Master's Thesis

Title

Service Function Reallocation Method for Low-latency Video Live Streaming in Multi-access Edge Computing

Supervisor

Professor Masayuki Murata

Author

Junichi Kaneda

February 8th, 2019

Department of Information Networking

Graduate School of Information Science and Technology

Osaka University

Master's Thesis

Service Function Reallocation Method for Low-latency Video Live Streaming in Multi-access Edge Computing

Junichi Kaneda

Abstract

In recent years, with the progress of IoT (Internet of Things), many new applications and services that process data at remote base have appeared. In such new services, it is important to reduce application-level delays, i.e. end-to-end delays experienced by the users, in order to improve the users' quality of experience (QoE). The concept of Multi-access Edge Computing (MEC) is expected that responsiveness to services will be improved by relaxing load concentration and reducing the delay due to geographical factors. However, the processing capability of edge servers is generally lower than that of data centers. Service functions capable of effectively reducing application-level delays should be deployed on edge servers. For future deployment of MEC, it is therefore necessary to clarify application-level delays and its occurrence factors in a MEC environment, and to investigate how to make the application-level delay lower by MEC configurations such as the locations and reallocations of the functions.

In this thesis, we construct an MEC environment using OpenStack and Amazon Web Service (AWS), then operate a video live streaming service with real-time processing supposing a shopping agent service using robots. We measure the application-level delay of video live streaming and analyze the delay in detail. Results of our analysis shows that, for low-latency video live streaming, it is naturally important that the average of propagation delays is small, but it is also important to suppress the jitter caused by the network side. Based on this finding, we devise and implement a simple service reallocation method for low-latency video live streaming. In our method, we use a CPU load on edge server as the measure of increase of jitter. Our experimental results show the application-level delay is kept under 400 ms and video quality is comfortable by the service function reallocation method.

Keywords

Multi-access Edge Computing

Video Live Streaming

Real-time Processing

Application-level Delay

Live Migration of Virtual Machine

Contents

List of Figures

List of Tables

1 Introduction

In recent years, with the progress of IoT (Internet of Things), many new applications and services have appeared and information networks are rapidly changing. In addition, modern mobile devices are equipped with many sensors and cameras. New services use information from such mobile devices and provide realistic experience for the users. Information is first transferred to a data center and processed there [1]. Then, the results are returned to mobile devices as necessary [2]. Telexistence is one of such new service, which uses a remote robot, a VR headset and a haptic feedback device as end devices.

In such new services, it is important to reduce application-level delays, i.e. end-to-end delays experienced by the users, in order to improve the users' quality of experience (QoE). However, processing at a data center can lead to penalties in the form of large application-level delay due to geographical factors and load concentration [3]. In the experiment described later, the application-level delays of about 520 ms to 750 ms are occurred in the video live streaming service with real-time processing.

The concept of Multi-access Edge Computing (MEC) has been introduced to mitigate delays [3–5]. MEC virtualizes service functions and deploys them on virtual machines on edge servers. An edge server is a secondary data center located at the network edge, closer to the user. Incorporating Network Function Virtualization (NFV) to MEC is expected to allow flexible changes in resources and deployment locations of virtual machines [4–6]. In a MEC environment, service applications use functions at edge servers rather than at data centers. As a result, it is expected that responsiveness to services will be improved by relaxing load concentration and reducing the delay due to geographical factors.

However, the processing capability of edge servers is generally lower than that of data centers. Service functions capable of effectively reducing application-level delays should be deployed on edge servers, since it is impossible to deploy all service functions on an edge server. For future deployment of MEC, it is therefore necessary to clarify application-level delays and its occurrence factors in a MEC environment, and to investigate how to make the application-level delay lower by MEC configurations such as the locations and reallocations of the functions. It is also needed to clarify the effects of service function reallocation on the application-level delay. Moreover, it is important to obtain policies for

service function reallocation for the purpose of low-latency service provision.

In this thesis, we construct an MEC environment using OpenStack and Amazon Web Service (AWS), then operate a video live streaming service with real-time processing supposing a shopping agent service using robots. A real-time image processing function is deployed on an edge server in the constructed MEC environment. By manually changing the location of the function, we measure the application-level delay of video live streaming and analyze the delay in detail. As a result of the analysis, it is revealed that the increase of buffering time for absorbing jitter, i.e. a variation in packet arriving intervals, increases application-level delays significantly. Therefore, in order to realize low-latency video live streaming, it is naturally important that the average of propagation delays is small, but it is also important to suppress the jitter caused by the network side. Based on this finding, we devise a service function reallocation method for low-latency video live streaming. In our method, we focus on the increases of jitter and application-level delay caused by the increase of the CPU load of the edge server the function is deployed. And the timing of service function reallocation is determined based on the CPU load of edge servers. We implement a simple service reallocation method and confirm the effect of the service function reallocation method on application-level delay.

The remainder of this thesis is organized as follows. Section 2 describes the implementation of a MEC environment and a video live streaming service with real-time processing. In Section 3, we measure application-level delays and analyze the factors of occurring the delays. In Section 4, we devise a service function reallocation method and confirm the effect of the method through our MEC environment. In Section 5, we present our conclusions and future works.

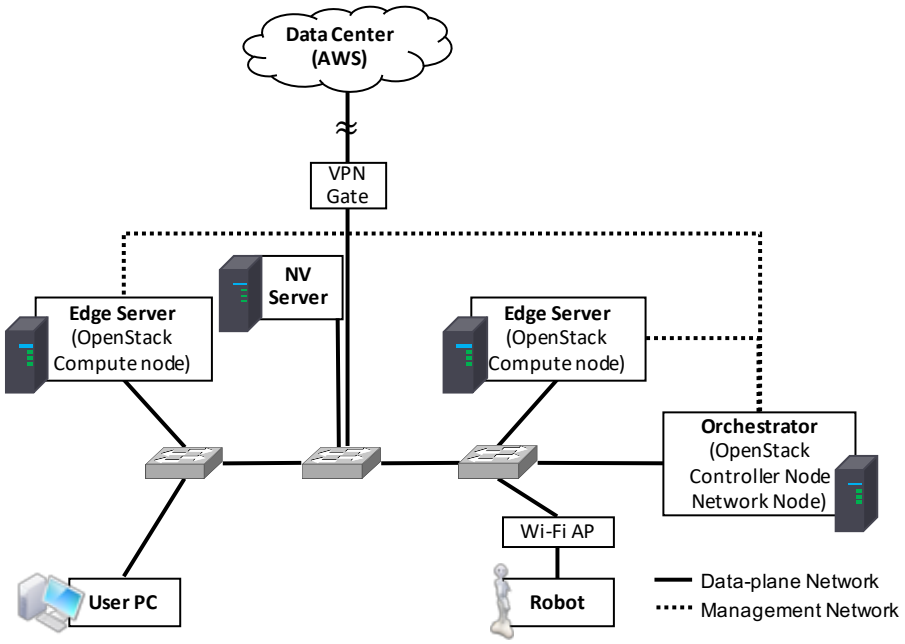


Figure 1: Configuration of the MEC environment

2 Implementation

In this section, we explain the construction of a MEC environment using OpenStack and the configuration of applications operating on the MEC environment.

2.1 Multi-access Edge Computing Environment

To build a MEC environment, we use OpenStack, which is open-source software for creating virtualization environments. The white paper about MEC framework [6] states that MEC incorporates NFV for virtualization. Taking advantage of virtualization allows orchestrating MEC services and reallocating service functions based on the load of edge servers and on the users' movement. Since OpenStack has already been adopted in many implementation projects of NFV [7, 8], it is considered appropriate to use OpenStack to build a virtualization environment for MEC. Also, the OpenStack development community expects OpenStack to be applied to the edge [9].

We built the MEC environment in our laboratory by connecting four similarly configured server machines, a user's PC and a robot with switches. These devices are also

Table 1: Specifications of Server Machines and Virtual Machines

	Server Machines	VM on OpenStack	VM on AWS (t2.medium)
# of CPU Cores	28	2	2
Main Memory	64 GB	4 GB	4GB
Storage	720 GB	20 GB	-
Network Performance	10 Gbps	1 Gbps	(Unspecified)
Operating System	CentOS 7	CentOS 7	CentOS 7

connected to virtual machines on Amazon Web Service (AWS). An overview of the MEC environment is shown in Figure 1. Three of the four similarly configured machines are OpenStack nodes, one operating as an OpenStack controller and network node, and the other two as OpenStack compute nodes. The controller node operate as a control unit such as infrastructure manager and orchestrator in a virtualization environment. We call this node the orchestrator. The compute nodes operate as edge servers. In the shopping agent service, since the robot and the user are geographically separated, the two edge servers are prepared as processing bases close to each other. CentOS 7 and Kernel-based Virtual Machine (KVM) are installed on the compute nodes as a host OS and a hypervisor, respectively. Applications are executed in virtual machines with 4 GB of memory and 32 GB of storage on the compute nodes. Storage files of all virtual machines are shared by Network File System (NFS) among three OpenStack nodes. The fourth server machine is not virtualized for comparison. Hereafter, we refer to this non-virtualized server as the NV server. We also use virtual machines on AWS as data centers at the center of network. The specifications of the server machines and the virtual machines are shown in Table 1.

The robot, the user PC, the virtual machines on the edge servers and AWS, and the NV server communicate using a data-plane network. The robot is wirelessly connected to the data-plane network via a Wi-Fi access point. The devices and the virtual machines on the edge servers are connected on local area network (LAN) scale. The virtual machines on AWS are connected to the data-plane network using Virtual Private Network (VPN) via

the Internet. The orchestrator uses an out-of-band network to communicate with the edge servers so that they are not at all involved in the data-plane communication. Hereafter, we call this out-of-band network the management network.

OpenStack is open-source software for creating virtualization environments. Its architecture is one in which several modularized software operate in cooperation through application programming interfaces (API). Modularized software in OpenStack are called "projects", each performing one of many functions in the virtualized environment. Projects are divided into smaller agents and deployed on one or more of three physical node types: a controller, network, or compute node. Controller nodes are those on which agents managing the overall OpenStack system are deployed. Networking-related agents are deployed on network nodes. Compute nodes are those on which virtual machines are actually executed. Agents managing virtual machines and providing network connectivity are also deployed on compute nodes. Each node type is connected by a network, and agents deployed on each node communicate with each other, realizing operation of the virtualization environment.

We use OpenStack Ocata, released in February 2017. Seven projects are installed on the nodes: Nova, Neutron, Keystone, Glance, Horizon, Ceilometer, and Heat. On the two OpenStack compute nodes, virtual routers and virtual switches allow the virtual machines to connect to the data-plane network. We use Open vSwitch for switching function. In addition to basic Layer 2 and 3 functions, distributed virtual router (DVR) is enabled. The DVR allows distributing virtual routers to all compute nodes, while only a single virtual router is deployed on the network nodes by default configuration.

2.2 Video Live Streaming Service with Real-time Processing

As a potential new service, we consider realization of a shopping agent service using robots. In this service, robots go to a physical store, and users can shop from home as if they were actually there. Using AR/VR technology, product information is superimposed on the video taken by a camera mounted on the robot and presented to the user. This process is performed on an external server, and product information is acquired from cloud. Sensing technology can also used to present tactile sensations of products and to control the robots.

In this thesis, we assume only video live streaming from the "Pepper" robot [10] to the

user. In the service, the video which is taken by a camera on the robot is compressed into MPEG2 format using FFmpeg [11], which runs on the operating system of the robot. The video is then transferred to an edge server and text information is added using FFmpeg running on a virtual machine. Another virtual machine relays the video stream to a virtual machine hosting FFserver, a streaming server application, to stream it to the user PC for real-time user viewing using FFplay. To simplify the implementation, we do not insert the superimposition of product information, but insert a simple text. The text is inserted using drawtext filter of FFmpeg. Note that FFserver uses UDP and TCP transport protocols for reception and transmission, because of its specification.

Pepper is a humanoid robot capable of recognizing users' emotions. Its software development kit (SDK) and API are publicly available, so internal and external applications using its equipped cameras, sensors, and motion modules can be developed easily.

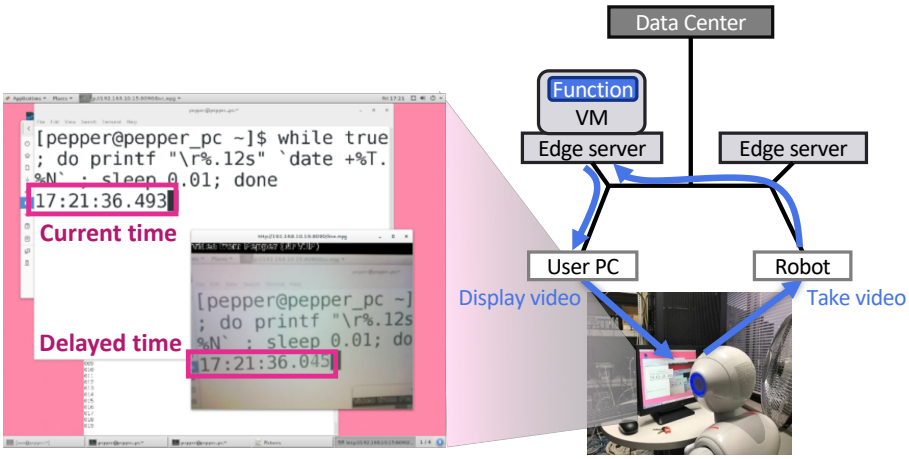


Figure 2: Method for Measuring Application-level Delay

3 Measurement and Analysis of Application-level Delay

In this section, we focus on application-level delay between end devices for video live streaming service with real-time processing. Then, we measure application-level delays and analyze to clarify the factors of occurring the delays.

3.1 Measurement Setting

3.1.1 Measurement Method

To measure the application-level delay of video live streaming, we use the time difference of the two clocks in the live-streamed video. First, a millisecond-precision digital clock is displayed in front of the robot. Then, the video captured by the robot is live-streamed to the user PC via real-time processing on an edge server. The digital clock shown to the robot is displayed on the monitor of the user PC for time synchronization. The digital clock and the live-streamed video from the robot are arranged on the monitor. Next, a screenshot is taken per second for 100 seconds and the time differences of the two clocks displayed on the monitor are calculated by each screenshot. At this time, the two displayed times are converted mechanically from an image to a numerical value. Specifically, a OCR engine called Tesseract digitizes trimmed screenshots. The screenshots are trimmed at the portions that the times are displayed, and binarized in black and white. If it cannot be

digitized mechanically due to image distortion but we can recognize it as a numerical value, digitize it manually. Finally, we calculate the mean of the time differences to measure the application-level delay of video live streaming. Figure 2 shows the method for measuring the application-level delay.

3.1.2 Deployment Scenarios for Real-time Processing

Seven scenarios are set to investigate the occurrence factors and the volume of application-level delays due to the differences in the locations where service functions are deployed. For each scenario, we changed the forms of service provision, such as the existence of a virtualization environment, distance of TCP communication, and the location of service functions. In two edge scenarios, with names beginning with "edge," the service functions are deployed on an edge server. In three DC scenarios, they are deployed on a virtual machine on AWS. The "Non-Virtualized" and "Direct" scenarios are used for comparison. Note that TCP path length represent the number of links on the TCP path. Our preliminary experiments, which are not shown in this thesis, to measure the communication delay of UDP and TCP showed that the delay of TCP is about twice the delay of UDP even in our LAN network. Therefore, we focus on difference of the TCP path lengths.

- **Edge-User-Side:** In this scenario, applications that perform text insertion, relay, and streaming are deployed on the user-side edge server and executed on virtual machines. Placing service functions on the server reduces TCP communication distances. The TCP path length is 2. Figure 3a shows the location of the service functions and the communication paths in this scenario.

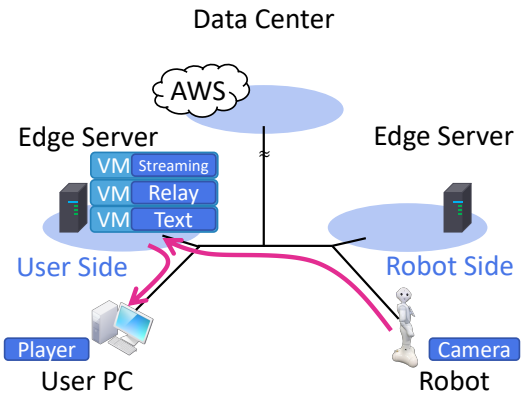
- **Edge-Robot-Side:** In this scenario, applications that perform text insertion, relay, and streaming are deployed on the robot-side edge server and executed on virtual machines. Placing the service functions on the server increases TCP communication distances. The TCP path length is 4. Figure 3b shows the location of the service functions and the communication paths in this scenario.

- **DC-{Ohio, Singapore, Tokyo}:** In the three DC scenarios, applications that perform text insertion, relay, and streaming are deployed on AWS as a data center

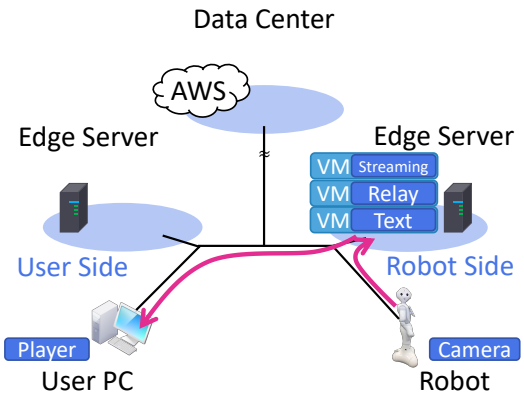
and executed on a virtual machine. For scenarios DC-Ohio, DC-Singapore and DC-Tokyo, we use AWS regions of Ohio, Singapore and Tokyo respectively. Of the three DC scenarios, the TCP communication distance is the longest in DC-Ohio and the shortest in DC-Tokyo. Even in DC-Tokyo, it is considerably longer than the other scenarios: Edge-User-Side, Edge-Robot-Side, Non-Virtualized, because the packets go through the Internet. Ohio, Singapore and Tokyo are about 10800 km, 4900 km and 400 km away from Osaka respectively. Figure 3c shows the location of the service functions and the communication paths in the three DC scenarios.

- **Non-Virtualized:** In this scenario, applications that perform text insertion, relay, and streaming are deployed on the NV server and executed directly on that server. The TCP communication distance is larger than in the scenario Edge-User-Side and smaller than in the scenario Edge-Robot-Side. The TCP path length is 3. Figure 3e shows the location of the service functions and the communication paths in this scenario.

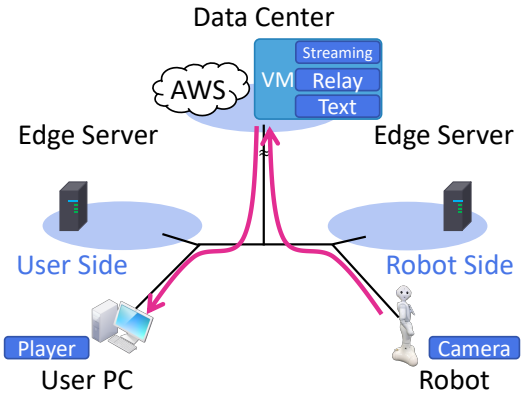
- **Direct:** In this scenario, no applications that perform text insertion, relay, or streaming are deployed. The video is directly sent from FFmpeg on the robot to FFplay on the user PC. We aim to measure the processing time at the end devices. All communications use UDP. Figure 3e shows the communication paths in this scenario.



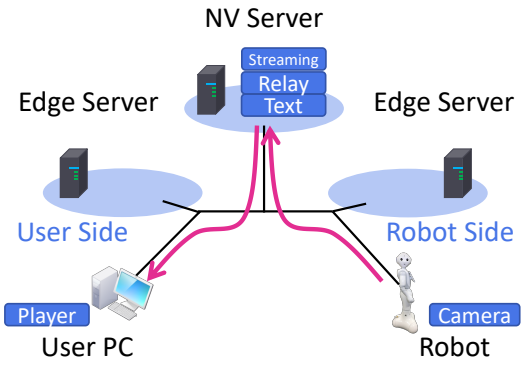
(a) Edge-User-Side



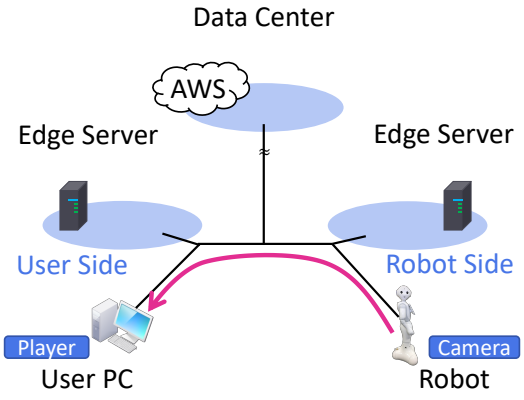
(b) Edge-Robot-Side



(c) DC-Ohio, DC-Singapore and DC-Tokyo



(d) Non-Virtualized



(e) Direct

Figure 3: Location of Service Functions and Communication Paths in Each Scenario

Table 2: Application-level Delay of Video Live Streaming in Each Scenario

Scenario	Application-level Delay [ms]		
	1st	2nd	Mean
Edge-User-Side	476.19	482.76	479.48
Edge-Robot-Side	482.76	500.99	491.88
DC-Ohio	745.18	761.13	753.15
DC-Singapore	617.68	641.64	629.66
DC-Tokyo	526.56	515.21	520.89
Non-Virtualized	467.35	477.93	472.64
Direct	419.54	430.84	425.19

3.2 Measurement Results and Analysis

In the seven scenarios, we measure the application level delays of video live streaming twice in the way described in Section 3.1.1. The results of the two measurements are shown in Table2. The mean of application-level delays are 479.48 ms and 491.88 ms for Edge-User-Side and Edge-Robot-Side respectively. In the three DC scenarios, they are 753.15 ms, 629.66 ms, and 520.89 ms for DC-Ohio, DC-Singapore and DC-Tokyo respectively[1]. In the comparisons, they are 472.64 ms and 425.19 ms for Non-Virtualized and Direct.

The results of the analysis is explained in the following subsections.

3.2.1 Delay due to TCP Path Length

The result of the scenario Edge-Robot-Side is 12.4 ms lager than that of the scenario Edge-User-Side and the TCP path length is 2 units longer. Therefore, the application-level delay increases by 6.20 ms for each TCP path length unit even in the LAN scale.

Table 3: RTTs of ICMP Packets from Osaka University to AWS Regions of Ohio, Singapore and Tokyo

Region	Distance from Osaka [km]	With VPN [ms]	Without VPN [ms]
Ohio	10800	174.94	174.64
Singapore	4900	68.32	68.06
Tokyo	400	12.84	13.19

3.2.2 Delay due to Long Communication Distance

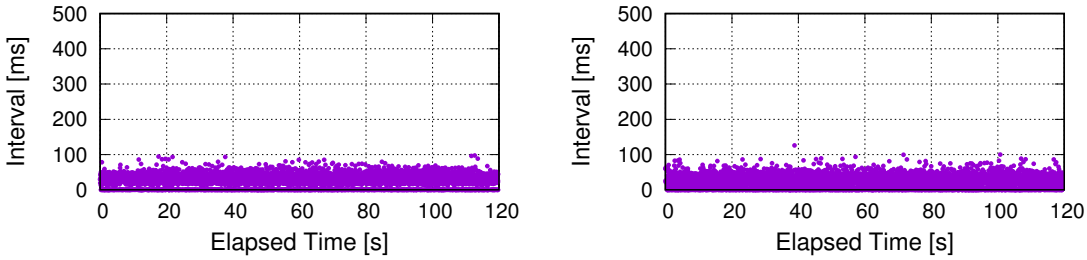
When using a data center, it can be anticipated that the application-level delay increase by as much as the increase of propagation delay, such as the RTT of packets, from the user PC to the data center. In practice, however, the increase of application-level delay cannot be explained by a simple increase of propagation delay in a large scale. For the scenarios DC-Ohio, DC-Singapore and DC-Tokyo, the differences from the application-level delay in Edge-User-Side are 273.67 ms, 150.18 ms and 41.41 ms respectively. On the other hand, the RTTs of ICMP packets from the user PC to the virtual machines on AWS are 174.94 ms, 68.32 ms and 12.83 ms for Ohio, Singapore and Tokyo regions, respectively, as shown in Table 3. The RTT from the user PC to the edge server is about 0.5 ms, and the increase of application-level delay are obviously larger than the increases of RTTs.

The cause of those unexpected increases of application-level delay is jitter, i.e. a variation in packet arriving intervals. Since it is difficult to make the intervals perfectly constant, the receiver buffers the packets and absorbs the variation. As the variation gets bigger, the receiver needs to buffer them for a longer time.

We newly measure packet arriving intervals on the user PC and application-level delays[2]. Figure 4 shows the packet arriving intervals on the user PC in Scenario Edge-User-

[1]The dates and times (UTC) of the measurements are 8:20 a.m. on May 31, 2018 and 5:10 a.m. on June 1, 2018 for Ohio region, 10:00 a.m. on May 31, 2018 for both measurements on Singapore region, and 4:00 a.m. on June 15, 2018 for both measurements on Tokyo region.

[2]The dates and times (UTC) of the measurements are 8:30 a.m. on October 2, 2018 for Ohio region, 6:30 a.m. on October 2, 2018 for Singapore region and Edge-User-Side, and 6:45 a.m. on October 3, 2018 for Tokyo region.



(a) Edge-User-Side

(b) DC-Ohio

(c) DC-Singapore

(d) DC-Tokyo

Figure 4: Packet Arriving Intervals in Four Scenario: Edge-User-Side, DC-Ohio, DC-Singapore and DC-Tokyo

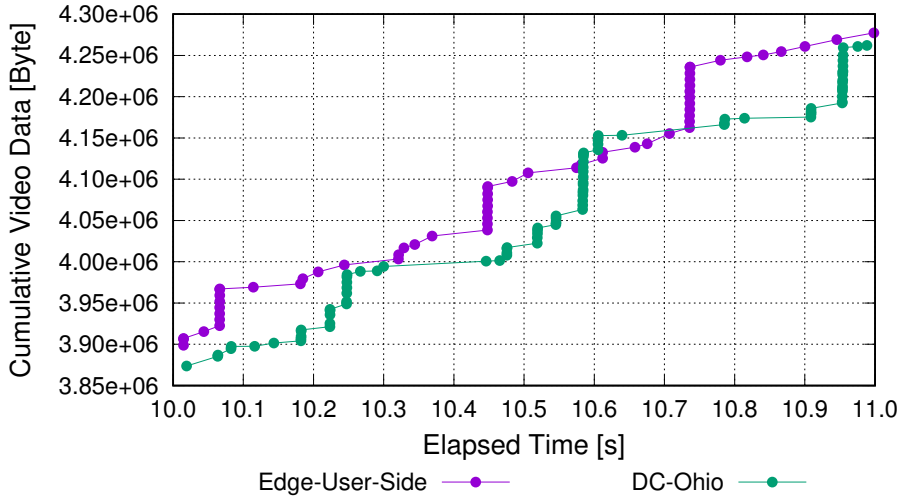


Figure 5: Cumulative Video Data Received in Scenario Edge-User-Side and Scenario DC-Ohio

Side, DC-Ohio, DC-Singapore and DC-Tokyo. Figure 5 shows the cumulative video data received on the user PC in Scenario Edge-User-Side and Scenario DC-Ohio. Points in Figure 5 indicate reception of packets at the data link layer. Receiving a lot of data at the same time shows reception of an MPEG I frame (Intra-coded frame) that is a complete frame having all information of the frames. A packet with a small amount of data periodically reached indicates reception of a P frame (Predicted frame) that is a frame having only difference information with respect to an I frame. As can be seen from Figure 5, when using a data center in Ohio, the arrival of I frames and the arrival interval of P frames are scattered.

Moreover, it is revealed that application-level delay increases almost in proportion to the coefficient of variation (C.V.)[3] of packet arriving intervals in our configuration, as shown in Table 4 and Figure 6. In Table 4 and Figure 6, the application-level delays are different from the results stated in Table 2. This is because the date and time of the two measurements are different and the state of the Internet was different.

To summarize the above, the factor of delay increase in the case of long communication distance is divided into propagation delay and buffering time for absorbing jitter. Noted that the time taken for VPN processing is about 0.3 ms, measured by ICMP Ping (Table 3), and there is almost no influence on the application-level delays.

3.2.3 Time for Real-time Processing

Considering the TCP path length and comparing the the result in the scenario Non-virtualized with that of the scenario Direct, the difference in the application-level delays is 28.85 ms. Therefore, the result show that the time required for text insertion and streaming server processing is 28.85 ms.

In addition, a buffering time of 194 ms occurs. It is explained in Section 3.2.5.

3.2.4 Increase of Delay due to Virtualization

Comparing the mean of the results in the two edge scenarios with the result of the scenario Non-Virtualized, the difference is 13.04 ms. Therefore, the results show that software

[3]The coefficient of variation (C.V.) is defined as the ratio of the standard deviation (S.D.) to the mean.

Table 4: Details of Packet Arriving Interval and the Unexpected Increase of Application-level Delay

	Edge-User-Side	DC-Ohio	DC-Singapore	DC-Tokyo
Application-level Delay [ms]	480.42	668.25	611.31	637.23
RTT of ICMP Packets [ms]	0.47	170.12	72.93	12.58
Unexpected Increase [ms]	**0**	**18.19**	**58.44**	**144.71**
Mean of Intervals [ms]	26.05	22.27	17.20	23.07
S.D. of Intervals [ms]	17.88	18.01	22.35	40.45
C.V. of Intervals	**0.69**	**0.81**	**1.30**	**1.75**

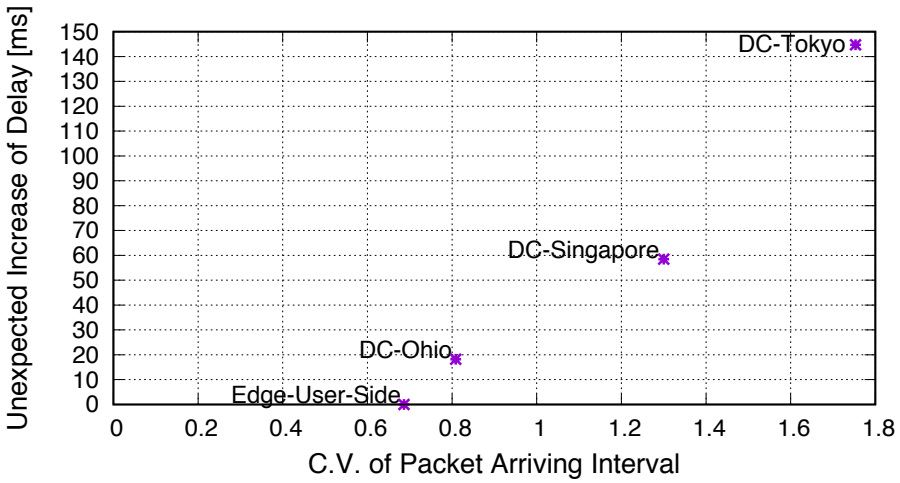


Figure 6: Relation between Coefficient of Variation of Packet Arriving Interval and the Unexpected Increase of Application-level Delay

operation in the virtualized environment increases application-level delay by 13.04 ms. The TCP path length need not be considered, since the mean of the TCP path length in the two edge scenarios is 3. It is equal to that in the scenario Non-Virtualized.

We also measure the increase of server processing time due to software operation in the virtualized environment. Since it is difficult to directly measure server processing times, we use the difference in captured times between incoming and outgoing time packets of a server. Therefore, the video live streaming is started and stopped repeatedly while packets are captured at the server's network interface. Using those captured packets, we can calculate the differences in captured times between first incoming and first outgoing packets. We also calculate the differences in captured times between last incoming and last outgoing packets. Note that we only know the increase of server processing time, not the processing time itself, since it cannot be guaranteed that the data carried by the incoming packets and the data carried by the outgoing packet are the same. In other words, if we regard the differences in captured times as server processing time, the time that data is buffered in the application will not be taken into account. When the live streaming stops, the data transmitted by the last incoming packet is buffered and not send out. It will be sent when streaming is resumed. Therefore, we can only know the increase in server processing time due to virtualization by measuring and comparing the differences in captured times between incoming and outgoing time packets of a server for scenario Edge-User-Side, Edge-Robot-Side and Non-Virtualized. As a result, the increase in server processing time due to virtualization is 4.00 ms.

3.2.5 Processing Time at End Devices

Although the scenario Direct is set to measure the processing time at the end devices, we also modified FFplay application on the user PC to output timestamps in order to investigate the details of the processing time. Analyzing the outputted timestamps, the delay occurred in FFplay when receiving TCP packets is about 80 ms and that when receiving UDP packets is about 274 ms. The difference of 194 ms is the time for buffering. From its source code, it is revealed that, when receiving packets using UDP, FFplay buffers them longer. We also confirmed that about 170 ms out of the delay of 194 ms for buffering can be reduced by modifying the source code. Note that the measurement of the processing

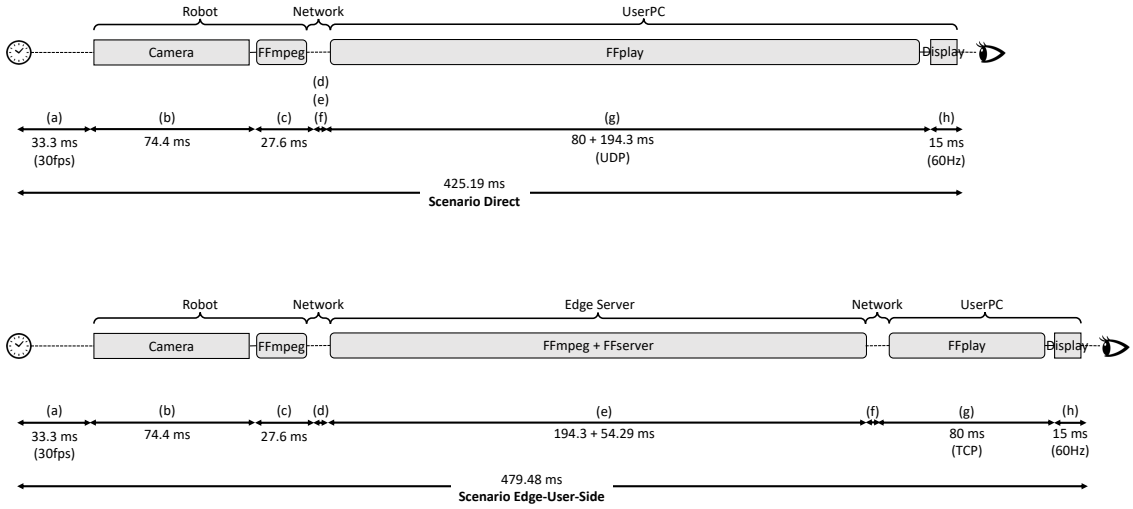


Figure 7: Details of Processing Delay in End-to-End Video Streaming

time by the timestamps can not completely clarify the processing time at the user PC. This is because it does not include the time until presentation timestamps (PTS), that is used for identifying data in FFplay, are extracted from arrived packets. In particular, it is difficult to measure the buffering time below the application layer.

Since, only in the scenario Direct, FFplay receives UDP packets, the processing time in FFplay is 80 ms. However, in the other scenarios, the delays of 194 ms for buffering occur in FFmpeg running on a virtual machine on an edge server or AWS. This is because FFmpeg running on them receives packets using UDP, and FFmpeg shares most of its source code with FFplay. Therefore, in the scenarios except Direct, the delay of 194 ms for buffering occurs in the virtual machine on edge servers or AWS, and the processing time of 80 ms occurs in FFplay on the user PC. Also, since the refresh rate of the display of the user PC is 60 Hz, a delay of up to 15 ms is likely to be occurred [12].

The camera and FFmpeg on the robot will generate delay of about 136 ms, that is the difference between the total processing time currently confirmed and the result of the scenario direct. FFmpeg modified to output timestamps shows that its processing time is 27.6 ms. The remaining 108 ms should be occurred in the camera of the robot, and then a delay of up to 33 ms is likely to be occurred just before capturing since the frame rate is 30 fps [12]. However, they cannot be confirmed because of the specification of the robot.

The details of processing delay in end-to-end video streaming is shown in Figure 7.

The processing times in all the scenarios except Direct are the same as in Edge-User-Side. The delay occurs in "Network" or "Edge Server" varies depending on the distance to the data center or whether there is virtualization environment.

3.3 Discussion for Low-latency Video Live Streaming

Based on the result in Section 3.2, we can obtain policies for service function reallocation aimed at reducing application-level delay of video live streaming service with real-time processing. The following three solutions are considered.

First, application-level delay can be reduced by deploying a service function of real-time processing on edge servers, that is a key idea of Multi-access Edge Computing. Processing at a data center can lead to penalties in the form of a large propagation delay due to geographical factors. In a new service with real-time processing, that delay significantly damages the user's QoE and is required to be reduced. In our experiment, a RTT of 12 to 170 ms occurred between the user PC and the data centers. Using a function on the edge can almost eliminate that delay.

Second, it is necessary to deal with jitter at network side rather than at end devices. Conventionally, applications have buffered the packets and have absorbed jitter in order to ensured quality of service (QoS) and user's QoE. Contrary to that conventional circumstance, in new services which demand high real-time performance, the time for buffering will lead to degradation of QoS and user's QoE. Therefore, it is required not to generate jitter at the network. Alternatively, it is needed to select a route with small jitter and reallocate the service function accordingly. In Section 3.2.2, the maximum increase of the application-level delays due to the buffering time for absorbing jitter was 144 ms. By reducing jitter, this delay is eliminated. In our experiment, the impact of jitter was clarified by the result of providing service at data center. If the cause of jitter is congestion of the route to the date center, the effect can be reduced by MEC. However, even using an edge server, there is a possibility of occurring jitter due to high load of the edge server. In Section 4, we devise a service function reallocation method based on the CPU load of edge server.

The third solution is to improve the performance of end devices. Cameras capable of taking high frame rate video and high refresh rate monitors can push the limits of

application-level delay reduction. For example, if the frame rate of video is changed from 30 fps to 60 fps, a delay of up to 16.6 ms (= 1000 × (1 sec / 30 fps – 1 sec / 60 fps)) is reduced only by taking a video. Also, if the refresh rate of monitor is changed from 60 Hz to 240 Hz, a delay of up to 12.5 ms (= 1000 × (1 sec / 60 Hz – 1 sec / 240 Hz)) is reduced only by playing a video. In addition, improving the processing performance of end devices leads to reduction of compression time and decompression time of MPEG videos. However, even if the performance of end devices improves, the delay reduction is on the order of a few tens of milliseconds, which is smaller than the reduction of propagation delay or buffering time for absorbing jitter.

4 Service Function Reallocation Method

In this section, we devise a service function reallocation method that reduces application-level delay. In our method, we focus on the increases of jitter and application-level delay caused by the increase of CPU load of the edge server, and the timing of service function reallocation is determined based on the CPU load of edge servers. Also, we confirm the effect of the service function reallocation method on application-level delay.

4.1 Live Migration of Virtual Machine

Reallocation of a service function is realized by live migration of virtual machine. It is a technique to migrate a virtual machine from one host to another host while the virtual machine is running. By copying a virtual machine with its memory instead of switching only the communication to another virtual machine, the state of application can be taken over to the migration destination. Live migration is roughly divided into two types [13]: pre-copy migration of copying memory before functions of the virtual machine are migrated and post-copy migration of copying memory after functions are migrated. In our experiment, we use pre-copy migration as it is widely used in the main-stream virtual machine monitors (VMM)/hypervisors, because of its robustness.

In pre-copy migration, new virtual machine is started at the destination host, and the memory of the original virtual machine is copied to the new virtual machine. Memory pages that changed at the original virtual machine during copying are repetitively copied to the new virtual machine. When the number of dirtied pages that are memory pages that have not been copied yet becomes small enough to minimize their transmission time, the hypervisor stops the original virtual machine and instantly copies them. External communication pauses for a moment while reconnecting. The duration that the migrated virtual machine is out of service is called downtime.

The total migration time is basically determined by the size of the memory of the virtual machine and the bandwidth of the network used for the migration. For instance, when migrating a virtual machine with 4 GB of memory using a 10 Gbps network, it takes about 3.2 seconds (= 4 GB $\times$ 8 bits / 10 Gbps). However, in practice, it takes longer time due to overhead and repetitive copying of changed memory. The downtime

is determined by the network bandwidth and the threshold at which repetitive copying stops. The threshold is the number of dirty pages, and it is often calculated based on a target downtime.

In our MEC environment, the OpenStack compute nodes are used as the edge servers, as explained in Section 2.1. The controller node of OpenStack (the orchestrator) instructs KVM on the compute node to live-migrate a virtual machine. The size of memory is 4 GB as shown in Table 1 and the network bandwidth is 10 Gbps. The maximum permitted downtime is set to 500 ms in our configuration. The document of OpenStack states that the downtime is initialized to a small value, typically around 50 ms, depends on the size of the virtual machine and that it is gradually increased when it notices that the memory copy does not make sufficient progress.

4.2 Service Function Reallocation Method

As described in Section 3.2.2, it is revealed that the increase of buffering time for absorbing jitter increases application-level delay as well as the increase of propagation delay in video live streaming service. Therefore, in order to realize low-latency video live streaming, it is naturally important that the average of propagation delays is small, but it is also necessary to suppress or avoid the occurrence of jitter on the network side. Based on this finding, we devise a service function reallocation method that reduces application-level delay. If the cause of jitter is congestion of the route to the date center, the effect can be reduced by MEC. However, even using an edge server, there is a possibility of occurring jitter due to high load on the edge server. Therefore, we focus on the increase of jitter caused by the increase of CPU load of the edge server.

In our method, the orchestrator monitors the CPU utilization of the edge server as an index of CPU load and determine when start to live-migrate the virtual machine in order to avoid increase of application-level delay of video live streaming. The reason why the orchestrator does not directly measure jitter is that it is difficult for the orchestrator to directly access the network statistics of the user's device in an actual MEC environment. In our method, selection of virtual machine to be migrated, selection of destination, prediction of CPU load are out of scope. Our method only determines the timing that the orchestrator issues an instruction to start live migration.

There are two factors to determine the beginning of live migration: CPU load at which application-level delay start to increase and the total migration time at that CPU load. It is appropriate that live migration is finished just before the CPU load of the edge server reaches the point at which application-level delay begins to increase. For that purpose, it need to be started before the total migration time in advance, since the virtual machine is operated at the source host during copying in pre-copy migration.

Firstly, we investigate the value of CPU utilization at which jitter and application-level delay start to increase. The CPU load of the edge server is given by using a stress tool called "stress-ng", and the packet arriving intervals and the application-level delays are measured when the CPU load is 0, 50, 80, 95 and 100 percent. Figure 8 shows the packet arriving intervals at FFplay on the user PC in each CPU load. Figure 9 shows the cumulative video data received on the user PC at CPU load 0% and 100%. The charts in Figure 8 and Figure 9 are plotted in the same way as in Figure 4 and Figure 5, but the configuration of the applications has been changed for a reason to be described in Section 4.3.1, and the video is received using UDP. Therefore, the distribution of packet arriving intervals has been changes slightly. Table 5 and Figure 10 show the relation between the CPU load of edge server and packet arriving intervals. The packet arrival intervals increase because there are times when the processing delay on the edge server increases. The results show that jitter increases from when the CPU load exceeds 80%.

Figure 11 shows the application-level delay of video live streaming on the user PC in each CPU load. Table 6 and Figure 12 show the relation between the CPU load of edge server and the application-level delay of video live streaming. In the box plot of Figure 12, the bottom line of the whiskers, the bottom line of the boxes, the upper line of the boxes, and the upper line of the whiskers represent the value of minimum, 25th percentile, 75th percentile, and maximum, respectively. The results show that the application-level delays start to increase from when the CPU load exceeds 95%. In Figure 12, the notable values are mean and median. Both values increase from 95%. Two reasons are considered why the increase of application-level delay are slower than that of jitter. First, FFplay buffer of the minimum size can afford to absorb jitter occurring when the CPU load is 80% at the maximum. Second, the buffering time is increasing, but the processing time may be decreasing. The CPU clock of the edge server is variable depending on its load.

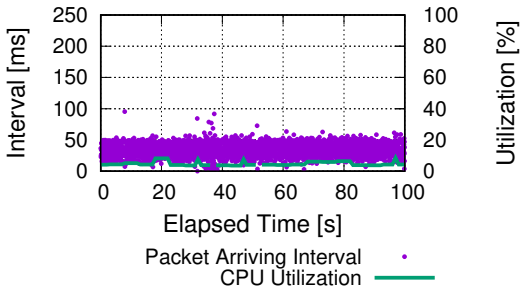
Note that, the increase of the application-level delay may include not only the increase of buffering time but also the increase of processing time of the edge server. To investigate the breakdown of the increase is a future work. However, in any case, application-level delay increases due to the increase of CPU load.

We also executed three different calculations of instructions per cycle (IPC) to give the CPU a load, since there is an opinion [14] that CPU utilization does not correctly indicate the load of CPU and IPC is a better index. Stress-ng allow us to choose a CPU-stress method that is a calculation to give the CPU a load. We chose the calculations of Jenkin's hash function, inverse discrete cosine transform (IDCT) and π. When using them to give 100% load to CPU, IPC is 1.76, 1.48 and 0.69 respectively. Note that, in the above measurement, we did not specify a CPU-stress method. In that case, all CPU-stress methods are exercised sequentially and IPC is 1.16. Figure 13 and Figure 14 show the results when CPU load is given using those three CPU-stress methods. In those figures, the result when the CPU load is not given, i.e. the background CPU load is 0%, is the same data as in Figure 10 and Figure 12. The results show that the effect of CPU load does not change even with different CPU-stress methods of IPC. In all the subsequent experiments of this thesis, the CPU-stress method is not specified.

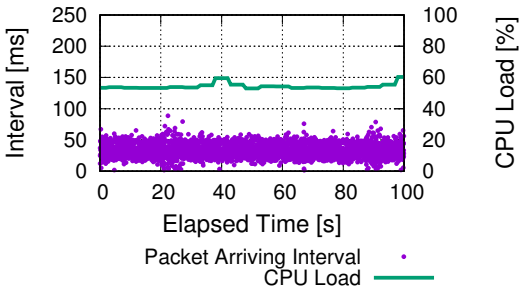
Next, we investigate the total time of live migration in different CPU load of source host. Since the process of live migration itself uses CPU, there is a possibility that total migration time increases or migration fails when the CPU load is high. To state the result, the total migration time is 1.5 times longer than the shortest case, as shown in Figure 15. We measured it by repeating live migration 10 times in each different CPU load of the source host while the CPU load of the destination host fixed at 0%. As auxiliary experiments, we also measure the total migration times when the CPU load of the destination host is fixed at 50%. As shown in Figure 16, the total migration times are almost the same as when it is fixed at 0%. Table 7 summarizes the total migration times in all the cases. Note that, live migration fails if it has been running too long time. The timeout is calculated by the memory size. In our configuration, a virtual machine with 4 GB of memory has a time of 4×800 seconds.

From the results, it is appropriate to start live migration at 8 seconds before the CPU load of source host reaches 95%. This is because application-level delay start to increase

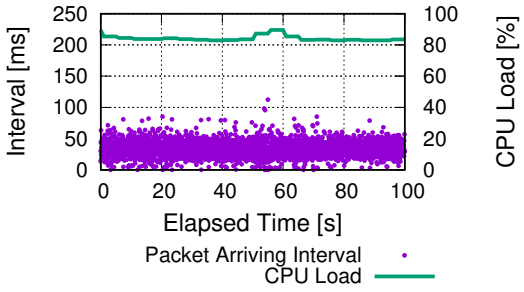
from when the CPU load exceeds 95% due to occurrence of jitter, and the total migration time is about 8 seconds at that CPU load.



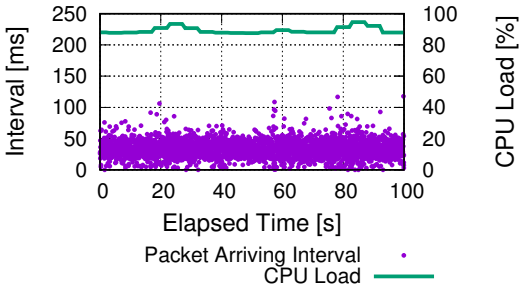
(a) CPU Load 0%



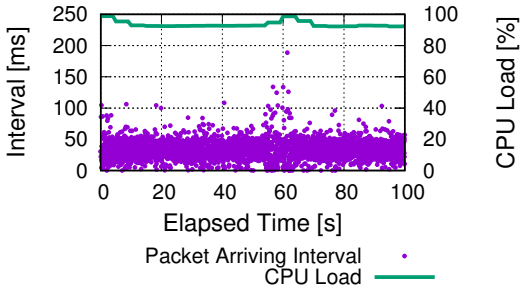
(b) CPU Load 50%



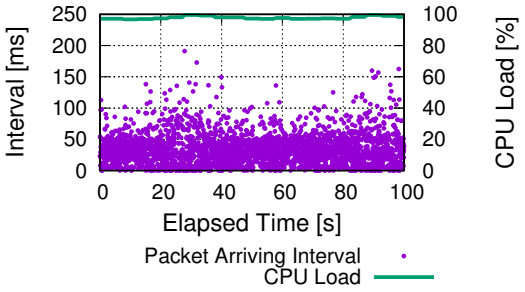
(c) CPU Load 80%



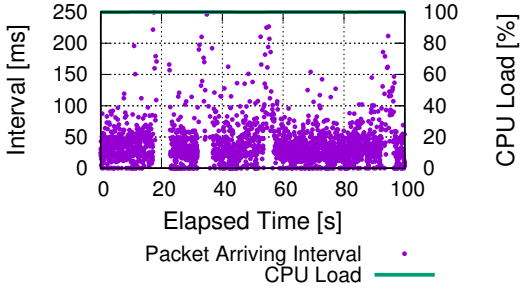
(d) CPU Load 85%



(e) CPU Load 90%



(f) CPU Load 95%



(g) CPU Load 100%

Figure 8: Packet Arriving Intervals in Each CPU Load

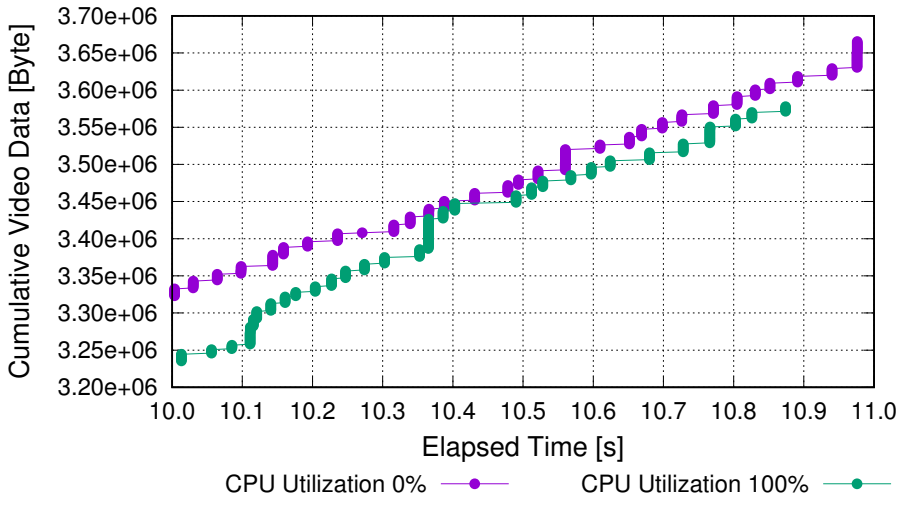


Figure 9: Cumulative Video Data Received at CPU Load 0% and 100%

Table 5: Details of Packet Arriving Intervals in Different CPU Load

	CPU Load						
	0%	50%	80%	85%	90%	95%	100%
Mean [ms]	33.33	33.33	33.36	33.33	33.33	33.33	33.27
S.D. [ms]	10.31	10.74	12.56	13.64	15.48	24.33	53.76
C.V.	0.31	0.32	0.38	0.41	0.46	0.73	1.62

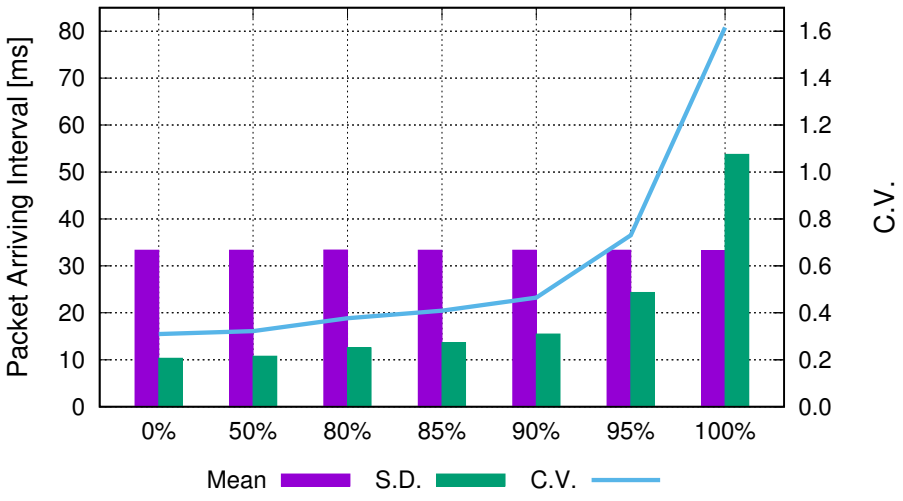
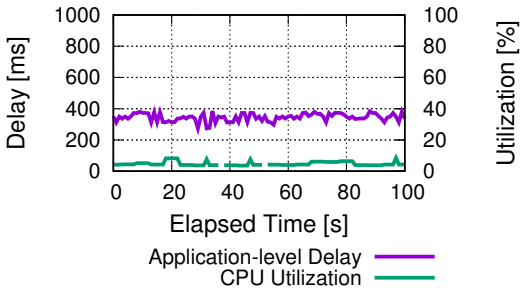
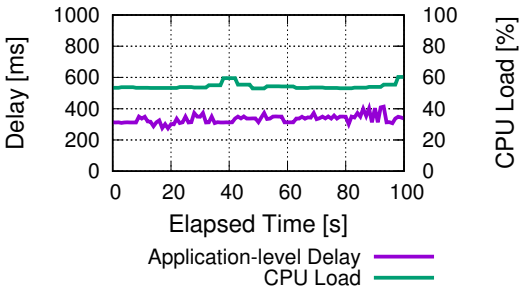


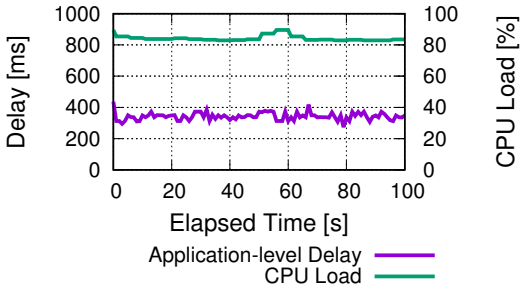
Figure 10: Relation between CPU Load and Packet Arriving Intervals



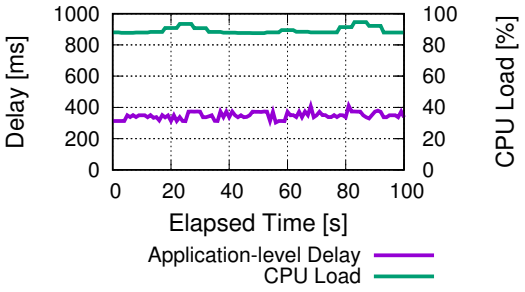
(a) CPU Load 0%



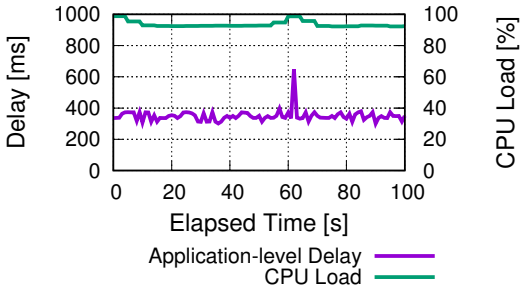
(b) CPU Load 50%



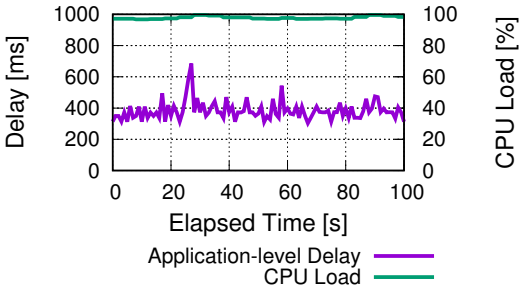
(c) CPU Load 80%



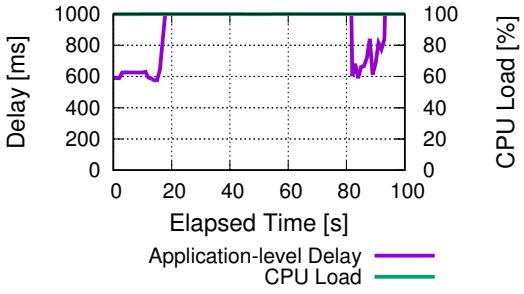
(d) CPU Load 85%



(e) CPU Load 90%



(f) CPU Load 95%



(g) CPU Load 100%

Figure 11: Application-level Delay in Each CPU Load

Table 6: Details of Application-level Delay in Different CPU Load

	CPU Load						
	0%	50%	80%	85%	90%	95%	100%
Mean [ms]	344.55	333.55	342.26	347.35	348.74	382.37	3383.25
Max [ms]	385	413	436	410	649	685	8067
75th Percentile [ms]	371	349	350	372	366	409	5026
Median [ms]	349	337	338	349	348	373	3534
25th Percentile [ms]	335	313	334	337	336	349	773
Min [ms]	273	274	273	303	300	303	575

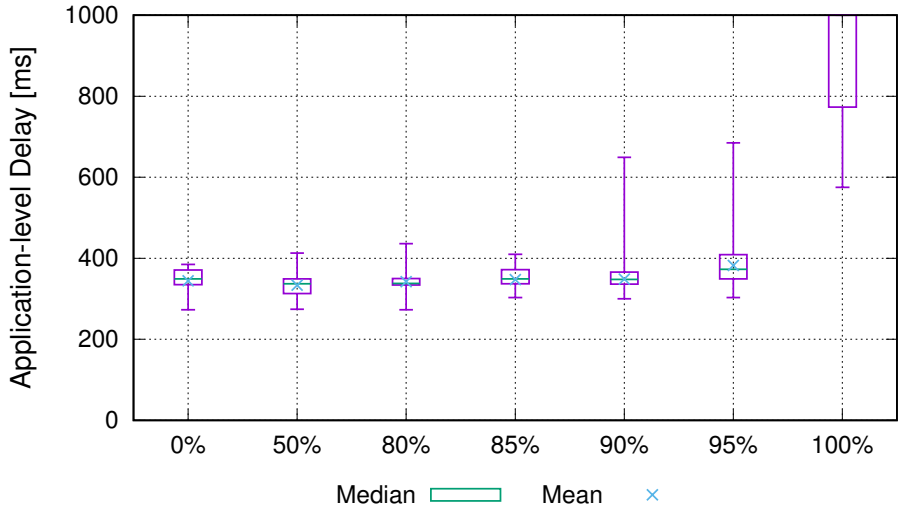
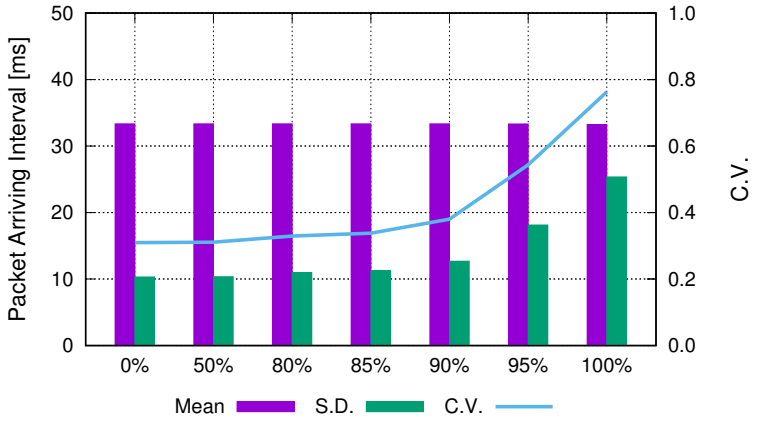
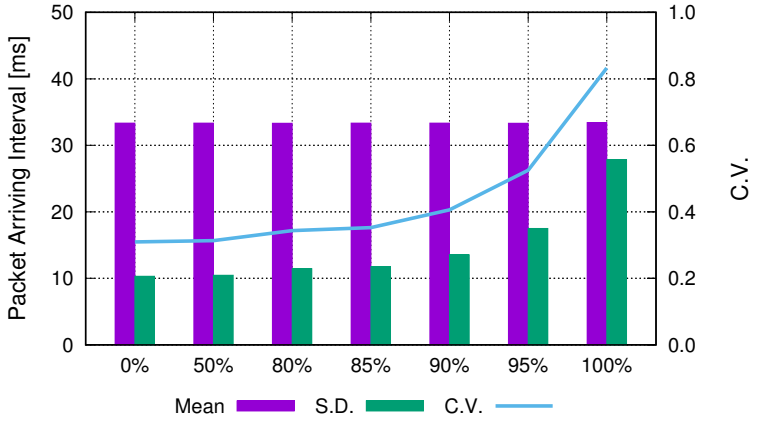


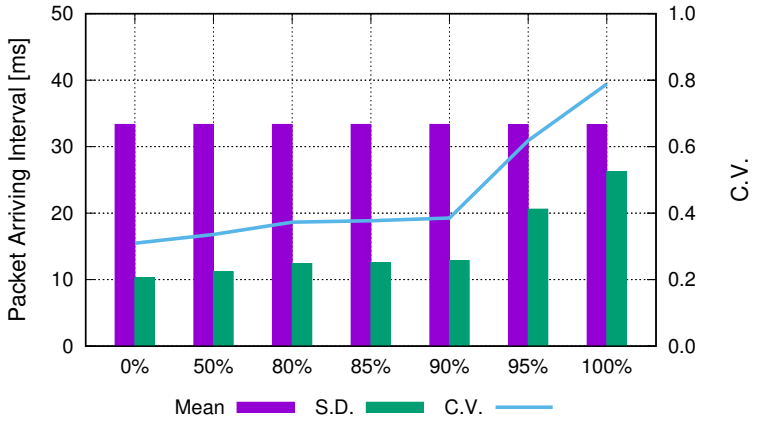
Figure 12: Relation between CPU Load and Application-level Delay



(a) Jenkin's Hash Function (IPC: 1.76)

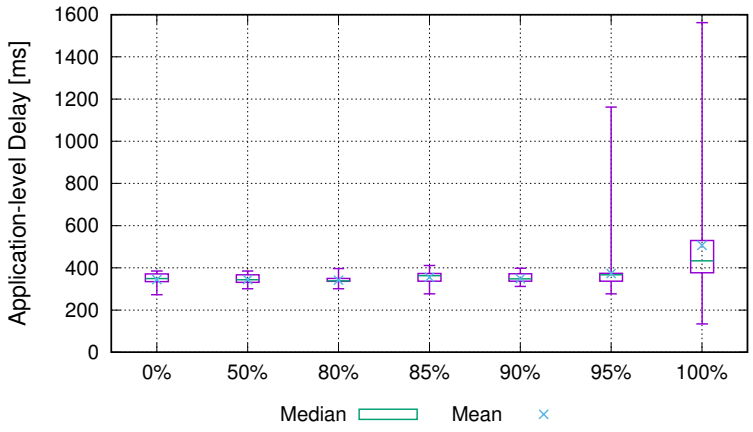


(b) IDCT (IPC: 1.48)

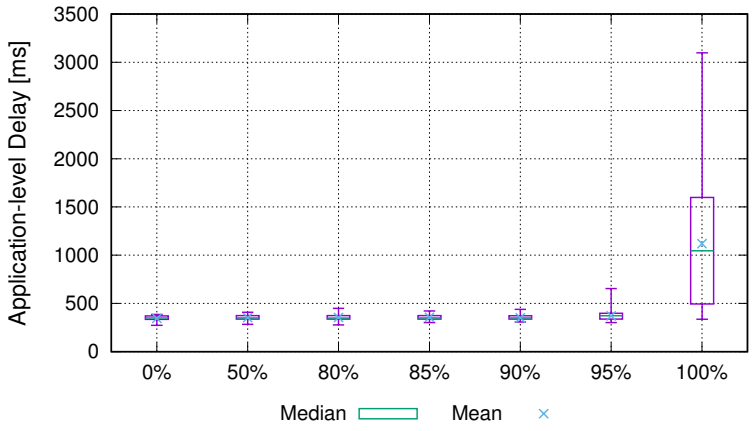


(c) π (IPC: 0.69)

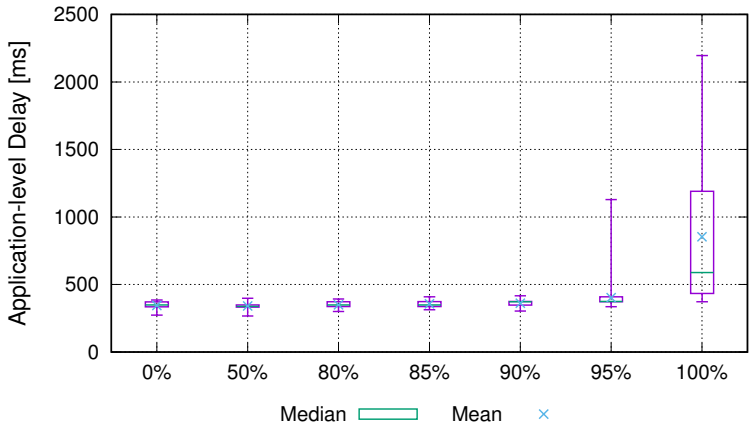
Figure 13: Packet Arriving Intervals with Different IPC Load



(a) Jenkin's Hash Function (IPC: 1.76)



(b) IDCT (IPC: 1.48)



(c) π (IPC: 0.69)

Figure 14: Application-level Delay with Different IPC Load

Table 7: Details of Total Migration Time in Different CPU Load of Source Host

CPU Load (Destination)	CPU Load (Source) 0%	50%	80%	85%	90%	95%	100%
0%	6.35	6.06	6.17	6.32	6.89	8.13	10.8
50%	6.07	6.13	6.23	6.28	6.75	7.80	9.53
100%	6.40	6.34	6.56	6.65	7.10	7.99	11.00

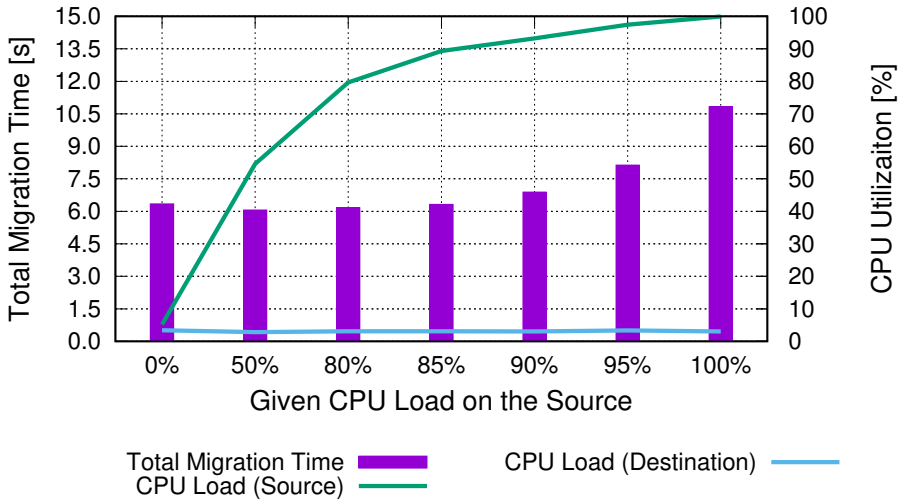


Figure 15: Relation between CPU Load of Source Host and Total Migration Time

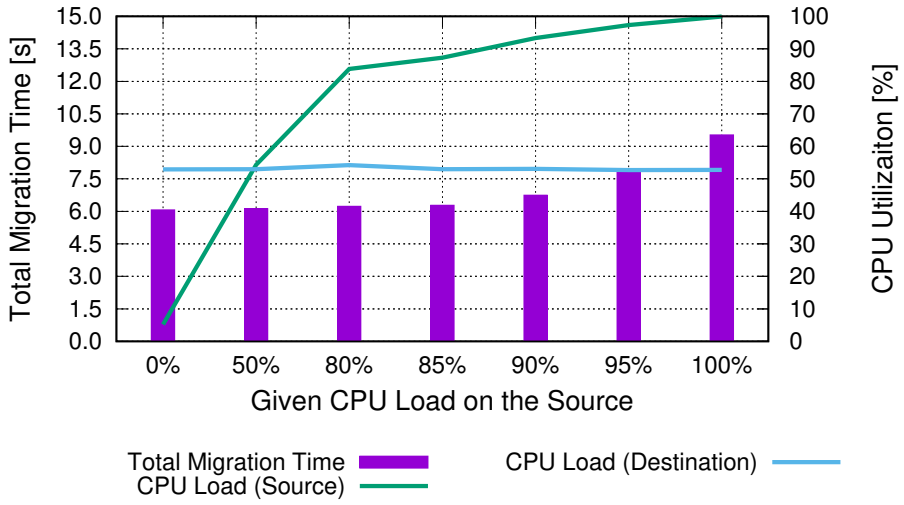


Figure 16: Relation between CPU Load of Source Host and Total Migration Time at CPU Load 50%

4.3 The Effect of Service Function Reallocation

4.3.1 Setting and Scenarios

To confirm the effect of the service function reallocation method on application-level delay, we develop a program running on the orchestrator. That is a sample program for simple service reallocation method that dynamically reallocates service functions based on the CPU load of edge servers for the purpose of low-latency video live streaming. The program monitors the CPU utilization of the edge servers using Simple Network Management Protocol (SNMP). In our environment, data acquired by SNMP is updated every 5 seconds. The program also instructs KVM on the edge server to start live migration. The virtual machine that to be migrated is executing real-time text inserting processing using FFmpeg, and it is migrated from the edge server of the user side to that of the robot side. To simplify the situation, there is only that virtual machine. There are no virtual machines for relay or FFserver. Therefore, all the communications use UDP. The source code of FFplay on the user PC has been modified to decrease 170 ms of the buffer time when receiving UDP.

As findings in Section 4.2, the appropriate timing of start to live migration is about 8 seconds before the CPU load of source host reaches 95%. However, our method does not predict when the CPU load of edge servers reaches 95%. Therefore, it starts migration when the CPU load of the source reaches 90% under a condition that the CPU load is increased as plotted in Figure 17. The CPU load of the destination is fixed at 0%, since an edge server with low CPU load should be selected as the destination when reallocating a service function in an actual MEC environment. It is also shown a case where live migration is disabled under the same CPU load situation.

4.3.2 Results

Figure 18 shows the packet arriving intervals and the application-level delay when live migration is enabled. In Figure 18, the migration is started when the CPU load reaches 90%. The packet arriving intervals are scattered before/during the migration, but they are calm after it. The application-level delay is stable overall, though it slightly increases after the CPU load reaches 80% and during the migration.

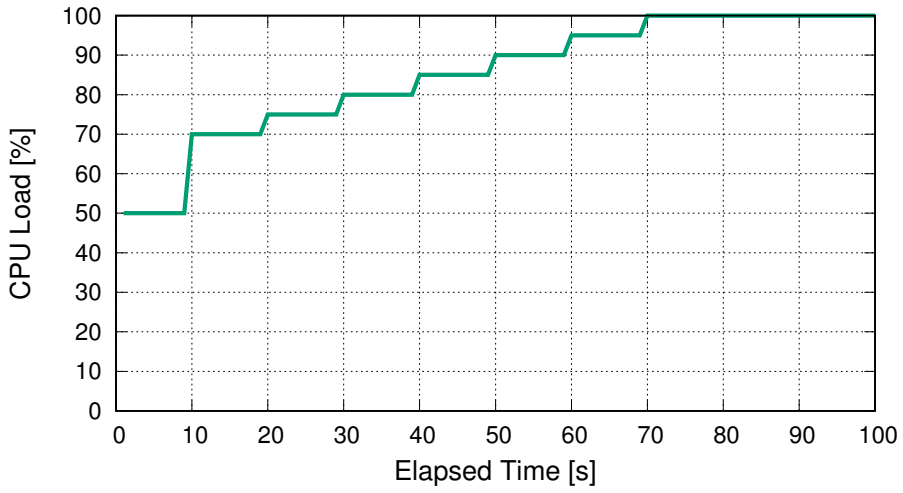
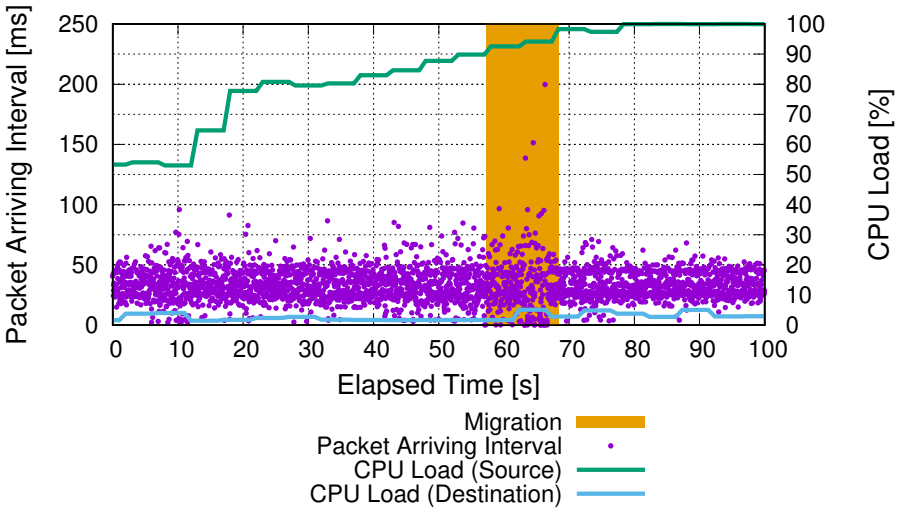


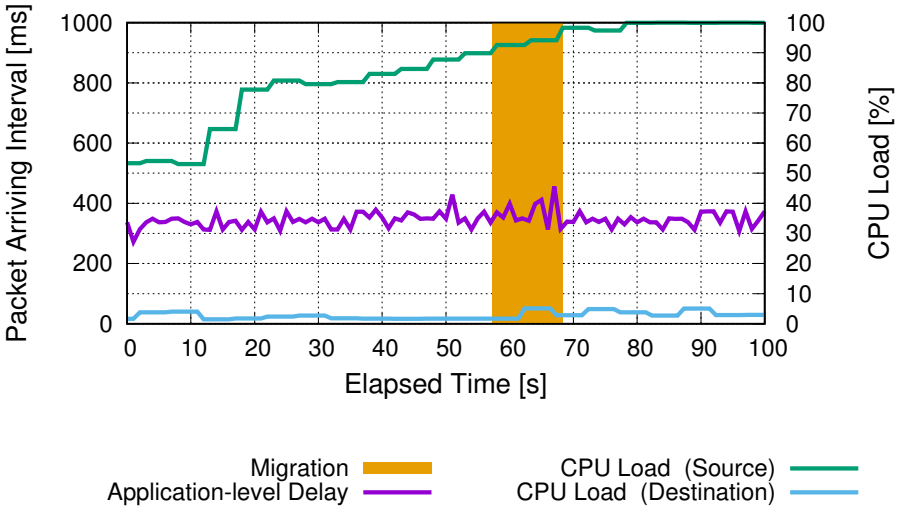
Figure 17: Increase of CPU Load of Source Host

Figure 19 shows the packet arriving intervals and the application-level delay when live migration is disabled. The packet arriving intervals are greatly scattered after the CPU load reaches 90%. The application-level delay also increases after the CPU load is over 85%. After the CPU load reaches 100%, the application-level delay increases significantly. That indicates the video is almost freezing. At around time 30, scattered packet arriving intervals and increased application-level delay are observed. It is possible that some background process instantaneously increased the CPU load.

As a result, it was possible to prevent the occurrence of large jitter and the increase of application-level delay in advance.

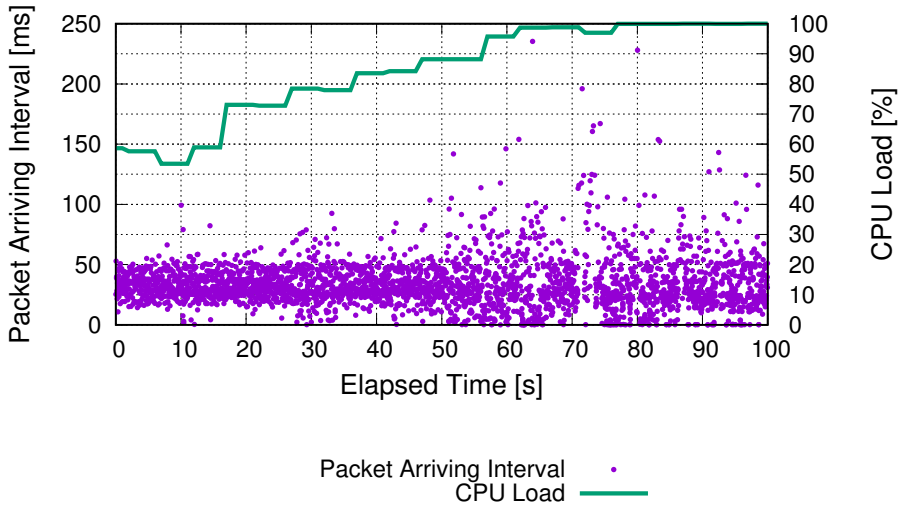


(a) Packet Arriving Intervals

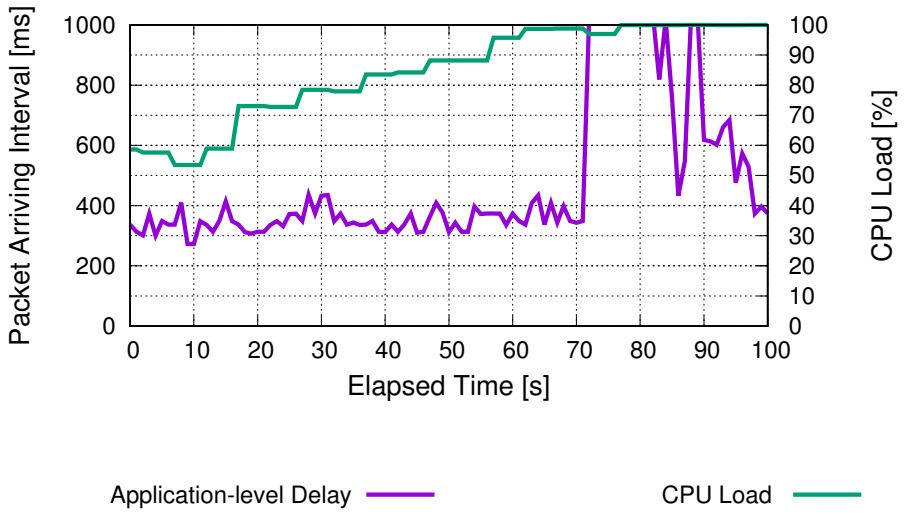


(b) Application-level Delay

Figure 18: Packet Arriving Intervals and Application-level Delay with Live Migration



(a) Packet Arriving Intervals



(b) Application-level Delay

Figure 19: Packet Arriving Intervals and Application-level Delay without Live Migration

5 Conclusion

In this thesis, we measured the application-level delays of video live streaming and analyze the delays in detail. As a result of the analysis, it was revealed that, for low-latency video live streaming, it is naturally important that the average of propagation delays is small, but it is also important to suppress the jitter caused by the network side. Therefore, in order to realize low-latency video live streaming, it is naturally important that the average of propagation delay is small, but it is also necessary to suppress or avoid the occurrence of jitter on the network side. Based on this finding, we devised a service function reallocation method for low-latency video live streaming. In our method, we focused on the increases of jitter and application-level delay caused by the increase of CPU load, and the timing of service function reallocation was determined based on the CPU utilization of edge server. Also, we confirmed that our method prevent the occurrence of large jitter and the increase of application-level delay in advance.

After all, for low-latency video live streaming, shortening the communication distance is a top priority in the three solutions we suggested in Section 3.3. This is because jitter may be occurring due to long communication distance. Service provision of using edge servers can reduce propagation delay and reduce the probability of occurrence of jitter. However, there is a possibility that jitter may occur even with edge servers because of their load concentration. Jitter has various causes, and it is scary because orchestrators are hard to detect its occurrence. Also, jitter may degrade video quality, since the application cannot wait for a delayed packet and deals with it as a lost packet. It thus is important for developers of MEC applications and operators of MEC environments to decide the policies of service function placement and reallocation based on the characteristics of the MEC services. For example, processing at edge server is important for services requiring high real-time processing such as real-time two-way communication service. It is also important in systems such as autonomous vehicle control and robot control via networks. Those systems expect to receive data frequently, periodically and accurately.

Recently, QoE has been used to evaluate users' perceived quality of services [15]. Unlike QoS metrics, which are measured on the network side, QoE is based on user experience, perception, and expectations regarding application and network performance. The work

in our thesis does not focus on QoE, but the measurement and analysis of application-level delay will contribute to understanding QoE in MEC environments, because QoE metrics include application-level delay [15].

As a future work, we will perform live migration in larger scales such as metropolitan area network (MAN) and wide area network (WAN) environments. Live migration between far-off edge servers and between edge servers and data centers should be considered in our work for future deployment of MEC. In larger scales, total migration time and downtime will be larger [16] than what is shown in Section 4.2. Also, we need to perform post-copy migration that can shift service functions immediately after starting the migration. Since post-copy migration can degrade the robustness, it may have a significant impact on application-level delays during migration. As a further prospect, we will develop a two-way communication application. In such an application, real-time performance becomes even more important. Therefore, we evaluate the effect of application-level delay on the user's QoE. Moreover, we would like to evaluate the effect in a service providing non-visual information using tactile sensors.

Acknowledgments

I sincerely express my greatest appreciation to Professor Masayuki Murata of Osaka University. Thanks to his farsighted idea, my research began. Also, his continuous supports, including insightful advice and meticulous comments, were indispensable in developing my research. It has certainly been an honor to be a student of him. I greatly appreciate him.

Furthermore, my heartfelt appreciation goes to Associate Professor Shin'ichi Arakawa of Osaka University for contributing to the progress of my research with his valuable advice, technical supports and constructive discussion. This thesis would not be accomplished without his supports.

Associate Professor Yuichi Ohsita and Assistant Professor Daichi Kominami of Osaka University gave me objective comments and feedback. All the comments and feedback were helpful for me to evaluate my research from diversified perspectives. I offer my special thanks to them.

Moreover, Mr. Kouki Inoue instructed me how to write papers and make presentations. Discussion with Mr. Kohdai Kanda was an enormous help to me. Dr. Toshihiko Ohba, Mr. Koki Sakamoto, Ms. Shiori Takagi, Mr. Yuki Tsukui and all the other members of the laboratory also supported me. My fruitful university days with all my friends and colleagues are irreplaceable time in my life. I am thankful to all of them.

Finally, I show my deep gratitude to my parents for providing me an opportunity to study in Osaka University.

References

[1] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, "The Rise of Big Data on Cloud Computing: Review and Open Research Issues," *Information Systems*, vol. 47, pp. 98–115, Jan. 2015.

[2] Z. Huang, W. Li, P. Hui, and C. Peylo, "CloudRidAR: A Cloud-based Architecture for Mobile Augmented Reality," in *Proceedings of ACM Workshop for Mobile Augmented Reality and Robotic Technology-based Systems*, Jun. 2014, pp. 29–34.

[3] A. C. Baktir, A. Ozgovde, and C. Ersoy, "How Can Edge Computing Benefit from Software-Defined Networking: A Survey, Use Cases & Future Directions," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2359–2391, Jun. 2017.

[4] "Mobile Edge Computing - A Key Technology Towards 5G," ETSI, Sep. 2015.

[5] 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 Architecture & Orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, May 2017.

[6] "Mobile-Edge Computing (MEC); Framework and Reference Architecture," ETSI GS MEC 003 V1.1.1, Mar. 2016.

[7] "Network Functions Virtualisation - White Paper #3," ETSI, Oct. 2014.

[8] R. Mijumbi, J. Serrat, J.-l. Gorricho, S. Latré, M. Charalambides, and D. Lopez, "Management and Orchestration Challenges in Network Functions Virtualization," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 98–105, Jan. 2016.

[9] Cloud Edge Computing: Beyond the Data Center. [Online]. Available: https://www.openstack.org/assets/edge/OpenStack-EdgeWhitepaper-v3-online.pdf

[10] SoftBank Robotics. [Online]. Available: https://www.ald.softbankrobotics.com/en/robots/pepper

[11] FFmpeg. [Online]. Available: https://ffmpeg.org/

[12] "Latency in Live Network Video Surveillance," AXIS Communications, 2015.

[13] F. Zhang, G. Liu, X. Fu, and R. Yahyapour, "A Survey on Virtual Machine Migration: Challenges, Techniques and Open Issues," *IEEE Communications Surveys & Tutorials*, Jan. 2018.

[14] B. Gregg, "CPU Utilization is Wrong," Lightning talk at the 16th annual Southern California Linux Expo, Mar. 2018.

[15] S. Winkler and P. Mohandas, "The Evolution of Video Quality Measurement: FromP-SNR to Hybrid Metrics," *IEEE Transactions on Broadcasting*, vol. 54, no. 3, pp. 660–668, Jun. 2008.

[16] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. De Laat, J. Mambretti, I. Monga, B. Van Oudenaarde, S. Raghunath, and P. Y. Wang, "Seamless Live Migration of Virtual Machines over the MAN/WAN," *Future Generation Computer Systems*, vol. 22, no. 8, pp. 901–907, Oct. 2006.