first was the *blocking probability* (P_b), which is the ratio of rejected workloads to total workloads. The second was the *decision error* (ϵ) of the permissive system, defined as the ratio of incorrect decisions to all accepted workloads. Accepted workloads that actually exceeded the given latency allowance were counted as incorrect decisions. The third was the *modified blocking probability* (P'_b), which combines the blocking and error into one metric for representing the true performance of the system: $P'_b = P_b + \epsilon - P_b \cdot \epsilon$.

The model described above is based on the ideal assumption that the system can monitor every component in real time and make a correct decision. In practice, the system requires a monitor component that handles the whole system. In [15], [16], we extended the system in [14] by introducing a controller, as shown in Fig. 4. The controller includes a monitor resource function and edge-selecting policies. The source node communicates with the controller and asks for a destination server for its workload. The controller checks the resources of each server and verifies the condition under which to accept the workload or not; then it sends its decision to the source node.

The simulation results in the previous works pointed that the permissive system outperformed the strict one in terms of lower the modified blocking probability (P'_b). Selecting edge node with the lowest latency achieved the best performance among three policies.

4. Design and Implementation of Prototype System

Our ultimate objective is to guarantee the latency performance. To do so, we have proposed a system using edge computing and we have evaluated the system through simulations as described in Sects. 1, 2, and 3. In this paper, we designed and implemented a testbed system of MEC including every necessary entity guaranteeing latency performance as a proof of concept and compared its performance with simulation results.

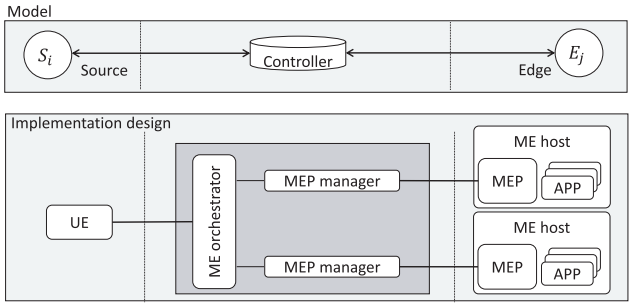


Fig. 5 MEC architecture applied system model.

To do so, we implemented three major components of the architecture, i.e., the ME orchestrator, ME host, and UE (as a source of requests). To identify operation functions of each component, we mapped the ETSI MEC in Fig. 2 to our model, as shown in Fig. 5.

- *UE: A source node* that represents a group of mobile users. It sends a workload to edge hosts, but first it must contact the controller and ask it for the edge destination. This is the same operation as in the UE of the ETSI MEC. It connects to the MEC through an ME orchestrator.
- *Orchestrator: A controller* that monitors the status of the whole system and selects a designated edge node for each workload in accordance with a defined policy. It satisfies the constraints of mobile applications. The orchestrator receives messages from the UEs and makes decisions, while each ME host is monitored by an individual manager.
- *ME Host: An edge node* that acts as a computational entity. It can be an ME host that provides services to each ME application.

The system model has to check application constraints, e.g., the allowable latency. This should actually be a function of the MEPM. However, we made it a function of the orchestrator in the testbed, as we have not implemented the MEPM as yet.

We defined three policies in the simulation — random, lowest latency, and minimum unfinished work — and two systems — strict and permissive. The minimum unfinished work and strict system examines every processing workload and estimates the time needed to finish each one. This task is complicated and the implementation cost would be too high in a real system. As a result, we decided to deploy only random and lowest latency policies and the permissive system in the testbed system.

The rest of the MEC components could be provided by developed from existing open source software, such as VI and VIM.

As mentioned in Sect. 2, the framwork of ETSI was a standardized architecture for MEC development. We there-

fore focus on ETSI MEC architecture illustrated in Fig. 2.

4.1 Communication Latency

Communication latency is an important consideration when selecting the locations of servers [30]. The simulation determined it in terms of the hop count and the average hop delay of each link. However, the authors of [31] argued that the hop count measures different aspects besides latency. They argued that although these measures are correlated, they are unsuitable for experiments on real networks.

The system should be able to accurately estimate both the source-to-edge and edge-to-source communication latency to make the decisions of the orchestrator more precise. A survey [32] classified latency measurements in mobile networks into active and passive methods. The active methods always monitor the latency from the server or source side, while the passive methods rely on statistical values. We designed our experiment to measure communication latency in the active method for getting near-current values.

The testbed system requires a tool that accurately estimates latency. For instance, there is one-way latency [33] and round-trip latency [34]. Moreover, ICMP, UDP, or TCP can be used as measurement protocols. Here, we measured the round-trip latency using TCP for simplicity. The orchestrator commands that each host update the round-trip time measured by TCP communication.

4.2 Computing Latency

Computing latency is always difficult to predict, because it depends on many factors, such as the CPU architecture and the characteristics of the running program and other background processes. In the simulations, we used a PS model, which was explained in Sect. 3, and an equation for calculating the latency. We still relied on PS in the implementation. When an edge host received a workload, it created a new process separately in response to the request.

However, the simulation assumed a single processor for PS, whereas an actual CPU may have multiple cores that can handle two or more processes simultaneously. We therefore took into account the number of CPU cores in the equation for computing latency (P):

$$P_{k,j}(t) = \left(\frac{b_k}{\mu_j}\right)\left(\frac{n_j(t)+1}{c_j}\right), \tag{1}$$

where c_j is the number of CPU cores of the edge host E_j; it becomes 1 when $n_j(t) < c_j$. b_k is the size of workload w_k. μ_j and n_j are respectively the processing rate and the current number of workloads of E_j. The orchestrator uses Eq. (1) to estimate computing latency of each workload.

4.3 Edge Service

Measuring computing latency requires an actual application to be run. That means for a realistic system, we need to define the services the edge host can provide. Different applications have different features and workloads. A simple feature such as text analysis may depend on mostly the CPU and not as much on memory or I/O, while a more complex function like image processing may need more resources.

To identify an application or service that is typical of latency-sensitive services, we have to consider the trend of mobile applications, which is toward real-time services which are more and more latency-sensitive. An attractive application for our purpose is public or road safety monitoring, which processes images or video rapidly in order to detect objects. Such applications sometimes require *face recognition*, which basically provides face detection, modification, and identification functions. In fact, various computer languages offer libraries for such functions, making their implementation easy.

4.4 Implementation

As described in the previous subsections, we designed the core entities and functions of MEC in the testbed system. The other entities could be built from open sources, such as OPENVIM for VI and VIM. In this subsection, the implementation details of each core entity are addressed.

The system mainly includes a source node or UE, an orchestrator, and an edge host. The implementation of the MEC testbed follows Fig. 5 and the design described in the previous subsections.

We developed a source node program in Python. It generated workloads with a rate of λ in accordance with a Poisson process. The source program was always connected to the orchestrator over a TCP connection. When there was a new workload, the source program sent a request to the orchestrator; then it waited for the decision. If the orchestrator accepted the workload, the source would forward it to the selected edge host over a TCP connection. The results were eventually returned to the source, after which the connection was closed.

The edge host program was also developed in Python. A server application was always running that listened for messages from sources and the orchestrator. When the edge program established a TCP connection for the source node to transmit a workload, it immediately created an individual thread for executing the arriving workload. To apply PS as described in Sect. 3, all current threads shared the server resources for each process. After finishing a workload, the edge host sent back the result to the source node.

The edge host also communicated with the orchestrator. When it received a request for a status update from the orchestrator, it sent a message including the latest average computing latency and the estimated communication latency of each source.

The orchestrator maintained a separate thread for updating the status of each edge host. The thread connected to the edge host via an SSH connection. It periodically updated the average computing latency. Through the SSH connection, the orchestrator commanded the edge host sent a probe

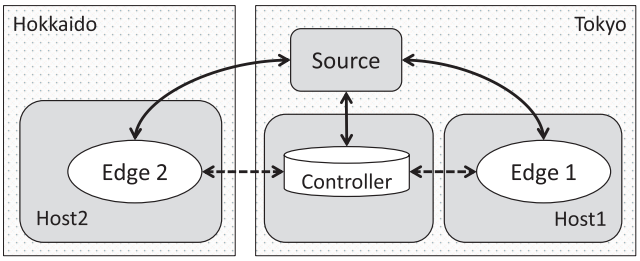


Fig. 6 System architecture.

message to the source for measuring the communication latency.

As a service at the edge host, we installed the face recognition program in the Python library [35]. This program is of the sort used by mobile applications to support features such as real-time video and mobile gaming with the front camera. The service detected human faces in photos and showed the result of how many people faces appeared in the photo frames. In the experiment, we put the application in the source node, and it would send requests to the edge server asking it to send a photo as quickly as possible to the face detection service at the edge host.

The implementation topology is depicted in Fig. 6. Two computers (Intel(R) i5-7500 CPU and 8GB RAM) operated as computational nodes of the edge-host servers. One (named host 1) was located in Tokyo, Japan, the other (host 2) was in Hokkaido, Japan. They ran processes over an Ubuntu 14.0 server operating system. The source program was run on Windows 10 on an Intel(R) i3-3120M and 4 GB RAM. The orchestrator was installed on a computer with Ubuntu 16.04.4, Intel(R) i3-3220, 4 GB RAM. The source and orchestrator were at the same location as host 1.

5. Evaluation

We empirically evaluated our testbed. The important parameters are described in Table 2. We fixed the number of sources and edge hosts and varied the maximum allowable latency to assess its impact on our model. The orchestrator updated the host status at intervals of 0.05 s. In other words, it sent an update request message to each host every 0.05 s, and the host subsequently sent back its current status. Unlike the simulation, we fixed the total number of workloads in one experiment to 30,000 instead of measuring run time. We measured the processing time $(1/\mu)$ by running each host for one day without control from the orchestrator. A single processor on average spent 0.222 s on one workload. Since there were two hosts with 4 CPU cores each, we set the workload rate to 20 workloads per second in order to determine the system capacity, e.g., $\lambda/(c\mu M) < 1$. Note that all workloads were of the same size ($b_k = 1$). The orchestrator measured the communication latency by sending a ping message during the update interval. In fact, sending packets from a source to the server in Hokkaido took on average 0.03 s while sending packets from a source to the server in Tokyo took about 0.003 s.

Table 2 Experimental parameters.

Parameter	Value
Number of sources(N)	1
Number of edges(M)	2
Workload rate (λ)	20 workloads/s
Max. allowance latency (θ)	0.3–1.3 s
Processing rate (μ)	0.222^{-1} workloads/s
CPU cores (c)	4
Orchestrator interval (σ)	0.05 s
Total number of workloads	30,000

We measured three matrices in the same way as described in Sect. 3. The pure and modified blocking probabilities had the same definitions, but the errors of the real system included not only decision errors but also other errors such as socket timeouts and connection losses.

In this section, we examine the performance of the different policies; then we compare the results of the experiment and the simulation. Finally, we investigate the accuracy of the method of estimating the computing latency.

5.1 Policy Results

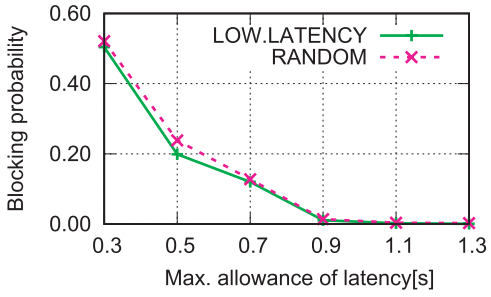
Figure 7 depicts the experimental results, showing the impact of the maximum allowable latency on the blocking probability in Fig. 7(a), on errors in Fig. 7(b) and on the modified blocking probability in Fig. 7(c). These figures show that a longer allowable latency gave better performance in regard to blocking and errors since the system could use all of its resources more efficiently.

Comparing the lowest latency and random policies, it is clear that both performed very nearly the same for all matrices, although the lowest latency yielded less blocking and less errors compared with the random policy.

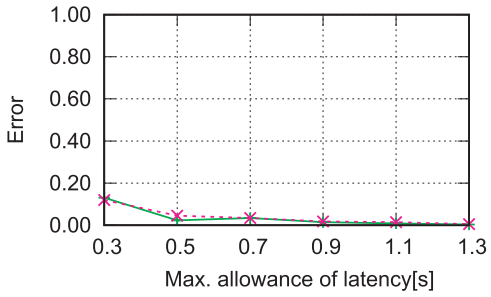
To verify the behavior of each policy in the experiment, Table 3 describes the ratio of selected hosts (Host1 or Host2) by the controller when we set the maximum allowable latency to 1.5 s that means the controller can accept all the workloads. It shows that the controller with the random policy dispatched workloads to both hosts with roughly 50% of the probability while the one with lowest latency policy likely preferred Host1 rather than Host2. That corresponds to design of each policy and suggests that the logic of policy correctly works in the real environment.

5.2 Comparison of Experimental and Simulated Results

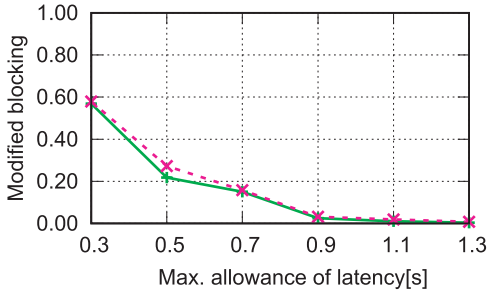
As we described in Sects. 1.1 and 4, our objective is to guarantee the latency performance including computing and communication ones. To prove that our testbed system can meet our goal, we have to evaluate its performance. Along with this, to analyze the detailed behavior of the system, we also conducted simulation with the same parameters as the implementation. The communication latency was fixed between two nodes by using the average round trip time from the experiment. We calculated the average computing time of a single workload and used its inverse to denote the processing rate in the simulation. Additionally, the previous



(a) Blocking probability

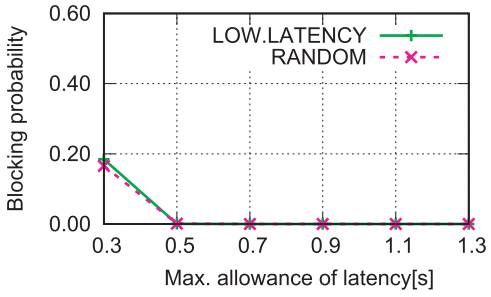


(b) Error

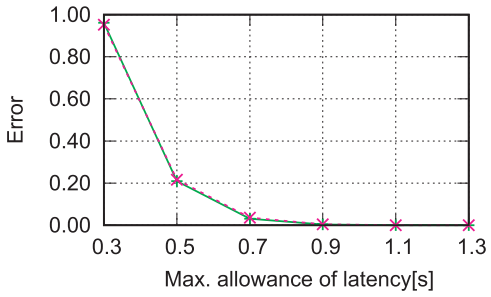


(c) Modified blocking probability

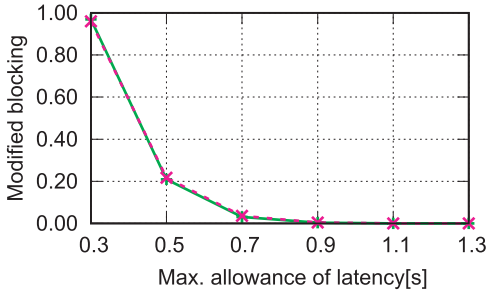
Fig. 7 Experimental results.



(a) Blocking probability



(b) Error



(c) Modified blocking probability

Fig. 8 Simulation results with experimental parameters.

Table 3 The ratio of selected hosts by controller (max. latency = 1.5 s).

Policy	Host 1	Host 2	Total
Random	15,310 (51%)	14,690 (49%)	30,000
Lowest latency	16,719 (56%)	13,281 (44%)	30,000

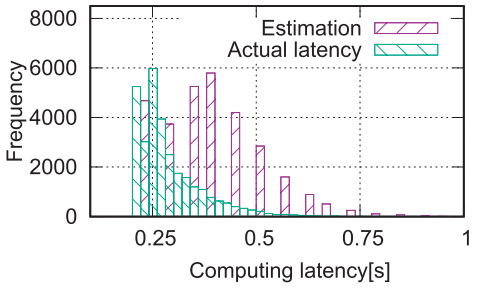
simulation executed workloads by using a PS with a CPU having a single core, while the real CPU architecture was able to process using four cores simultaneously. We thus incorporated a multi-core processor into the simulation.

The results in Fig. 8 are from the simulation described above. The X-axes and Y-axes of the subfigures correspond to those of Fig. 7. A comparison of Figs. 7 and 8 indicates that their modified blocking probabilities had the same tendency and the results of the two policies were not distinct. However, Fig. 8(b) shows that the simulation suffered the most errors, while Fig. 8(a) shows that it reduced the blocking probability to zero or near zero as the maximum latency equaled 0.5 s. The experimental results (Figs. 7(a) and 7(b)), on the other hand, showed high blocking probabilities with some errors.

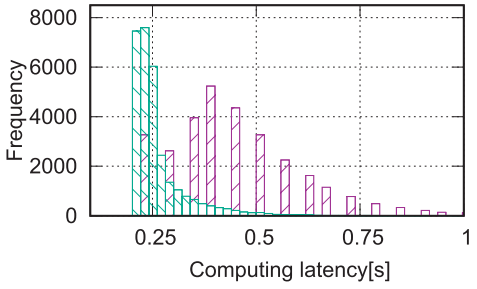
The experimental blocking probability and errors were rather different from the results of the simulation, though the modified blockings were similar. To investigate why the real system performed the way it did, we monitored the estimates and the actual latencies of both policies. Because the communication latency was estimated using the TCP round-trip time, which is very accurate, the computational latency should be carefully determined.

5.3 Estimation Accuracy

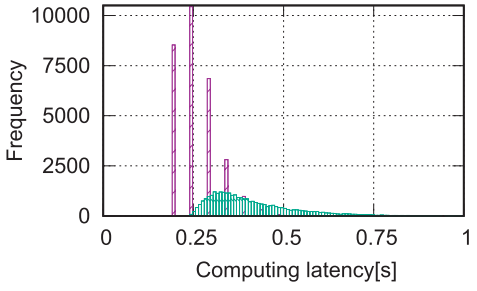
Finally, we examined the estimation accuracy. By probing messages, the estimation of communication latency was very precise. The main difference resulted from the computing latency. We, then, combined computing latency from all hosts and showed the distribution in Fig. 9 when we set the maximum latency to 1.5 s within which we did not see any rejections in either simulation or experiment. We monitored the estimated computing latency when the orchestrator cal-



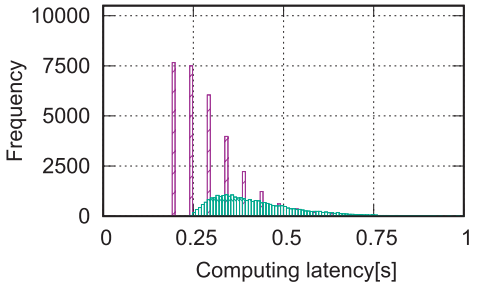
(a) Experiment with lowest latency policy



(b) Experiment with random policy



(c) Simulation with lowest latency policy



(d) Simulation with random policy

Fig. 9 Computational latency distribution.

culated each workload, and we checked the actual latency when each workload finished. We created histograms of the estimated and actual latency as in Fig. 9. The orchestrator (controller) in the simulation (Figs. 9(c) and 9(d)) underestimated the latency because of the PS formulation. The actual latency may lengthen the processing time quite a bit. That is why there were large errors in the simulation. In contrast, the distribution from the experiment (Figs. 9(a) and 9(b)) shows that the orchestrator most often overestimated the computing latency. That led the orchestrator to reject workloads with less error. This is because the real server did not perform perfectly like a PS. It was affected by many factors, such as its CPU architecture, service algorithm, and memory accesses.

Table 4 Statistics of actual update latency in [s] (max. latency = 1.5 s).

	Host 1		Host 2	
	Random	Lowest latency	Random	Lowest latency
Mean	0.262	0.363	0.320	0.413
Standard deviation	0.075	0.102	0.099	0.118
Min	0.109	0.206	0.100	0.195
Max	0.647	0.869	0.869	0.922

Our question in the previous section Sect. 5.2 is the reason of the different tendency of experimental and simulation results. The analysis about Fig. 9 reveals a part of the question. Namely, the estimation tendency of experiments and simulations are different, i.e., overestimation occurred in experiments whereas underestimation occurred in simulations.

In this way, the accuracy of the estimated computing latency seems to be a significant factor affecting the orchestrator's decisions and the system performance. Even in the simulation, using PS with multiple CPU cores could not ensure that the estimation was accurate (see Figs. 9(c) and 9(d)). A real implementation must consider this in a more serious way. To improve the estimation method, we should investigate the factors that can lead to underestimations of computing latency and find a way to shift the distribution in Figs. 9(s) and 9(b) closer to the actual latency distribution.

To reveal a reason of the overestimation in the experiment, one of possible factors is a process inside the orchestrator, so log files of the orchestrator were further investigated. In the experiment, the actual update interval (σ) was set to 0.05s, which means the orchestrator would frequently update the host information within the average computing latency per a workload. We measured the actual update latency, which is all the latency about updates, i.e., it consists of the update interval, the communication latency to the host, and the processing delay at the orchestrator. Table 4 describes the statistic of the actual update latency when we set the maximum allowable latency to 1.5 s. It shows that the orchestrator spent longer time to update the information of each host than the update interval (0.05 s). That suggested there are other factors here.

As we described, the actual update latency includes the update interval, the communication latency to each host, and the processing delay in the orchestrator. As an example, we look at the mean value of Host2 with Random policy, 0.32 s. In this case, if we assume the communication latency between the controller and Host2 is the same with the latency between the controller and source (0.03 s), we see the processing time of the orchestrator equal to $0.320 - 0.05 - 0.03 = 0.24$ s, which becomes a major part of the update latency. The orchestrator might be lacking

of resources. It must handle many tasks such as connections to the hosts, requests about workloads. The CPU of the orchestrator machine in this experiment was Intel(R) i3-3120M, which may not enough to execute all processes simultaneously, so we can see longer update latency than the interval. Our previous work [16] has discussed how long latency from the orchestrator affects the edge computing performance. It proved by simulations that longer processing time of the orchestrator leads more overestimations of the computing latency about workloads. Consequently, the orchestrator becomes a bottleneck that affects the accuracy of the estimations.

6. Conclusion

CPS will enable many applications in 5G and Beyond-5G networks. It requires MEC in order to provide low-latency services to many users, so the MEC concept and architecture should be thoroughly investigated before such systems are deployed in the real world. Although many studies have used simulations for such analyses, few studies have targeted actual implementations and evaluations in real environments.

We designed a prototype MEC architecture guaranteeing the latency performance based on the framework standardized by ETSI and our previous simulation studies. The necessary functions were identified, and they guided the design of each MEC entity. We mapped functions and entities that were evaluated in our previous simulation to the MEC system design. We implemented a testbed based on this design and evaluated its performance in a real environment.

The evaluation in the testbed showed that the lowest-latency policy would outperform a random policy. The orchestrator has to estimate the computing latency to check the latency conditions of applications. Simulations conducted with the same parameters showed a lower blocking probability but more errors in comparison with the experimental results. This implies that there are hidden but important factors in real network environments.

To investigate these factors, we drew computational latency distributions, which showed that the simulations probably underestimated the latency, while the experiment overestimated it. We found phenomena wherein the simulation could not perfectly simulate the real system. The real system had a higher blocking probability and fewer errors than what could be expected from the simulation. Therefore, when we design an MEC system for CPS, we should be aware of whether or not a higher blocking rate is acceptable for applications. One reason is that an equation of PS is too simple; it cannot accurately estimate the actual computing latency of the server's complicated processing architecture. In the future, the accuracy of the latency estimation should be improved by conducting a deeper analysis of the processing architecture.

Acknowledgments

This was supported in part by JSPS KAKENHI Grant-in-Aid for Scientific Research (B) Number 19H04103.

References

[1] K. Intharawijitr, K. Iida, H. Koga, and K. Yamaoka, "Implementation and preliminary evaluation of low-latency network architecture," IEICE Technical Report, IA2017-51, Nov. 2017.
[2] R. Raj, "Cyber-physical systems: The next computing revolution," Proc. IEEE Design Automation Conference (DAC'10), Anaheim, CA, USA, pp.731–736, June 2010. DOI: 10.1145/1837274.1837461
[3] E.A. Lee, "Cyber physical systems: Design challenges," Proc. IEEE Int'l Symp. Object and Component-Oriented Real-Time Distributed Computing (ISORC), Orlando, FL, USA, pp.363–369, May 2008. DOI: 10.1109/ISORC.2008.25
[4] J. Lee, B. Bagheri, and H.A. Kao, "A cyber-physical systems architecture for industry 4.0–based manufacturing systems," Manufacturing Letters, vol.3, pp.8–23, Jan. 2015. DOI: 10.1016/j.mfglet.2014.12.001
[5] M. Maier, M. Chowdhury, B.P. Rimal, and D.P. Van, "The tactile internet: Vision, recent progress, and open challenges," IEEE Commun. Mag., vol.54, no.5, pp.138–145, May 2016. DOI: 10.1109/MCOM.2016.7470948
[6] G.I. Klas, "Open fog computing and mobile edge cloud gain momentum," Y. I Readings, http://yucianga.info/?p=938, Nov. 2015.
[7] ETSI MEC ISG, "Mobile edge computing (MEC); Framework and reference architecture," ETSI, GS MEC 003, https://www.etsi.org/deliver/etsi_gs/mec/001_099/003/01.01.01_60/gs_mec003v010101p.pdf, March 2016.
[8] T. Kondo, K. Isawaki, and K. Maeda, "Development and evaluation of the MEC platform supporting the edge instance mobility," Proc. IEEE Annual Computer Software and Applications Conference (COMPSAC), Tokyo, Japan, vol.2, pp.193–198, July 2018. DOI: 10.1109/COMPSAC.2018.10228
[9] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D.S. Nikolopoulos, "Challenges and opportunities in edge computing," Proc. IEEE Int'l Conf. Smart cloud (Smartcloud), New York, NY, USA, pp.20–26, Nov. 2016. DOI: 10.1109/Smartcloud.2016.18
[10] K. Iida, "Current R & D status of edge computing and its future direction," IEICE Technical Report, IA2017-16, Aug. 2017 (in Japanese).
[11] S. Sarkar, S. Chatterjee, and S. Misra, "Assessment of the suitability of fog computing in the context of internet of things," IEEE Trans. Cloud Comput., vol.6, no.1, pp.46–59, Jan.–March 2018. DOI: 10.1109/TCC.2015.2485206
[12] T.X. Tran and D. Pompili, "Joint task offloading and resource allocation for multi-server mobile-edge computing networks," IEEE Trans. Veh. Technol., vol.68, no.1, pp.856–8681, May 2018. DOI: 10.1109/TVT.2018.2881191
[13] P. Zhao, H. Tian, S. Fan, and A. Paulraj, "Information prediction and dynamic programming-based RAN slicing for mobile edge computing," IEEE Wireless Commun. Lett., vol.7, no.4, pp.614–617, Aug. 2018. DOI: 10.1109/LWC.2018.2802522
[14] K. Intharawijitr, K. Iida, and H. Koga, "Simulation study of low latency network architecture using mobile edge computing," IEICE Trans. Inf. & Syst., vol.E100-D, no.5, pp.963–972, May 2017. DOI: 10.1587/transinf.2016NTP0003
[15] K. Intharawijitr, K. Iida, H. Koga, and K. Yamaoka, "Practical enhancement and evaluation of a low-latency network model using mobile edge computing," Proc. IEEE Annual Computer Software and Applications Conference (COMPSAC2017), Turin, Italy, pp.567–574. July 2017. DOI: 10.1109/COMPSAC.2017.190

[16] K. Intharawijitr, K. Iida, H. Koga, and K. Yamaoka, "Simulation study of low-latency network model with orchestrator in MEC," IEICE Trans. Commun., vol.E102-B, no.11, pp.2139–2150, Nov. 2019. DOI: 10.1587/transcom.2018EBP3368
[17] R. Schuster, and P. Ramchandran, "Open edge computing: From vision to reality," https://wiki.opnfv.org/download/attachments/6819410/2016-06-21%20Open%20Edge%20Computing%20-%20published%20final.pdf, June 2016.
[18] M. Syamkumar, P. Barford, and R. Durairajan, "Deployment characteristics of 'The Edge' in mobile edge computing," Proc. ACM Workshop on Mobile Edge Communications (MECOMM'18), Budapest, Hungary, pp.43–49, Aug. 2018. DOI: 10.1145/3229556.3229557
[19] I. Farris, T. Taleb, H. Flinck, and A. Iera, "Providing ultra-short latency to user-centric 5G applications at the mobile network edge," Trans. Emerging Telecommunications Technologies, vol.29, no.4, 13 pages, March 2017. DOI: 10.1002/ett.3169
[20] V. Sciancalepore, F. Giust, K. Samdanis, and Z. Yousaf, "A double-tier MEC-NFV architecture: Design and optimisation," Proc. IEEE Conf. Standards for Communications and Networking (CSCN), Berlin, Germany, 6 pages, Nov. 2016. DOI: 10.1109/CSCN.2016.7785157
[21] Z. Yu, J. Wang, Q. Qi, H. Sun, and J. Zou, "A boundless resource orchestrator based on bontainer technology in edge computing," Proc. IEEE Int'l Conf. Computer Communication and Networks (ICCCN), Hangzhou, China, 2 pages, July 2018. DOI: 10.1109/ICCCN.2018.8487377
[22] R. Mijumbi, J. Serrat, J. Gorricho, S. Latre, M. Charalambides, and D. Lopez, "Management and orchestration challenges in network functions virtualization," IEEE Commun. Mag., vol.54, no.1, pp.98–105, Jan. 2016. DOI: 10.1109/MCOM.2016.7378433
[23] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration," IEEE Commun. Surveys Tuts., vol.19, no.3, pp.1657–1681, Thirdquarter 2017. DOI: 10.1109/COMST.2017.2705720
[24] J. Poderys, M. Artuso, C. Michael Oest Lensbol, H. Lehrmann Christiansen, and J. Soler, "Caching at the mobile edge: A practical implementation," IEEE Access, vol.6, pp.8630–8637, Feb. 2018. DOI: 10.1109/ACCESS.2018.2809490
[25] X. Li, J. Wan, H.N. Dai, M. Imran, M. Xia, and A. Celesti, "A hybrid computing solution and resource scheduling strategy for edge computing in smart manufacturing," IEEE Trans. Ind. Informat., vol.15, no.7, pp.4225–4234, Feb. 2019. DOI: 10.1109/TII.2019.2899679
[26] N. Wang, B. Varghese, M. Matthaiou, and D.S. Nikolopoulos, "ENORM: A framework for Edge NOde Resource Management," IEEE Trans. Services Comput., vol.13, no.6, pp.1086–1099, Nov.-Dec. 2020. DOI: 10.1109/TSC.2017.2753775
[27] Y. Kao, B. Krishnamachari, M. Ra, and F. Bai, "Hermes: Latency optimal task assignment for resource-constrained mobile computing," IEEE Trans. Mobile Comput., vol.16, no.11, pp.3056–3069, Nov. 2017. DOI: 10.1109/TMC.2017.2679712
[28] H.A. Alameddine, S. Sharafeddine, S. Sebbah, S. Ayoubi, and C. Assi, "Dynamic task offloading and scheduling for low-latency IoT services in multi-access edge computing," IEEE J. Sel. Areas Commun., vol.37, no.3, pp.668–682, March 2019. DOI: 10.1109/JSAC.2019.2894306
[29] S.F. Yashkov, "Processor-sharing queues: Some progress in analysis," Queueing Systems, vol.2, no.1, pp.1–17, March 1987. DOI: 10.1007/BF01182931
[30] M. Kwon, Z. Dou, W. Heinzelman, T. Soyata, H. Ba, and J. Shi, "Use of network latency profiling and redundancy for cloud server selection," Proc. IEEE Int'l Conf. cloud Computing, Anchorage, AK, USA, pp.826–832, July 2014. DOI: 10.1109/CLOUD.2014.114
[31] K. Obraczka and F. Silva, "Network latency metrics for server proximity," Proc. IEEE Global Telecommunications Conference (GLOBECOM'00), San Francisco, CA, USA, pp.421–427, Nov. 2000. DOI: 10.1109/GLOCOM.2000.892040
[32] V. Bajpai and J. Schonwalder, "A survey on internet performance measurement platforms and related standardization efforts," IEEE Commun. Surveys Tuts., vol.17, no.3, pp.1313–1341, Thirdquarter 2015. DOI: 10.1109/COMST.2015.2418435
[33] G. Almes, S. Kalidindi, and M. Zekauskas, "A one-way delay metric for IPPM," IETF RFC2679, https://tools.ietf.org/html/rfc2679, Sept. 1999.
[34] G. Almes, S. Kalidindi, and M. Zekauskas, "A round-trip delay metric for IPPM," IETF RFC2681, https://tools.ietf.org/html/rfc2681, Sept. 1999.
[35] A. Geitgey, "Face recognition," https://github.com/ageitgey/face_recognition, Accessed Feb. 10, 2019.

Krittin Intharawijitr received the B.E. degree in Computer Engineering from Chulalongkorn University, Bangkok, Thailand in 2013, and received the M.E. and D.E. degrees both in Communications and Computer Engineering from Tokyo Institute of Technology, Tokyo, Japan in 2016 and 2019, respectively. Presently, he is a researcher in Autonomous Networking Research & Innovation Dept., Network Division, Rakuten Mobile, Inc., Japan. His research interests lie in the fields of network architecture, mobile networks, and cloud computing.

Katsuyoshi Iida received the B.E., M.E. and Ph.D. degrees in Computer Science and Systems Engineering from Kyushu Institute of Technology (KIT), Iizuka, Japan in 1996, in Information Science from Nara Institute of Science and Technology, Ikoma, Japan in 1998, and in Computer Science and Systems Engineering from KIT in 2001, respectively. Currently, he is an Associate Professor in the Information Initiative Center, Hokkaido University, Sapporo, Japan. His research interests include network systems engineering such as network architecture, performance evaluation, QoS, and mobile networks. He is a member of the WIDE project and IEEE. He received the 18th TELECOM System Technology Award, and Tokyo Tech young researcher's award in 2003, and 2010, respectively.

Hiroyuki Koga received the B.E., M.E. and D.E. degrees in computer science and electronics from Kyushu Institute of Technology, Japan, in 1998, 2000, and 2003, respectively. From 2003 to 2004, he was a postdoctoral researcher in the Graduate School of Information Science, Nara Institute of Science and Technology. From 2004 to 2006, he was a researcher in the Kitakyushu JGN2 Research Center, National Institute of Information and Communications Technology. From 2006 to 2009, he was an assistant professor in the Department of Information and Media Engineering, Faculty of Environmental Engineering, University of Kitakyushu, and then has been an associate professor in the same department since April 2009. His research interests include performance evaluation of computer networks, mobile networks, and communication protocols. He is a member of the ACM and IEEE.

Katsunori Yamaoka received the B.E., M.E., and Ph.D. degrees from the Tokyo Institute of Technology in 1991, 1993 and 2000, respectively. He left Ph.D program in 1994 and joined the Tokyo Institute of Technology as an assistant professor at that time. In 2000, he joined the National Institute of Multimedia Education (NIME) in Japan as an associate professor. Since 2001, he has been an associate professor at the Tokyo Institute of Technology. Currently, he is a professor at Tokyo Institute of Technology since 2018. He has also been a visiting associate professor of the National Institute of Informatics (NII) in Japan since 2004. His research interests are network QoS control for multimedia communications.