Master's Thesis

Title

Implementation and Evaluation of a Network-oriented Mixed Reality Service based on Core/Periphery Structure

Supervisor

Professor Masayuki Murata

Author

Shiori Takagi

February 5th, 2020

Department of Information Networking

Graduate School of Information Science and Technology

Osaka University

Master's Thesis

Implementation and Evaluation of a Network-oriented Mixed Reality Service based on Core/Periphery Structure

Shiori Takagi

Abstract

In recent years, many different new network-oriented services has developed, and Multi-access Edge Computing (MEC) is standardized to improve the responsiveness of services. When deploying services in MEC environment, it is necessary to consider a service structure that can switch service behaviors flexibly to meet various users' demands and can change the service behavior to real-world environment at low implementation cost. In this thesis, I introduce a Core/Periphery structure, which is known as a model for a flexible behavior of biological systems, of service components, and design and implement a network-oriented mixed reality service based on the Core/Periphery structure. I investigated what kind of functions can be developed due to users' demands, real environment where devices are placed, and the development of new devices. Then, to utilize the flexibility of the Core/Periphery structure, I regarded functions whose behaviors do not change even when users' demands or environmental changes occurs as core functions, and functions whose behaviors can be changed due to users' demands or environmental changes as peripheral functions. I divided and deployed the service functions in MEC environment, and evaluated the effect of the design based on Core/Periphery structure by our experiment environment in our laboratory. My experiment reveals that the implementation cost is reduced while increase of service response time is less than 5.7 [ms]. This result shows that taking advantage of Core/Periphery structure enables to divide service functions appropriately and place the functions in MEC environment appropriately, with a little penalty on latency and with low implementation cost.

Keywords

Core/Periphery Structure

Multi-access Edge Computing (MEC)

Implementation Costs

Mixed Reality (MR)

Telexistence Service

Network Robots

Contents

List of Figures

List of Tables

1 Introduction

In recent years, with development of IoT (Internet of Things), many different new network-oriented services have developed and information network has changed rapidly. In new network-oriented services, we can send information of real world retrieved from cameras and sensors to cloud, or perform high-load processing such as image recognition and/or voice/sound recognition. For example, telexistence services using robots and VR (Virtual Reality) technology or MR (Mixed Reality) technology are now being developed. In ANA Avatar project [1], using robotics and techniques to send tactile feeling, development of services in which users can communicate with remote places by operating avatar robots as if they existed there is investigated.

In these applications, application-level delay is a significant factor for service quality. However, application-level delay significantly increases in cloud computing environment due to communication distance and load concentration [2]. Recently, Multi-access Edge Computing (MEC) [2–4] is standardized to relax increase of the application-level delay for delay-sensitive services. In edge computing environment, computing resources and storage are allocated at the edge of the network, so that processing end devices require is performed at the place closer to the end devices. This leads to improvement of application responsiveness by shortening communication distance and load distribution.

Because many different new network-oriented services have developed to meet users' various demands, it is important to consider a service design that can accommodate as many services as possible when deploying network services in MEC environment. However, if developers reconstruct whole services to meet different users' demands or to adapt to the environmental variation such as device evolution, implementation cost increases. Moreover, resources in the MEC environment are not necessarily same as those in cloud computing environment. Resources in the MEC environment are limited by spatial restrictions, and thus, it is difficult to locate all-possible services in advance such that services on edge can adapt to each user's demand and environmental variation. Therefore, it is necessary to consider a service structure that can change behaviors of services in a flexible manner.

My research group has been investigated a Core/Periphery structure [5, 6] of service components to effectively adapt to each user's demand and environmental variation. Core/Periphery structure is a model for a flexible and efficient information processing mechanism in biological systems. Information processing units with Core/Periphery structure is classified as Core or Periphery. Core is composed densely with system constraint, and process information more efficiently, whereas the Periphery, which is connected with Core, can have various configurations, flexibly adapts to environmental changes surrounding the system, and builds flexible and efficient information processing mechanisms with Core. The advantage of Core/Periphery structure on accommodating information services, represented by chains of functions, is numerically investigated in [7], and the results show that Core/Periphery structure leads to less developmental costs for accommodating various kind of information services.

In this thesis, based on the advantage of Core/Periphery structure, I aim to realize a service system which adapts service behaviors to users' various demands, environmental changes such as real environment where the end devices are located, or various devices. Unlike the model-based evaluation of [7], I design and implement a service based on Core/Periphery structure in this thesis. Then, I focus on a shopping service using mixed reality (MR) devices and robots. I implement the service using actual devices, and evaluate the effect of designing services based on Core/Periphery structure by experiment.

When designing services based on the Core/Periphery structure, it is necessary to consider which functions should be implemented as Core and which functions should be implemented as Periphery. First, in the shopping service, I investigated what kind of functions can arise due to users' demands, real environment where devices are placed, and the development of new devices. In order to utilize the flexibility of the Core/Periphery structure, I regard functions whose behaviors do not change even if users' demands or environmental changes occurs as core functions, and functions whose behaviors can be changed due to users' demands or environmental changes as peripheral functions. The core functions enable us to adapt to the emergence of new services by adding or changing some peripheral functions instead of recreating whole services. My experiment proved that the implementation cost is reduced without increasing the service response time compared to the case where the service functions are not divided, because core functions deployed on edge servers cooperate with peripheral functions deployed on end devices.

In addition, I got the guidelines for service function placement from Core/Periphery structure. Taking advantage of Core/Periphery structure enables to divide service functions appropriately, and to deploy functions on different servers or devices. If all functions are not divided and deployed in cloud or end devices, whole services must be recreated when developing a new service in order to adapt to users' various demands or device evolution. Furthermore, allocating core functions on edge servers and peripheral functions on end devices is the most effective in terms of responsiveness of services and implementation cost, because it is possible to form feedback loops only by short-distance communication between end devices and edge servers located near the end devices and adapt to environmental changes in real world.

The remainder of this thesis is organized as follows. Section 2 describes services that are currently being developed or are expected to be developed in the future as related works. Section 3 describes the service targeted in this thesis and the service design based on Core/Periphery structure. Section 4 describes the details of the service implementation and evaluation. Finally, Section 5 describes the conclusions and future works.

2 Network-oriented Mixed Reality Services: Current and Future Perspectives

This section describes network-oriented service that has been developed recently or are expected to be developed in the future.

2.1 Current Services

In recent years, telexistence services have been actively developed, and momentum for social implementation has been rising. The concept of telexistence is that people can feel as if they were actually at remote places. TELESAR V [8] is telexistence master-slave system that enables a user to feel present in a remote environment by transmitting not only video and audio, but also haptic sensation. ANA AVATAR [1] is conceived as a "new mode of instantaneous transportation", that enables humanity to communicate and work as if they were remote places. It uses robotics and technology to send tactile feeling and enables them to operate remote robot. For example, ANA has begun testing "ANA AVATAR MUSEUM", that users enjoy remote aquariums and "ANA AVATAR FISHING", that users enjoy fishing remotely . Furthermore, a telexistence application using drones is developed [9].

2.2 Future Services

With the development of the 6th Generation (6G), new services using technologies that will be difficult to support in 5th Generation (5G) is expected to be developed. Within 10 years, the current remote interaction technologies are obsolete, and a new form of interaction that enable immersion in a remote place will be developed and lead to holographic communication and five sensory communication [10]. Tactile Internet and full-sensory digital reality can be realized by 6G [11]. Authors of [11] also state that the 6G will support underwater and space communications that enable deep sea sightseeing and space travel.

In these applications, application-level delay is a significant factor for service quality. However, application-level delay significantly increases in cloud computing environment due to communication distance and load concentration [2]. Therefore, Multi-access Edge Computing (MEC) [2–4] is expected to be standardized. In edge computing, computing resources and storage are allocated at the edge of the network, so that processing end devices require is performed at the place closer to the end devices. This leads to improvement of application responsiveness by shortening communication distance and load distribution. My research group revealed that the service quality of network-oriented mixed reality service is improved in MEC environment [12]. ETSI ISG (Industry Specification Group) [13] describes video content delivery, video stream analysis and Augmented Reality (AR) as key use cases in MEC, and gives guidelines for software developers.

In current services, audio and video transmission is the mainstream, but considering that the transmission of the five senses information can be realized, it is necessary to construct a service system that can handle multiple input and output. In this thesis, we obtain the guideline of service function placement from the Core/Periphery structure, which is a biological model to process information flexibly and efficiently.

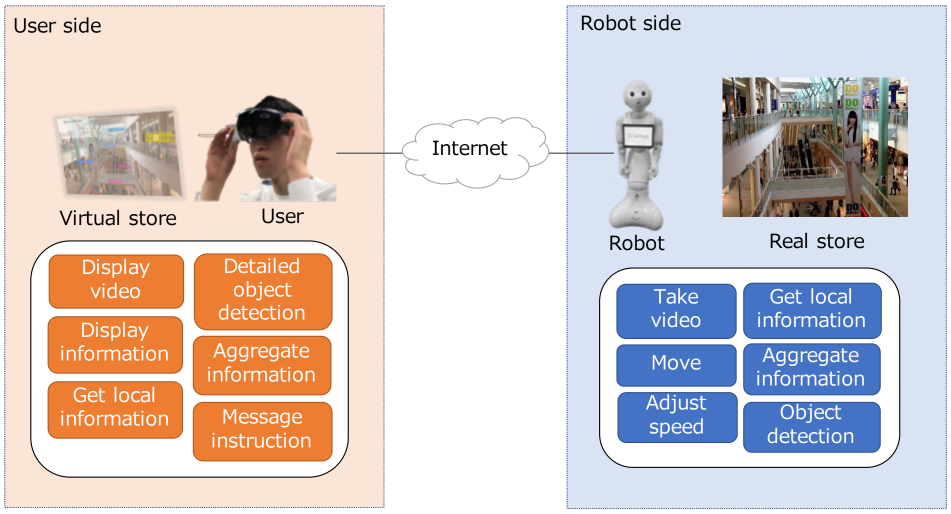


Figure 1: Supposed service and functions.

3 Service Design based on Core/Periphery Structure

This section describes a service design based on Core/Periphery structure.

3.1 Network-oriented Mixed Reality Services under Study

The supposed service is a shopping service using MR and robots. Robots are places in real stores, and users enjoy shopping as if they were in the stores though they are actually at home. Robots take video of the real store while moving as per instructions of users. Real-world information on store-side is attached on video and sent to users. Users can move robots with controllers, gestures, or their gaze. Figure 1 shows the whole service image and service functions.

Robot-side application requires functions for moving, taking video, processing images, collecting and aggregating information around the robot, and adjusting moving speed so as not to hit people or objects. On the other hand, user-side application requires functions for displaying video, messaging instruction to robots, collecting and aggregating information around users, and detecting object based on the granularity users want.

For video processing and live streaming system, I suppose that users' demands are, for example. to watch real-time video, to watch high resolution video, to adopt high accuracy object detection methods, and to deteriorate frame rate or bit rate when communication

Table 1: Examples of demands and functions in video system.

Function	Users' demand	Behavior variation
Capture and output video	Real-time video	Change frame/bit rate
	High-resolution video	Change resolution
Perform object detection	Fast and standard method	Adopt preferred method
	New but slow method	
Distribute video	Send video to one user	Send video with UDP
	Distribute video on a large scale	Distribute video with HTTP

quality become worse. In addition, I suppose that demands from people who place robots in stores, or who provide video are, for example, to distribute video on a large scale, to send video to a single user, and limit bit rate per user of video.

In order to meet the above demands, the video system provides functions for capturing and outputting video, for performing object detection, and for distributing video to users.

Table 1 shows correspondence of uses' demand to functions in video system.

In robot operation system, I suppose that users' demands are, for example, to select which robot to access, to change robot speed, to move robots' arms, to select how to operate robots by users' gestures or controllers, and to follow users' gaze to change the direction of robots. Service behavior can be changed due to variation of real-world environment, such as communication quality, obstacles, or crowd of people. Furthermore, new devices, such as new controllers, new robots or drones, can be used.

In order to meet the above demands, the robot operation system provides functions for recognizing users' instruction such as gestures, gazes, and controller status, for sending message from users, for accessing APIs, for adjusting robot speed in order to avoid obstacles, for collecting and aggregating information obtained from robots.

Table 2 shows correspondence of uses' demand/real-world environment to functions in robot operation system.

Table 2: Examples of demands and functions in robot operation system.

Function	Users' demand/Real world	Behavior variation
Recognize users' instruction	Use gestures to move robot Use controllers to move robot Use gazes to move robots' gaze	Change information to get
Access to APIs	Operate robots Operate drones	Switch APIs to access
Adjust robot speed	where of no obstacles or crowd	Move robot speedily
	where of obstacles or crowd	Move robot slowly

3.2 Function Placements based on Core/Periphery Structure

This section describes function placements based on Core/Periphery structure. In order to place service functions appropriately, I considered which functions described in Section 3.1 are core functions and which functions are peripheral functions, based on the concept of Core/Periphery structure that Core processes information more efficiently, and Periphery has various configurations and flexibly adapts to environmental changes surrounding the system.

For video processing and live streaming system and robot operation system, respectively, I show examples of data flow based on functions supposed in Section 3.1, and show the system structure based on Core/Periphery structure.

3.2.1 Video Processing and Live Streaming

Figure 2, Figure 3, and Figure 4 show examples of data flow in video processing and live streaming system. Arrows represent flows of video data from cameras to users, and dots represent functions. In these data flows, the function for inputting/outputting video is the common function. Figure 2 shows an example of data flow to change frame rate or resolution. Both users and distributors of video can change frame/bit rate, and behaviors of these functions can change due to users' demands. Figure 3 shows an example of data flow for video distributors to select protocol. Video distributors use UDP to send video to a single user, otherwise use HTTP. Protocols and video format can change along service

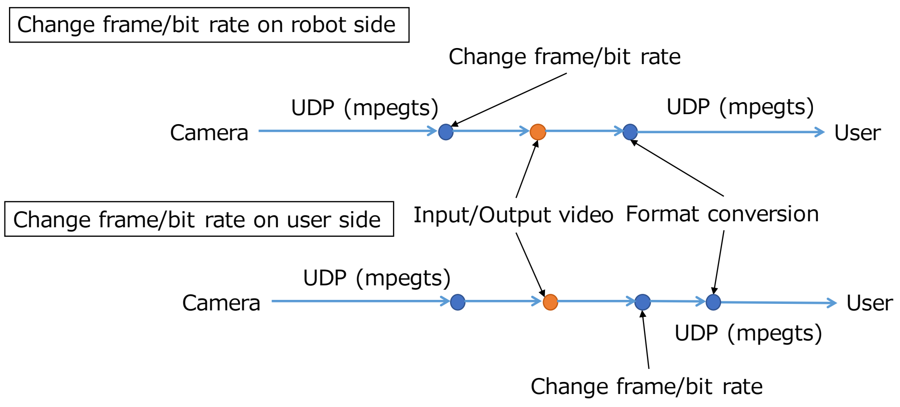


Figure 2: Data flow to change resolution or frame rate.

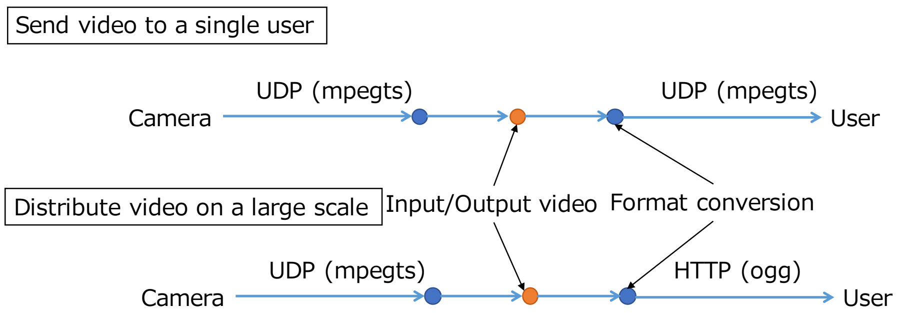


Figure 3: Data flow for video distributors to select protocol.

providers' demands. Figure 4shows an example of data flow for object detection. There are different object detection methods, such as YOLOv3 [14], which is fast and widely used, and Mask R-CNN [15], which is detailed but slow. Users adopt a method they prefer. Among functions shown in Figure 2, 3, and 4, orange-colored functions are common, and they are core functions. Other blue-colored functions are peripheral functions.

Figure 5 shows Core/Periphery structure of video system. Orange-colored field represents Core, and blue-colored fields represent Periphery. Video is sent from camera, and its frame/bit rate is adjusted based on providers' demands as peripheral function. Then, the video passes through core function including functions for inputting video, outputting video, and standard part of object detection. Finally, the video format and distribution protocol are selected along users' demands, and sent to users. By taking flexibility of

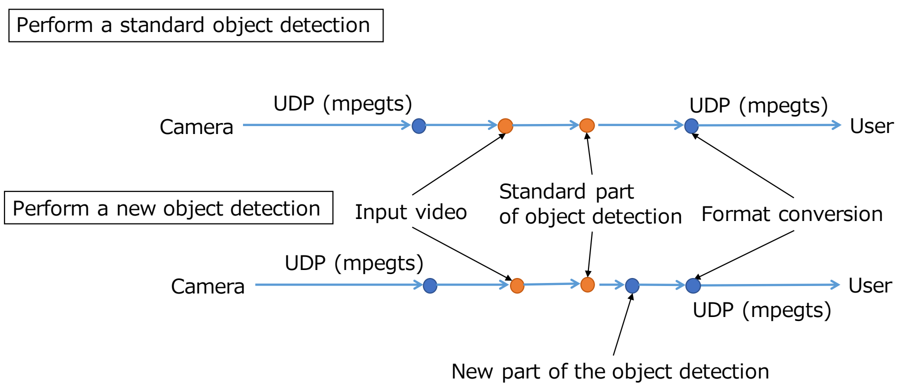


Figure 4: Data flow for object detection.

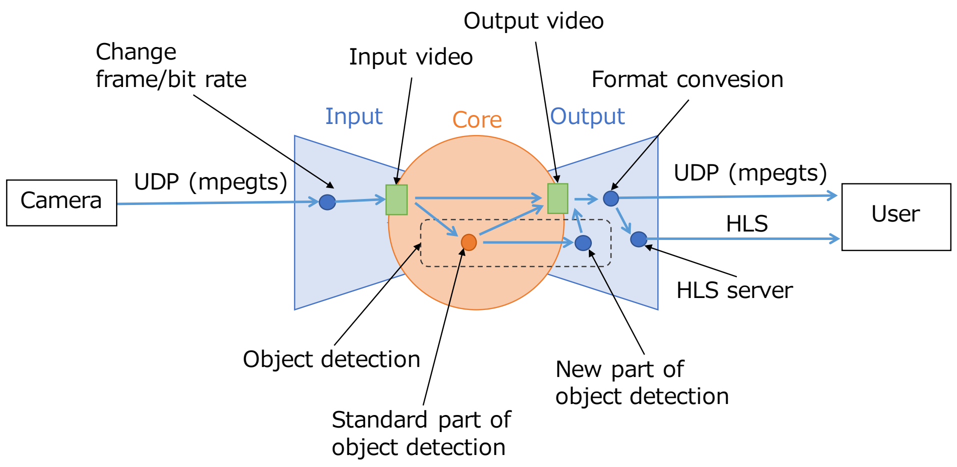


Figure 5: Video system based on Core/Periphery structure.

Core/Periphery structure, all developers have to do is remake or add peripheral functions in order to adapt to different user demands, changes in the real environment where devices are placed, or device evolution.

3.2.2 Robot Operation

Figure 6, Figure 7, and Figure 8 show examples of data flow. Arrows represent flows of instruction messages from users to robots, and dots represent functions. Figure 6 shows an example of data flow select how to operate robot, controller or gesture. Along the way users select, users' device recognizes their instructions and send messages. Figure

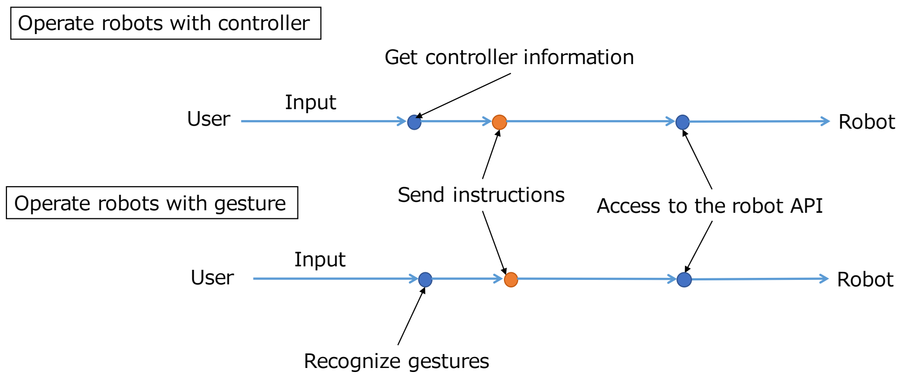


Figure 6: Data flow to how to operate robot.

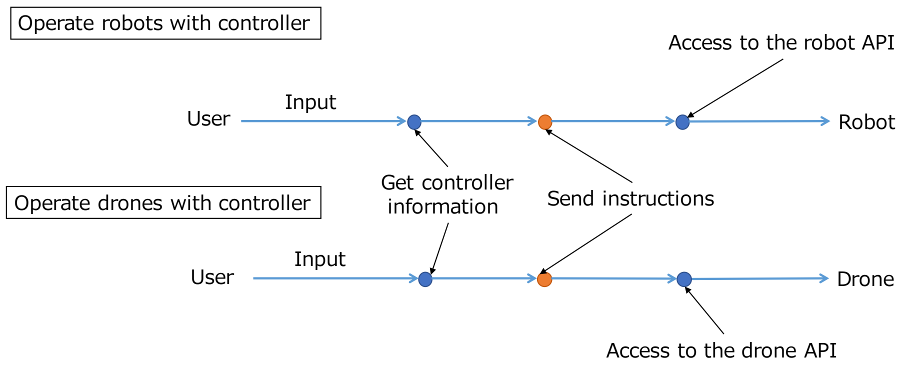


Figure 7: Data flow for different devices..

7 shows an example of data flow for operation of different devices, robots and drones. Service behavior after receiving messages from users changes and accesses to robots' or drones' API. Figure 8 shows an example of data flow to adjust robot speed based on the environment around robots. If there are no obstacles or crowds, users can move robots speedily, otherwise, robots slow down their speed to avoid hitting obstacles and people.

In these data flows, the function for send messages is the common function. Therefore, this function should be divided, rather than whole service is performed as an all-in-one function.

Figure 9 shows Core/Periphery structure of robot operation system. The function to send instruction messages from users and aggregating information obtained from robots are common, so they are core functions. Functions that can change to adapt user demands

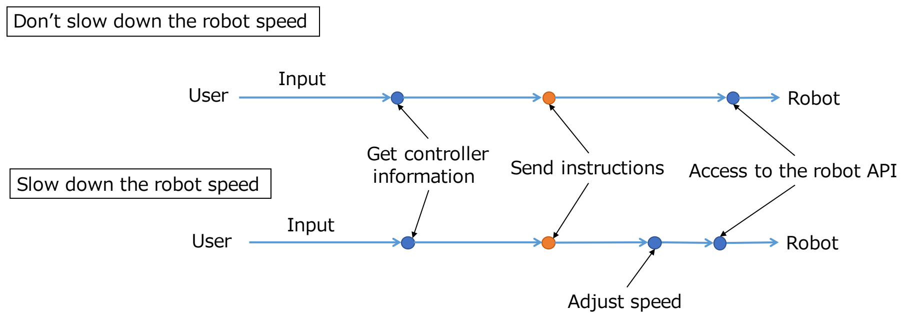

Don't slow down the robot speed
User
Input
Robot
Get controller information
Send instructions
Access to the robot API
Slow down the robot speed
User
Input
Robot
Adjust speed

Figure 8: Data flow to adjust robot speed.

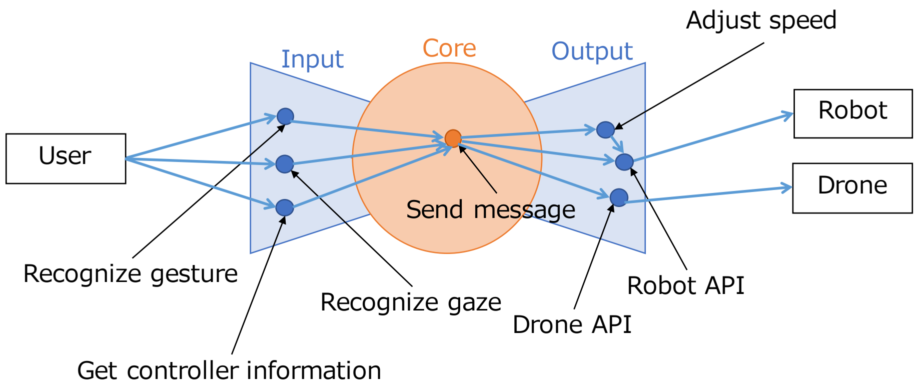


Figure 9: Robot operation system based on Core/Periphery structure.

and changes in the real environment, such as how to input users' instructions are peripheral functions. In addition, functions to access the API of robots, functions to collect information such as current robot position, and functions to adjust the speed are peripheral function because they change due to the type of devices and real-world environment where devices are placed. Because taking flexibility of Core/Periphery structure, all developers have to do is remake or add peripheral functions in order to adapt to different user demands, changes in the real environment where devices are placed, or device evolution.

3.3 Service Scenarios

This subsection describes service scenarios and function placements in MEC environment. First scenario described in 3.3.1 is behavior based on the real environment on the robot

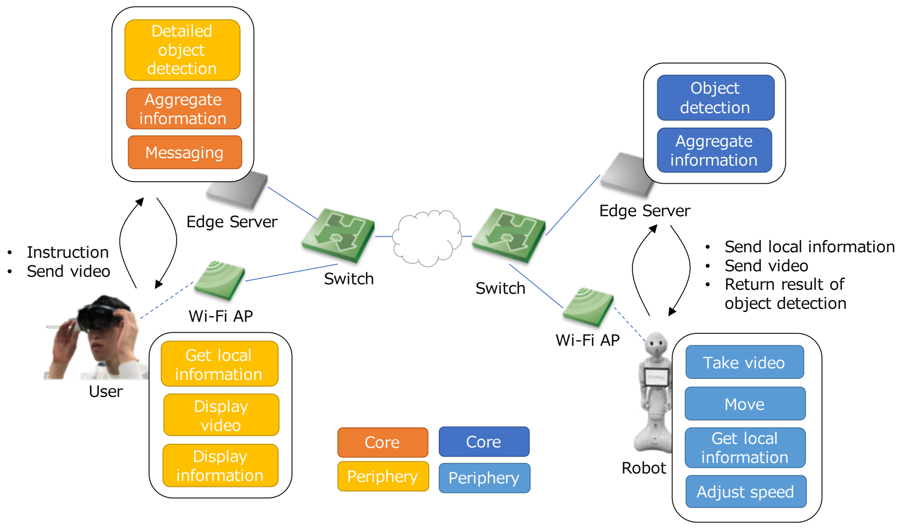


Figure 10: Scenario: behavior based on the real environment on the robot side.

side. Second scenario described in 3.3.2 is behavior based on the real environment on the user side.

3.3.1 Behavior based on the Real Environment on the Robot Side

I describe a scenario in which behavior based on real environment where robots are placed. Functions for robot operation, the functions of Core/Periphery are as follows, respectively.

- Core: functions for transmitting instructions from the user to the robot and functions for object detection
- Periphery: functions to get information near the robot, functions to adjust the movement speed of the robot, and functions to aggregate information sent from multiple robots

Figure 10 shows this scenario. There are users with MR headsets, robots, cameras, and edge servers on robot side. Blue-colored functions are core functions on robot side, and light blue-colored functions are peripheral functions on robot side. Orange-colored functions are core functions on user side, and light orange-colored functions are peripheral functions on user side. Users send their instruction to robot, and move bodies and heads of the robots by controllers, gestures, and their gaze. Robots can detect nearby obstacles

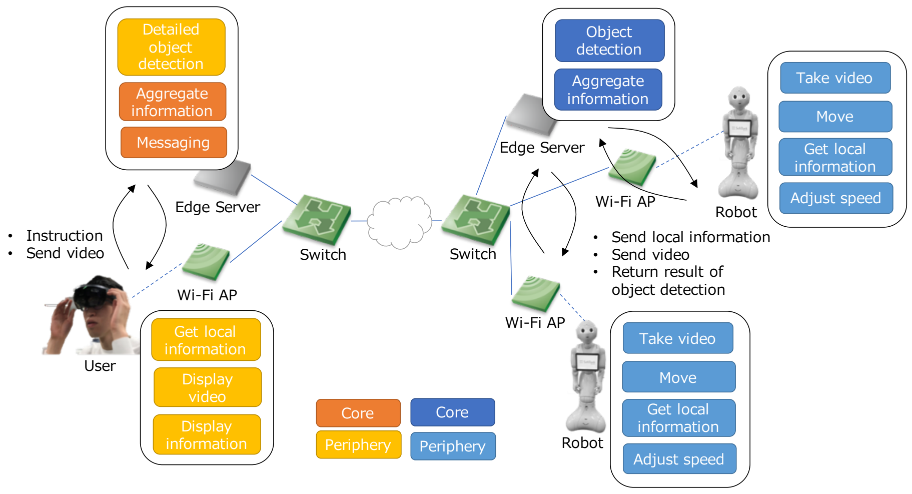


Figure 11: Scenario: behavior based on the real environment on the user side

and stop using sensors mounted on them. In addition, video captured by the camera on robots are sent to the edge servers. Edge servers perform object detection on the video and recognize objects and persons around robots. Object detection function, one of core functions, needs to be performed in real time, and required high specification servers. Therefore, object detection function should be deployed not on end devices but on edge servers. Results of the object detection is returned to robots. For example, if robots know that there are many people around them, The robot reduce speed in order to avoid collision.

Moreover, information obtained from robots can be aggregated in the edge server and shared with other robots. Information sharing enables users to avoid other robots while operating their robots.

3.3.2 Behavior based on the Real Environment on the User Side

I describe a scenario in which behavior based on real environment on user side. Functions of Core/Periphery are as follows, respectively.

- Core: functions for instructions from the user to robots and function to aggregate information of multiple robots
- Periphery: function for displaying video, detailed information of objects, information

about each robot

Figure 11 shows this scenario. Blue-colored functions are core functions on robot side, and light blue-colored functions are peripheral functions on robot side. Orange-colored functions are core functions on user side, and light orange-colored functions are peripheral functions on user side. Information of stores and robots such as product information or communication status is collected at the edge severs on user side. Users select which robot to operate only by communicating with their own edge servers while seeing the aggregated information of the stores and robots.

As a core function, the video sent from cameras is roughly classified into each object type on edge servers on the robot side. Then, as a peripheral function, detailed object detection is performed on the edge servers or users' devices. Users' devices collect personal information such as the users' tastes, information about what the user has, and purchase history. Using the personal information, contents which users want to know are displayed. For example, if purchase history has a commodity that is regularly purchased, the application recommends the commodity based on the history, or the expiration date of a food in users' home is displayed.

4 Implementation and Evaluation of the Service based on Core/Periphery Structure

This section describes implementation detail and evaluation of the service based on 3.3.

4.1 Implementation of the Service based on Core/Periphery Structure

4.1.1 Video Processing and Live Streaming

Video from cameras is sent to the edge server on robot side. The video was captured with OpenCV [16]. Then, object detection with YOLOv3 [14] and PyTorch, a library for deep learning is performed. Figure 12 shows a result of object detection with YOLOv3. Mask R-CNN (Region-based Convolutional Neural Networks) [15], an algorithm that not only surrounds certain areas where objects are detected with a rectangle but also recognizes the type of object for each pixel and colors it, can be used. Figure 13 (citation from [15]) shows a result of object detection with Mask R-CNN. The processed video is transmitted to the HoloLens [17], MR headset worn by users using ffmpeg [18] with UDP and displayed on HoloLens.

HoloLens is a standalone head mounted computer made by Microsoft. It gives users MR experience by displaying holograms and recognizing users' gaze and gestures.

4.1.2 Robot Operation

Information of the controller by HoloLens is transmitted with MQTT (Message Queuing Telemetry Transport), which is a publish/subscribe type protocol and developed for messaging frequently exchanged between IoT devices. I used Xbox controller, which can connect to HoloLens, as the user's controller. I built an MQTT messaging system on the edge server on user side with mosquitto [19], an open source message broker, and Node-RED [20], a programming tool for event-driven applications. The MQTT broker gets controller information from HoloLens and send it to Choregraphe, the programming tool for Pepper [21], robot. A program on Choregraphe accesses to Pepper API when it gets messages from MQTT broker.

Pepper gets lists of object names from the edge server which performs object detection,

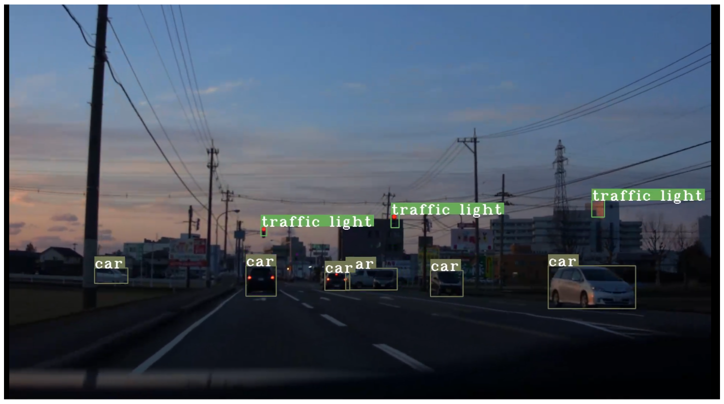


Figure 12: A result of object detection with YOLOv3.

and if there is a person, it reduces speed.

Furthermore, Pepper has a map of an area where it can move, and get its position regularly. Each Pepper's position is aggregated in the edge server and shared with the other Pepper. Pepper's position is also plotted on map, sent to users' HoloLens, and displayed on HoloLens.

Pepper is a humanoid robot made by Softbank Robotics. APIs and the development tool are provided, and developers can create applications executed on Pepper.

Figure 14 shows a screenshot of application on HoloLens. Users can see video with result of object detection and a map made by Pepper displayed at the top left. Green-colored dot represents Pepper's position.

4.2 Evaluation and Experiment Settings

In this subsection, I describe evaluation indexes and experiment settings. Evaluation indexes are implementation cost and responsiveness of the service.

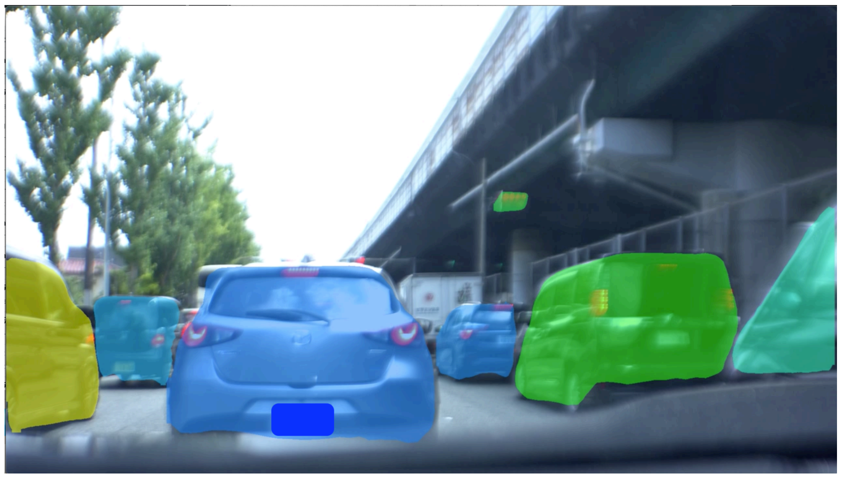


Figure 13: A result of object detection with Mask R-CNN.

4.2.1 Implementation Cost

With the implemented service, I show that the implementation cost is reduced by adopting to the Core/Periphery structure. If all functions are deployed on an end device, the implementation is based on the type of end device, and source code to adapt each devices' features and APIs. Dividing service function of into core function and peripheral function, enables to adapt to device variation because developers have to remake only peripheral function, and can reduce implementation cost. For example, consider the amount of source code for sending users' instruction. Function for messaging users' instruction is deployed on the edge server as core function. Therefore, developers have to write only peripheral functions, the part necessary to access HoloLens and Pepper APIs, and various devices can be handled without adding source code.

I compare amount of the source code when functions are divided to Core and peripheral functions with when whole service is implemented on an end device to evaluate the implementation cost.

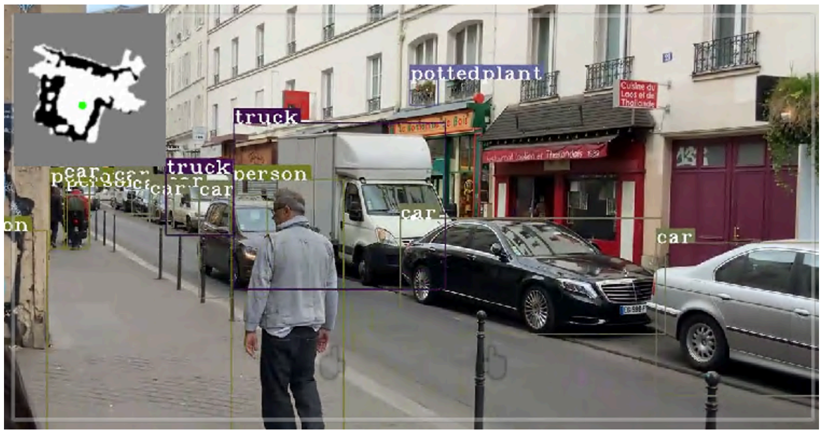


Figure 14: A screenshot of the application on HoloLens.

4.2.2 Responsiveness of the Service

The advantage of Core/Periphery structure is that, in addition to reducing the implementation cost, the service function can be divided and allocated on different servers and devices. The improvement of service responsiveness by placing peripheral functions on end devices and core functions on edge servers evaluated is as an effect of Core/Periphery structure.

When allocating core functions and peripheral functions respectively, core function can be deployed on the cloud or on the edge server, and the peripheral function can be deployed on the edge server or on the end devices. Core functions perform real-time image processing or aggregation of information, and cannot be executed on end devices. In order to improve responsiveness, Core functions should be placed on the edge server instead of the cloud.

I measured the time from when a user inputs an instruction to when the robot starts moving in MEC environment and in cloud environment, respectively. Then, I compared the time and evaluate the effect of allocating core functions on the edge server. I sent

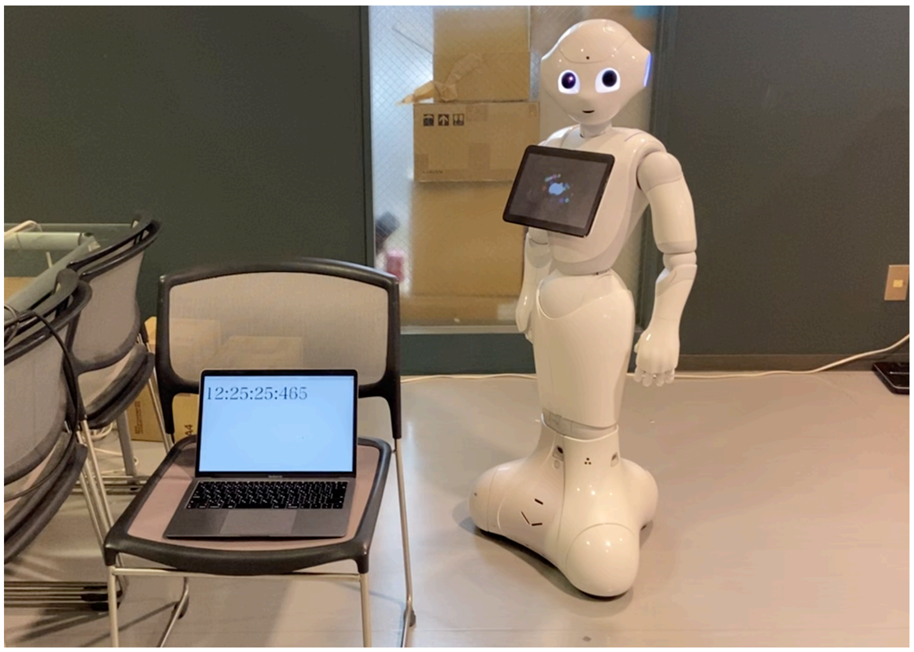


Figure 15: Experiment to measure application-level delay.

messages about 50 times regularly from the HoloLens application and save each time when it sent messages as t_1, t_2, ..., t_n, where n is number of inputs. On the other hand, a digital clock that can display milliseconds is displayed on a laptop nearby Pepper15. I took a video and made records of each time when Pepper started moving as t'_1, t'_2, ..., t'_n. Then, I calculated average of $t'_1 - t_1$, $t'_2 - t_2$, ..., $t'_n - t_n$ as the absolute time from each input to the start of Pepper with time difference between devices. Since the clocks of HoloLens and Pepper are not synchronized, the absolute time from each input to the start of Pepper cannot be measured. Therefore, I calculated the penalty of using the edge server by subtraction these two average times. The reason for adopting this way is that different between two system time cannot be canceled by obtaining the system time on HoloLens and Pepper. When sending messages directly from HoloLens to Pepper, I can only obtain the time when HoloLens sends message and the time when Pepper stops moving both in HoloLens. When sending messages via the edge server, the time when HoloLens sends message can be obtained only in HoloLens and the time when Pepper stops moving can be obtained only in Pepper.

Since application-level delay may increase when users' instructions are sent via the edge

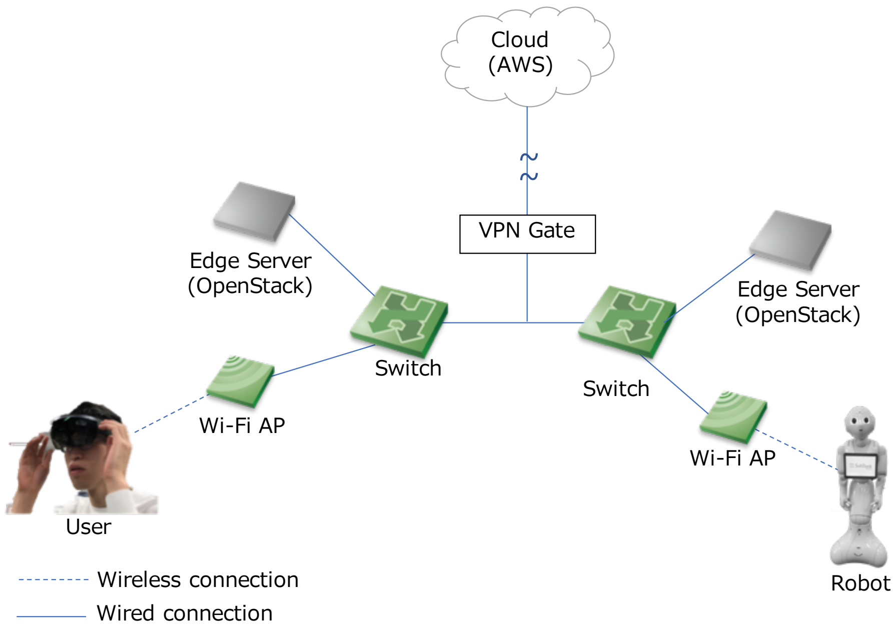


Figure 16: Configuration of MEC environment.

server, compared with when they are sent directly to robots, I measured and evaluated application-level delay as penalty of using edge server. I sent messages about 50 times regularly from the HoloLens application and save each time when it sent messages as same as previously noted. For each input, Pepper saved each time when it started moving. Then, I calculated average the absolute time from each input to the start of Pepper with time difference between devices. Finally, the application-level delay in cloud environment by subtraction these two average times.

Figure 16 shows configuration of the MEC environment. I constructed MEC environment using OpenStack and Amazon Web Service (AWS). The AWS cloud is in Ohio, and connected to OpensStack environment via VPN gate.

4.3 Results

4.3.1 Implementation Cost

I show extracts of source code to implement robot operation system. The whole source code is described in Appendix A. Source code 1 shows a part of source code of HoloLens application, which connects to robot Pepper or another robot, and Source code 2 shows a part which connects to a MQTT broker. After line 8 of Source code 1 represents the code needed to make anther robot usable in the application. Source code 3 and Source code 4 show parts of source code of HoloLens application, which sends messages of Xbox controller to robot Pepper or another robot. Source code 3 and Source code 1 are not designed based on Core/Periphery structure, and perform all in one application from get message to move robots. Since I have not implemented using another drone, Source code 3 is based on the supposition of connecting to estimated another robot. After line 37 of Source code 3 represents the code get information from another robot in the application. Source code 4 and Source code 2 are designed based on Core/Periphery structure. Service functions are divided and MQTT is used to send messages. Compared to the case where the functions are not divided, in case the functions are divided and MQTT is used, developers do not need to change nor add source code for accessing each device APIs to establish connections, to disconnect, and to move devices in remote places, and settings of relationship between information of controller and movement distance of devices (parameter “move_scalefactor” in Source code 3), when the type of devices increases. However, preparing topics on the MQTT broker, connecting to the MQTT with devices in remote places, and writing source code for processing after subscribe messages from the MQTT broker are needed.

For amount of the entire source code, the implementation cost is reduced due to the effect of the connection establishment part, shown in Source code 2. In terms of the cost for implementing the application on the user side, the case when devices on user side establish connections directly with devices in remote device, developers cannot write many parts of source code unless they know APIs of different devices.

Therefore, developers can implement applications more easily by adopt Core/Periphery structure and use MQTT.

Source code 1: Establish connection to robots.

```
//Pepper
if(!string.IsNullOrEmpty(pepperIP)){
  _session = QiSession.Create(tcpPrefix + pepperIP + portSuffix);
  if (!_session.IsConnected){
    Debug.Log("Failed to establish connection");
    return;
  }
//another robot (estimated)
}else if(!string.IsNullOrEmpty(robotIP)){
  session_robot = RobotSession.Create(tcpPrefix + robotIP + portSuffix);
  if (!_session.IsConnected){
    Debug.Log("Failed to establish connection");
    return;
  }
}
```

Source code 2: Establish connection to MQTT broker.

```
BrokerAddress = "192.168.10.73";
clientId = Guid.NewGuid().ToString();
client = new MqttClient(BrokerAddress);
client.ProtocolVersion = MqttProtocolVersion.Version_3_1;
try{
  client.Connect(clientId);
} catch (Exception e){
  Debug.Log(string.Format("Exception has occurred in connecting to MQTT 
{0} ", e ));
  throw new Exception("Exception has occurred in connecting to MQTT", e.
InnerException);
}
```

Source code 3: Send messages Directly.

```
if(_session.IsConnected){
  if (eventData.XboxLeftStickHorizontalAxis != 0 || eventData.
  XboxLeftStickVerticalAxis != 0){
    var motion = _session.GetService("ALMotion");
    motion["moveTo"].Call(eventData.XboxLeftStickHorizontalAxis * move_scalefactor,
  eventData.XboxLeftStickVerticalAxis * (-1) * move_scalefactor, 0f);
  }
  if (eventData.XboxLeftBumper_Pressed){
    var motion = _session.GetService("ALMotion");
    motion["moveTo"].Call(0f, 0f, rotation_scalefactor);
  }else if (eventData.XboxRightBumper_Pressed){
    var motion = _session.GetService("ALMotion");
    motion["moveTo"].Call(0f, 0f, (-1) * rotation_scalefactor);
  }
  if (eventData.XboxB_Pressed){
    if (Time.time - first_buttonpressed > timeBetweenbuttonpressed){
      var motion = _session.GetService("ALMotion");
      motion["setAngles"].Call("HeadYaw", angle, 0f);
    }
    first_buttonpressed = Time.time;
  }
  if (eventData.XboxX_Pressed){
    if (Time.time - first_buttonpressed > timeBetweenbuttonpressed){
      if (pepperIP == "192.168.10.49"){
        _session.Close();
        _session.Destroy();
        pepperIP = "192.168.10.48"
        _session = QiSession.Create(tcpPrefix + pepperIP + portSuffix);
      }else{
        _session.Close();
        _session.Destroy();
        pepperIP = "192.168.10.49"
        _session = QiSession.Create(tcpPrefix + pepperIP + portSuffix);
      }
    }
    first_buttonpressed = Time.time;
  }

  //another robot (estimated)
  }else if(session_robot.isConnected){
    if (eventData.XboxLeftStickHorizontalAxis != 0 || eventData.
  XboxLeftStickVerticalAxis != 0){
      CallRobotsAPI_move(eventData.XboxLeftStickHorizontalAxis *
  move_scalefactor_robot, eventData.XboxLeftStickVerticalAxis * (-1) *
  move_scalefactor_drone, 0f);
    }
  if (eventData.XboxLeftBumper_Pressed){
    CallRobotsAPI_rotate(0f, 0f, rotation_scalefactor);
  }else if (eventData.XboxRightBumper_Pressed){
    CallRobotsAPI_rotate(0f, 0f, (-1) * rotation_scalefactor);
  }
  if (eventData.XboxB_Pressed){
    if (Time.time - first_buttonpressed > timeBetweenbuttonpressed){
      CallRobotsAPI_rotate(0f, 0f, 0f);
      }
      first_buttonpressed = Time.time;
    }
    if (eventData.XboxX_Pressed){
      if (Time.time − first_buttonpressed > timeBetweenbuttonpressed){
        if (robotIP == "192.168.10.50"){
          session_robot.Close();
          session_robot.Destroy();
          robotIP = "192.168.10.51"
          session_robot = RobotSession.Create(tcpPrefix + robotIP + portSuffix);
        }else{
          session_robot.Close();
          session_robot.Destroy();
          robotIP = "192.168.10.50"
          session_robot = RobotSession.Create(tcpPrefix + robotIP + portSuffix);
        }
      }
      first_buttonpressed = Time.time;
    }
  }

```

Source code 4: Send message via MQTT.

```
if (eventData.XboxLeftStickHorizontalAxis != 0 || eventData.XboxLeftStickVerticalAxis
    != 0){
      msg = string.Format("{1}␣{0}␣0␣{2}", (−0.2) ∗ eventData.
    XboxLeftStickHorizontalAxis, (−0.2) ∗ eventData.XboxLeftStickVerticalAxis,
    deviceIP);
      client.Publish(topicPublishPath, Encoding.UTF8.GetBytes(msg), MqttMsgBase.
    QOS_LEVEL_AT_MOST_ONCE, true);
    }
    if (eventData.XboxLeftBumper_Pressed){
      msg = string.Format("0␣0␣15␣{0}", deviceIP);
      client.Publish(topicPublishPath, Encoding.UTF8.GetBytes(msg), MqttMsgBase.
    QOS_LEVEL_AT_MOST_ONCE, true);
    }
    else if (eventData.XboxRightBumper_Pressed){
      msg = string.Format("0␣0␣-15␣{0}", deviceIP);
      client.Publish(topicPublishPath, Encoding.UTF8.GetBytes(msg), MqttMsgBase.
    QOS_LEVEL_AT_MOST_ONCE, true);
    }
    if (eventData.XboxB_Pressed){
      if (Time.time − first_buttonpressed > timeBetweenbuttonpressed){
        msg = string.Format("b");
        client.Publish(topicPublishPath_button, Encoding.UTF8.GetBytes(msg),
    MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE, true);
      }
      first_buttonpressed = Time.time;
    }
    if (eventData.XboxX_Pressed){
      if (Time.time − first_buttonpressed > timeBetweenbuttonpressed){
        if (deviceIP == 51){
          deviceIP = 48;
        }else{
          deviceIP++;
```

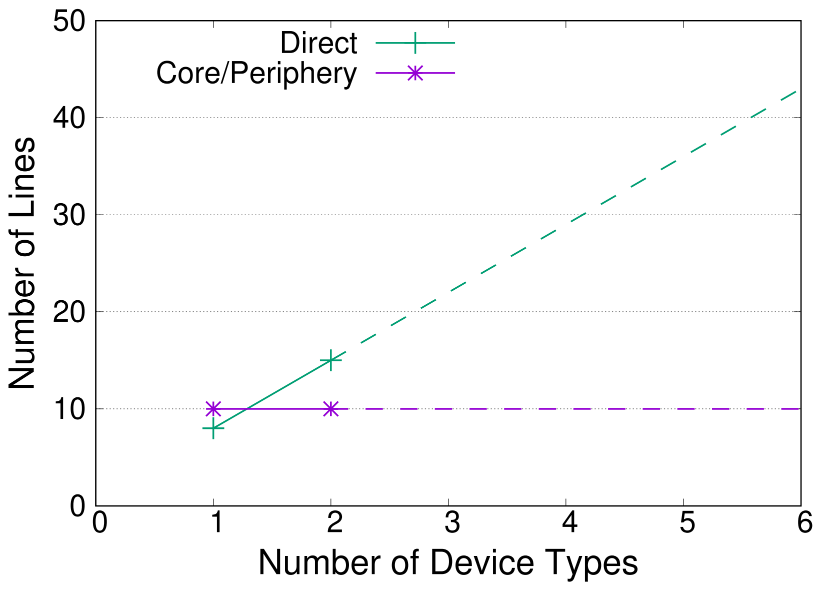
```
            }
          }
          first_buttonpressed = Time.time;
        }

```

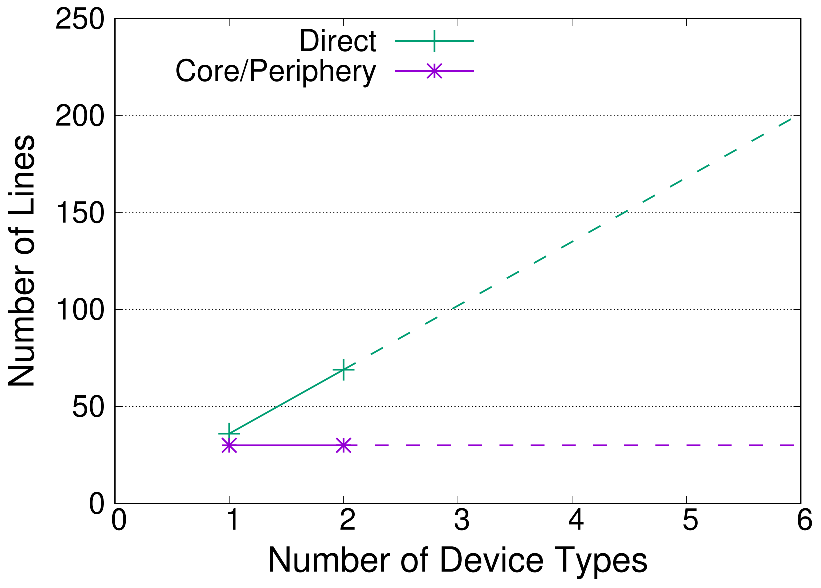
Figure 17 shows the relation between the number of device types and the number of lines of source code. Both Figure (a), the connection establishment part, and Figure (b), the messaging part, show that designing services based on Core/Periphery structure is effective when the number of device types.

Not only variation of device in remote places, but also variation of devices on user side. In both services based on Core/Periphery and services not based on Core/Periphery structure, developers have to add source code to get information of controller, because this function is peripheral. However, the more types of controllers, the more parts of source code to add when the types of devices on the remote side increase, effect obtained by designing services based on Core/Periphery structure.

Furthermore, I consider the implementation cost for sharing information among robots. Service structure without Core/Periphery structure does not have edge servers. In order to share information such as robot positions, robots have to establish connections each other. Therefore, each time when new robot appears, developers have to edit source code to connect the new robot on each robot. By designing based on Core/Periphery structure, since robots only send its information to edge servers and edge servers aggregate information, source code does not need to be changed even when new robots appear.



(a) Connection establishment part



(b) Messaging part

Figure 17: Number of lines of source code.

4.3.2 Responsiveness of the Service

Figure 18 shows the difference of application-level delay between the case the HoloLens application directly connects the robot Pepper, and the case of the HoloLens application connects Pepper via the edge server. Result shown Table 4 revealed that the delay due to

Table 3: Result of the experiment to measure the delay due to the cloud.

	Core on edge	Core on cloud
Number of input	49	50
Average of interval (with time difference) [ms]	-10.4	463.9

Table 4: Result of the experiment to measure the penalty of using edge server.

	Direct	Core on edge
Number of input	41	48
Average of interval (with time difference) [ms]	343.9	349.6

MQTT on the edge server was less than 5.7 [ms]. Combining with the results shown in4.3.1, the service design based on Core/Periphery structure enables to reduce the implementation cost without deteriorating responsiveness of the service much.

Result shown Table 3 and 19 revealed that the difference of application-level delay between using the edge server and using the cloud was 474.3 [ms]. The difference of RTT between PC-to-Edge and PC-to-Cloud is 140 [ms], and the difference of application-level delay is about 3 times larger. Therefore, deploying core functions on edge servers is significant.

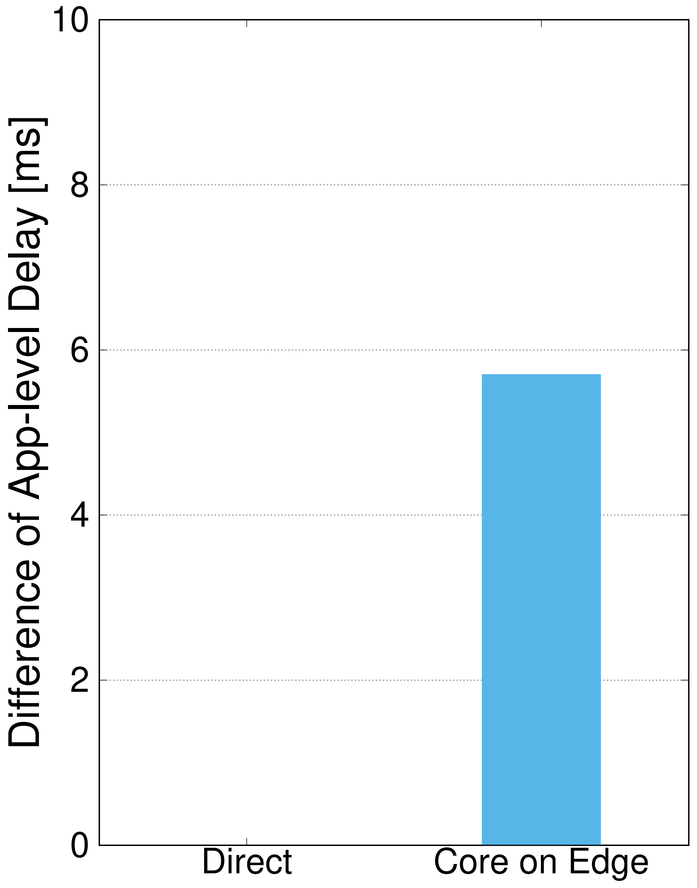


Figure 18: Difference of application-level delay.

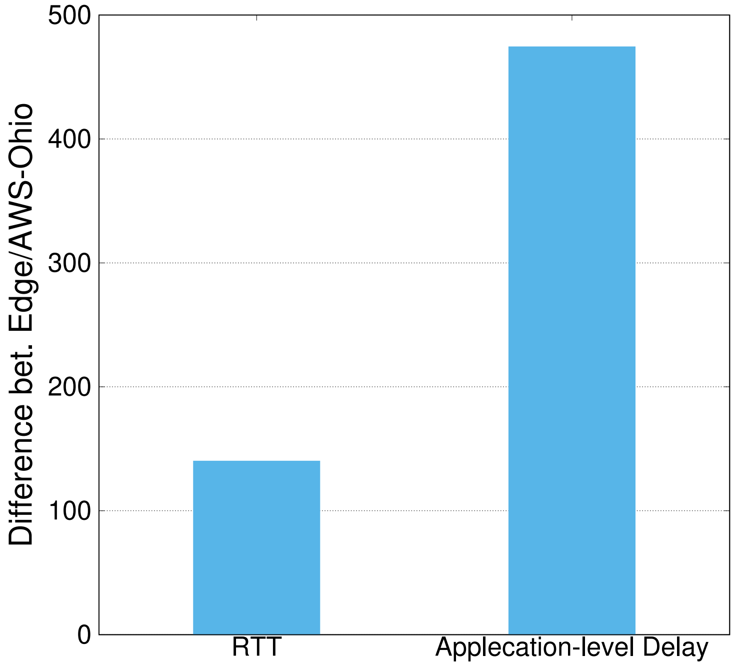


Figure 19: Difference between RTT and application-level delay in cloud environment.

5 Conclusion

In this thesis, I introduced a Core/Periphery structure, which is known as a model for a flexible behavior of biological systems, of service components, and designed and implemented a network-oriented mixed reality service based on the Core/Periphery structure.

First, in the supposed service, I investigated what kind of functions can be developed due to users' demands, real environment where devices are placed, and the development of new devices. In order to utilize the flexibility of the Core/Periphery structure, I regarded functions whose behaviors do not change even if users' demands or environmental changes occurs as core functions, and functions whose behaviors can be changed due to users' demands or environmental changes as peripheral functions. Second, I implemented applications based on scenarios described in Section 3.3, and evaluated the effect of the design based on Core/Periphery structure by our experiment environment in our laboratory.

My experiment revealed that the application-level delay due to MQTT on the edge server was less than 5.7 [ms]. Taking advantage of Core/Periphery structure enables to divide service functions appropriately and place the functions in MEC environment appropriately, so that implementation cost is reduced with little penalty.

As future works, I will evaluate the implementation cost and for object detection and feedback to robots, for sharing information among robots. Furthermore, implementation and evaluation of the service using different devices such as robots rather than Pepper is needed. Service design based on Core/Periphery structure is more efficient when there are various devices, but in this thesis, I implemented and evaluated the service using one kind of headset and robot.

Acknowledgments

This thesis would not accomplish without a lot of great support from many people. First, I would like to express my deepest gratitude to Professor Masayuki Murata of Osaka University, for providing me with the opportunity to research with a talented team of researchers. His creative suggestions, insightful comments, and patient encouragement have been essential for my research activity. 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. I would like to thank Dr. Koki Inoue, Mr. Junichi Kaneda, and all the members of Advanced Network Architecture Research Laboratory. Last, but not least, I thank my parents for their invaluable support and constant encouragement during my master studies.

References

[1] "ANA Avatar." `https://ana-avatar.com`.

[2] A. C. Baktir, A. Ozgovde, and C. Ersoy, "How can edge computing benefit from software-defined networking: A survey, use cases, and future directions," *IEEE Communications Surveys Tutorials*, vol. 19, pp. 2359–2391, June 2017.

[3] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing a key technology towards 5G," *ETSI White Paper*, Sept. 2015.

[4] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration," *IEEE Communications Surveys Tutorials*, vol. 19, pp. 1657–1681, May 2017.

[5] P. Csermely, A. London, L.-Y. Wu, and B. Uzzi, "Structure and dynamics of core-periphery networks," *Journal of Complex Networks*, vol. 1, pp. 93–123, Sept. 2013.

[6] V. Miele, R. Ramos-Jiliberto, and D. P. Vázquez, "Core–periphery dynamics in a plant–pollinator network," *bioRxiv*, July 2019.

[7] Y. Tsukui, "On network function virtualization for dynamically changing service requests based on a core/periphery structure," Master's thesis, Graduate School of Information Science and Technology, Osaka University, Feb. 2020.

[8] S. Tachi, "Telexistence: Enabling humans to be virtually ubiquitous," *IEEE Computer Graphics and Applications*, vol. 36, pp. 8–14, Jan. 2016.

[9] X. Xia, C. Pun, D. Zhang, Y. Yang, H. Lu, H. Gao, and F. Xu, "A 6-DOF telexistence drone controlled by a head mounted display," in *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 1241–1242, Mar. 2019.

[10] E. C. Strinati, S. Barbarossa, J. L. Gonzalez-Jimenez, D. Ktenas, N. Cassiau, L. Maret, and C. Dehos, "6G: The next frontier: From holographic messaging to artificial intelligence using subterahertz and visible light communication," *IEEE Vehicular Technology Magazine*, vol. 14, pp. 42–50, Sept. 2019.

[11] Z. Zhang, Y. Xiao, Z. Ma, M. Xiao, Z. Ding, X. Lei, G. K. Karagiannidis, and P. Fan, "6G wireless networks: Vision, requirements, architecture, and key technologies," *IEEE Vehicular Technology Magazine*, vol. 14, pp. 28–41, Sept. 2019.

[12] S. Takagi, J. Kaneda, S. Arakawa, and M. Murata, "An improvement of service qualities by edge computing in network-oriented mixed reality application," in *2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)*, pp. 773–778, Apr. 2019.

[13] D. Sabella, V. Sukhomlinov, L. Trang, S. Kekki, P. Paglierani, R. Rossbach, X. Li, Y. Fang, D. Druta, F. Giust, L. Cominardi, W. Featherstone, B. Pike, and S. Hadad, "Developing software for multi-access edge computing," *ETSI White Paper*, Feb. 2019.

[14] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," *CoRR*, vol. abs/1804.02767, Apr. 2018.

[15] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proceedings of 2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2980–2988, Oct. 2017.

[16] "OpenCV." `https://opencv.org`.

[17] "Microsoft HoloLens." `https://www.microsoft.com/ja-jp/hololens`.

[18] "FFmpeg." `https://www.ffmpeg.org/`.

[19] "Eclipse Mosquitto." `https://mosquitto.org`.

[20] "Node-RED." `https://nodered.org`.

[21] "Pepper the humanoid robot - SoftBank Robotics." `https://www.softbankrobotics.com/emea/en/pepper`.

Appendix

A Source code for evaluation

Source code 5 is a program for HoloLens to send messages of Xbox controller directly to Pepper and a drone. Source code6 is a program for HoloLens to send messages of Xbox controller to the MQTT broker. Source code 7 is a program for Pepper to receive messages from MQTT broker and move.

Source code 5: HoloLens connects to Pepper directly.

```
using UnityEngine;
using UnityEngine.UI;
using MiniJSON;
using System;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using Baku.LibqiDotNet;
using LibraryForDrone; //fictious

namespace HoloToolkit.Unity.InputModule.Tests{
  public class xbox_direct : XboxControllerHandlerBase{
    [Header("Xbox Controller Test Options")]
    [SerializeField]
    private float movementSpeedMultiplier = 1f;

    [SerializeField]
    private float rotationSpeedMultiplier = 1f;

    [SerializeField]
    private XboxControllerMappingTypes resetButton = XboxControllerMappingTypes.
      XboxY;

    private Vector3 initialPosition;
    private Vector3 newPosition;
    private Vector3 newRotation;

    public string pepperIP; //192.168.10.48 - 49
    public string droneIP; //192.168.10.50 - 51

    public float rotation_scalefactor = 0.785f;

    //Pepper
    private const string tcpPrefix = "tcp://";
    private const string portSuffix = ":9559";
    private QiSession _session;
    public float move_scalefactor = 3.0f;

    //Drone (fictious)
    private DroneSession session_drone;
```

```
public float move_scalefactor_drone = 5.0f;

void Start(){
  initialPosition = transform.position;
  //Pepper
  if(!string.IsNullOrEmpty(pepperIP)){
    _session = QiSession.Create(tcpPrefix + pepperIP + portSuffix);
    if (!_session.IsConnected){
      Debug.Log("Failed to establish connection");
      return;
    }
  //Drone (fictious)
  }else if(!string.IsNullOrEmpty(droneIP)){
    session_drone = DroneSession.Create(tcpPrefix + droneIP + portSuffix);
    if (!_session.IsConnected){
      Debug.Log("Failed to establish connection");
      return;
    }
  }
}

public override void OnXboxInputUpdate(XboxControllerEventData eventData){
if (string.IsNullOrEmpty(GamePadName)){
  Debug.LogFormat("Joystick {0} with id: \"{1}\" Connected", eventData.
  GamePadName, eventData.SourceId);
}

base.OnXboxInputUpdate(eventData);

/*get information from xbox controller*/

if(_session.IsConnected){
  if (eventData.XboxLeftStickHorizontalAxis != 0 || eventData.
  XboxLeftStickVerticalAxis != 0){
    var motion = _session.GetService("ALMotion");
    motion["moveTo"].Call(eventData.XboxLeftStickHorizontalAxis * move_scalefactor,
  eventData.XboxLeftStickVerticalAxis * (-1) * move_scalefactor, 0f);
  }
  if (eventData.XboxLeftBumper_Pressed){
    var motion = _session.GetService("ALMotion");
    motion["moveTo"].Call(0f, 0f, rotation_scalefactor);
  }else if (eventData.XboxRightBumper_Pressed){
    var motion = _session.GetService("ALMotion");
    motion["moveTo"].Call(0f, 0f, (-1) * rotation_scalefactor);
  }
  if (eventData.XboxB_Pressed){
    if (Time.time - first_buttonpressed > timeBetweenbuttonpressed){
      var motion = _session.GetService("ALMotion");
      motion["setAngles"].Call("HeadYaw", angle, 0f);
    }
    first_buttonpressed = Time.time;
  }
  if (eventData.XboxX_Pressed){
    if (Time.time - first_buttonpressed > timeBetweenbuttonpressed){
      if (pepperIP == "192.168.10.49"){
        _session.Close();
        _session.Destroy();
        pepperIP = "192.168.10.48"
```

```
          _session = QiSession.Create(tcpPrefix + pepperIP + portSuffix);
        }else{
          _session.Close();
          _session.Destroy();
          pepperIP = "192.168.10.49"
          _session = QiSession.Create(tcpPrefix + pepperIP + portSuffix);
        }
      }
      first_buttonpressed = Time.time;
    }

    //Drone (fictious)
    }else if(session_drone.isConnected){
      if (eventData.XboxLeftStickHorizontalAxis != 0 || eventData.
    XboxLeftStickVerticalAxis != 0){
        CallDronesAPI_move(eventData.XboxLeftStickHorizontalAxis *
    move_scalefactor_drone, eventData.XboxLeftStickVerticalAxis * (-1) *
    move_scalefactor_drone, 0f);
      }
    if (eventData.XboxLeftBumper_Pressed){
      CallDronesAPI_rotate(0f, 0f, rotation_scalefactor);
    }else if (eventData.XboxRightBumper_Pressed){
      CallDronesAPI_rotate(0f, 0f, (-1) * rotation_scalefactor);
    }
    if (eventData.XboxB_Pressed){
      if (Time.time - first_buttonpressed > timeBetweenbuttonpressed){
        CallDronesAPI_rotate(0f, 0f, 0f);
      }
      first_buttonpressed = Time.time;
    }
    if (eventData.XboxX_Pressed){
      if (Time.time - first_buttonpressed > timeBetweenbuttonpressed){
        if (droneIP == "192.168.10.50"){
          session_drone.Close();
          session_drone.Destroy();
          droneIP = "192.168.10.51"
          session_drone = DroneSession.Create(tcpPrefix + droneIP + portSuffix);
        }else{
          session_drone.Close();
          session_drone.Destroy();
          droneIP = "192.168.10.50"
          session_drone = DroneSession.Create(tcpPrefix + droneIP + portSuffix);
        }
      }
      first_buttonpressed = Time.time;
    }
  }
 }
}
```

Source code 6: HoloLens sends message via MQTT broker.

```
using UnityEngine;
using UnityEngine.UI;
using MiniJSON;
using System;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using uPLibrary.Networking.M2Mqtt;
using uPLibrary.Networking.M2Mqtt.Messages;

namespace HoloToolkit.Unity.InputModule.Tests{
 public class XboxController : XboxControllerHandlerBase{
  [Header("Xbox Controller Test Options")]
  [SerializeField]
  private float movementSpeedMultiplier = 1f;

  [SerializeField]
  private float rotationSpeedMultiplier = 1f;

  [SerializeField]
  private XboxControllerMappingTypes resetButton = XboxControllerMappingTypes.
   XboxY;

  private Vector3 initialPosition;
  private Vector3 newPosition;
  private Vector3 newRotation;

  MqttClient client;
  string clientId;
  string topicPublishPath;
  string topicPublishPath_button;
  string topicSubscribePath;
  string BrokerAddress;
  private string msg;
  float first_buttonpressed = 0f;
  float timeBetweenbuttonpressed = 0.3f;

  public static int deviceIP { get; set; }; //192.168.10.48 - 51

  protected virtual void Start(){
   initialPosition = transform.position;

   //MQTT broker
   BrokerAddress = "192.168.10.73";
   clientId = Guid.NewGuid().ToString();
   client = new MqttClient(BrokerAddress);
   client.ProtocolVersion = MqttProtocolVersion.Version_3_1;
   topicPublishPath = "HoloLens/message/push";
   topicPublishPath_button = "Xbox/button";
   topicSubscribePath = "sub/HoloLens";

   try{
    client.Connect(clientId);
   }
   catch (Exception e){
      Debug.Log(string.Format("Exception has occurred in connecting to MQTT {0} ", e ));
      throw new Exception("Exception has occurred in connecting to MQTT", e.InnerException);
    }

    //Subscribe
    client.Subscribe(new string[] { topicSubscribePath }, new byte[] { 2 });
  }

  public override void OnXboxInputUpdate(XboxControllerEventData eventData){
    if (string.IsNullOrEmpty(GamePadName)){
      Debug.LogFormat("Joystick {0} with id: \"{1}\" Connected", eventData.GamePadName, eventData.SourceId);
    }

    base.OnXboxInputUpdate(eventData);

    /*get information from xbox controller*/

    if (eventData.XboxLeftStickHorizontalAxis != 0 || eventData.XboxLeftStickVerticalAxis != 0){
      msg = string.Format("{1} {0} 0 {2}", (-0.2) * eventData.XboxLeftStickHorizontalAxis, (-0.2) * eventData.XboxLeftStickVerticalAxis, deviceIP);
      client.Publish(topicPublishPath, Encoding.UTF8.GetBytes(msg), MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE, true);
    }
    if (eventData.XboxLeftBumper_Pressed){
      msg = string.Format("0 0 15 {0}", deviceIP);
      client.Publish(topicPublishPath, Encoding.UTF8.GetBytes(msg), MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE, true);
    }
    else if (eventData.XboxRightBumper_Pressed){
      msg = string.Format("0 0 -15 {0}", deviceIP);
      client.Publish(topicPublishPath, Encoding.UTF8.GetBytes(msg), MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE, true);
    }
    if (eventData.XboxB_Pressed){
      if (Time.time - first_buttonpressed > timeBetweenbuttonpressed){
        msg = string.Format("b");
        client.Publish(topicPublishPath_button, Encoding.UTF8.GetBytes(msg), MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE, true);
      }
      first_buttonpressed = Time.time;
    }
    if (eventData.XboxX_Pressed){
      if (Time.time - first_buttonpressed > timeBetweenbuttonpressed){
        if (deviceIP == 51){
          deviceIP = 48;
        }else{
          deviceIP++;
        }
      }
      first_buttonpressed = Time.time;
    }
  }
}
```

```
}
```

Source code 7: Pepper moves after getting message from MQTT broker.

```
import datetime
import os, sys

class MyClass(GeneratedClass):
  def __init__(self):
    GeneratedClass.__init__(self, False)
    self.framemanager = ALProxy("ALFrameManager")
    self.motion = ALProxy("ALMotion")
    self.positionErrorThresholdPos = 0.01
    self.positionErrorThresholdAng = 0.03
    self.memory = ALProxy("ALMemory")

  def onInput_onStart(self,p):
    labels = self.memory.getData("count")
    pcount = labels.count("person")
    self.logger.info("person count: "+ str(pcount))
    if argslist[3] == self.getParameter("robotIP"):
      import almath
      initPosition = almath.Pose2D(self.motion.getRobotPosition(True))
      targetDistance = almath.Pose2D(float(argslist[0]), float(argslist[1]), float(argslist[2]) *
       almath.PI / 180)
      expectedEndPosition = initPosition * targetDistance
      enableArms = True
      self.motion.setMoveArmsEnabled(enableArms, enableArms)
      self.motion.moveTo(float(argslist[0]), float(argslist[1]), float(argslist[2]) * almath.PI /
      180,[["MaxVelXY", speed]])
```