Computing (2023) 105:2773–2792
https://doi.org/10.1007/s00607-023-01204-7

REGULAR PAPER

Design, implementation and evaluation of a network-oriented service with environmental adaptability based on core/periphery structure

Shiori Takagi[1] · Shin'ichi Arakawa[1] · Masayuki Murata[1]

Received: 5 August 2021 / Accepted: 22 July 2023 / Published online: 10 August 2023

Abstract

Many new network-oriented services have been developed in recent years, and they are expected to be virtualized in multi-access edge computing (MEC) environments, which are being standardized along with fifth generation (5 G). Because many new network-oriented services have been developed to meet various user requests, and service-oriented development, wherein service functions are divided and combined, is expected to facilitate the development of flexible services at low costs. A core/periphery structure is an information processing system in biological systems consisting of core units, that is densely connected and provide efficient process, and peripheral units that can accommodate a variety of inputs and outputs. In this paper, we introduce a core/periphery structure into the service design since the service based on this structure can adapt to various inputs and outputs with only modifying peripheral functions. We expect that development cost is reduced by designing services based on core/periphery structure, because the entire service is not modified against environmental changes. Besides, we also consider the balance between the penalty and the reduction of development costs since dividing functions and placing them in different devices creates extra communication paths and degrades service responsiveness. We designed and implemented two service scenarios for our shopping service with a remote robot based on a core/periphery structure. Using the implemented services, we show that the design using the core/periphery structure is effective in terms of implementation cost and overhead for information sharing among remote robots. Furthermore, we measured the penalty through experiments on actual devices and showed that it is tolerable.

Keywords Core/periphery structure · Multi-access edge computing (MEC) · Telexistence, Mixed reality (MR) · Network robot

Mathematics Subject Classification 68U35 · 68M11

Extended author information available on the last page of the article

1 Introduction

In the recent years, many new network-oriented services have emerged, and information networks have been changing rapidly. For example, telexistence [25] services, which allow human users to experience being present in a remote place virtually as if it is their own body by operating a remote robot and using virtual reality (VR) or mixed reality (MR) technologies are currently developed. They use real-world information from cameras and sensors sent to the cloud, or conduct high-load processing such as image recognition, voice, and sound recognition. Several different services that are yet to be envisioned are expected to emerge in the future. Such services will need to be virtualized in the multi-access edge computing (MEC) [4, 10, 29] environment, which is currently being standardized along with fifth generation (5 G) technologies.

The service design should be adaptable to user requirements and environmental changes for accommodating a large number of services at low cost. When design decisions that are expedient in the short term, the costs of maintaining and adapting this system in future increase, which is known as technical debt [12]. To reduce the costs, module-based design has been widely introduced. In module-based design, modules connect each other via interfaces , and they are able to be decoupled [5] and reused for several products.

The disadvantage of module-based design is that the development becomes complex and difficult to maintain the products. Albers et. al. [2] pointed out that the interdependencies among modules used in different products increase because the change of the module affects other products. As a result, the module's development becomes complex, and it becomes difficult to maintain the module and products. Although Albers et al. [2] explains about vehicle development, a module-based design of service also complicates the development. Modules are connected on equal basis and have interdependencies. More importantly, modules, in our case service functions, are sometimes developed by several different developers. Thus, it is difficult for developers to observe the scope of effect when they modify their modules. This will lead to the difficulties of maintenance and modification of services and/or modules.

We investigated a core/periphery structure [6, 15] that allows service components to effectively adapt to each user request and environmental variation. The core/periphery structure is a model for flexible and efficient information-processing mechanisms in biological systems; the information processing units in the core/periphery structure are classified as core or peripheral units. The advantage of the core/periphery structure is that it helps reduce the costs for maintaining or changing services by distinguishing the service functions into core and periphery functions. Unlike module-based design, a service design based on the core/periphery structure can adapt to environmental changes while modifying only peripheral functions and reusing the core functions as shown in Fig. 1.

The advantages of a core/periphery structure for accommodating information services, represented by chains of functions, were numerically investigated in our previous work [30]. From this biological point of view, the core/periphery

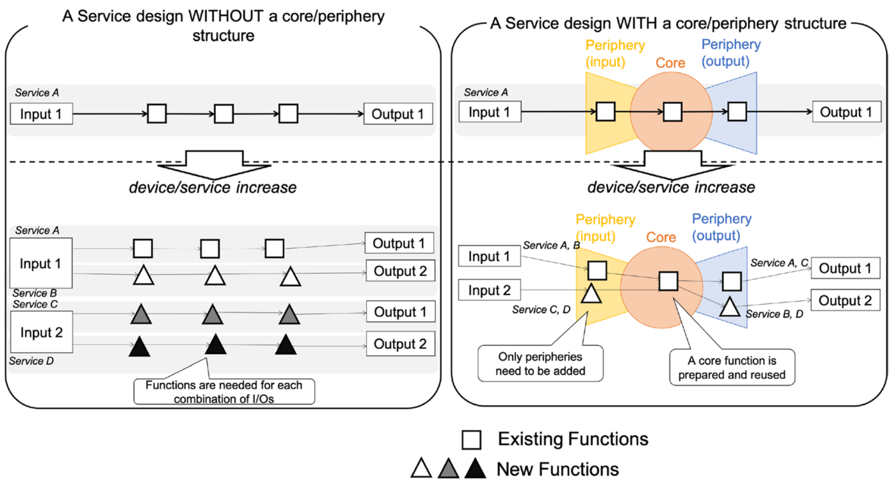

Fig. 1 Effectiveness of service design using core/periphery structure. Consider adding Input 2 and Output 2 to a service that has Input 1 and Output 1. Fewer additional functions are required to support various inputs and outputs when core functions are provided

structure is expected to achieve an efficient and adaptive behavior against environmental changes. Figure 1 indicates that service design based on a core/periphery structure can support variations in inputs and outputs with fewer additional implementations. However, from the service-design perspective, dividing the functions and placing them in different devices creates extra communication paths and can degrade service responsiveness, which is cannot be ignored. Therefore, when we apply the core/periphery structure to the service design, we consider the balance between the penalty and the reduction of development costs.

In [27, 28], we implemented a shopping experience service using mixed reality (MR) and robots as a use case for realizing a service scenario based on [30], and we implemented the service with an actual device, and demonstrated that the service design using a core/periphery structure is effective for robot operation when the numbers of types of devices on the user side and remote side increase.

In this paper, we perform evaluation in the following two aspects more pragmatically than previous works. The first aspect is the service scenario to use in our experiment. In our previous works [30] and [27, 28], we considered a service scenario that included only information processing; however, commonly used applications today not only process information obtained from devices, they also share information among such devices. Therefore, we focus on the information sharing in this paper. We consider a service scenario that includes information processing and information sharing among remote robots and users, and we evaluate our service design in terms of the complexity of the source code and the overhead for information sharing. We implement a service and measure the penalty through experiments on actual devices for investigating the amount of penalties on sharing information. The second aspect is the metric to represent the implementation cost. In our previous works [30] and [27, 28], we used the number of lines of source code for the user-side applications as the implementation cost. The number of lines can be used to evaluate the effort required to adapt the service to the environment; however, it cannot help evaluate the extent to which the logic of the application is simplified because the amount of source code only represents the implementation results of efforts. Therefore, we introduce the complexity of the program as a factor in the cost of adapting to environment because the complexity is especially important when multiple people develop the service, i.e., in a modern software development. We use the cyclomatic complexity [13], which represents the number of independent paths from the start to the end of the program as the metric for evaluating the implementation cost.

The remainder of this paper is organized as follows. Section 2 describes the related works. Section 3 describes the services targeted in this paper as well as the service scenarios. Section 4 describes our service design based on a core/periphery structure and service implementation. Section 5 describes the evaluation of our service design. Finally, Sect. 6 provides some concluding remarks and areas of future study.

2 Related work

We present network-oriented services and architectures as the related work.

Existing conventional MR services, such as ANA Avatar [3], are not designed for supporting various inputs and outputs because it is provided by a single vendor; however, it is believed that a modular division of services is in place.

These functions are distributed across various devices when functions created by multiple vendors are combined to form a service. Such a distribution is accelerated by a multi-access edge computing (MEC) [10, 22]. The ETSI Industry Specification Group suggests video content delivery, video stream analysis, and augmented reality (AR) as key use cases for MEC, and it also suggests guidelines for software developers. The virtual-reality peripheral network (VRPN) [11], which is used for developing VR services, is similar to the concept of core/periphery structure because it can absorb the differences between VR devices. VRPN can be used as a core function when creating services that support various inputs and outputs. In this paper, we use message queuing telemetry transport (MQTT) [16], which is a messaging protocol that can absorb the differences between devices as a core function.

In recent years, the concept of dividing and combining service functions such as microservices [26] and service-oriented architecture (SOA) [20], have been proposed. In these architectures, each function is loosely connected and adaptable to changing demands; however, the development costs are higher for supporting various inputs/outputs because the functions are finely divided. In this paper, these architectures are referred to as "No Core" because they are considered as service structures with only peripheral functions.

3 Service scenario

3.1 Basic service

Our basic service is a shopping experience that uses a robot in a remote location. The remote robot captures images of a store or streets lined with stores, and it provides a processed video to the user using an object detection function. The user operates the robot while watching the video and the information sent from a remote location.

3.2 Additional functions to improve the service

We consider adding the following service scenarios to improve this basic service by adapting it to a real environment:

– A service scenario that adjusts the movement of the robot based on where it is walking to avoid collisions in crowded or small areas, and
– a service that obtains information based on the attention level of the user.

Our services and functions are illustrated in Fig. 2. These services scenarios require the functions to conduct object detection, store and share information using the results of the object detection, and adjust the manner in which the robot moves. These functions are combined when realizing such services.

3.3 Service design without a core/periphery structure

In designs without a core/periphery structure, the specifications of each service function are specific to a particular environment, for example, the API for a

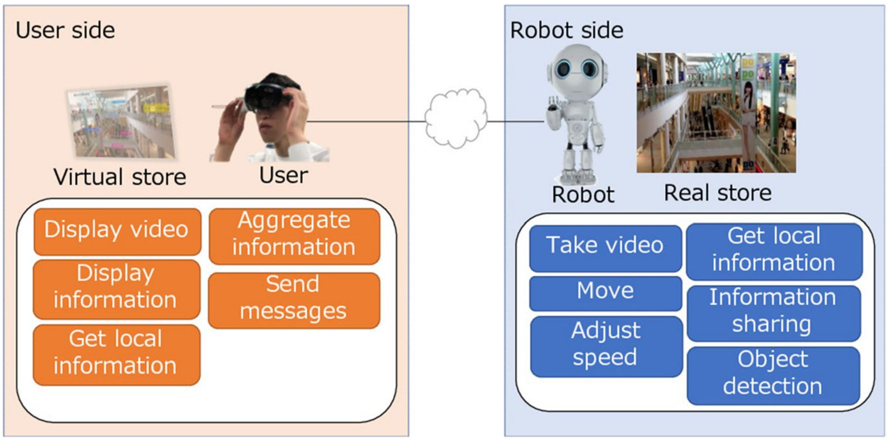

Fig. 2 Our service with added functions

particular robot. The robot is accessed directly from the user devices when implementing a design without a core/periphery structure for robot operation; the program is changed to adapt to the environment. In information sharing, each device stores information in a different format because the method of storing information is not unified, and it is therefore difficult to store information collected from a variety of robots and provide information to the users.

4 Service design and implementation

4.1 Design scenario

We evaluated the effectiveness of the service design using a core/periphery structure under the following three design scenarios.

No Core: All functions are peripheral and specific to each device.
Core on Robot: Common functions are implemented as core functions on the robots.
Core on Edge Server: Common functions are implemented as core functions on edge servers.

As we described in Sect. 2, we there call the structures such as microservices [26] and service-oriented architecture (SOA) [20] “No Core” because they are considered as service structures with only peripheral functions.

A monolithic structure wherein all functions are tightly coupled is another candidate for the design scenarios. Thus we interpret the monolithic design as a design consisting only of core functions. Monolithic designs are more difficult to maintain and scale compared to designs with microservices because the monolithic services are larger and more complex than microservices [26]. This paper does not compare with the monolithic design because its development cost is greater than that of a structure consisting of only peripheral functions.

4.2 Service design based on core/periphery structure

We designed our service scenarios in Sect. 3.2 based on a core/periphery structure. In a service design without a core/periphery structure, all functions are implemented as peripheral functions, and are implemented specifically for each service scenario or real environment; this makes it difficult to add more functions or change the combination of functions. In a service design with a core/periphery structure, we implemented common functions such as messaging from the user to the robot, object detection for the video, storage of the object detection results as core functions, and connect other functions peripherally to the core functions. Therefore,

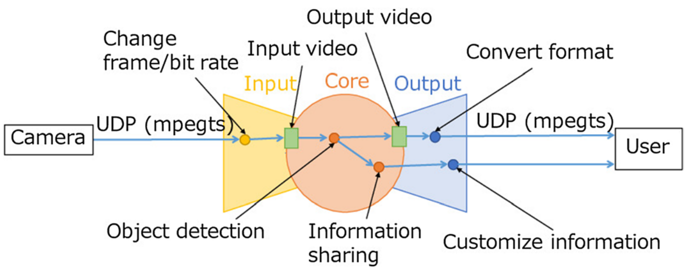

Fig. 3 Core/periphery structure for video processing and information storage

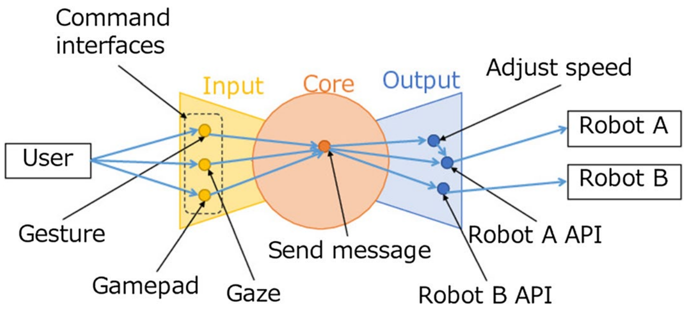

Fig. 4 Core/periphery structure for robot operation

when environmental changes occur, only peripheral functions require to be changed within a short period of time.

Figure 3 and 4 show the core/periphery structure based on our service design. The dots represent service functions, and the arrows represent service function connections. The service functions are used in the order of connection from the input side (the camera and the user in Figs. 3 and 4, respectively) toward the output side (the user and the robots in Figs. 3 and 4, respectively). The green rectangles in Fig. 3 represent the interfaces for video input and output. Figure 3 shows the core/periphery structure of the service functions and their connections for processing the video captured by a camera, information storage and providing users with the processed video and the customized information. For video processing and information storage, the object detection function is commonly used in our service scenario, and therefore, both it and the tightly connected information storing function are considered core components; the peripheral functions provide how the information is used, as described in 4.3.3 and 4.3.4. Figure 4 shows the core/periphery structure based on our service design for robot operation. For the robot operation, the function to send user commands to the robot is a common function. Thus it is considered as a core function; and the peripheral function adjusts the speed of the robot and processes the messages based on each API.

4.3 Implementation

4.3.1 Object detection and information sharing (core function)

The video input from the cameras are transmitted to the robot-side edge server, which are then captured using OpenCV [18]; subsequently, object detection is then applied using a PyTorch implementation of YOLO v3 [21]. The processed video is transmitted with FFmpeg [9] through the UDP to the HoloLens [14] worn by the users and displayed. HoloLens is a standalone head-mounted computer developed by Microsoft that displays holograms and recognizes user gaze and gestures to provide an MR experience. From the results obtained by the object detection function, the type, coordinates, and shooting time of the products sold in the store is remote information that is stored as shared information among the robots.

4.3.2 Messaging (core function)

The controller information is transmitted through MQTT [16], which is a publish /subscribe-type protocol developed for frequent message exchanges between IoT devices. The MQTT broker receives controller commands through the HoloLens and sends them to a program running on the Pepper robot [24].

The users use Xbox controllers that can connect to HoloLens. We develop an MQTT messaging system on a user-side edge server using Mosquitto [7], an open-source message broker, and Node-RED [19], a programming tool for event-driven applications.

4.3.3 Speed adjustment (peripheral function)

The basic service provides a function to move at a constant speed. We add a function to adjust the speed of the robot to change in the way the robot moves according to the surrounding environment; we ensure that the robot does not bump into people or obstacles by using the information of the surroundings obtained via the object detection function. The results of object detection are returned from the edge server to the robot, which uses this information to adjust its movement speed (e.g., slowing down in crowded areas). The core function is to perform object detection on the images sent from the robot and to return the results; the peripheral function involves reducing the speed when there are many people in the area. The peripheral function is used to process the results, and therefore, the robot can choose to avoid obstacles besides people, based on its location. The edge server stores a list of objects and their sizes, and the robot refers to only the information of the object to be avoided.

4.3.4 Displaying information to users (peripheral function)

The basic service displays information about an object using object detection with a learned model. We add a function to detect gaze and display information using a database of products to display detailed information about the surrounding objects based on the user's attention. A database that integrates the product list of each store is prepared, and the region of interest (i.e., the object that the user is gazing at in the video) is cut out and enlarged. Further, a recommendation is made for a product of the same type, which the user is yet to see. The peripheral function customizes the information for each user by selecting the necessary information from the information stored in the core function, or by deleting the object that the user has already gazed at.

5 Evaluation

We evaluated the effectiveness of the service design using a core/periphery structure in terms of the implementation cost and overhead for information sharing by using the three design scenarios described in 4.1.

5.1 Implementation cost

We evaluate the implementation cost for increasing the number of device types. In our service, where n types of remote robots and m types of devices are connected on the user side.

In [27, 28], we have measured the implementation cost of the core and periphery functions based on the amount of source code. However, the amount of source code does not fully capture the effort required to adapt to the environment because it only represents the implementation results of the efforts. Thus, we introduce the complexity of the program as a factor in the cost of adapting to the environment; the complexity is especially important when multiple people develop the service, such as in modern software development.

We assume that all implementations under the No Core scenario are written in the program of the user-side device, according to the implementation of the program that directly connects the HoloLens MR headset with the Pepper robot. Therefore, the API program for each robot is modified when modifying the program to control other robots. When users have $m + 1$ types of devices, the process for n types of robots for the $m + 1$th device is provided. In addition, the process for the $n + 1$th robot in each program of the m types of user devices is added when the number of robot types increases by 1. The more we add to the program, the more complex the program becomes, and, the longer it takes to read and write to add the source code. Thus, the implementation cost is considered to not only be based on the amount of new source code that has to be written, but also based on the complexity of the program that we have to read when we modify the program. Therefore, we define the cost of reading a program as the implementation cost, and we use the cyclomatic complexity [13] to evaluate the implementation cost. Cyclomatic complexity represents the number of independent paths from the start to the end of the program. The higher the number of branches, the higher is the value, and the harder it is to read. Figure 5 shows cyclomatic complexity of application source

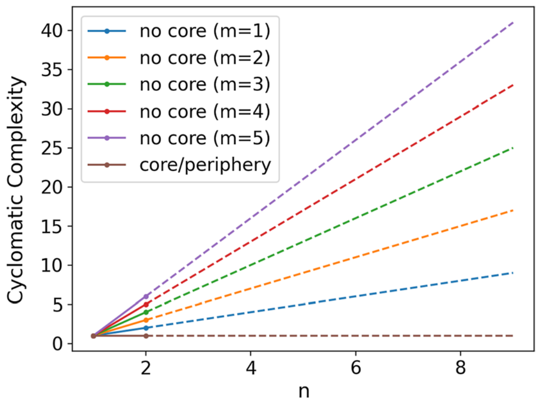

Fig. 5 Cyclomatic complexity of application source code when the number of robots n is increased

code when the number of robots n is increased. The horizontal axis represents the number of robot types and the vertical axis represents the cyclical complexity.

We show excerpts of the source code used to establish connections with robots. Source Code 1 shows a part of the source code of the HoloLens application under the No Core scenario. This example represents the case where $n = 2$ and $m = 1$. All processes for all robots' APIs are written within a single program to access each robot directly from the HoloLens application. The second and ninth lines represent branches based on the environment (in this case, the type of device), and they become more complex with an increase in n and m.

Source Code 1: Establish connection to robots.

```
 1  //Pepper
 2  if(!string.IsNullOrEmpty(pepperIP)){
 3     _session = QiSession.Create(tcpPrefix + pepperIP + portSuffix);
 4     if (!_session.IsConnected){
 5        Debug.Log("Failed␣to␣establish␣connection");
 6        return;
 7     }
 8  //Another Robot
 9  }else if(!string.IsNullOrEmpty(RobotIP)){
10     session_robot = RobotSession.Create(tcpPrefix + RobotIP + portSuffix);
11     if (!_session.IsConnected){
12        Debug.Log("Failed␣to␣establish␣connection");
13     return;
14     }
15  }
```

Source Code 2 shows a part of the source code of the HoloLens application, and Source Code 3 shows a part of the source code of the Pepper application with a core/periphery structure. The messaging function with MQTT is provided as a core function, and therefore, only the process for connecting to the MQTT broker from each device is required.

Source Code 2: Establish connection from User to MQTT broker.

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

Source Code 3: Establish connection from Robot to MQTT broker.

```
mqttc = mqtt.Client()
mqttc.Connect(brokerIP, portSuffix, keepalive)
```

Under the No Core scenario, the entire source code for accessing the robots is written into the HoloLens program, and therefore, the more robots are used, the higher is the number of branches added into the program. Thus, the cyclomatic complexity for the entire service becomes $O(m \times n)$. Under the Core on Robot/Edge Server scenarios, the value does not change and remains at 1 because the program has no branch. Therefore, the difference in the cyclomatic complexity between the No Core and Core on Robot/Edge Server design scenarios becomes more significant with an increase in the number of robot types.

5.2 Overhead for information sharing

We measure the number of messages sent for information sharing in the three design scenarios assuming that we keep the information stored by the information sharing function up-to-date and that robots R_1, R_2,..., R_n share information with each other. Further, we show that the number of messages is the lowest when the system is implemented as a core function and deployed on an edge server.

Figure 6 shows the configuration of the experimental environment in our laboratory. We constructed the MEC environment using OpenStack version 3.8.1 [17]. The user-side edge server is an OpenStack virtual machine (192.168.10.73), and the robot-side edge server is a physical machine (192.168.10.39). We built MQTT brokers on the edge servers with Mosquitto version 1.4.15, which is an MQTT version 3.1.1/3.1 broker [7]. Messages sent from the user to the robot, e.g., controller operation or gaze operation, are sent through the MQTT broker on the user side. The MQTT server on the robot side is used to send information from the edge server to the robot and from the robot to the edge server. The robot Pepper (192.168.10.51) is connected to the MEC environment. Pepper has an embedded camera with 320 × 240 resolution, however, it is too low to enjoy the video

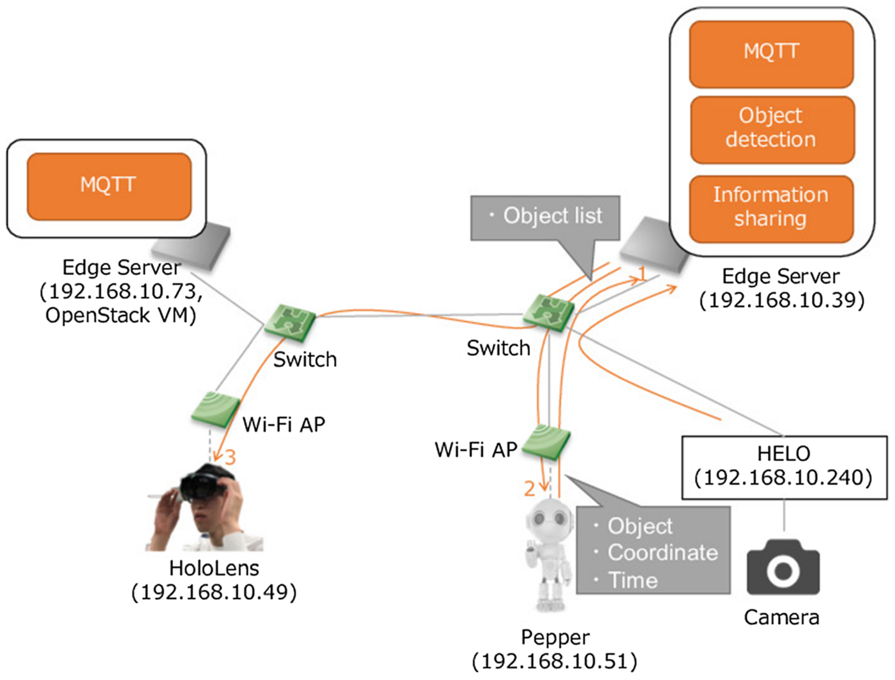

Fig. 6 Experimental environment for measuring the messages to perform information sharing. The numbers correspond to each phase

streaming at the HoloLens. Therefore, in this experiment, we attached an external camera on the head of the Pepper, and this camera is connected to Aja HELO [1] via an SDI cable to perform the H.264 encoding for the video streaming. The video from the attached camera is input to the HELO at 60 fps, 1920 × 1280, and output at 30 fps, 1080 × 720. The video encoded by HELO is sent to the robot side edge server using UDP in the mpegts format. Once the processing is completed, the video is output to standard output as raw data, and then, it is encoded into mpegts using FFmpeg, and sent to the HoloLens worn by the user.

The messages in our experiment are the results of object detection obtained from the captured video. Therefore, we measured the number of messages using the video of a certain length and calculated the average overhead per frame. Then, we divided the process, from the time the new information was acquired until it was reflected to the user, into the following three phases; subsequently, we measured the number of messages during each phase.

1. The object detection function is executed and each robot obtains new object information. This information is a list of objects seen by the camera used in our application. This phase is not applied under the No core and Core on Robot scenarios because the robot performs the object recognition function directly. The edge server conducts the object detection function and sends the list to the robot

via MQTT for integration with the information held by the robot under the Core on Edge Server scenario.

2. Information is sent from the robot to the information sharing function. The robot Pepper adds the coordinates and time information obtained by the robot to the object information. The robot sends information to all other robots under the No Core and Core on Robot scenarios. The robot sends information to the edge server via MQTT under the Core on Edge Server scenario.
3. Information is provided from the information sharing function to the user when the user gazes at a specific object. The robot sends information to the user under the No Core and Core on Robot scenarios; the edge server sends information to the user under the Core on Edge Server scenario.

The information-sharing function is implemented in a specific manner under the No Core scenario based on the type of robot applied. Each robot shares information through flooding when there is an update in the surrounding information held by the robot to ensure that the information stored in the information-sharing function on all robots is up to date. Because each robot, R_1, R_2,..., R_n, sends information to $n - 1$ robots other than itself, $O(n^2)$ messages are sent for information sharing. Information-sharing functions are implemented as periphery functions, and the method of storing information for each robot is not unified; therefore, it is necessary to break down and transmit each type of information, such as the name and coordinates of the object. Therefore, $O(k \times n^2)$ messages are sent, where k represents the number of information types.

Under the Core on Robot scenario, multiple types of information can be shared in a single flooding because the information-sharing function is unified as a core function. We assume that the information to be stored in the information sharing function of all robots needs to be up to date in the same way as that under the No Core scenario, and therefore, we assume that $O(n^2)$ messages are sent since each n robot sends information to the $n - 1$ robots other than itself.

Figure 7 shows the change in the number of messages for information sharing when the number of robots n is increased. We measured the number of messages for $n = 1$ and calculated the number of messages for each frame, which is the

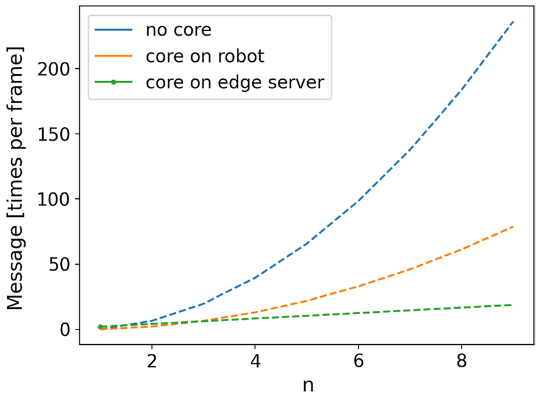

Fig. 7 Number of messages for information sharing when the number of robots n is increased

coefficient. For Phase 1, overhead occurs only in the Core on Edge Server scenario.. The method of storing and sharing information is unified because the information-sharing function is implemented as a core function; all information obtained from one frame can be sent to the robot concurrently. Therefore, one message per frame is applied. For Phase 2 (Core on Edge Server scenario), we measured the number of messages using a video 26.36 s long video (790 frames) captured at 29.97 fps, and we revealed that the edge server sent 854 messages. Thus, 1.08 messages were sent every time the information was updated(i.e, at every frame). For Phase 3, 11 gazes occurred in the same video compared to that used in the Phase 2 experiment. Because a recommendation is sent once for each gaze, the rate is approximately 0.014 times per frame. These results indicate that the number of messages in the Core on Edge Server scenario is $2.094 \times n$ times per frame.

When extrapolating the results of our experiment shown in Fig. 7, the messages occur at a rate of $k \times 1.094 \times n \times (n - 1)$ times per frame (where k represents the number of information types, which in this implementation is three) under the No Core scenario, and at $1.094 \times n \times (n - 1)$ times per frame under the Core on Robot scenario. For a single robot, the number of messages under the Core on Edge Server scenario is the largest. However, as n increases, the number of messages under the No Core and Core on Robot scenarios increases, and the overhead is considered to become larger than the number of messages under the Core on Edge Server scenario.

5.3 Penalty in the core on edge server scenario

Dividing functions and placing them in different devices creates extra communication paths and a penalty of service responsiveness. We conducted another experiment to investigate this penalty; in this experiment, the penalty of communicating through an edge server when robots share information is calculated.

In this experiment, we evaluate the application-level delay using the service scenario of information sharing among robots. Figure 8 shows our experimental environment; the robot NAO (192.168.10.44) [23] is connected as an additional robot. The application-level delay in this service is the delay between the robot obtaining information about a new object, storing the information in the edge server, and providing the information based on the attention of the user. Among these delays, the penalty for using the edge server to provide information to the user via is about the same as that measured in [27] because the extra path in the Core on Edge Server scenario is between the robot-side switch and the robot-side edge server. Therefore, we measured time as an application-level delay that indicates the time required for the information obtained by the object detection function to be reflected in the information sharing function under the Core on Edge Server scenario; this is represented by orange arrows in Fig. 6. Further, we divided the service process into five phases to understand the application-level delay more clearly. Figure 9 shows the five phases, starting from the time the new information is acquired at the edge server, and ending with the time the information is stored in the edge server. The five phases are listed below:

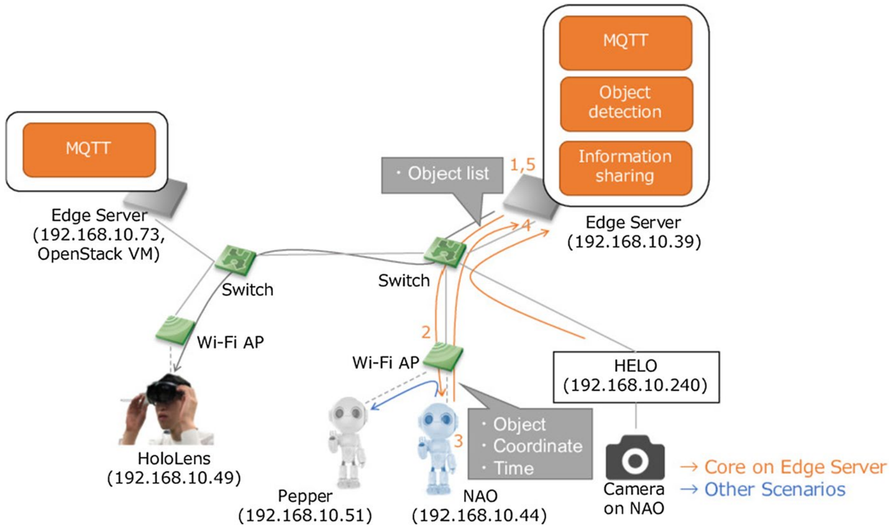

Fig. 8 Experimental environment to measure the penalty attributed to the extra communication path. The numbers correspond to each phase

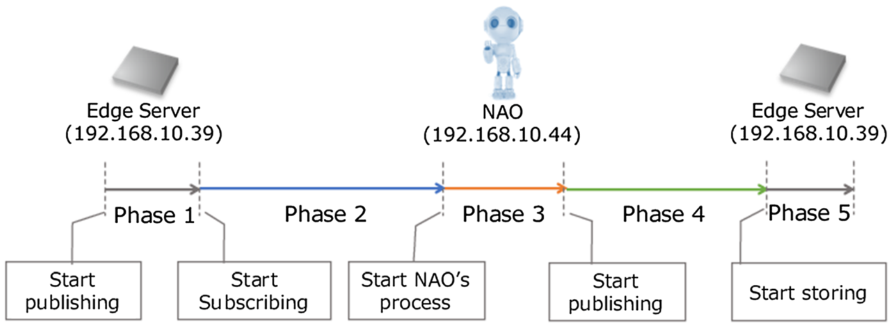

Fig. 9 Each phase for measuring the the delay

1. From the time the object detection function completes execution to the time it completes publishing to the MQTT broker under the Core on Edge Server scenario. Under other design scenarios, this phase is not applied because the robot execute the object detection function.
2. Until Pepper/NAO receives the information obtained by the object detection function. The information is a list of objects seen by the camera used in our application. Under other design scenarios, this phase is not applied because the robot execute the object detection function. Under the Core on Edge Server scenario, the edge server conducts the object detection function and then sends the list to the robot via MQTT for integration with the coordinate and time information held by the robot.

3. Until Pepper/NAO adds the coordinates and time information obtained by the robot to the object information. This is a common process for all design scenarios.
4. Until the object information that is added by the robot is received by the information sharing function. Under the No Core and Core on Robot scenarios, the robot sends information to all other robots. Under the Core on Edge Server scenario, the robot sends information to the edge server via MQTT.
5. Until the information sent from Pepper/NAO is stored. This is a common process for all design scenarios.

The sum of the time taken for the five phases represents the delay taken in the Core on Edge Server scenario. We measure the time it takes for an object detection function in the edge server to complete publishing a list of objects to the MQTT broker in the same edge server. Phases 1, 2, and 4 require extra processing compared to other design scenarios. Therefore, the time taken in these phases considered a penalty in the Core on Edge Server scenario. Phase 2 represents Pepper/NAO's MQTT subscription process. Since there are different system clocks among edge servers and robots, it is difficult to accurately measure time in Phase 2. Thus, we calculate the difference between the total time taken in Phases 1 through 5 and the time taken in Phases 1, 2, 3, and 5, and we measure the time taken in Phase 2. For Phase 4, we measure the time taken for Pepper/NAO to complete publishing each information including the object, coordinate, and time, to the MQTT broker on the edge server. We measure the total time for all phases by recording the time when the edge server executes the object detection function and the time when the edge server receives the message with the coordinate and time information added by NAO for each video frame. In our experiment, we used 70 frames of video stored in the edge server; in this video, 1024 objects are detected in total.

5.3.1 Result

We show the result of our experiment using the robot NAO. Table 1 presents the average, maximum, and minimum values for total time, the time for each phase and the penalty by time incurred under the Core on Edge Server scenario. The total measured time, 104 ms on average, is the application-level delay for

Table 1 Time it taken for each phases

	Avg [ms]	Max [ms]	Min [ms]
Total	104	496	10
Phase 1	0	0	0
Phase 2	76	430	1
Phase 3	4	23	1
Phase 4	24	106	3
Phase 5	0	0	0
Sum of Phases 1, 2 and 4	99	536	4

information sharing among robots. The penalty under the Core on Edge Server scenario is 99 ms on average, and it can reach up to 536 ms. The application-level delay penalty in the users' operation of the robot is 31 ms on average [27]. In [27], the extra path is between the edge server and switch at the user side because the penalty is for user-robot communication. In this experiment, the extra communication path is between the robot and the robot-side edge server; it is three times the number of hops, and has a larger penalty. The time taken for Phases 1 and 5 was 0 ms, because the functions performed in these phases communicate with the MQTT broker on the same edge server without communication delay. The delay of Phase 3 changes drastically because the robot does not have a good CPU and does process our information processing in addition to the fundamental control of the robot head and arms. Further, the robot sequentially adds and sends information for every item in the list of objects; the more objects there are in one frame, or the longer it takes to send other objects, the larger is the maximum value. This is because the waiting time required to send the object at the end of the list becomes large. Thus, we evaluate the penalty attributed to the extra communication path for Phases 2 and 4 compared to 4 ms, which is the average time taken for Phase 3.

Figure 10 shows the time for Phases 2, 3, and 4 under the Core on Edge Server scenario compared to the average time for Phase 3 only. The average communication delay is 99 ms which occupies about 95% of the application-level delay and is about 25 times longer than the delay for Phase 3. Wi-Fi communication is used in Phases 2 and 4, and therefore, the delay varies due to congestion, obstacles, and the distance from the access point. This result indicates that the penalty is tolerable because the interaction delay tolerance is 100 ms [8]. The latency of

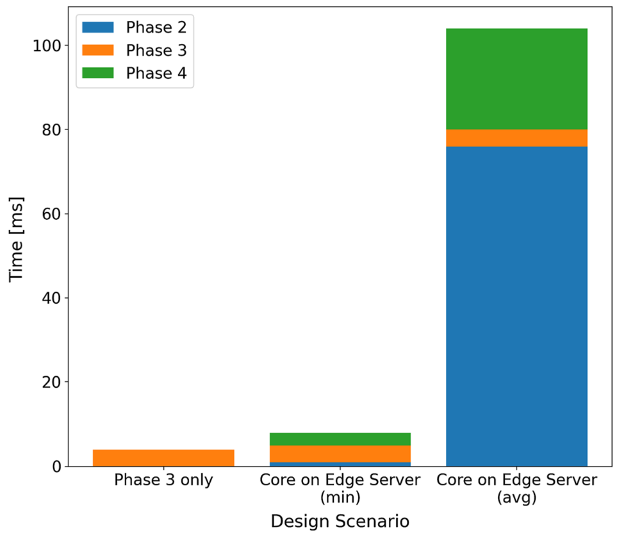

Fig. 10 Minimum and average time for the Core on Edge Server scenario

wireless communication is expected to improve with the ultralow latency offered by 5 G.

5.4 Hierarchical core/periphery structure

In the implementation discussed in this paper, communication is applied only between the periphery and core functions; however, in actual services, communication may be achieved between the core functions on the edge servers. For example, the core function on the edge servers can aggregate the information held by neighboring robots, which can communicate with each other to share information over a wide area. In this case, we can consider the information collection function at each edge server as a periphery function, and the information-sharing function among edge servers as a core function. Therefore, we can find a hierarchical core/periphery structure in a large-scale service configuration where multiple edge servers exist. Considering the same scenario as illustrated in Fig. 7, it is is effective to place the core functions in the edge servers of a higher hierarchy when aggregating the information of four or more edge servers. However, this figure assumes that the information is exchanged every frame. In a real service, information sharing among edge servers is infrequent, and the edge servers for the core functions of higher hierarchy are only used when information is shared among an extremely large number of edge servers.

6 Conclusion

We introduced a core/periphery structure to actualize a flexible and adaptive service composition in an MEC environment, and we designed and implemented a network-oriented service based on this structure. To this end, we designed and implemented multiple service scenarios and evaluated them using cyclomatic complexity and overhead for information sharing. We demonstrated that the source code did not become complex when we added functions to access different devices using the core/periphery structure; further, we measured the penalty through experiments on actual devices and showed that it is tolerable.

Although this paper focused on a shopping service, a service design based on a core/periphery structure is not limited to shopping and can be applied to other network services.

In a future study, we will consider a service design that can adapt to larger environmental changes, such as moving to another location in the real world. Specifically, we design a service that can adapt to larger environmental changes by reconstructing both the core and peripheral functions.

Acknowledgements This work was partly supported by the National Institute of Information and Communications Technology (NICT) in Japan. The first author is supported by Di-CHiLD (Daikin Information Science Research Unit) at Osaka University, funded by Daikin Industries, Ltd. in Japan.

Funding Open access funding provided by Osaka University.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. aja (2021) HELO. https://www.aja-jp.com/products/helo Accessed 13 July 2021
2. Albers A, Bursac N, Scherer H et al (2019) Model-based systems engineering in modular design. Design Sci 5:1–33. https://doi.org/10.1017/dsj.2019.15
3. ANA (2018) ANA Avatar. https://ana-avatar.com, Accessed 12 November 2021
4. Baktir AC, Ozgovde A, Ersoy C (2017) How can edge computing benefit from software-defined networking: a survey, use cases, and future directions. IEEE Commun Surv Tutor 19(4):2359–2391. https://doi.org/10.1109/COMST.2017.2717482
5. Baldwin CY, Clark KB (1997) Managing in an age of modularity. Harvard Bus Rev 75:84–93
6. Csermely P, London A, Wu LY et al (2013) Structure and dynamics of core-periphery networks. J Complex Netw 1:93–123. https://doi.org/10.1093/comnet/cnt016
7. Eclipse Foundation (2021) Eclipse Mosquitto. https://mosquitto.org Accessed 13 July 2021
8. ETSI (2021) 5G; Extended Reality (XR) in 5G. 3GPP TR 26928 version 1610 Release 16
9. FFmpeg (2021) FFmpeg. https://www.ffmpeg.org/ Accessed on 31 July 2021
10. Hu YC, Patel M, Sabella D, et al (2015) Mobile edge computing a key technology towards 5G. ETSI White Paper (11)
11. Ii R, Hudson T, Seeger A, et al (2001) VRPN: A device-independent, network-transparent VR peripheral system. In: Proceedings of the ACM symposium on virtual reality software and technology, pp 55–61. 10.1145/505008.505019
12. MacCormack A, Sturtevant DJ (2016) Technical debt and system architecture: the impact of coupling on defect-related activity. J Syst Softw 1(120):170–182. https://doi.org/10.1016/j.jss.2016.06.007
13. McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng SE-2 4:308–320. https://doi.org/10.1109/TSE.1976.233837
14. Microsoft (2021) Microsoft HoloLens. https://www.microsoft.com/ja-jp/hololens Accessed 13 July 2021
15. Miele V, Ramos-Jiliberto R, Vazquez DP (2019) Core-periphery dynamics in a plant-pollinator network. bioRxiv 10.1101/543637
16. MQTT org (2021) MQTT. https://mqtt.org Accessed 13 July 2021
17. Open source cloud computing infrastructure - OpenStack (2021) OpenStack. https://www.openstack.org/ Accessed 13 July 2021
18. OpenCVteam (2021) OpenCV. https://opencv.org Accessed 13 July 2021
19. OpenJS Foundation (2021) Node-RED. https://nodered.org Accessed 13 July 2021
20. Papazoglou M, van den Heuvel W (2007) Service oriented architectures: approaches, technologies and research issues. VLDB J 16:389–415
21. Redmon J, Farhadi A (2018) YOLOv3: An incremental improvement. CoRR abs/1804.02767. http://arxiv.org/abs/1804.02767
22. Sabella D, Sukhomlinov V, Trang L, et al (2019) Developing software for multi-access edge computing. ETSI White Paper (20)

23. Softbank Robotics (2021a) NAO the versatile humanoid robot. https://www.softbankrobotics.com/emea/sites/default/files/press-kit/NAO-press-kit-EN.pdf Accessed 31 July 2021
24. Softbank Robotics (2021b) Pepper press kit. https://www.softbankrobotics.com/emea/sites/default/files/press-kit/Pepper-press-kit_0.pdf Accessed 13 July 2021
25. Tachi S (2016) Telexistence: enabling humans to be virtually ubiquitous. IEEE Comput Graph Appl 36(1):8–14
26. Taibi D, Lenarduzzi V, Pahl C (2017) Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation. IEEE Cloud Comput 4(5):22–32. https://doi.org/10.1109/MCC.2017.4250931
27. Takagi S, Arakawa S, Murata M (2020) Design, implementation and evaluation of core/periphery-based network-oriented mixed reality services. J Int Serv Appl 13(1):1–10
28. Takagi S, Arakawa S, Murata M (2020b) On the implementation and evaluation of a network-oriented mixed reality service based on core/periphery structure. In: IEICE technical report (NS2019-218) pp 221–226
29. Taleb T, Samdanis K, Mada B et al (2017) On multi-access edge computing: a survey of the emerging 5G network edge cloud architecture and orchestration. IEEE Commun Surv Tutor 19(3):1657–1681. https://doi.org/10.1109/COMST.2017.2705720
30. Tsukui Y, Arakawa S, Takagi S et al (2020) Design and placements of virtualized network functions for dynamically changing service requests based on a core/periphery structure. IEEE Access 8:166294–166303

Authors and Affiliations

Shiori Takagi[1] · **Shin'ichi Arakawa**[1] · **Masayuki Murata**[1]

✉ Shiori Takagi
s-takagi@ist.osaka-u.ac.jp

Shin'ichi Arakawa
arakawa@ist.osaka-u.ac.jp

Masayuki Murata
murata@ist.osaka-u.ac.jp

[1] Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka, Japan