IEICE TRANS. INF. & SYST., VOL.E105–D, NO.9 SEPTEMBER 2022

PAPER

Design and Implementation of an Edge Computing Testbed to Simplify Experimental Environment Setup*

Hiroaki YAMANAKA[†a)], **Yuuichi TERANISHI**[†], **Eiji KAWAI**[†], **Hidehisa NAGANO**[†], ***and*** **Hiroaki HARAI**[†], ***Members***

SUMMARY Running IoT applications on edge computing infrastructures has the benefits of low response times and efficient bandwidth usage. System verification on a testbed is required to deploy IoT applications in production environments. In a testbed, Docker containers are preferable for a smooth transition of tested application programs to production environments. In addition, the round-trip times (RTT) of Docker containers to clients must be ensured, according to the target application's response time requirements. However, in existing testbed systems, the RTTs between Docker containers and clients are not ensured. Thus, we must undergo a large amount of configuration data including RTTs between all pairs of wireless base station nodes and servers to set up a testbed environment. In this paper, we present an edge computing testbed system with simple application programming interfaces (API) for testbed users that ensures RTTs between Docker containers and clients. The proposed system automatically determines which servers to place Docker containers on according to virtual regions and the RTTs specified by the testbed users through APIs. The virtual regions provide reduced size information about the RTTs in a network. In the proposed system, the configuration data size is reduced to one divided by the number of the servers and the command arguments length is reduced to approximately one-third or less, whereas the increased system running time is 4.3 s.

key words: *edge computing, testbed, Kubernetes, low response time*

1. Introduction

Edge computing is a crucial computing paradigm for IoT applications. Edge computing offers ultra-low response times and efficient bandwidth usage by leveraging a server's proximity to the client hosts. These features are preferable for IoT applications, e.g., augmented reality [1] and surveillance systems [2], because such applications generate a large amount of video data and require low latency for data analysis.

To encourage the deployment of IoT applications in production environments, a testbed system of edge computing has important roles that reveal system bugs and evaluate system effectiveness and performance. We propose a testbed system that allows IoT application developers (i.e., testbed users) to run server programs of the IoT applications in experimental edge computing environments. Instead of burdens such as server and network configuration, the testbed users can obtain the experimental edge computing environments through resource requests via an application programming interface (API) of the testbed system.

The edge computing testbed system needs to provide an experimental environments, in which the network round-trip time (RTT) between a server and a client satisfies a condition specified by a testbed user. The response time to a client is one of the significant performance metrics of IoT applications and the required response time varies, e.g., 16 ms for augmented reality and 2 ms for remote-controlled robots [3]. The response time consists of the network RTT between a server and a client, and the processing time in the server. To observe an impact of the network RTT on the response time, the testbed system must support providing an experimental environment, in which the RTT between a server and a client is specified by a testbed user.

Both EdgeNet [4] testbed system and Akraino [5] infrastructure system support Docker containers [6] by utilizing Kubernetes (k8s) [7] to manage Docker containers across multiple servers. However, the systems do not support satisfying RTT conditions between clients and servers running Docker containers. Krishnaswamy *et al.* [8] proposed a method to determine servers to place Docker containers on according to the disclosed latency and the RTT condition for IoT applications. However, this method forces the users to use large configuration data including network latency. Here, the data size is proportional to the product of the numbers of wireless base station nodes and servers.

In this paper, we propose an easy-to-use edge computing testbed system providing experimental environments via APIs. In the provided environments, the network RTTs between Docker containers and wireless base station nodes are less than and close to the RTTs of conditions specified by testbed users. The proposed system employs the virtual region-based APIs [9], which reduces the amount of configuration data for testbed users to satisfy RTT conditions. For the RTT input by a testbed user, the testbed system provides the virtual regions presenting relation between the edge clouds and the wireless base station nodes under the RTT condition. For a request of a Docker container in the virtual region by the testbed user, the testbed system automatically determines the server that satisfies the RTT condition and configures the Docker container. Compared to the conventional method, the configuration data size is reduced to one divided by the number of the servers and the command argument length is reduced to approximately one-third or less, whereas the system running time merely increases by approximately 4.3 s.

Manuscript received March 31, 2022.
Manuscript publicized May 27, 2022.
[†]The authors are with the National Institute of Information and Communications Technology, Koganei-shi, 184–8795 Japan.
*This is paper on system development.
a) E-mail: hyamanaka@nict.go.jp
DOI: 10.1587/transinf.2022EDK0003

This paper is an extended version of the previous work [10]. The extensions in this paper are the network isolation function to accept multiple testbed users (Sects. 3.2.2 and 4.2.4) and the quantitative evaluation using real machine (Sects. 5.4–5.6).

The remainder of this paper is organized as follows. Section 2 introduces previous work involving edge computing infrastructure systems. Sections 3 and 4 describe design considerations and the implementation of the proposed edge computing testbed, respectively. Section 5 evaluates the testbed using the implementation. Finally, Sect. 6 concludes the paper.

2. Related Work

2.1 Real-World Testbed Deployments

EdgeNet [4] is a set of globally deployed edge clouds running at sites across the US, Canada, and the EU. EdgeNet is a master node that manages a k8s cluster, i.e., all worker nodes at the sites. Here, a testbed user obtains Docker containers using the Kubernetes Dashboard of EdgeNet. EdgeNet allows testbed users to select the geographical locations of servers to create Docker containers on; however, the RTTs between the servers and wireless base station nodes are not disclosed to the testbed users. Thus, testbed users cannot ensure the RTT between the Docker containers and clients.

2.2 Application-Level Proximity Control

Chiu *et al.* [11] proposed a server task allocation algorithm to obtain a short response time for clients, where the distributed tasks are allocated to multiple servers according to the communication latency to the servers and the tasks' processing times. In addition, CEF [12] provides an API for IoT application programmers to specify the geographical location of Docker containers running processes in a program. The methods improve the response time while leveraging the task scheduling of an application program. However, with these methods, testbed users are limited to using the application programming frameworks or middleware that are supported by these methods. In contrast, with the proposed testbed system, testbed users have no such limitation because Docker containers are accepted, and they can package arbitrary application programming framework and middleware.

2.3 Infrastructure Systems without Proximity Control

KubeEdge [13] extends k8s to servers in cloud data centers and edge clouds in a single virtual network. This feature allows testbed users to use both cloud and edge resources in an integrated manner. However, the KubeEdge scheduler does not control the RTT between Docker containers and wireless base station nodes. Dreibholz [14] proposed a virtual network function (VNF) testbed on OpenAirInterface-based [15] evolved packet core networks, i.e., software-based LTE mobile core networks. This testbed simplifies and automates the configuration of VNF and OpenAirInterface using the Open Source Management and Orchestration (OSM) framework [16]. In this testbed, the server programs for IoT applications are deployed as VNFs. This testbed assumes LTE networks; thus, the VNFs are always placed in the same packet data network gateways, where the RTTs to wireless base station nodes are always the same. Both the KubeEdge and Dreibholz testbed systems are unsuitable for edge computing testbed systems. In these systems, testbed users cannot change the RTT between Docker containers and clients; however, user-defined experimental RTT conditions vary according to tested IoT applications, e.g., 16 ms for augmented reality and 2 ms for remote-controlled robots [3].

2.4 Infrastructure Systems with Proximity Control

The method proposal by Haja *et al.* [17] extends the k8s scheduler to ensure RTT among Docker containers. This method measures the RTTs among servers to k8s, and the scheduler uses the measured RTTs to determine where to place a Docker container. However, the scheduler does not ensure the RTTs between Docker containers and clients.

The method proposed by Ceselli *et al.* [18] fixes an edge cloud to host Docker containers for each wireless base station. The edge cloud for the wireless base station is determined according to the target RTT to the wireless base station node. However, the RTT between Docker containers and wireless base station nodes may not match the testbed user's preferences because the target RTT is predetermined by the testbed administrator rather than the testbed users.

FocusStack [19] is a cloud-edge orchestrator that utilizes Docker containers. According to a client request, the orchestrator seeks nearby servers that satisfy the given computation performance criteria (e.g., CPU cores, memory size, and geographical area), and then a Docker container image is sent and instantiated on the selected server. Akraino [5], which is a Linux Foundation project, integrates open source computing and networking resource management software, including k8s, and encourages the smooth deployment of edge computing environments in production networks. FocusStack and Akraino allow testbed users to specify the geographical location and the tier of the location (e.g., client devices, public buildings, and teleco network) of servers to host Docker containers, respectively. However, these systems only loosely ensure response times because not consider the actual RTTs in a network to select servers.

Krishnaswamy *et al.* [8] proposed an infrastructure framework for network function virtualization that discloses the RTT between wireless base station nodes and edge clouds to testbed users. The testbed user can specify an edge cloud to host a Docker container. However, to ensure the RTT to clients, the testbed user must deal with a large amount of RTT data whose size is linear to the product of the number of wireless base station nodes and the number

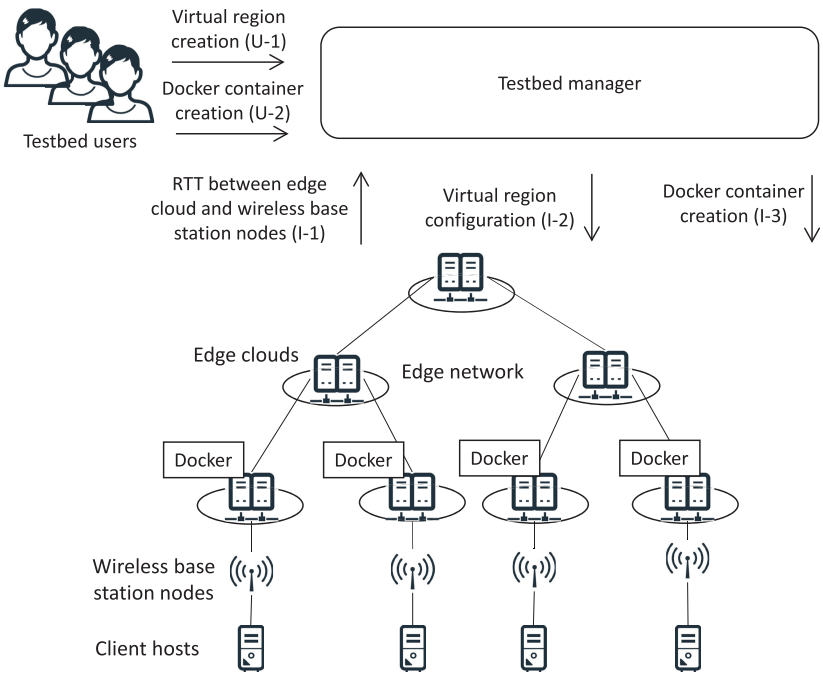

Fig. 1 Testbed system components.

of edge clouds.

2.5 Infrastructure System with Easy Proximity Control

The virtual region-based interface [9] ensures the RTT between edge clouds and wireless base station nodes while reducing the configuration data. This method determines groups of wireless base station nodes, which are referred to as virtual regions. Here, a virtual region has the same edge cloud, where the RTTs to the wireless base station nodes in the virtual region are the same or less than the threshold of the testbed user's application. Here, the testbed user selects the virtual region to request a Docker container. The data size of the virtual regions is bound by the number of wireless base station nodes. For the request, the Docker container is created in the edge cloud, and the RTTs to the wireless base station nodes in the virtual region do not exceed the threshold. In the proposed testbed system, we use the virtual region-based interface to ease the setup of the experimental environment, which is described in the following.

3. Proposed Edge Computing Testbed System

We developed an edge computing testbed that enables a simple system operation to satisfy the RTT conditions between Docker containers and clients in provided experimental environments.

3.1 System Components and Operation

The proposed edge computing testbed system provides networked Docker containers [6] and client hosts to testbed users. The Docker containers and client hosts include server and client programs, respectively. As shown in Fig. 1, the proposed testbed system comprises a testbed manager, a network, edge clouds, wireless base station nodes, and client hosts. The testbed manager is the original module interfacing with the testbed users and managing the edge clouds. Each of the edge clouds is a k8s [7] cluster with additional modules for compatibility with the testbed manager. The k8s cluster includes servers to run the Docker containers. The network connects the edge clouds and wireless base station nodes. The client host accesses the Docker container through the wireless base station node and the network. In our testbed deployment, although the wireless base station nodes are wired network nodes, we refer to those nodes as the "wireless base station" nodes because their positions in the network correspond to wireless base stations in a mobile backhaul network.

The networked edge clouds in the proposed testbed system imitate an edge computing infrastructure. The network corresponds to a mobile backhaul network. This imitated infrastructure is compatible with the edge computing infrastructure model in the 5G network proposed by ETSI ISG MEC [20]. As UPF (user plane functions) in the 5G network infrastructure model do the edge clouds locate at any node in the edge computing testbed's network.

The proposed testbed system is operated according to the following processes.

1. The testbed user prepares an original Docker image and sets the client hosts.
2. Using interface U-1 (Fig. 1), the testbed user requests the creation of virtual regions with a condition of the RTT r_v between the wireless base station nodes and a Docker container.

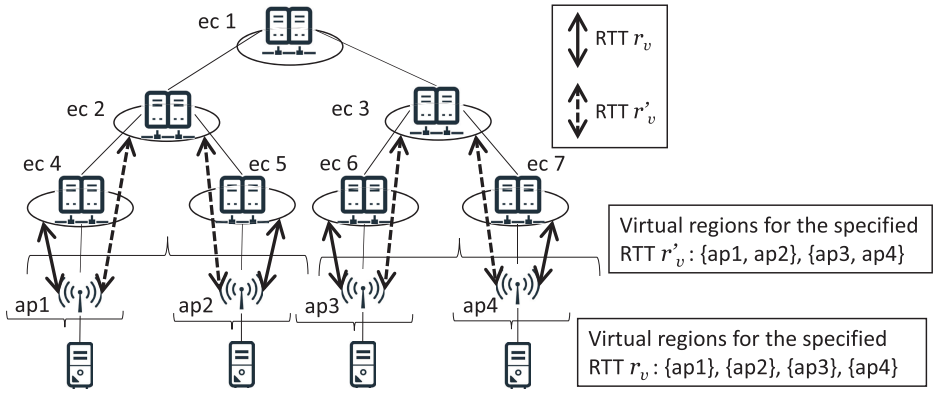

Fig. 2 Example virtual regions.

3. The testbed manager determines the virtual regions and presents them to the testbed user.
4. Using interface U-2 (Fig. 1), the testbed user requests a Docker container for each virtual region.
5. The Docker container is created and allocated to the testbed user.
6. The testbed user experiments using communication among the client hosts and the Docker container.

In process 2, the testbed user determines the condition of the RTT r_v so that the RTT between client hosts and provided Docker containers is short enough to keep the response time from the Docker containers to the client hosts acceptable for an experiment. The condition of the RTT r_v is determined as follows:

$$r_v = r_d^c - r_w^c,$$

where r_w^c is the RTT between the client hosts and their connecting wireless base station nodes, and r_d^c is the maximum allowable network latency in the acceptable response time in the experiment.

3.2 Design Approach

3.2.1 RTT Condition Satisfaction

To satisfy RTT conditions, edge clouds to place Docker containers need to be selected appropriately. The RTT between the selected edge clouds and the wireless base station nodes must be the same or less than r_v. As described in Sect. 2.4, direct selection of the edge clouds forces testbed users to deal with a large amount of network RTT data whose size is linear to the product of the numbers of the wireless base station nodes and the edge clouds. Instead, we employ the virtual region-based interface [9], which reduces the size of the configuration data to be bounded by the number of the wireless base station nodes.

The virtual regions are groups of wireless base station nodes. Here, the virtual regions are determined, according to the condition of the RTT r_v such that the nodes in a group are associated with the same edge cloud reachable within r_v. Figure 2 shows a testbed environment for two different RTT conditions in the network. Here, four edge clouds ec4–ec7 are associated with the four individual virtual regions of the four wireless base station nodes (ap1–ap4), where the condition of the RTT is r_v. Edge clouds ec2 and ec3 are associated with the two virtual regions of ap1 and ap2, and ap3 and ap4, where the condition of the RTT is r'_v. For a testbed user's request to create a Docker container for a virtual region, the proposed testbed system creates the Docker container in an edge cloud associated with the virtual region. The details of the virtual region-based interface are described in Sect. 4.1.1.

Although the proposed testbed system takes away the configurability of a k8s system due to using the virtual region-based interface, it is enough for testbed users who are IoT application developers. The main lost configurability is direct server selection to place Docker containers and network policy setting to control communication among Docker containers. However, IoT application developers do not need this configurability because, typically in IoT applications, communication occurs between clients and servers nearby proximity while the proposed testbed system supports that communication pattern.

In terms of human resources for testbed system management, the proposed testbed system has extra maintenance costs in addition to that of the k8s system because a system administrator needs to maintain both the k8s and the testbed manager. However, both systems keep running without human operations. Thus, in normal operations, the human resources for maintaining the proposed testbed system are almost the same as that of k8s.

3.2.2 Multi-Tenancy

To isolate experimental environments of different testbed users (i.e., multi-tenancy), communication among Docker containers and client hosts of different testbed users is blocked. Figure 3 illustrates the communication control. This example assumes the Docker containers of testbed users A and B. Here, Dockers α-1, α-2, and α-3 belong to testbed user A, and Dockers β-1 and β-2 belong to testbed user B. In addition, client hosts α and β belong to testbed

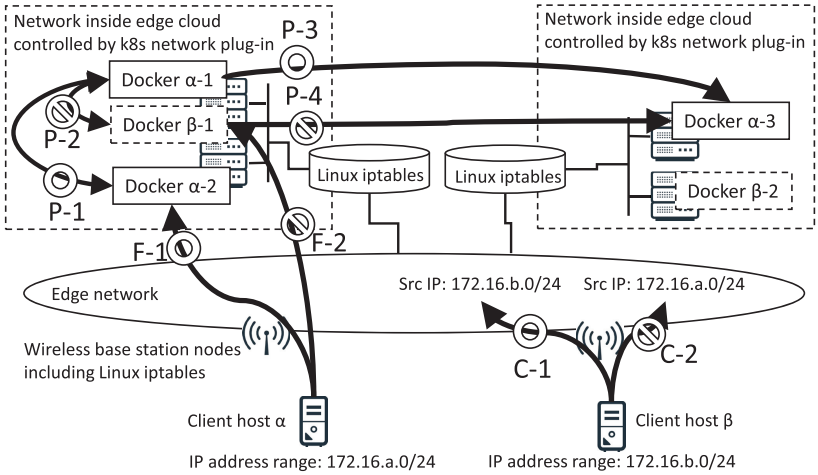

Fig. 3 Summary of communication control.

users A and B, respectively.

To allow communication among the Docker containers of the same testbed users (P-1 and P-3 in Fig. 3) and deny that of different testbed users (P-2 and P-4), we use a function of a network plugin of k8s, i.e., Calico [21]. To allow communication between the client hosts and the Docker containers of the same testbed users (F-1) and deny that of different testbed users (F-2), IP address-based filtering by iptables [22]. A testbed system manager assigns IP address ranges for the client hosts to the testbed users. A firewall by iptables at an edge cloud allows to access the Docker container if the source IP address is in the range assigned to the testbed user of the Docker container. Otherwise, the access is denied (see Sect. 4.2.4).

In addition, we deploy Open vSwitch [23] at the wireless base station nodes (i.e., network ingress for client hosts) to block data packets from client hosts with IP addresses outside the assigned range of the testbed user (C-1 and C-2 in Fig. 3). Because root privileges on the client hosts are given to the testbed users to allow installing arbitrary software, the testbed users can set arbitrary IP addresses of the client hosts. The blocking in the wireless base station nodes prevents unauthorized access by changing the IP addresses to be in the range of another testbed user.

4. Details of Implementation

4.1 Testbed Manager

The testbed manager determines the virtual regions and edge clouds on which to place the Docker containers according to requests from testbed users through the APIs. Here, the APIs for the testbed users are the REST APIs of the virtual region creation and Docker container request, as shown in Table 1. We programmed the testbed manager using Python.

4.1.1 Determine Virtual Regions

When a testbed user requests to obtain virtual regions through API U-1 (process 2 in Sect. 3.1), the testbed manager determines the virtual regions for the condition of the RTT r_v in the message body. Virtual region determination (process 3 in Sect. 3.1) is described as follows. Here, let $W = w_1, \cdots, w_{|W|}$ be the set of the wireless base station nodes. The computation is clustering of wireless base station nodes W. The "distance" $d(w_i, w_j, r_v)$ between wireless base stations $w_i, w_j \in W$ for the condition of the RTT r_v is defined as follows:

$$d(w_i, w_j, r_v) = \begin{cases} \frac{1}{s(w_i, w_j, r_v)} & (s(w_i, w_j, r_v) > 0) \\ \infty & (s(w_i, w_j, r_v) = 0) \end{cases},$$

where $s(w_i, w_j, r_v)$ is the total number of allocatable Docker containers in the edge clouds that can communicate with w_i and w_j within RTT r_v. The RTTs between the edge clouds and wireless base station nodes are obtained from the edge clouds through API I-1 in Fig. 1 (see Sect. 4.2.1). The wireless base station nodes are divided into the virtual regions via hierarchical clustering computation [24], where clusters are updated by iterations. Here, let $v_k = \{c_1 \subseteq W, \cdots, c_{|v_k|} \subseteq W : c_l \cap c_m = \emptyset\ (l \neq m), c_1 \cup \cdots \cup c_{|v_k|} = W\}$ be the cluster after the kth iteration. In this clustering process, each of the wireless base station nodes is a cluster at initial, i.e., $v_0 = \{\{w_1\}, \{w_2\}, \cdots, \{w_{|W|}\}\}$. The clusters are integrated by the following iterations. In an iteration, the two clusters with the minimum distance but not ∞ are integrated into a single cluster. The distance $d_c(c_l, c_m)$ between clusters $c_l, c_m \in v_k$ is expressed as follows:

$$d_c(c_l, c_m) = \frac{\sum_{w_l \in c_l, w_m \in c_m} d(w_l, w_m, r_v)}{|c_l| \cdot |c_m|}$$

when $d(w_l, w_m, r_v) \neq \infty, \forall w_l \in c_l, \forall w_m \in c_m$. When there are wireless base stations $w_l \in c_l, w_m \in c_m$ such that $d(w_l, w_m, r_v) = \infty$,

$$d_c(c_l, c_m) = \infty.$$

The iterations stop when no clusters are integrated, i.e., all distances among clusters are ∞. When the iteration stops,

Table 1 REST APIs for testbed users.

Objective	URL	Method	Message body
Virtual region creation (U-1 in Fig. 1)	/ <testbed user name>/virtual_regions	POST	latency: <RTT r_v (ms)>
Docker container request (U-2 in Fig. 1)	/<testbed user name>/virtual_regions/ <virtual region name>/containers	POST	name: <container name> image: <container image name> num: <the number of containers> protocol: <TCP or UDP used for the container service> port: <the port number of the container service>

the resulting clusters are the virtual regions.

According to the determined virtual regions, the testbed manager notifies the condition of the RTT r_v to the edge clouds corresponding to the virtual regions through API I-2 in Fig. 1. Here, the RTT between the corresponding edge cloud and all wireless base station nodes in the virtual region is not greater than r_v. The notified edge clouds use the RTT r_v for future Docker container placements (see Sect. 4.2.2).

After receiving the responses for the notifications from the corresponding edge clouds through API U-1 in Fig. 1, the testbed manager responds to the testbed user with the determined virtual region.

4.1.2 Determine the Edge Cloud to Place a Docker Container

When the testbed user requests a Docker container through API U-2 as process 4 in Sect. 3.1, the testbed manager decides on an edge cloud to create the Docker container. Here, the virtual region is specified by the URL of API U-2. The RTTs between the decided edge cloud and all the wireless base station nodes in the virtual region are not greater than r_v. The testbed manager requests the determined edge cloud using an API of the edge cloud (I-3 in Fig. 1) to create the Docker container (see Sect. 4.2.3). The RTTs between all edge clouds and all wireless base station nodes are periodically notified from the edge clouds through API I-1 in Fig. 1 (see Sect. 4.2.1).

4.2 Edge Cloud

An edge cloud comprises a k8s cluster with a virtual region plugin and a gateway node (Fig. 4). The k8s cluster comprises a master node and worker nodes. The master node has a scheduler to decide on a worker node to place a Docker container. The k8s cluster uses the k8s REST API to receive requests to create the Docker container. The virtual region plugin interfaces with the k8s cluster and the testbed manager. The gateway node includes the functions of network isolation and measuring RTTs between the worker nodes and the wireless base station nodes.

4.2.1 Measure Network RTT

The gateway node periodically sends ICMP requests to all the wireless base station nodes and all the worker nodes, receives ICMP replies, and obtains the RTTs to those nodes.

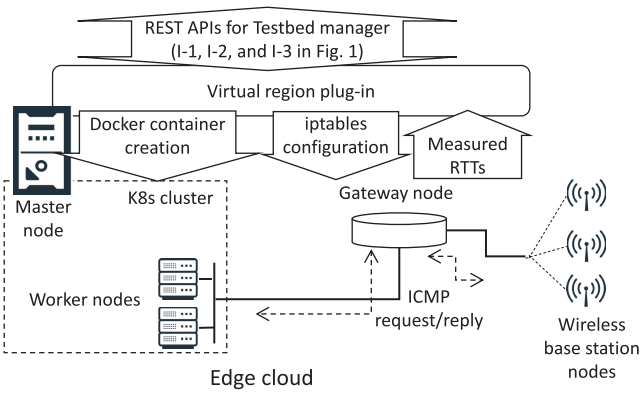

Fig. 4 Edge cloud components.

Then, the gateway node calculates the minimum sum of the RTTs to the worker nodes and the RTT to each of the wireless base station nodes. We call the minimum sum as the RTT between the edge cloud and the wireless base station node, which implies that at least one worker node in the edge cloud can communicate with the wireless base station node within that RTT. The RTTs between the edge clouds and all the wireless base station nodes are notified to the testbed manager via the virtual region plugin and interface I-1 (Fig. 1).

4.2.2 Virtual Region Configuration

According to the virtual regions determined by the testbed manager, the virtual region plugin determines candidate worker nodes for future Docker container creation. The condition of the RTT r_v and the wireless base station nodes in the virtual region are notified through API I-2 in Fig. 1. Using the RTT measurement results (Sect. 4.2.1), the virtual region plugin instructs the master node to label worker nodes with the name of the virtual region, where the RTT to the wireless base station nodes in the virtual region is not greater than r_v. After labeling the candidate worker nodes, the virtual plugin responds to the testbed manager through API I-2. We used the labeling process of a k8s function. To instruct the labeling, the virtual region plugin employs the k8s REST API of the master nodes. We used the Python k8s client library [25] to program the virtual region plugin.

4.2.3 Docker Container Creation

The virtual region plugin sends a request to the master

node to create a Docker container through API I-3 in Fig. 1 when the testbed manager sends a request as described in Sect. 4.1.2. The request message includes the image name, the service port number, and the service protocol. Here, the master node randomly selects a worker node from among worker nodes labeled with the virtual region name. The RTT between any selected worker node and the wireless base stations in the virtual region is not greater than r_v because all labeled worker nodes are. Then, the Docker container is created on the worker node. During the creation of the Docker container, the Docker container image is downloaded from the private registry if the worker node has not downloaded the image previously.

4.2.4 Network Setting of Edge Clouds for Multi-Tenancy

Access control from client hosts: To control access from client hosts, an original Python program in the gateway node dynamically configures rules of iptables to accept access from authorized client hosts while a default rule to block all access from client hosts is set. When a Docker container is created, the virtual region plugin inserts a rule to accept data packets of the source IP address range assigned to the testbed user of the created Docker container. This rule also includes the service port number of the gateway node corresponding to those of the created Docker container to accept the access from the client hosts because the gateway node functions as a network address port translation (NAPT) node. Note that the service port number of the gateway node is selected randomly by the virtual region plugin when the Docker container is created.

Access control inside an edge cloud: Here, the k8s network plugin, Calico [21], is used to control the network inside a k8s cluster. The virtual region plugin configures Calico to allow communication of Docker containers labeled by the same testbed user name in the same edge cloud, i.e., the same k8s cluster. In addition, Calico is set to deny the communication of Docker containers labeled by different testbed user names. Here, when Docker containers are created, they are labeled with the requesting testbed user name.

Access control for Docker containers in different edge clouds: The virtual region plugin dynamically configures Calico to allow a Docker container to access Docker containers in other edge clouds while the default configuration for Docker containers is denying access outside the edge clouds. When a Docker container is created in an edge cloud, the virtual region plugins of the remaining edge clouds allow Docker containers of the same testbed user to access the external IP address and the service port number corresponding to the created Docker container.

5. Evaluation

We evaluated the proposed edge computing testbed system experimentally using real machines to investigate its effectiveness and overhead. Here, effectiveness refers reducing the amount of the settings required by a testbed user to obtain a Docker container, and overhead refers to the system's running time to obtain a Docker container.

5.1 Experimental Infrastructure

In this experiment, we deployed the proposed testbed system using a server and network experimental system in a data center, i.e., StarBED [26]. We constructed an edge computing testbed comprising 11 edge clouds (ec1–ec11), six wireless base station nodes (ap1–ap6), 12 client hosts, the testbed manager, and the testbed user hosts (Fig. 5). Here, a tree topology was used for the edge network, in which ec1 was the root and ap1–ap6 were the leaves. Two client hosts of different testbed users were connected per wireless base station node, and the testbed user hosts used the testbed API of the testbed manager. All hosts were virtual machines running the Ubuntu 20.04 LTS operating system. Here, the testbed user hosts, gateway nodes in the edge clouds, and the wireless base station nodes had two virtual

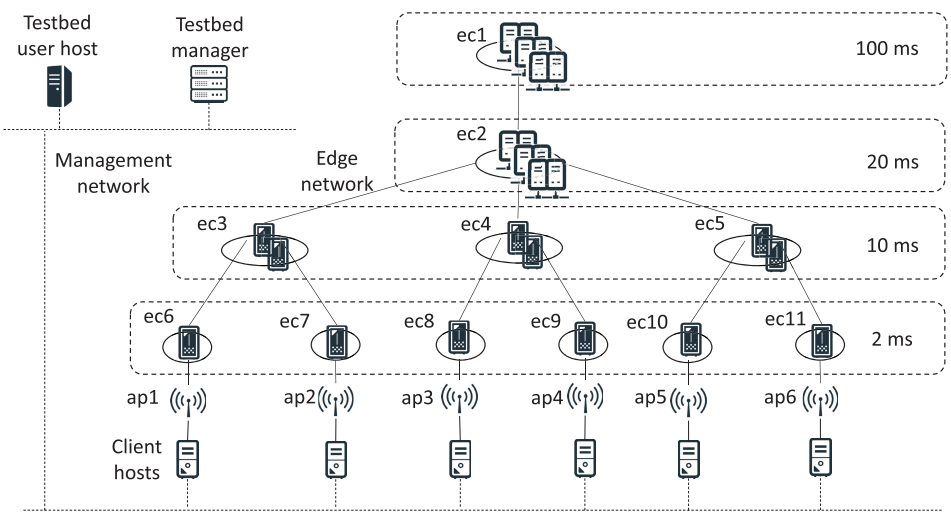

Fig. 5 Testbed system constructed in StarBED.

CPU (vCPU) cores of Intel Xeon E312xx at 1.99 GHz and 2 GB of memory. The testbed manager, master nodes in the edge clouds, and the client hosts had four vCPU cores of the same model and clock frequency and 8 GB of memory. There were three worker nodes in each of ec1 and ec2, and each worker node had 20 vCPU cores of Intel Core Processor (Broadwell, IBRS) and 2.09 GHz, and 100 GB of memory. In each of ec3–ec5, there were two worker nodes with the 10 vCPU cores of the same model and clock frequency and 50 GB of memory. In each of ec6–ec11, there was one worker node with the same vCPU cores and clock frequency, and the same amount of memory.

To emulate edge clouds deployed over a wide-area network, we added latency to interfaces of the bridges forming the edge network using Linux tc command [27] while the StarBED network was a local network. By adding latency, the RTT between an edge cloud (i.e., a gateway node) and the wireless base station nodes in a sub-tree whose root was the edge cloud was set as shown in Fig. 5. For example, the RTTs between ec1 and wireless base station nodes ap1–ap6 were 100 ms, the RTTs between ec3 and wireless base station nodes ap1 and ap2 were 10 ms, and the RTT between ec6 and wireless base station node ap1 was 2 ms. The RTTs 100 ms and 20 ms assumed communication with the cloud data center in a foreign country and the same country, respectively. The RTT 10 ms assumed communication with edge cloud in the same prefecture in Japan. The RTT 2 ms assumed communication with an edge cloud directly linked to a wireless base station.

The experimental infrastructure had 240 vCPU cores in all the worker nodes in total and could offer 240 Docker containers to testbed users in total when each Docker container used a single vCPU core and each testbed user set r_v to 100 ms or greater. According to the edge network topology and the network RTT, the RTT between any edge cloud and wireless base station node was 100 ms or less.

To evaluate system loads versus system scales, we conducted the same experiments using experimental infrastructures of different scales. We call the infrastructure described above (Fig. 5) the "large-scale infrastructure", in below. We call an infrastructure, in which ec5, ec10, ec11, ap5, and ap6 with their connecting links and client hosts are from the large-scale infrastructure, a "middle-scale" infrastructure. We call an infrastructure, in which ec4, ec8, ec9, ap3, and ap4 with their connecting links and client hosts are from the middle-scale infrastructure, a "small-scale" infrastructure.

5.2 Compared Conventional System

We compared the proposed testbed system to a conventional system (CS), in which a testbed user directly decided k8s worker nodes to place Docker containers and set up Docker containers using k8s (i.e., kubectl) and iptables commands. Because state-of-the-art systems described in Sect. 2 have no schedulers for Docker container placement to satisfy the RTT conditions, the testbed user directly decided worker nodes to place Docker containers, seeing the network RTTs between all worker nodes and wireless base station nodes.

5.3 User Costs on Docker Container Creation

We evaluated user costs on creating a Docker container by estimating command argument length, the number of values to be considered to decide the arguments, and the number of times to run the commands. We believe that the lengths of the argument values (i.e., the numbers of characters) affect the user's working time of input. This is because the argument values are different according to each Docker container while the commands and default values of arguments and options can be written in script files. The number of values to be considered and the number of times to run the commands also affect the user's working time. We compared those of CS and the proposal. Table 2 summarizes the results. In the estimation, we assumed that the length of the arguments defined by strings, which were names of a worker node, a Docker image, a testbed user, and a virtual region, was five, i.e., five characters. The maximum service port number was 49151 (i.e., five characters), according to port number assignment by Internet Assigned Numbers Authority [28].

In the CS, the testbed user ran three kubectl commands in master nodes and one iptables command in a gateway node when creating a Docker container. The first kubectl command was to create the Docker container and had arguments of the testbed user name, the worker node, the service port number, and the Docker image name. The testbed user name was used to label the Docker container. Because directly specifying the worker node to place the Docker container, the testbed user needed to know the RTTs between all the worker nodes and all the wireless base station nodes to satisfy an RTT condition. The number of the RTT values was $|W| \cdot m$, where $|W|$ and m were the numbers of wireless base station nodes and worker nodes in the testbed system, respectively. For example, in the large-scale experimental infrastructure, $|W|$ was 6 and $|W| * m$ was 108, which had a large impact on the number of values seen by a human. The second kubectl command allowed communication among Docker containers of the same testbed user in the same edge clouds and required the testbed user name for labeling the Docker containers to identify their owner. The iptables command configured the external service port number of the gateway node to allow the created Docker container to accept access from client hosts of the testbed user. The command had the argument of the external service port number. The testbed user needed to select the service port number that was not used for other Docker containers in the same edge cloud. The third kubectl command allowed Docker containers of the same testbed user in the remaining edge clouds to access the created Docker container. The arguments were the testbed user name, external IP address, and service port number of the created Docker container. The argument length of the IP address and the service port number was 20 consisting of 15 characters of

Table 2 User commands and costs on creating a Docker container.

	Command	Argument length	Values to be considered to decide the arguments and their number	Required number of times to run the command
CS	kubectl for Docker container creation	Testbed user name: 5 Worker node: 5 Service port number: 5 Docker image name: 5	The RTTs between a all the wireless base station nodes and all the edge clouds: $\|W\| \cdot m$, where $\|W\|$ is the number of and m is the number all the worker nodes	Once
	kubectl for network policy in the same edge cloud	Testbed user name: 5	None	Once.
	iptables configuration	External service port number: 5	Used listening ports for Docker containers in the same edge cloud: the number of existing Docker containers in the same edge cloud	Once
	kubectl for network policy in other edge clouds	Testbed user name: 5 External IP address and service port number of the created Docker containers: 20	None	The product of the number of created Docker containers and the remaining edge clouds
Proposal	REST API for virtual region creation	Condition of the RTT: 3	None	Once
	REST API for Docker container creation	Virtual region: 5 Service port number: 5 Docker image name: 5	Virtual region names: $\|W\|$ at most	Once

an IPv4 address including dots and five characters of a service port number.

In the proposal, the testbed user employed the two REST APIs. The first REST API had the argument of the RTT condition and provided virtual regions. Because the RTT condition was an original requirement of the testbed user, the testbed user did not need to see any data to decide the argument. The second REST API created a Docker container. The testbed user needed to see at most $|W|$ virtual region names provided by the first REST API to select the virtual region included in the arguments of the second REST API.

According to the results, the total argument length of CS was at least 55 while that of the proposal was 19. The length of CS becomes longer as existing Docker containers in the remaining edge clouds increase. Thus, the proposal reduces the argument length to approximately one-third or less.

5.4 System Load of RTT Measurement

We measured the CPU load of a gateway node. In the implementation, at every three minutes, the gateway nodes sent ICMP request packets to the worker nodes in the same edge clouds and the wireless base station nodes ten times, calculated the average RTT, and determined the RTTs between the worker nodes and the wireless base station nodes, as described in Sect. 4.2.1.

We measured CPU load of gateway nodes using sysstat tools [29] at every one second in the RTT measurement process. Figure 6 shows the CPU loads averaged over ten measurement processes in gateway nodes, ec1, ec2, ec3, and ec6 in the large-scale experimental infrastructure. The gateway nodes were at different levels of the edge network tree. The average CPU load increased as the total waiting

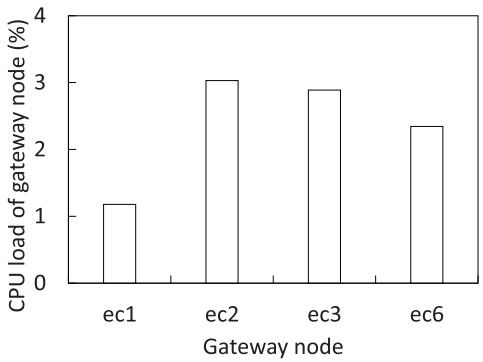

Fig. 6 CPU load of gateway nodes during RTT measurement.

time of ICMP replies during the measurement process decreased. The average total RTTs of ICMP replies over ten measurement processes were 611.7 ms, 128.3 ms, 156.7 ms, and 190.9 ms for ec1, ec2, ec3, and ec6, respectively. The RTTs between the edge cloud and the wireless base stations nodes mainly affected the total waiting time. The maximum CPU load was 3.0% at most, which had a small impact.

Network traffic loads in the edge network by ICMP packets were the maximum when all the 11 gateway nodes sent ICMP packets simultaneously. The Ethernet frame length of an ICMP request or reply was 64 B. Thus, the maximum data size sent to a wireless base station node in the large-scale experimental infrastructure was 704 B, which had a small impact on the typical link bandwidth, e.g., 1 Gbps or more.

5.5 System Load of Virtual Region Creation

We measured the time of the virtual region creation processes in the large, middle, and small-scale experimental infrastructures. Here, the measured time was between send-

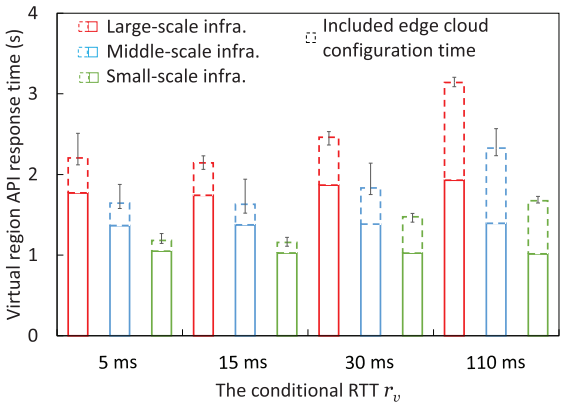

Fig. 7 Virtual region creation time: the times were merely several seconds.

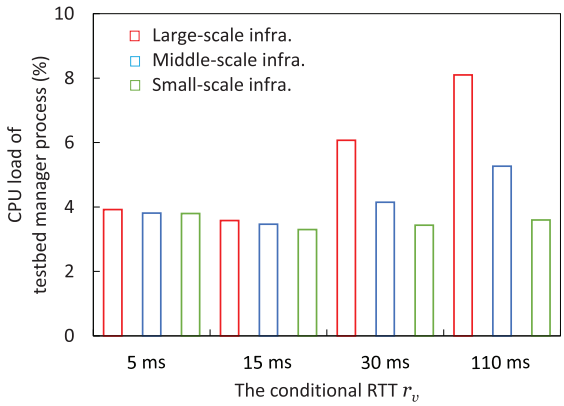

Fig. 8 CPU load of testbed manager during virtual region creation.

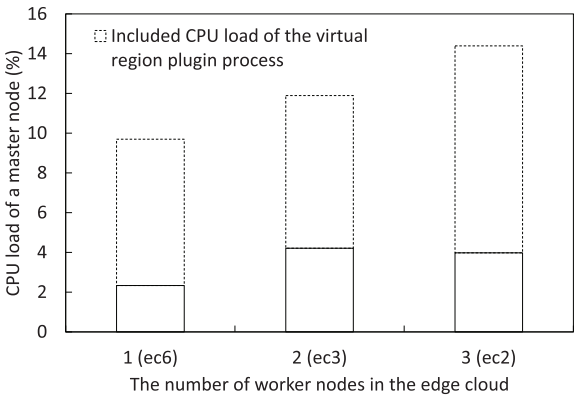

Fig. 9 CPU load of master node during virtual region creation.

ing a request to create virtual regions using the API from the testbed user host and receiving the reply. In this evaluation, the condition of the RTT r_v was 5 ms, 15 ms, 30 ms, or 110 ms. The RTT conditions of 5 ms and 15 ms corresponded to connected cars [30] and AR applications [3], respectively. The RTT conditions of 30 ms and 100 ms assumed cloud environments just for testing server programs of IoT applications in the cloud environments, e.g., performance comparison of the same server program running in edge and cloud environments. Figure 7 shows the measured times versus the condition of the RTT r_v. For each, the average, maximum, and minimum over the ten same experiments are shown. Within the measured time, the average time of configuring edge clouds according to the created virtual regions (Sect. 4.2.2) is shown by dashed line as a part of the average time. The configuration time was measured in the testbed manager program.

In a comparison of the times versus the condition of the RTTs in the same experimental infrastructures, the API response time primarily depended on the time of configuring the edge clouds while the remainder including virtual region computation time was almost the same for all the RTT conditions. The time required to configure the edge clouds increased as the number of configured edge clouds increased. Edge clouds to be set for a virtual region are all the edge clouds, where the RTT to all wireless base stations in the virtual region is not greater than the specified r_v. For instance, when r_v was 5 ms, six edge clouds needed to be set in the large-scale experimental infrastructure. Here, the six wireless base station nodes (ap1–ap6) were individual virtual regions. The corresponding edge clouds of the virtual regions were ec6–ec11. In addition, when r_v was set to 15 ms, three edge clouds were set. Here, there were three virtual regions, i.e., ap1 and ap2 were in a virtual region, ap2 and ap3 were in another virtual region, and ap4 and ap5 were in a third virtual region. ec3–ec5 were the corresponding edge clouds, respectively. When r_v was set to 30 ms, there were four edge clouds to set. Here, there was a single virtual region including all wireless base station nodes, and the four edge clouds were ec2–ec5. When r_v was set to 110 ms, all the 11 edge clouds were set for a single virtual region.

A testbed user creates virtual regions merely once in advance of Docker container creation. Thus, the time of virtual region creation has a negligible impact on the testbed user.

In a comparison of the times of the different experimental infrastructure scales in the same RTT conditions, the time increased as the scale became large. As a result, the times of the virtual region computation and the edge cloud configuration became longer because there were more edge clouds in a larger scale infrastructure.

In addition to the time measurements, we measured average CPU usage at every second in the processes in the testbed manager and the master nodes. Figure 8 shows the average CPU usage of the testbed manager process versus the RTT conditions in the different experimental infrastructure scales. The CPU usage of the testbed manager had the same tendency of the processing time results in Fig. 7. The CPU load became large when the experimental infrastructure or the number of configured edge clouds increased. Figure 9 shows the CPU loads of the master nodes including the load of the virtual region plugin processes versus the number of worker nodes in the edge clouds. The remaining CPU load included the CPU load of k8s processes. The CPU load increased as the number of worker nodes in the same edge clouds increased. When a virtual region was config-

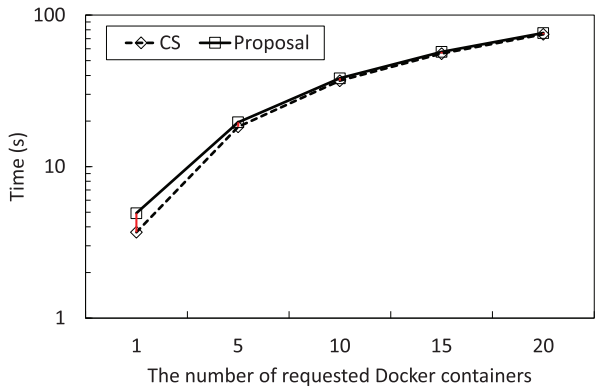

Fig. 10 Docker container creation time: the increased ratio of the proposal to CS was small for increased number of requested Docker containers.

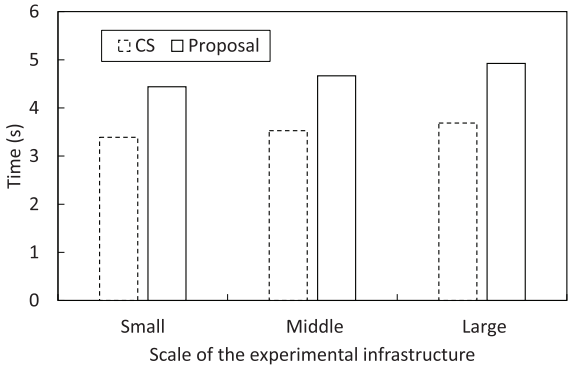

Fig. 11 Docker container creation time vs. the experimental infrastructure scales.

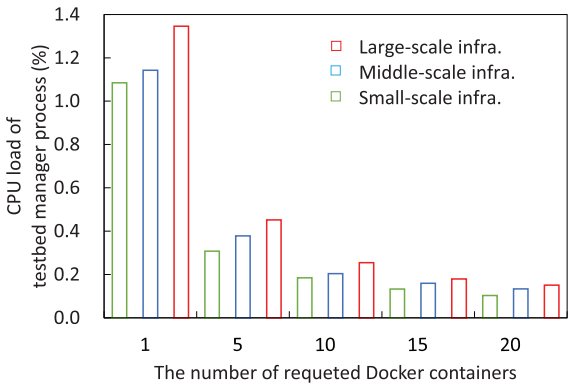

Fig. 12 CPU load of the testbed manager during Docker container creation.

ured for the edge cloud, each worker node was labeled with a virtual region name. The load for the labeling increased in both the virtual region plugin and the k8s processes. All the CPU loads on virtual region creation in a testbed manager and master nodes were less than 15%, which had a small impact.

5.6 System Load of Docker Container Creation

We measured the Docker container creation time and compared it to that of the CS. In the proposed testbed system experiment, the measured time was between sending a request to create a Docker container using the API from the testbed user's host and receiving the corresponding reply. In the CS experiment, the measured time was the running time of the kubectl commands on the master nodes and the iptables commands on the gateway nodes in the edge clouds. Here, the kubectl commands created Docker containers and configured the network policy, and the iptables commands configured to control clients' access, as the proposed system did. The number of requested Docker containers (i.e., "num" specified in the request API in Table 1) was 1, 5, 10, 15, or 20. Note that "image," "protocol," and "port" specified in the request API were an original image running httpd, TCP, and 80, respectively. In the experiments, the used image was downloaded from the private registry on all worker nodes in advance. Thus, the measured Docker container creation time did not include the time of downloading the Docker image.

Figure 10 shows the Docker container creation times versus the number of requested Docker containers on the large-scale experimental infrastructure. The times are the average over the ten same experiments. The overhead time by the proposed system was about 1.3 s when one Docker container was requested. The overhead time increased up to about 1.5 s as the number of requested Docker containers increased.

Figure 11 shows the Docker container creation time versus the experimental infrastructure scales. The number of requested Docker containers was one. As the scale became larger, the times of both of the methods became longer because the remaining edge clouds to be configured to allow access to the created Docker container increased. The overhead time by the proposal was 1.0 s, 1.1 s, and 1.2 s for the small, middle, and large-scale experimental infrastructures, respectively. The total overhead time was the sum of 1.2 s for Docker container creation and 3.1 s for virtual region creation in the large-scale experimental infrastructure, i.e., 4.3 s. The overhead time has a negligible impact on testbed users.

Figure 12 shows the average CPU load of the testbed manager during the Docker container creation versus the number of requested Docker containers in the large, middle, and small-experimental infrastructures. The average CPU load decreased as the number of requested Docker containers increased because the time of waiting for Docker container creation in an edge cloud increased. When the number of requested Docker containers was the same, the CPU load increased as the experimental infrastructure became large because of configuring the remaining edge clouds to allow accessing the created Docker containers. Figure 13 shows the average CPU load of the master node that created the requested Docker containers versus the number of requested Docker containers. The shown CPU load includes that of the virtual region plugin process while the remaining CPU load includes that of the k8s processes. As the number of

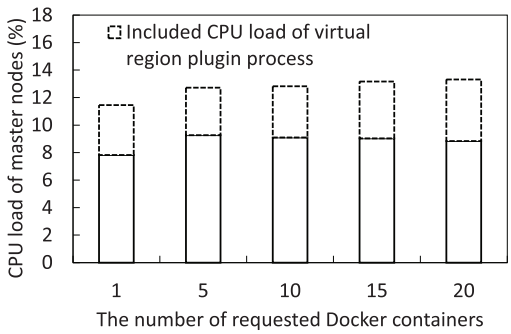

Fig. 13 CPU load of the master node during Docker container creation.

requested Docker containers increased, the CPU loads of the virtual region and the k8s processes increased. All the CPU loads on the Docker container creation processes in the testbed manager and the master node were less than 15%, which had a small impact.

6. Conclusion

In this paper, we have proposed a testbed system to experimental environment setup and facilitate IoT application deployment in production environments. The virtual region-based interface and automated k8s configuration of the proposed testbed system enable easy configuration of experimental environments, in which the RTTs between the Docker containers and clients are ensured to guarantee response times for tested applications. In addition, the proposed testbed system supports multi-tenancy by isolating the network communication of different testbed users. The experimental results demonstrate that the testbed user burden is reduced by reducing the amount of configuration data that must be considered to run commands when creating experimental environments. In addition, the experimental results demonstrate that the overhead of the proposed system in terms of system run time is small.

In the future, the proposed system must provide a more realistic edge computing environment. The current system network is created using all wired links; however, in real-world environments, clients communicate using wireless communication technologies. Thus, we plan to consider using a wireless emulation technique to simulate wireless communication in the experimental environment. We will consider techniques to guarantee the required RTT, e.g., network QoS control. We will also consider guaranteeing system performance according to testbed user preferences. For example, resource allocation and isolation techniques could be applied to provide guaranteed CPU, memory, and storage resources for Docker containers, as well as network bandwidth.

References

[1] T. Verbelen, P. Simoens, F.D. Turck, and B. Dhoedt, "Adaptive deployment and configuration for mobile augmented reality in the cloudlet," Journal of Network and Computer Applications, vol.41, pp.206–216, May 2014.
[2] H.D. Park, O.-G. Min, and Y.-J. Lee, "Scalable architecture for an automated surveillance system using edge computing," The Journal of Supercomputing, vol.73, no.3, pp.926–939, 2017.
[3] ITU-T, "The tactile internet." https://www.itu.int/dms_pub/itu-t/oth/23/01/T23010000230001PDFE.pdf, Aug. 2014. ITU-T Technology Watch Report.
[4] J. Cappos, M. Hemmings, R. McGeer, A. Rafetseder, and G. Ricart, "EdgeNet: A global cloud that spreads by local action," Proceeding of 2018 Third ACM/IEEE Symposium on Edge Computing (SEC 2018), pp.359–360, Oct. 2018.
[5] "Akraino - LF Edge." https://www.lfedge.org/projects/akraino/
[6] "Empowering App Development for Developers | Docker." https://www.docker.com/
[7] "Kubernetes." https://kubernetes.io/
[8] D. Krishnaswamy, R. Kothari, and V. Gabale, "Latency and policy aware hierarchical partitioning for nfv systems," 2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN), pp.205–211, 2015.
[9] H. Yamanaka, E. Kawai, Y. Teranishi, and H. Harai, "Proximity-aware IaaS in an edge computing environment with user dynamics," IEEE Transactions on Network and Service Management, vol.16, no.3, pp.1282–1296, Sept. 2019.
[10] H. Yamanaka, Y. Teranishi, E. Kawai, H. Nagano, and H. Harai, "Design of an edge computing testbed to simplify experimental setup," 2021 24th International Symposium on Wireless Personal Multimedia Communications (WPMC), Dec. 2021.
[11] T.-C. Chiu, W.-H. Chung, A.-C. Pang, Y.-J. Yu, and P.-H. Yen, "Ultra-low latency service provision in 5G fog-radio access networks," 2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), IEEE, Sept. 2016.
[12] Y. Nakata, M. Takai, H. Konoura, and M. Kinoshita, "Cross-site edge framework for location-awareness distributed edge-computing applications," 2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud), pp.63–68, IEEE, Aug. 2020.
[13] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with KubeEdge," 2018 IEEE/ACM Symposium on Edge Computing (SEC), pp.373–377, IEEE, Oct. 2018.
[14] T. Dreibholz, "Flexible 4G/5G testbed setup for mobile edge computing using OpenAirInterface and open source MANO," Advances in Intelligent Systems and Computing, vol.1150, pp.1143–1153, Springer International Publishing, March 2020.
[15] "OpenAirInterface." https://openairinterface.org/
[16] "Open source MANO." https://osm.etsi.org/
[17] D. Haja, M. Szalay, B. Sonkoly, G. Pongracz, and L. Toka, "Sharpening kubernetes for the edge," Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos on - SIGCOMM Posters and Demos '19, pp.136–137, ACM Press, Aug. 2019.
[18] A. Ceselli, M. Premoli, and S. Secci, "Mobile edge cloud network design optimization," IEEE/ACM Transactions on Networking, vol.25, no.3, pp.1818–1831, Feb. 2017.
[19] B. Amento, B. Balasubramanian, R.J. Hall, K. Joshi, G. Jung, and K.H. Purdy, "FocusStack: Orchestrating edge clouds using location-based focus of attention," 2016 IEEE/ACM Symposium on Edge Computing (SEC), pp.179–191, IEEE, Oct. 2016.
[20] S. Kekki, W. Featherstone, Y. Fang, P. Kuure, A. Li, A. Ranjan, D. Purkayastha, F. Jiangping, D. Frydman, G. Verin, K.W. Wen, K. Kim, R. Arora, A. Odgers, L.M. Contreras, and S. Scarpina, "ETSI White Paper No. 28 MEC in 5G networks." https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp28_mec_in_5G_FINAL.pdf, June 2018.
[21] "Project Calico – Secure Networking for the Cloud Native Era." https://www.projectcalico.org/
[22] "The netfilter.org "iptables" project." https://www.netfilter.org/projects/iptables/index.html

[23] “Open vSwitch.” http://openvswitch.org/
[24] R.R. Sokal and C.D. Michener, “A statistical method for evaluating systematic relationships,” University of Kansas Science Bulletin, vol.38, pp.1409–1438, 1958.
[25] “Github - kubernetes-client/python: Official python client library for kubernetes.” https://github.com/kubernetes-client/python
[26] “StarBED4 Project website.” https://starbed.nict.go.jp/en/index.html
[27] “tc(8) – Linux manual page.” https://man7.org/linux/man-pages/man8/tc.8.html
[28] “Service Name and Transport Protocol Port Number Registry.” https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml
[29] “sysstat(5) – Linux manual page.” https://man7.org/linux/man-pages/man5/sysstat.5.html
[30] “Metis deliverable d1.1: Scenarios, requirements and kpis for 5g mobile and wireless system.” https://www.metis2020.com/wp-content/uploads/deliverables/METIS_D1.1_v1.pdf, Nov. 2013.

Hiroaki Yamanaka received M.E. and Ph.D. degrees from Osaka University in 2008 and 2011, respectively. Since 2011, he has been a researcher at the National Institute of Information and Communications Technology (NICT). He has researched network virtualization and software-defined networking (SDN) technologies. His current research work focuses on edge computing. His research interests include infrastructure virtualization and resource management technologies.

Yuuichi Teranishi received his M.E. and Ph.D. degrees from Osaka University, Japan, in 1995 and 2004, respectively. From 1995 to 2004, he was engaged Nippon Telegraph and Telephone Corporation (NTT). From 2005 to 2007, he was a lecturer of Cybermedia Center, Osaka University. From 2007 to 2011, he was an associate professor of Graduate School of Information Science and Technology, Osaka University. Since August 2011, He has been a research manager of National Institute of Information and Communications Technology (NICT). He received IPSJ Best Paper Award in 2011. His research interests include technologies for distributed network systems and applications.

Eiji Kawai received Ph.D. from Nara Institute of Science and Technology in 2001. He joined the National Institute of Information and Communications Technology (NICT) in 2009 and is an executive researcher at ICT Testbed Research, Development and Operations Laboratory. He has been working for many years on a wide variety of information system and service technologies such as operating systems, networking, cloud computing, and application services. Especially, he is interested in the technologies for advanced testbeds.

Hidehisa Nagano received the B.Eng. and M.Eng. degrees in information and computer sciences in 1994 and 1996, respectively, and the Ph.D. degree in information science and technology in 2005, all from Osaka University, Japan. From 1996 to 2020, he worked for Nippon Telegraph and Telephone (NTT) Corporation, Japan. From 2011 to 2012, he was a Visiting Researcher at Queen Mary University of London, U.K. Since 2020, he has been working for National Institute of Information and Communications Technology (NICT), where he is currently the Director of ICT Testbed Research, Development and Operations Laboratory. His research interests include media information processing, AI, and AI network. Dr. Nagano is a senior member of IEEE and a member of IEICE and IPSJ. He was awarded the Maejima Award in 2010, the IEICE Achievement Award in 2017, and the Commendation for Science and Technology by the Minister of Education, Culture, Sports, Science and Technology of Japan (Prizes for Science and Technology, Research Category) in 2019.

Hiroaki Harai received a Ph.D. degree in information and computer science from Osaka University, Osaka, Japan in 1998. After he led R&D on innovative network architecture technologies, he is currently a director general at the National Institute of Information and Communications Technology, Tokyo, Japan, where he is leading R&D testbed construction and stable operation of networks and clouds. Dr. Harai was named an Outstanding Young Researcher at the 3rd IEEE ComSoc Asia- Pacific Young Researcher Award, 2007 (optical network topic). He received a Best Paper Award in ITU Kaleidoscope Academic Conference 2014 (mobile sensor network topic).