Adaptable and Evolvable Design of Network-oriented Services based on Core/Periphery Structure

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2023

Shiori TAKAGI

List of publication

Journal papers

1. Yuki Tsukui, Shin'ichi Arakawa, Shiori Takagi, and Masayuki Murata, "Design and placements of virtualized network functions for dynamically changing service requests based on a core/periphery structure," *IEEE Access*, vol. 8, pp. 166294-166303, September 2020.

2. Shiori Takagi, Shin'ichi Arakawa, and Masayuki Murata, "Design, implementation and evaluation of core/periphery-based network-oriented mixed reality services," *Journal of Internet Services and Applications*, pp.1-10, February 2022.

Refereed Conference Papers

1. Shiori Takagi, Junichi Kaneda, Shin'ichi Arakawa, and Masayuki Murata, "Improvement of Service Qualities by Edge Computing in Network-oriented Mixed Reality Application," in Proceedings of *6th IEEE International Conference on Control, Decision and Information Technologies (CoDIT' 19)*, pp.773-778, April 2019.

Non-Refereed Technical Papers

1. Shiori Takagi, Junichi Kaneda, Shin'ichi Arakawa, and Masayuki Murata, "On the Improvement of Service Quality by Edge Computing in Network-oriented Mixed Reality Application," *Technical Report of IEICE (NS2018-86)*, vol. 118, pp. 253-258, July 2018.

2. Shiori Takagi, Shin'ichi Arakawa, and Masayuki Murata, "On the Implementation and Evaluation of a Network-oriented Mixed Reality Service based on Core/Periphery Structure," *Technical Report of IEICE (NS2019-218)*, vol. 119, pp.221-226, March 2020.

3. Shiori Takagi, Shin'ichi Arakawa, and Masayuki Murata, "Implementation and Evaluation of a Network-oriented Service with Environmental Adaptability based on Core/Periphery Structure," *Technical Report of IEICE (NS2020-158)*, vol. 120, pp.208-213, March 2021.

Preface

Many new network-oriented services have been developed in recent years, and these services are expected to be virtualized in multi-access edge computing (MEC) environments, which are being standardized along with 5G. To accommodate large numbers of services at low cost, the service design needs to be adaptable to user requirements and environmental changes.

We investigate a core/periphery structure 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, which has been used to interpret the behaviors of biological systems, social networks, and internet systems. Some system components, called "core", do not change despite the composition of the entire system being changed with time and mediate the connection of non core system elements, called "periphery."

We first investigate the design principles and the placement policies that reduce the cost of designing and developing VNFs for accommodating new service requests. We introduce a Core/Periphery-Based Design (CPBD) that utilizes the core/periphery concept for developing VNFs. In addition, we examine the Center-Located Core/Periphery placement (CLCP) policy and the Geographically-Distributed Core/Periphery placement (GDCP) policy, and evaluate the long-term cost of the NFV system under resource restrictions to run VNFs. Our results show that CPBD reduces the long-term cost of design and development of VNFs by 23% compared to the design with no core VNFs. Moreover, in the case of no resource restrictions, both CLCP and GDCP reduce the long-term costs of placing and connecting VNFs by 15% compared to the existing VNF placement algorithm. With resource constraints, GDCP reduces the long-term costs over CLCP by 11%.

Second, we introduce a core/periphery structure into a network-oriented service. We design

and implement a network-oriented mixed reality service based on this structure. To utilize the flexibility of a core/periphery structure, we regard core functions as those whose behaviors remain unchanged even when there are changes in user requests or the environment. In contrast, peripheral functions are those whose behaviors can change under such circumstances. Experiments reveal that implementation costs are reduced while retaining increases in service response time to less than 31 ms. These results show that taking advantage of a core/periphery structure allows appropriate division of service functions and placement of functions in a MEC environment, with only small penalties on latency and at a low implementation cost.

Third, we evaluate the core/periphery-based service structure in the following two aspects more pragmatically. The first one is the service scenario to use in our experiment. We consider a service scenario that includes information processing and information sharing among remote robots and users. The second one is the metric to represent the implementation cost. 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., a modern software development. We use the cyclomatic complexity, which is the number of independent paths from the start to the end of the program as the metric to evaluate the implementation cost. Our experiment reveals that an information processing platform using a core/periphery structure is adaptable to environmental changes at a small cost by reusing the core and recreating only the periphery.

However, when large-scale environmental changes arise, it remains necessary to change the core/periphery roles of functions, and reconfiguration of only the peripheral functions may not be sufficient to adapt to such changes.

Therefore, we propose an "evolvable" structure of service functions network based on a core/periphery structure. Here, we refer to the network that can change the system at low cost with maintaining the ability to provide new services, as evolvable service structures. We show our proposed method achieves stable and high service chain accommodation ratios in multiple evolution patterns with low cost. This provides an advantage for changing the service functions structure in the future for a long period of time.

Acknowledgments

I would like to express my sincere appreciation to everyone who supported me in various ways throughout my Ph.D. This thesis could not have been accomplished without their assistance.

First of all, I express my grate gratitude to my supervisor, Professor Masayuki Murata of Graduate School of Information Science and Technology, Osaka University, for his insightful suggestions and valuable discussions. He brought me to an attractive research field, and made my research life fruitful.

I am heartily grateful to the members of my thesis committee, Professor Takashi Watanabe, Professor Toru Hasegawa, and Professor Hirozumi Yamaguchi of Graduate School of Information Science and Technology, Osaka University, and Professor Hideyuki Shimonishi of Cyber Media Center, Osaka University, for their multilateral reviews and perceptive comments.

Furthermore, I would like to owe my special thanks to Associate Professor Shin'ichi Arakawa of Graduate School of Information Science and Technology, Osaka University, for his continuous support and encouragement. He taught me the fun of thinking about and solving problems.

Moreover, I am deeply grateful to Assistant Professor Yuichi Ohsita, Assistant Professor Daichi Kominami, Assistant Professor Naomi Kuze, Assistant Professor Tatsuya Otoshi, Specially Appointed Assistant Professor Masaaki Yamauchi, of Osaka University for their appreciated comments and support. Their kindnesses on my behalf were invaluable, and I am forever in debt.

Also, I sincerely appreciate a lot of support of Mr. Yuki Tsukui of Osaka University for his numerical investigate.

I am really thankful to all of past and present colleagues, friends, and secretaries of the Advanced Network Architecture Research Laboratory, Graduate School of Information Science and

Technology, Osaka University. Finally, I express my thanks to my family for invaluable supports throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Many new network-oriented services have arisen, and information networks are rapidly changing. Using these new services, we can send real-world information from cameras and sensors to the cloud, or perform high-load processing such as image recognition or voice and sound recognition. For example, telexistence [1] services, which enables human users to feel present virtually in a remote place by operate a remote robot as if it is their own body and use robots and virtual reality (VR) or mixed reality (MR) technologies are now being developed, which 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. In such applications, application-level delay is a significant factor affecting service quality. However, communication distance and load concentrations can significantly increase application-level delay in cloud computing environments [2]. Recently, multi-access edge computing (MEC) [2–4] has been standardized to mitigate increases in application-level delay for delay-sensitive services. In an edge computing environment, computing resources and storage are allocated at the network edge so that processing required by end devices is performed at closer sites. This improves the responsiveness of applications by shortening communication distances and load distributions.

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 [5]. To reduce the costs, module-based design has been widely introduced. In module-based design, modules connect each other via interfaces and allow to decoupling them [6], and each module is 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. [7] 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. [7] 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 investigate a core/periphery structure [8, 9] 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, which has been used to interpret the behaviors of biological systems, social networks, and internet systems. Some system components, called "core", do not change despite the composition of the entire system being changed with time and mediate the connection of non core system elements, called "periphery as shown in Fig. 1.1." 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. When designing services that require efficient processing of a various input/output data based on environmental changes, designing inspired by the biological core/periphery structure is expected to enable the service adaptable to various inputs and outputs efficiently. A core/periphery structure is also found in 5G (fifth Generation Mobile Communication system). Network Exposure Function

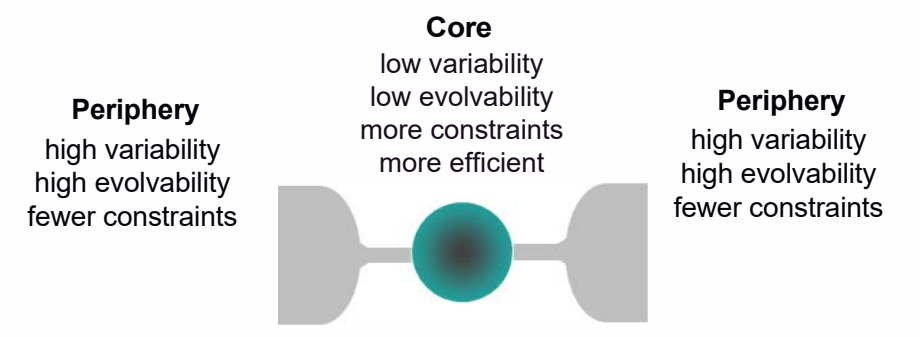

Figure 1.1: Concept of core/periphery structure

(NEF) [10] is located between the 5G core network and external third-party application functions (AF) and manages external open network data. Opening the network function (NF) to third-party applications provides a connection between the network function and business requirements and optimizes the allocation of network resources. Since all external applications that want to access the internal data of the 5G core always pass through the NEF, it is considered that the NEF exists as an interface between the AF and the NF and behaves as a core in the overall network that includes both AF and NF. In this thesis, we control the core and peripheral functions based on considering only the information exchange that occurs only on the AF side, but it is expected that a structure consisting of core and peripheries can be found in the network on the NF side as well.

1.2 Adaptable and Evolvable Network-oriented Service Structure

To accommodate large numbers of services at low cost, the service design needs to be adaptable to user requirements and environmental changes. Because many new network-oriented services have developed to meet various user requests, it is important to consider service designs that can accommodate as many services as possible when deploying network services in a MEC environment. However, implementation costs increase if developers must reconstruct entire services to meet different user requests or to adapt to environmental variation such as device evolution. MEC environment resources are limited by spatial restrictions, making it difficult to locate all possible services,

such those on the edge that can adapt to each user request and environmental variation. It is therefore necessary to consider service structures that can change service behaviors in a flexible manner. Service function placement in MEC environments has been studied in, for example, [11, 12], but most of them correspond to user mobility. Therefore, we introduce a core/periphery structure into the service design and reveal that this structure leads the service to environmental changes by modifying only the peripheral functions.

In addition, against large-scale environmental changes, the structure of the service function requires evolution. Here, we refer to service structures that can change the system at low cost with maintaining the ability to provide unknown services, as evolvable service structures.

We assume that there are many service functions created by software developers, and that the service functions are connected through the interfaces are connected to each other. When environmental changes occur, the functions commonly used to make up service chains change or new functions are required to be added. Adapting to environmental changes requires the addition of interfaces between service functions or the development of new service functions. Efficient accommodation provides the service with short chain length using only the minimum service functions. For example, if the network of service functions is provided in a full mesh, the shortest chain can be configured, but the number of interfaces between functions to maintain that density when adding new functions is large. This makes the development cost significant and makes it impossible to maintain the services in the future. If all functions are designed sparsely, the cost when adding new functions is small, but the chain length is long since it requires extra functions to accommodate service chains. Therefore, we propose a service structure that can efficiently accommodate various service chains with low development cost by controlling the density of service functions.

1.3 Outline of Thesis

The overview of this thesis is shown in Fig. 1.2.

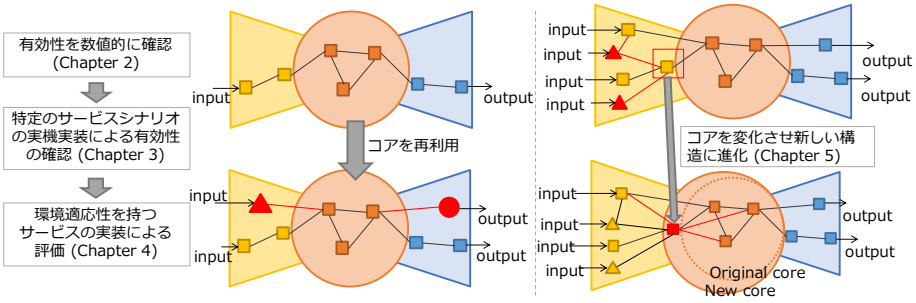

Figure 1.2: Overview of this thesis

Design and Placements of Virtualized Network Functions based on a Core/Periphery Structure [13]

In Chapter 2, we numerically investigate the design principles and the placement policies that reduce the cost of designing and developing virtual network functions (VNFs) for accommodating new service requests. As for the design policy, we introduce a Core/Periphery-Based Design (CPBD) that utilizes the core/periphery concept for developing VNFs. In CPBD, "core" VNFs are developed in advance and repeatedly used to accommodate future service requests. While "core" VNFs are common to current and future service requests, "periphery" VNFs are developed and customized for each service request. Next, we investigate the placement policies of VNFs for CPBD to fully utilize the nature of their core/periphery structure. In addition, we examine the Center-Located Core/Periphery placement (CLCP) policy and the Geographically-Distributed Core/Periphery placement (GDCP) policy, and evaluate the long-term cost of the NFV system under resource restrictions to run VNFs. Our results show that CPBD reduces the long-term cost of design and development of VNFs by 23% compared to the design with no core VNFs. Moreover, in the case of no resource restrictions, both CLCP and GDCP reduce the long-term costs of placing and connecting VNFs by 15% compared to the existing VNF placement algorithm. With resource constraints, GDCP reduces the long-term costs over CLCP by 11%.

Design, Implementation and Evaluation of Core/Periphery-based Network-oriented Mixed Reality Services [14, 15]

The advantages of a core/periphery structure for accommodating information services, represented by chains of functions, were numerically investigated in Chapter 2. From the biological point of view, the core/periphery structure is expected to achieve an efficient and adaptive behavior against environmental changes.

However, from the service design point of view, dividing functions and placing them in different devices creates extra communication paths. This degrades service responsiveness and cannot be ignored. Therefore, when we apply the core/periphery structure to the service design, we need to consider the balance between the penalty and the reduction of development costs. In this chapter, we show that a core/periphery structure allows services to adapt to increasing the number of device types with low implementation cost, and evaluate the actual penalty of locating core functions on edge servers with regards to the service responsiveness through a service implementation.

In Chapter 3, we implement a network-oriented mixed reality (MR) service based on a core/periphery structure using actual MR devices and robots. Our implementation focuses on a shopping service, but service design based on a core/periphery structure is not limited to the shopping service and can be applied to other network services. We investigate what kinds of functions should be developed to accommodate user requests in conjunction with various types of devices and real-world environments in which users and devices are located. To utilize the flexibility of a core/periphery structure, we regard core functions as those whose behaviors remain unchanged even when there are changes in user requests or the environment. In contrast, peripheral functions are those whose behaviors can change under such circumstances. Experiments reveal that implementation costs are reduced while retaining increases in service response time to less than 31 ms. These results show that taking advantage of a core/periphery structure allows appropriate division of service functions and placement of functions in a MEC environment, with only small penalties on latency and at a low implementation cost.

Design, Implementation and Evaluation of a Network-oriented Service with Environmental Adaptability based on Core/Periphery Structure [16, 17]

We evaluate in the following two aspects more pragmatically than chapter 3. The first one is the service scenario to use in our experiment. In our previous works, we considered a service scenario that includes only information processing; however, commonly used applications today not only process information obtained from devices, they also share information among such devices. We focus on the information sharing in this chapter. We consider a service scenario that includes information processing and information sharing among remote robots and users and evaluate our service design in terms of the complexity of the source code and overhead for information sharing. To investigate the amount of penalties on sharing information, we implement a service and measured the penalty through experiments on actual devices. The second one is the metric to represent the implementation cost. In our previous works, 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 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., a modern software development. We use the cyclomatic complexity [18], which is the number of independent paths from the start to the end of the program as the metric to evaluate the implementation cost.

Evolvable Design of Network-oriented Services based on Core/Periphery Structure [19]

Our experiments in Chapter 3 and 4 have shown that an information processing platform using a core/periphery structure is adaptable to environmental changes at a small cost by reusing the core and recreating only the periphery. However, when large-scale environmental changes arise, it remains necessary to change the core/periphery roles of functions, and modifying only the peripheral functions may not be sufficient to adapt to such changes. In this chapter, we propose an evolvable service structure that can efficiently accommodate various service chains with low development cost

by controlling the density of service functions. We show our proposed method achieves stable and high service chain accommodation ratios in multiple evolution paths. In addition, the development cost used to apply our proposed method is independent of the number or length of future service chains. This provides an advantage for changing the service functions structure in the future for a long period of time, because other methods require different costs to accommodate depending on the number or length of service chains and it is difficult to predict service chains that will arise in the future.

Chapter 2

Design and Placements of Virtualized Network Functions based on a Core/Periphery Structure

2.1 Introduction

As service demands become increasingly diverse, Network Function Virtualization (NFV) is gaining attention. NFV can implement network functions, such as firewall and proxy server, as a Virtual Network Function (VNF), which is developed using software. VNFs can run on general-purpose hardware shared with other VNFs. NFV flexibly accommodates various service requests by connecting VNFs over networks.

In operating NFV systems, it is important to reduce the costs of accommodating network services. Many previous studies conducted on NFV have discussed placement algorithms that minimize the costs of placing VNFs [20, 21]. For example, Kim et. al. [20] used a genetic algorithm to minimize the power consumption and satisfy the service delay requirements of users. Nam et. al. [21] minimized the end-to-end service time by placing VNFs based on Zipf's law which models the frequency of VNFs use. Although these studies used different algorithms or approaches, they implicitly assumed that VNFs are developed in advance. However, in reality, service requests

may change drastically and require VNFs that have not yet been developed. Therefore, we need a suitable software design of VNFs and its placement to accommodate the current and future service requests at a lower cost. If VNFs are not appropriately designed, new VNFs will be added frequently depending on changes in service requests, which will lead to an increase in their cumulative development cost. Moreover, appropriate placement of VNFs is required to reduce opportunities for changing placement, such as adding, moving, and removing VNFs. In this thesis, we investigate a software design and placement method for VNFs that can reduce long-term development costs against changes in service requests.

In considering the software design of VNFs, we introduce a core/periphery structure [8, 9], which has been used to interpret the behaviors of biological systems, social networks, and internet systems. Some system components, called "core", do not change despite the composition of the entire system being changed with time and mediate the connection of non core system elements, called "periphery." We interpret VNFs based on a core/periphery structure and distinguish them into core and periphery VNFs. It is expected that designing core VNFs such that they can be repeatedly used will reduce the long-term development cost for accommodating future service requests. However, the development cost of each core VNF is higher than that of each periphery VNF, because core VNFs need to be generalized to be connected with other VNFs. Therefore, we introduce a model for deriving the development costs of NFV software systems and reveal the benefit of introducing core/periphery structures in VNF software design.

Next, we investigate how to place VNFs designed based on a core/periphery structure, as the deployment cost of VNFs can be reduced by appropriately placing the core VNFs in advance, so that they can be shared to accommodate future service requests. In fact, the existing method can reduce the number of VNFs to be placed by sharing common VNFs among the service requests [22]. We examineCenter-Located Core/Periphery placement (CLCP) policy and Geographically-Distributed Core/Periphery placement (GDCP) policy. CLCP places core VNFs at the center of physical networks, which increases the opportunity for core VNFs to accommodate many service-chain requests. In contrast, GDCP places core VNFs for each topological cluster, which prevents resource exhaustion resulting from accommodating service-chain requests. In addition, simulations are conducted for CLCP and GDCP, and the long-term cost of the NFV system is evaluated under

resource restrictions to run VNFs.

2.2 Design and Placement Problems of NFV Software Systems

2.2.1 Design problem

For the operation of an NFV system, it is important to design a VNF properly, to reduce costs. The NFV system comprises many VNFs and connects them to accommodate the service requests. There are some costs incurred in designing and developing VNFs, which disturb the flexible accommodation for service requests. A suitable software design of VNFs can reduce such costs.

A monolithic software design has been widely used for software such as networking software. In monolithic software design, multiple components form a single module [23]. These components are designed to compose a particular service and connect specific components. Thus, changing components can incur changes in other components, and thus, increase the development cost [5, 24–27]. Moreover, tight coupling makes it difficult to use the components already developed for accommodating new services. Existing studies [24, 25] have analyzed how software components have been designed and developed in the long term, such as Linux and Mozilla, and indicated that large-scale refactoring to reduce tight coupling and increase the generality of components will contribute to fastening the application development.

Recently, in the field of software engineering, microservices have gained attention due to the possibility of reducing the development cost [23, 27, 28]. In microservices, components are well independent and can be connected with other components to form various services. The developed components can be used to accommodate future services; thus, microservices are expected to reduce the number of components and costs for design and development.

However, in the case of networked software, such as NFV systems, sufficient discussions have not been conducted on software. This thesis discusses networked-software designs that have not attracted enough attention so far. To reduce development costs, such as those related to microservices, we design “core” VNFs to be used to accommodate new service-chain requests.

2.2.2 Placement problem

Many NFV studies have developed placement algorithms that can minimize the costs related to VNF placement and accommodate service requests efficiently . One such method [20] uses a genetic algorithm to solve the problem that minimizes power consumption while satisfying the service delay requirements of users. Another method [21] places VNF on a physical network based on Zipf's law, which models the frequency of use, to minimize the end-to-end service time. The existing placement algorithm, called affiliation-aware vNF placement (AaP), can reduce the number of VNFs to be placed by merging the service requests [22]. AaP places VNFs on the shortest path between the source and destination nodes in order of each merged request, for avoiding bandwidth consumption.

For example, in accommodating two service requests, such as VNF1 $\rightarrow$ VNF2 $\rightarrow$ VNF3 and VNF4 $\rightarrow$ VNF2 $\rightarrow$ VNF5, AaP merges these requests and assumes them as one service request, such as VNF1 $\rightarrow$ VNF4 $\rightarrow$ VNF2 $\rightarrow$ VNF3 $\rightarrow$ VNF5. Here, VNF2 is shared and the number of VNF2 placements is reduced from 2 to 1. These studies aim at optimization in terms of power consumption, end-to-end service time, and bandwidth consumption.

However, considering the long-term operation of an NFV system, it is more important to reduce costs to additionally place VNFs and change the VNF location. For accommodating the service requests, the NFV system places VNFs into a physical network and connects them in a suitable order. However, there are some costs incurred in placeing and connecting VNFs, which are called deployment costs. In an operating NFV system, service requests may change variously and require VNFs that have not yet been placed and connected. These VNFs require additional costs to newly place and connect them. Moreover, changing the placement of VNFs, such as adding, moving, and removing them, suspends their execution and causes a delay in data communication [29]. Such factors increase the deployment cost. However, the above mentioned existing placement methods [20–22] may frequently change the VNF placement, because they do not consider changing the placement depending on the variation in the service requests. In this chapter, we investigate a VNF placement method that can reduce long-term deployment costs against changes in service requests.

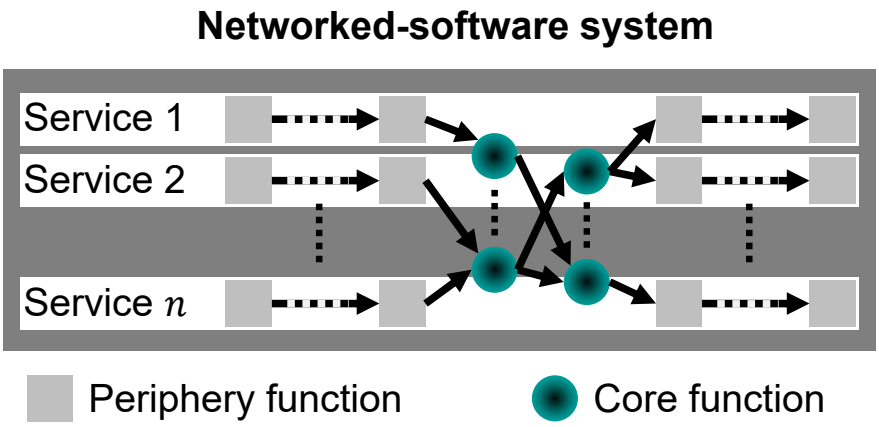

Figure 2.1: Example of NFV system

2.2.3 Approaches with core/periphery structure

In considering the software design of VNFs, we introduce a core/periphery structure [8, 9], which has been used to interpret the behaviors of biological, social, and internet systems. A system possessing a core/periphery structure operates stably. This is because some system components, such as the "core", do not change despite changes in the composition of the entire system with time, and mediate the connection of non core system elements, such as the "periphery." Fig. 1.1 shows the basic concept of a core/periphery structure.

We interpret VNFs based on a core/periphery structure, and distinguish them into core and periphery VNFs. It is expected that the additional VNFs and their development cost will be reduced by designing VNFs such that the core VNFs can be used repeatedly to accommodate future service requests. Moreover, the deployment cost can be reduced by suitably placing VNFs designed based on a core/periphery structure. To reduce the deployment cost, we place the core VNFs in advance, so that they can be used repeatedly to accommodate future service requests, as in the case of software design. In fact, the above-explained AaP can reduce the number of VNFs to be placed by sharing common VNFs among the service requests [22].

2.3 Core/periphery-based Design of NFV Systems

2.3.1 Core / Periphery Based Design

An NFV system has many VNFs and accommodates various service requests by appropriately connecting them. The connecting order of VNFs is called a service-chain. In general, some VNFs are frequently used for accommodating service-chain requests, and are regarded as core VNFs. Such a situation occurs when the VNFs are well-implemented, which is sufficient to be connected with many other VNFs. The other functions are regarded as periphery VNFs, which are used only for a specific service-chain request and implemented sufficiently to be connected with specific VNFs, such as receiving the process result of a VNF as input and passing it to another VNF as output. Fig. 2.1 illustrates the NFV system with core/periphery functions.

Such a VNF classification is based on a core/periphery structure, where the core part does not change despite the changes in service requests and mediates the connection of other system parts. The periphery part has higher variability and absorbs changes in service requests. Core VNFs are used to accommodate multiple service-chain requests and should not be changed frequently, owing to their generality. Periphery VNFs are used to accommodate service-chain requests that cannot be accommodated by core VNFs alone, and therefore, can absorb changes in service-chain requests. We call a software design that has both core VNFs and periphery VNFs as a Core/ Periphery-Based Design (CPBD).

Because core VNFs can be connected to other VNFs, they have more opportunity to accommodate service-chain requests. Preparing many core VNFs will lead to lesser development costs of periphery VNFs for accommodating new service-chain requests. This is because most of the service functionalities would be provided by core VNFs. However, the development cost of each core VNF will be higher than that of each periphery VNF because core VNFs should be generalized to be connected with other VNFs. Based on this observation, we model the development costs of NFV software systems, as presented in Sec. 2.3.2, and compare them with those of CPBD, as presented in Sec. 2.3.3.

2.3.2 Cost definitions

Let us consider an NFV system that accommodates n service-chain requests and the j-th service-chain request requires $k(j)$ VNFs on average. An NFV software system has $f_{all}(n)$ VNFs, which is the sum of the number of core VNFs, $f_c(n)$, and periphery VNFs, $f_p(n)$:

$$f_{all}(n) = f_c(n) + f_p(n). \tag{2.1}$$

The development cost of an NFV software system, $c_{all}(n)$, is

$$c_{all}(n) = \sum_{i=0}^{f_c(n)} c_c(i) + \sum_{j=0}^{n} k_p(j) c_p(j), \tag{2.2}$$

where $f_c(n)$ is the number of core VNFs and $c_c(i)$ is i-th core VNF's development cost. Moreover, the j-th service-chain request requires $k_p(j)$ periphery VNFs, and $c_p(j)$ is the development cost of each $k_p(j)$ periphery VNF. In Eq. (2.2), the first term represents the sum of the development costs of the core VNFs and the second term represents that of periphery VNFs because each periphery VNF serves only one service-chain request and the number of periphery VNFs equals $\sum_{j=1}^{n} k_p(j)$.

The variable $c_c(i)$ increases because of the ability to connect with many other VNFs, such as already implemented core VNFs. Thus,

$$c_c(i) = \alpha i, \tag{2.3}$$

where the parameter α determines how the development cost of newer core VNFs increases with the number of core VNFs.

Implementing more core VNFs decreases $c_p(j)$ because more service functionalities can be provided by the core VNFs as compared to the periphery VNFs. Thus,

$$c_p(j) = \exp(-\beta f_c(j)), \tag{2.4}$$

where β determines how the development cost of a periphery VNF decreases as the number of core

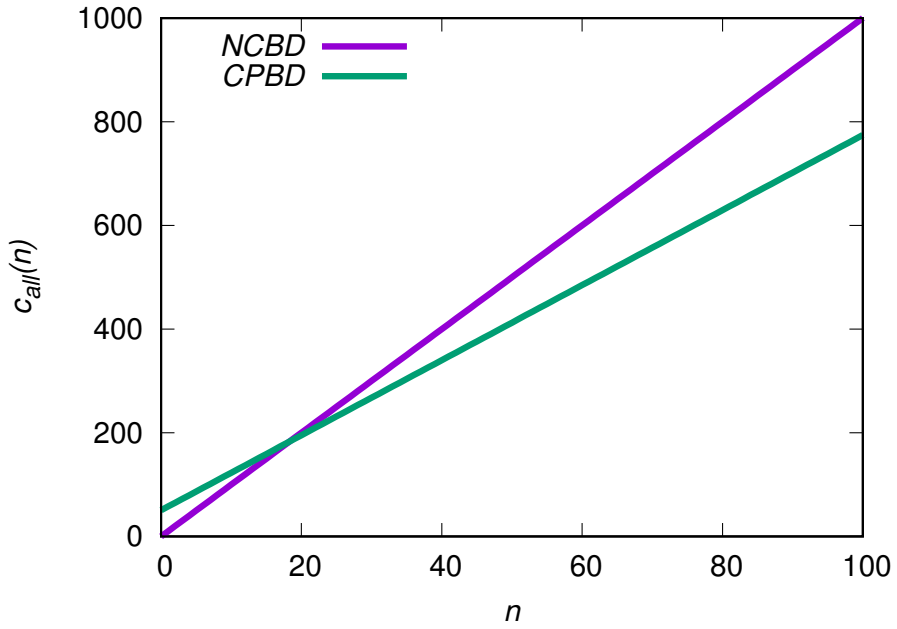

Figure 2.2: The development cost for two design scenarios

VNFs increases. For example, if a firewall is implemented as a periphery, its development cost can be reduced using core VNFs, such as pattern matching and session management. Such cases occur more frequently as the number of core VNFs increases. Note that not all implemented core VNFs can serve for a service-chain request, thus Eq.2.4 forms negative exponential.

Then, we represent $k_p(j)$ by $f_c(j)$ to observe the change in $c_{all}(n)$ against $f_c(n)$. We introduce a parameter γ ($0 < \gamma < 1/f_c(j)$), which represents how often the $f_c(j)$ core VNFs are repeatedly used among the service-chain requests, and $k_c(j)$ is written as

$$k_c(j) = k(j)\gamma f_c(j). \tag{2.5}$$

Then, $k_p(j)$ is obtained as k=$k_c(j)$ + $k_p(j)$;

$$k_p(j) = k - k\gamma f_c(j). \tag{2.6}$$

2.3.3 Benefit of CPBD for long-term development costs of NFV software system

We consider a scenario in which n increases from 0 to 100, which indicates that new networking services emerge dynamically over time. In this section, we set $k = 10$, $\alpha = 0.01$, $\beta = 0.001$, and $\gamma = 0.002$. When these parameters are changed, the slope of the graph changes, but CPBD reduces long-term development costs as this simulation settings.

We examine two design scenarios – noncore based design (NCBD) and CPBD – and discuss the development of each scenario by comparing $c_{all}(100)$ values of both NCBD and CPBD. NCBD uses only periphery VNFs to accommodate services without designing and developing core VNFs. That is, NCBD maintains $f_c(n) = 0$ regardless of the value of n. CPBD designs and develops 100 core VNFs even when n=0; that is, none of the service-chain requests are accommodated. Setting $k = 10$ and $\gamma = 0.002$ means that 2 among the 100 core VNFs are used, on average, to accommodate each future service-chain request.

Figure 2.2 shows the development cost of the NFV system $c_{all}(n)$ for each n. Note that the figure does not consider the addition of core VNFs; that is, $f_c(n)$ is always 100. The figure shows that the $c_{all}(0)$ of CPBD is 50 times higher than that of NCBD. This is because CPBD requires more costs to design and develop core VNFs before accommodating the service-chain request. However, CPBD can reduce the development cost by 23% compared to NCBD. This result suggests that CPBD can reduce long-term development costs by using the developed core VNFs to accommodate future service requests.

2.4 Placement Methods of Core/Periphery VNFs

2.4.1 Placement algorithms for a core/periphery-based software system

In the previous section, we revealed that CPBD reduces long-term development costs. Our next concern is where to deploy the core and peripheral VNFs in the physical network.

Because core VNFs are developed such that other VNFs can be reused, their placement is the most crucial problem in accommodating new service requests with lesser long-term costs. A method for obtaining a suitable placement of core VNFs is to solve the optimization problem that

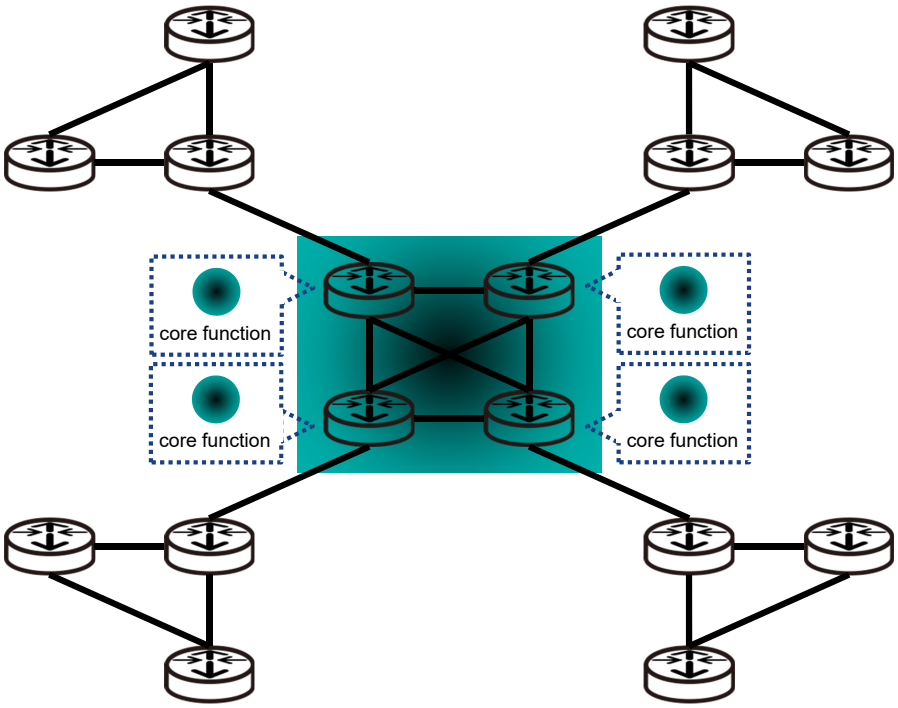

Figure 2.3: Example of CLCP

minimizes the deployment cost each time a service-chain request arises. Here, the deployment cost is the cost to place and connect VNFs, and not the development cost used in the previous section. Minimizing the long-term deployment costs is problematic because the placement of core VNFs affects the additionally placed VNFs for future service requests. Therefore, we use the following two heuristics, which focus on the topology of the physical network, and compare them.

- Duplications of core VNFs are placed at the center of the physical network. (CLCP: Center-Located Core/Periphery placement Policy)
- Duplications of core VNFs are distributed over the physical network. (GDCP: Geographically-Distributed Core/Periphery placement Policy)

Figures 2.3 and 2.4 illustrate the placement of core VNFs in cases of CLCP and GDCP, respectively, which are detailed in the following sections.

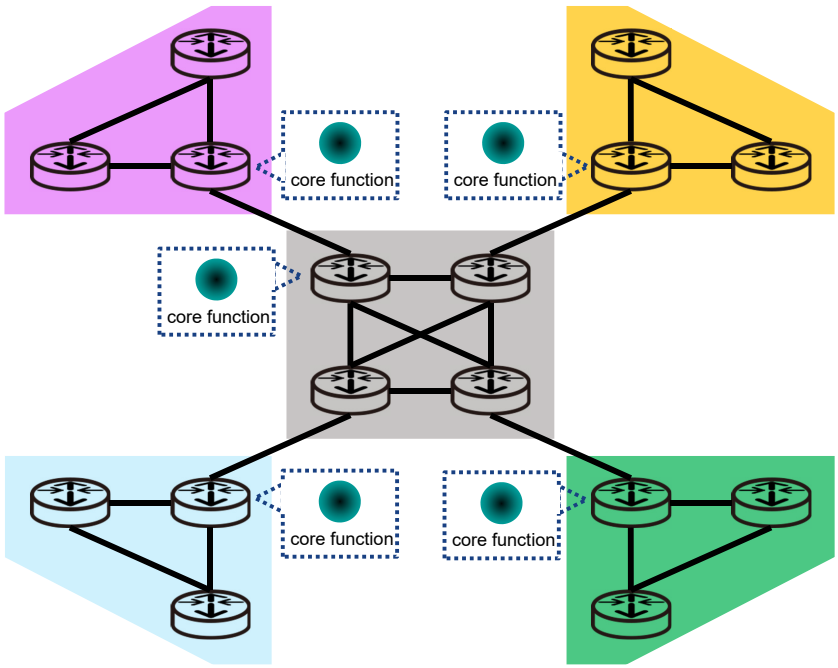

Figure 2.4: Example of GDCP

Notations

Table 2.1 shows the notations used in CLCP and GDCP.

The physical network is represented as $G = (V, E)$, where V is the node set and E is the link set. Given a G, a path set P that comprises the shortest paths between each source destination pair is calculated. Here, $P_{u,v} \in P$ represents the shortest paths between each source node $u \in V$ and each destination node $v \in V$.

In each time slot t, $C_v(t)$ represents the remaining node resources for each node $v \in V$ and $B_e(t)$ represents the remaining bandwidth resources for each link $e \in E$. Note that $C_v(0)$ and $B_e(0)$ represent the initially allocated node and bandwidth resources, respectively.

In each time slot t, λ type of service-chain requests are generated, and the required VNFs belonging to the set of all VNFs, M, are placed in the physical network. When VNF $m \in M$ is placed on a node v, $\hat{c}_{m,v}$ node resources are consumed from $C_v(t)$. We assume that $\hat{c}_{m,v}$ is uniform for any VNF $m \in M$ and node $v \in V$, and denote it by $\hat{c}$. Moreover, the deployment cost, $\alpha_{m,v}$,

Table 2.1: Table of notations

V	node set
E	set of links
P	set of all pre-calculated shortest path
$P_{u,v}$	set of shortest paths between $u, v \in V$, and $P_{u,v} \in P$
t	time slot index
T	max size of time slot
$C_v(t)$	remined node resources for each node $v \in V$
$B_e(t)$	remined bandwidth resources for each link $e \in E$
$\hat{c}$	node resource consumption when VNF is placed on node
R	set of all service-chain requests ($r = \{s_r, d_r, b_r, n_r\}, r \in R$)
R_t	set of service-chain requests at each time slot t
s_r	source node of $r \in R$
d_r	destination node of $r \in R$
b_r	bandwidth consumption when link is used for accommodating $r \in R$
$\vec{n}_r$	service-chain of $r \in R$
M	set of all VNF
X	core VNF set
Y	periphery VNF set
w_x	the number of duplications of $x \in X$
$U_{m,v}$	the remaining number of service-chain requests that can use a VNF m placed to node v at each time t
α	deployment cost to place a VNF on a node

is required to place VNF $m \in M$ on a node v. We also assume that $\alpha_{m,v}$ is uniform for any VNF $m \in M$ and node $v \in V$, and denote it by α. Note that the total deployment cost is the sum of $\alpha_{m,v}$ and increases with the time slot t.

R_t is the set of new service-chain requests at each time slot t, and R is a cumulative set of R_t over the time slot. That is, $R = R_1, R_2, \ldots, R_T$. Each service-chain request $r \in R$ is represented as $r = \{s_r, d_r, b_r, \vec{n}_r\}$, where $s_r \in V$ is the source node of a service-chain request r, $d_r \in V$ is the destination node of r, and b_r indicates the bandwidth resources consumed by $B_e(t)$ of each link $e \in E$.

Each VNF $m \in M$ is classified into either the core VNF or the periphery VNF. Denoting X as the core VNF set and Y as the periphery VNF set, X and Y satisfy $X \cup Y = M$ and $X \cap Y = \emptyset$. For each core VNF $x \in X$, w_x represents the number of duplications of x placed in the physical

network. $U_{m,v}(t)$ is the remaining number of service-chain requests that can use a VNF m placed at node v without processing overhead. When VNF $m \in M$ is placed on node $v \in V$, there is a limitation on running VNF m on that node. We denote $U_{m,v}(t)$ as the remaining number of service-chain requests that can use a VNF m placed on node v. $U_{m,v}(0)$ denotes the maximum number of service-chain requests that can use a VNF m on node v.

CLCP

Nodes located at the geographic center of the physical network have more opportunities for paths to go through. Placing core VNFs on such nodes increases the opportunity for them to accommodate many service-chain requests.

CLCP places the duplication of core VNFs on nodes in descending order of their *efficiency* [30], which is a metric for measuring how efficiently information exchange is performed on a node. A node with a high *efficiency* has a short hop count from/to other nodes, and is located at the center of the physical network.

Algorithm 1 shows the CLCP core placement algorithm of CLCP. For a loop from line 2 to line 10, the core VNFs are placed. From lines 5 to 7, we obtain a node v that has the highest *efficiency* to place x considering the resource restriction of the node. When placing the core VNF x on node v, $\hat{c}$ resources are consumed, and the remaining resource, $C_v(t)$, decreases by $\hat{c}$. When $C_v(t)$ is lower than $\hat{c}$, VNF x cannot be placed on v. When the node v, which exhibits the highest efficiency, does not satisfy the node resource restriction or x has already been placed on v, Algorithm 1 sets v to a node with the next highest *efficiency*. This process of placing the core VNF x is repeated w_x times, which is the number of duplications of the core VNF x.

Next, we explain where to place the periphery VNFs. Given the placements of core VNFs by Algorithm 1, we place and connect periphery VNFs sequentially from the source node to destination node via nodes where core VNFs are deployed. Note that a service-chain is composed of periphery-core-periphery VNFs as depicted in Fig. 2.1. Thus, there are three path segments between the source and destination nodes: a segment for periphery VNFs (source side), that for core VNFs, and that for periphery VNFs (destination side).

Algorithm 1 Core placement algorithm of CLCP

Input: $G = (V, E)$, X, w_x
1: **if** $t = 1$ **then**
2: **for** each $x \in X$ in descending order of $\hat{c}$ **do**
3: $v \Leftarrow$ node with the highest *efficiency*
4: **for** *loopcounter* $= 1$ to w_x **do**
5: **while** $\hat{C}_v(t) < \hat{c}$ or x has already been placed to v **do**
6: $v \Leftarrow$ node with a next higher *efficiency*
7: **end while**
8: place x to v
9: **end for**
10: **end for**
11: **end if**

Algorithm 2 calculates a set of available paths for each path segment, P_{avail}, and determines the possible nodes on which core VNFs can be placed, under the resource restriction of the node in line 7. In addition, it considers the bandwidth resource restriction in line 13. In using link e for accommodating a service-chain request r, b_r bandwidth resources are consumed by $B_e(t)$, which is defined as the resources remaining on link e. b_r should not exceed $B_e(t)$; otherwise, r cannot use e because of the lack of bandwidth resources. When $U_{m,v}$ is 0 for all m and v, or Algorithm 2 cannot obtain P_{avail} with the remaining bandwidth resources larger than b_r, a service-chain request r is rejected.

Algorithm 3 places the periphery VNFs along the P_{avail} obtained by Algorithm 2. However, when the hop count between the nodes in P_{avail} is too short, periphery VNFs cannot be placed because there are fewer opportunities to find a node to run them. To avoid such a situation, we consider detour paths other than the shortest path for each path segment from lines 11 to 16 . In more detail, when m is placed on v, which deploys core VNFs and does not retain sufficient resources, a detour path, p, is selected using the neighboring nodes of v.

GDCP

By placing the core VNFs at the center of a physical network, as in CLCP, they can be used more frequently for accommodating many service-chain requests. However, using the central nodes that

Algorithm 2 Algorithm to obtain available paths to place periphery VNFs

Input: $G = (V, E)$, X, P, r
Output: P_{avail}

1: $P_{avail} \Leftarrow \emptyset$
2: $s'_r \Leftarrow s_r$
3: **for** each $m \in \vec{n}_r$ **do**
4: **if** $m \notin X$ **then**
5: continue
6: **end if**
7: **if** $U_{m,v}(t) = 0$ for all VNF m and node v **then**
8: reject r
9: **end if**
10: $d'_r \Leftarrow$ node with the shortest path from s'_r, VNF m, and $\hat{r}_m(t) < \tilde{r}_m$
11: Obtain $p_{s'_r,d'_r} \in P_{s'_r,d'_r}$ with the highest remaining bandwidth resources
12: **for** each $e \in p_{s'_r,d'_r}$ **do**
13: **if** $\hat{B}_e(t) < b_r$ **then**
14: reject r
15: **end if**
16: **end for**
17: add $p_{s'_r,d'_r}$ to P_{avail}
18: $s'_r \Leftarrow d'_r$
19: **end for**
20: $d'_r \Leftarrow d_r$
21: Repeat from lines 11 to 17

Algorithm 3 The CLCP periphery placement algorithm

Input: $G = (V, E)$, X, P, R

1: **for** each $r \in R$ in descending order of b_r **do**
2: call Algorithm 2 and obtain P_{avail}
3: $p \Leftarrow$ first path of P_{avail}
4: $v \Leftarrow$ source node of p
5: **for** each $m \in \vec{n}_r$ **do**
6: **if** $m \in X$ **then**
7: $p \Leftarrow$ next path of P_{avail}
8: $v \Leftarrow$ source node of p
9: *continue*
10: **end if**
11: **if** v owns any core VNF **then**
12: **while** $C_v(t) < \hat{c}$ **do**
13: $v \Leftarrow$ neighbor node of v with the highest remaining node resource $C_v(t)$
14: **end while**
15: add a detour that can reach v to p
16: **end if**
17: **while** $C_v(t) < \hat{c}$ **do**
18: **if** v is the destination node of p **then**
19: reject r
20: **end if**
21: $v \Leftarrow$ next node of p
22: **end while**
23: place m to v
24: **end for**
25: **end for**

deploy core VNFs and their neighbors leads to resource exhaustion because such nodes are intensively used for accommodating service-chain requests. Therefore, we examine another placement algorithm that exhibits a distributed placement of core VNFs on the physical network.

GDCP divides the physical network into clusters to maximize modularity [31], and place duplication of core VNFs in each cluster. Modularity is a metric that reflects the density of the cluster density; a higher modularity leads to an increased ratio of the number of links between clusters and that in each cluster. Algorithm 4 shows the core placement algorithm of GDCP. In line 2, we divide the physical network into clusters using the Louvain algorithm [32] and obtain the number of clusters ζ. The Louvain algorithm can obtain optimized ζ to maximize modularity. Thus, line 3

Algorithm 4 Core placement algorithm of GDCP

Input: $G = (V, E)$, X

1: **if** $t = 1$ **then**
2: divide G into ζ cluster by using Louvain algorithm
3: $w_x \Leftarrow \zeta$
4: **for** each $x \in X$ **do**
5: **for** *loopcounter* $= 1$ to w_x **do**
6: Place a duplication of core VNF x to the node that has the highest efficiency in the *loopcounter*-th cluster
7: **end for**
8: **end for**
9: **end if**

sets w_x, which is the number of duplications of the core VNF x, to ζ. Finally, these duplications are distributed to each cluster, and placed on the node having the highest efficiency in the cluster in line 6. For the placement of periphery VNFs on GDCP, Algorithm 3 is used.

2.4.2 A model for service-chain requests

In the simulation for evaluating the algorithms, service-chain requests are dynamically generated. Each request comprises k VNFs, which is the sum of the numbers of core VNFs, k_c, and periphery VNFs, k_p. Here, k_c and k_p are obtained using the Eqs. 2.5 and 2.6, respectively.

When the time slot t is incremented, λ new types of service-chain requests are generated. Both the source and destination nodes of each service-chain request are selected by using a uniform random. k_c core VNFs are selected from all $|X|$ types of core VNFs using a uniform random. Note that service-chain requests do not have duplicate VNFs.

2.4.3 Results

We perform simulations to evaluate the CLCP and GDCP. AaP [22] is used as a placement policy for comparison. In this section, to reveal the basic characteristics of each placement policy, we first perform simulations when the resources are infinite. Next, we consider the case in which only node resources are finite and become a bottleneck for accommodating service-chain requests. Finally, we show the simulation results when both the bandwidth resources and node resources are finite.

Results with no resource restriction

We use a 7×7 grid network as the physical network, where both the initial node resources $C_v(0)$ and bandwidth resources $B_e(0)$ are infinite. The number of VNFs comprising a service-chain request, that is, chain length k, is decided by using a uniform random from the range $[4, 8]$. When t is incremented, new $\lambda = 10$ service-chain requests are generated. Because the Louvain algorithm divide the 7×7 grid network into five clusters, we set w_x of both CLCP and CDCP to 5. The deployment cost, α, is decided using a uniform random from the range $[1, 1.2]$.

Figure 2.5 shows the deployment cost of each placement policy when CLCP and GDCP place $|X| = 500$ core VNFs in advance, which are used for accommodations at a frequency of $\gamma = 0.001$. When the resources are infinite, the deployment costs of the CLCP and GDCP are the same, and thus, both are indicated by the CLCP / GDCP line in the figure. At $t \leq 80$, the deployment cost of AaP is lower, but at $80 \leq t$, the deployment cost of CLCP / CDCP reduces below that of AaP. This is because CLCP and GDCP reduce the number of additional VNFs to be placed, by using the already placed core VNFs to accommodate new service-chain requests.

Placing more core VNFs in advance reduces the deployment costs of CLCP and GDCP. Figure 2.6 shows the deployment cost of each placement policy at $|X| = 700$; that is, CLCP and GDCP place more core VNFs in advance. The deployment costs of CLCP/GDCP at $t = 150$ are 12.76% less than those of AaP. This is because placing more core VNFs increases the opportunity to use them to accommodate a service-chain request and reduce the opportunity to use periphery VNFs. Moreover, the parameter γ affects the deployment costs of CLCP and GDCP. As γ increases, the core VNFs are used more frequently to accommodate the service-chain requests.

Table 2.2 shows the average hop count of paths used by each placement policy to accommodate the service-chain requests. The average hop count of CLCP is smaller than that of GDCP. CLCP places core VNFs at the center of a physical networ, which has, on average, a short hop count from any node. In addition, it tends to use shorter paths through nodes that deploy core VNFs as compared to GDCP. Note that AaP is the placement policy with the shortest hop count because it uses the shortest path from the source node to destination node, while CLCP and GDCP use a detour path.

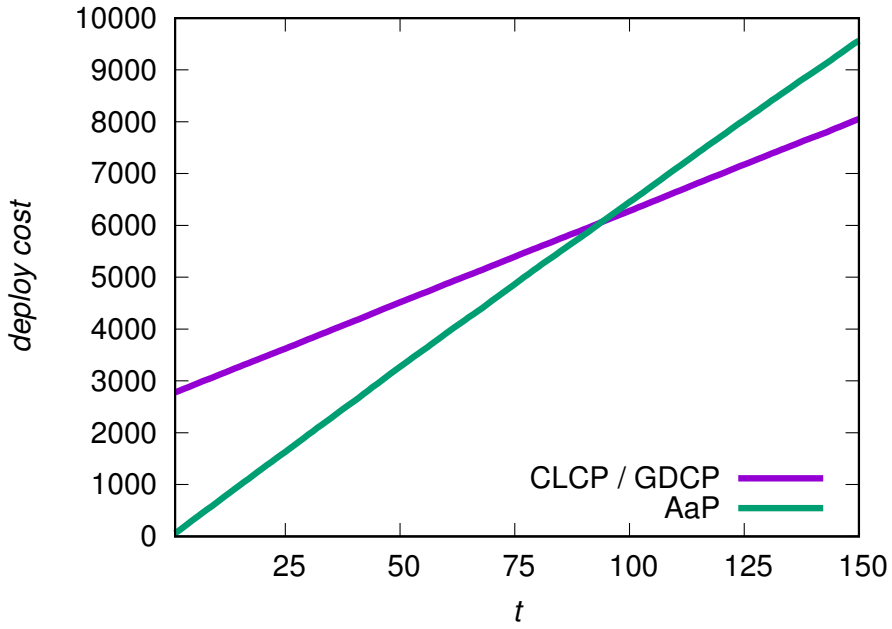

Figure 2.5: Deployment costs of each placement policy ($|X| = 500, \gamma = 0.001$)

Table 2.2: Average hop counts of paths used by each placement policy

setting	placement policy	average hop count
$\|X\| = 500, \gamma = 0.001$	CLCP	6.65
	GDCP	7.15
	AaP	6.50
$\|X\| = 500, \gamma = 0.0015$	CLCP	6.63
	GDCP	6.70
	AaP	6.49

Results with restrictions on computing and bandwidth resources

First, we consider a case in which only the node resources are finite and become a bottleneck to accommodate service-chain requests. A 7×7 grid network is again used for the physical network. We set the initial node resources $C_v(0)$ to 100 for each node $v \in V$ and the initial bandwidth resource $B_e(0)$ to infinity for each link $e \in E$. The node resource consumed by a placed VNF, $\hat{c}$, is decided using a uniform random from the range $[0.4, 1]$. The upper number of service-chain

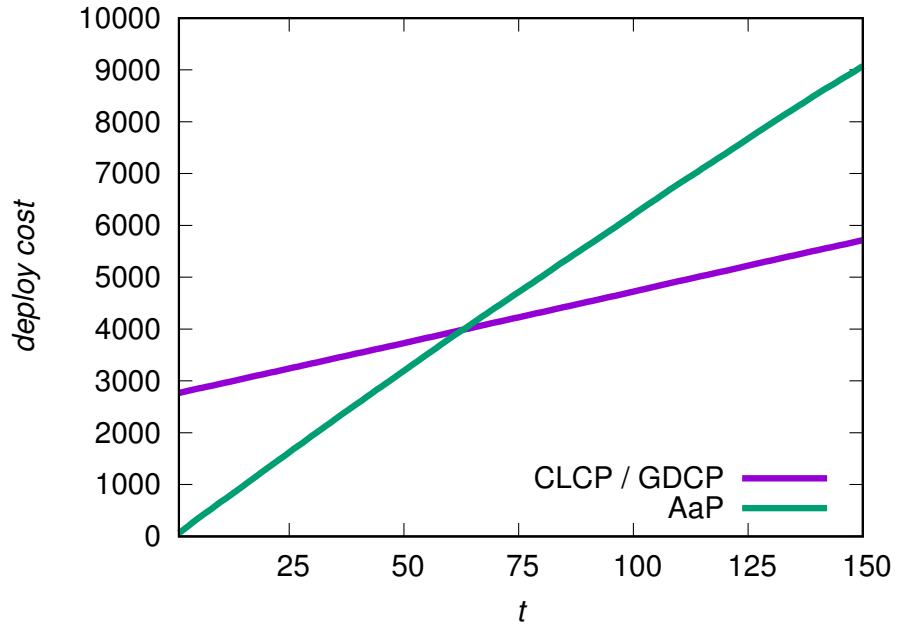

Figure 2.6: Deployment costs of each placement policy ($|X| = 700, \gamma = 0.001$)

requests that can use a VNF m placed at node v without processing overhead, $U_{m,v}(0)$, which is the maximum number of service-chain requests that can use a VNF m placed at node v, is set using a uniform random from the range $[4, 40]$. When the time slot t is incremented, $\lambda = 10$ service-chain requests are newly generated. The chain length, k, is decided using a uniform random from the range $[4, 8]$, and the deployment cost for each VNF, α, is decided using a uniform random from the range $[1, 1.2]$.

Figure 2.7 shows the deployment cost of each placement policy when $|X| = 500$, $\gamma = 0.001$ and $w_x = 5$. In the figure, the deployment costs per service-chain request accommodated by CLCP and GDCP decreases as t increases. This result indicates that CLCP and GDCP reduce the deployment cost by repeatedly using the core VNFs.

Note again that a reduction in the deployment costs appears when many service-chain requests are accommodated ($t \geq 90$). Otherwise, the initial costs to deploy the core VNFs are significant; the deployment cost of AaP at t = 1 is 6.82, whereas that of CLCP and GDCP is 278.69.

The deployment costs of GDCP are lower than those of CLCP, because they accommodate

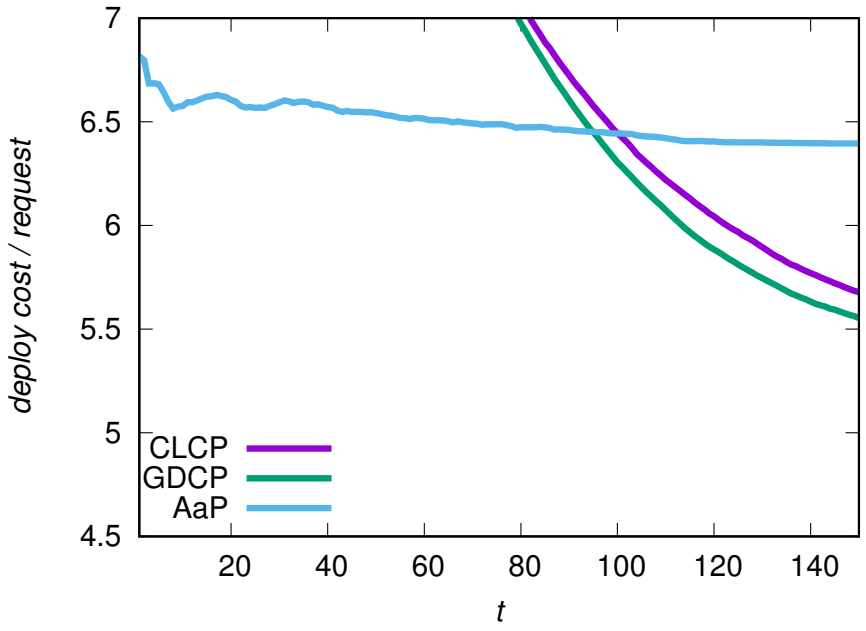

Figure 2.7: Deployment cost: ($C_v = 100, B_e$ is infinite, $|X| = 500, \gamma = 0.001, w_x = 5$

different numbers of service-chain requests due to the node resource restriction. CLCP places core VNFs only on nodes at the center of the physical network and intensively uses these nodes to accommodate service-chain requests; thus, node resource restrictions are likely to occur. In contrast, GDCP distributes core VNFs on the physical network, and thus, can use geographically distributed nodes and accommodate more service-chain requests than CLCP. This can be observed from Figure 2.8, which shows the amount of node resources consumed by the placed VNFs per accommodated service-chain request.

Next, we consider the case in which both the bandwidth resources and node resources are finite. Other settings are the same as those in the case where only the node resources are finite. Figure 2.9 shows the deployment cost of each placement policy. We set B_e to 500, keeping all other parameters same as those in Fig. 2.7. Looking at $t = 150$ in Figure 2.9, the deployment cost per service-chain request by GDCP exhibits the lowest value. The deployment cost of CLCP is 12.99% larger than that in Figure 2.7, while that of GDCP is only 4.26% larger. Figure 2.10 shows the amount of node resources consumed by the placed VNFs per accommodated service-chain request. In CLCP,

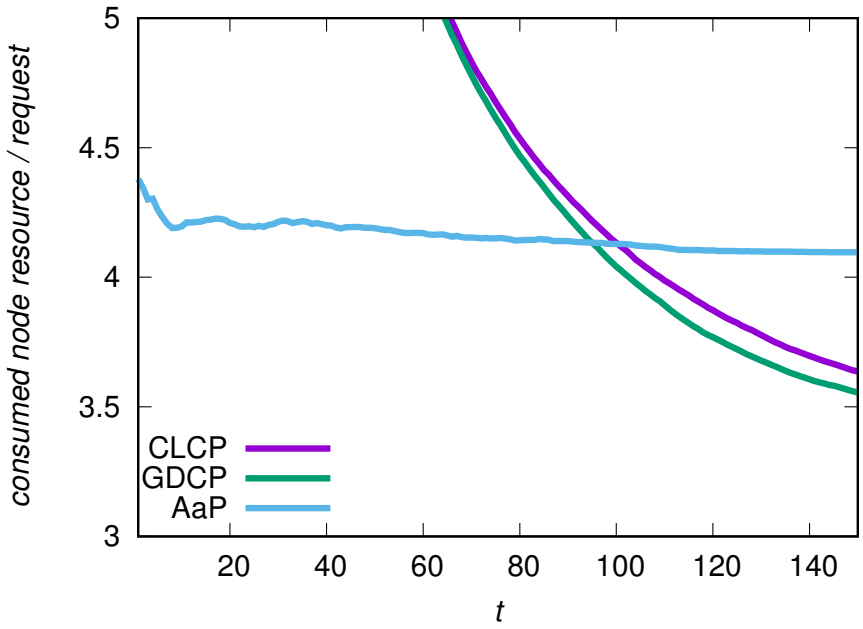

Figure 2.8: Amount of node resources consumed by the placed VNFs: $C_v = 100, B_e$ is infinite, $|X| = 500, \gamma = 0.001, w_x = 5$

nodes at the geographic center of the physical network are intensively used; thus, the links that can reach these nodes are also intensively used. As compared to CLCP, GDCP can use geographically distributed nodes and incurs fewer bandwidth resource restrictions. We have conducted simulations on 9×9 grid network, 49-node BA networks, and 40-node ternary trees, which are not shown here. Similar tendencies are observed for AaP/CLCP/GDCP.

Our results show that GDCP mostly reduces the deployment costs for CPBD when there are many service-chain requests. As CPBD also reduces the development costs, the core/periphery-based software design and distributed placement are suitable for NFV systems for accommodating future service-chain requests.

2.5 Conclusion

In this chapter, we investigated the software design and placement method of VNFs to reduce the long-term development and deployment costs against the change in service requests. We first

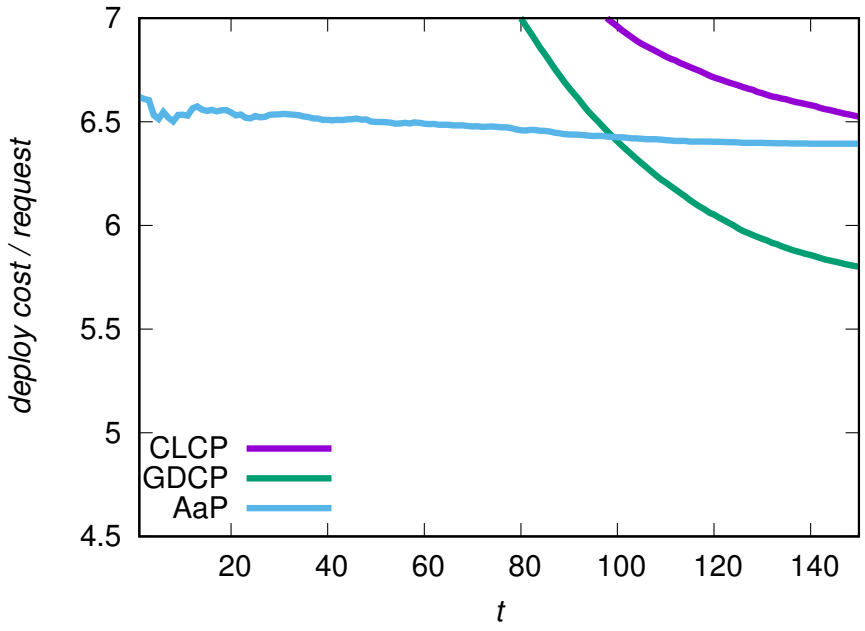

Figure 2.9: Deployment cost: $C_v = 100, B_e = 500, |X| = 500, \gamma = 0.001, w_x = 5$

considered designing an NFV system based on a core/periphery structure, and repeatedly used core VNFs to accommodate future service requests. Our evaluation results indicated that such software design based on the core/periphery structure can accommodate service-chain requests with lower development costs than that without core VNFs. Moreover, we investigated where to place the core VNFs in the physical network by examining CLCP and GDCP. Our results showed that GDCP is the best placement policy that can accommodate many service-chain requests with low deployment cost, and the difference between GDCP and CLCP is significant when there are resource constraints on nodes and/or links.

In this thesis, the incremental development of VNFs was considered. However, in reality, some of core VNFs would be no longer necessary as the time proceeds, because of the changes in service-chain requests. One of the future works is to consider removing the not required VNFs from the nodes.

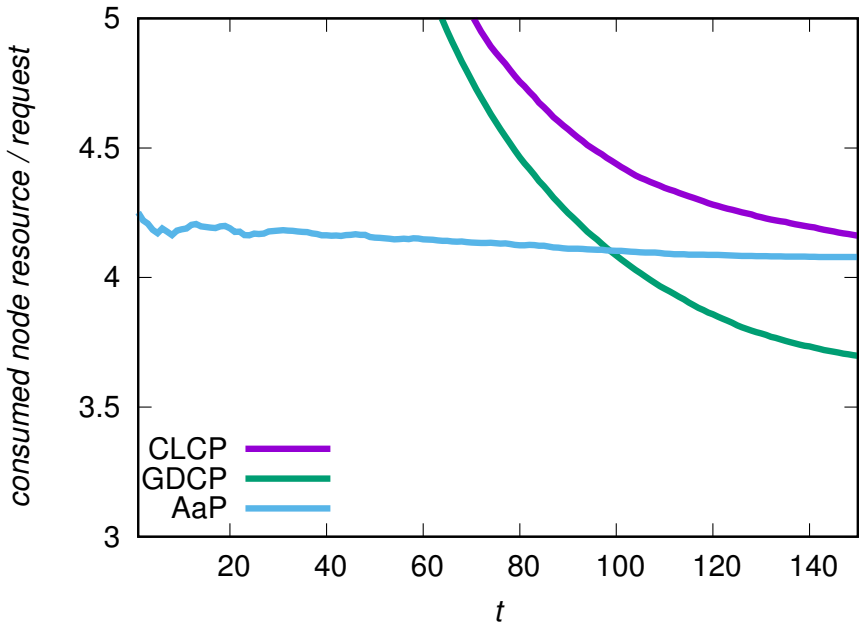

Figure 2.10: Amount of node resources consumed by the placed VNFs: $C_v = 100, B_e = 500, |X| = 500, \gamma = 0.001, w_x = 5$

Chapter 3

Design, Implementation and Evaluation of Core/Periphery-based Network-oriented Mixed Reality Services

3.1 Introduction

Because many new network-oriented services have developed to meet various user requests, it is important to consider service designs that can accommodate as many services as possible when deploying network services in a MEC environment. However, implementation costs increase if developers must reconstruct entire services to meet different user requests or to adapt to environmental variation such as device evolution. Moreover, MEC environment resources are not necessarily the same as those in a cloud computing environment. Specifically, MEC environment resources are limited by spatial restrictions, making it difficult to locate all possible services, such those on the edge that can adapt to each user request and environmental variation. It is therefore necessary to consider service structures that can change service behaviors in a flexible manner. Service function placement in MEC environments has been studied in, for example, [11, 12], but most of them

correspond to user mobility. We consider a service design where the developers can modify or add service functions in a flexible manner with less cost against changes of real environment and user requirements.

The advantages of a core/periphery structure [8, 9] for accommodating information services, represented by chains of functions, were numerically investigated in our previous work [13]. From the biological point of view, the core/periphery structure is expected to achieve an efficient and adaptive behavior against environmental changes. However, from the service design point of view, dividing functions and placing them in different devices creates extra communication paths. This degrades service responsiveness and cannot be ignored. Therefore, when we apply the core/periphery structure to the service design, we need to consider the balance between the penalty and the reduction of development costs. In this chapter, we show that a core/periphery structure allows services to adapt to increasing the number of device types with low implementation cost, and evaluate the actual penalty of locating core functions on edge servers with regards to the service responsiveness through a service implementation. Unlike model-based evaluations, we implement a network-oriented MR service based on a core/periphery structure using actual MR devices and robots. Our implementation focuses on a shopping service, but service design based on a core/periphery structure is not limited to the shopping service and can be applied to other network services.

When designing services based on a core/periphery structure, it is necessary to consider which functions should be implemented as core units and which should be implemented as periphery units. We first investigate what kinds of functions would be required in a shopping service. To utilize the flexibility of a core/periphery structure, we regard as core functions those whose behaviors remain unchanged under changes to user requests or the real-world environment, and peripheral functions as those whose behaviors can change under such circumstances. In this way, core functions allow adaptation to the emergence of new services by adding or changing some peripheral functions instead of recreating entire services. We next evaluate the design of a service based on a core/periphery structure in terms of implementation cost and service responsiveness. The results shows that as compared to a conventionally designed service, the implementation cost for adding new functions of a service design based on a core/periphery structure is reduced without increasing service responsibility. We close with a summary of the advantages of service design based on a

core/periphery structure, which are not numerically verified but are experienced through the service implementation.

3.2 Current and Future Network-oriented Mixed Reality Services

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

3.2.1 Current Services

Telexistence services have been actively developed in recent years, and momentum for their social implementation has been rising. Telexistence aims at allowing people to feel as if they are actually at a remote place. TELESAR V [1] is a telexistence master–slave system allowing users to feel present in a remote environment by transmitting not only video and audio, but also haptic sensations. ANA Avatar [33] is conceived as a new mode of instantaneous transportation allowing users to communicate and work as if actually present in remote places, using robotics and technologies for sending tactile sensations and allowing remote robot operations. ANA has begun testing via the ANA Avatar Museum, which allows users to view a remote aquarium, and ANA Avatar Fishing, through which users can remotely fish. A telexistence application using drones is also being developed [34].

Existing conventional MR services implement service functions targeted for specific devices and specific functionality. Flexible service development is necessary to easily adapt to changes in users' requests on the real-world side, such as future development of devices and shifting locations. Moreover, users of a MR service send related information to the centralized server, and the information is processed, and then, the results are sent to the remote devices. Typically, the centralized server is located on the cloud in services such as the above-mentioned ANA Avatar. Conceptually, the centralized server can be a remote device; users directly communicate with remote devices. By developing service functions with design of a core/periphery structure, less part of the program code needs to be modified. In addition, service functions can be placed separately on cloud/edge servers, users' devices, and remote devices. More importantly, placing the service functions on edge servers

has a potential to reduce application-level delay. Note, however, that separating service functions into core and periphery may lead to increased overhead at the function's processing delay and implementation cost. Therefore, it is necessary to design and implement the actual MR services by the concept of a core/periphery structure, and then measure the increased overhead.

VRPN (Virtual-Reality Peripheral Network) [35], which is used for developing VR services, is similar to the concept of core/periphery structure since it absorbs the difference of VR devices. We use an MQTT (Message Queuing Telemetry Transport), which is a messaging protocol to absorb the differences between devices as a core function.

In recent years, there is a concept of dividing and combining service functions such as microservices [36] and service-oriented architecture (SOA) [37]. In these architectures, each function is loosely connected and adaptable to changing demands, but because the functions are finely divided, development costs are higher to support various input/output. These architectures are called "No Core" because they are considered as service structures with only peripheral functions.

3.2.2 Future Services

Sixth-generation (6G) networks will allow development of services using technologies that would be difficult to support over fifth-generation (5G) networks. Within ten years, current remote interaction technologies will become obsolete, and new forms of interaction such as holographic and five-sensory communication will allow immersion in remote places [38]. Tactile Internet and full-sensory digital reality can be realized by 6G networks [39]. It has also been suggested that 6G networks will further support underwater and space communications, allowing deep-sea sightseeing and space travel [39].

Application-level delay, which significantly increases in cloud computing environments with communication distances and load concentrations, will be a significant factor for service quality in these applications [2]. MEC is therefore expected to be further standardized [2–4]. In edge computing, computing resources and storage are allocated at the network edge, so that processing required by end devices is performed at closer sites. This improves the responsiveness of applications by

shortening communication distances and optimizing load distributions. Our research group demonstrated that MEC environments improve the service quality of network-oriented MR services [40]. The ETSI Industry Specification Group [28] suggests video content delivery, video stream analysis, and Augmented Reality (AR) as key use cases for MEC, and suggests guidelines for software developers.

Current mainstream services include audio and video transmission, but realizing transmission of information for the five senses will require construction of service systems that can handle multiple inputs and outputs. In this thesis, we propose guidelines for service function placement in a core/periphery structure, a biological model for flexibly and efficiently processing information.

3.3 Service Design Based on a Core/Periphery Structure

This section describes a service design based on a core/periphery structure.

3.3.1 Supposed Service for Network-oriented Mixed Reality

We consider a shopping service using MR and robots. Robots are placed in an actual store to allow users to shop from home as if they were actually there. Robots provide a video feed while moving about the store under user operations. Real-world information on the store side is added to videos delivered to users. Users can move robots with gamepads, gestures, or gaze. Figure 3.1 shows an overview of the shopping service and its functions.

Robot-side applications provide functions for moving, taking video, processing images, collecting and aggregating information around the robot, and adjusting movement speed so as not to collide with people or objects. User-side applications provide functions for displaying video, sending instructions to the robot, collecting and aggregating information around users, and detecting objects at a user-defined granularity.

3.3.2 Service Decomposition Based on a Core/Periphery Structure

To design the service described in Sec. 3.3 based on a core/periphery structure, we divide the service functions into core and peripheral functions. This section discusses core and peripheral functions

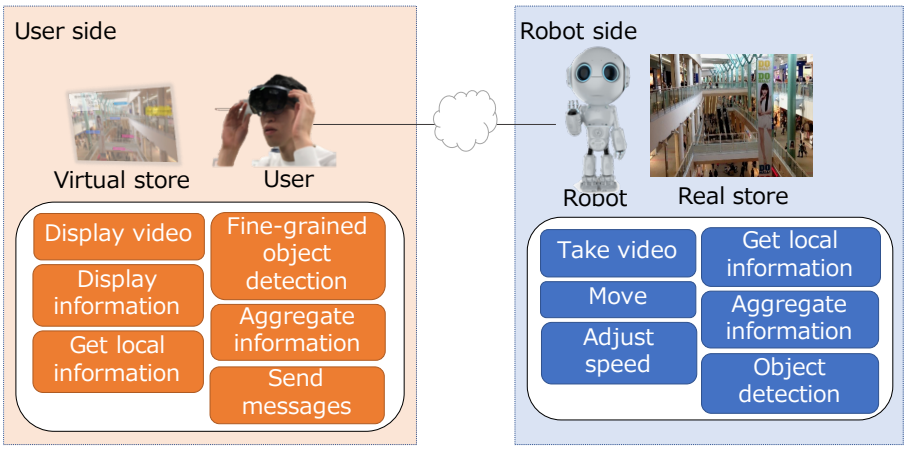

Figure 3.1: The presumed service and its functions.

for video transfer and robot operations, and explains the process at each function in detail.

The service function provides a functionality to services. The functionality ranges from primitive functionality to auxiliary functionality or specific functionality. A service is then provided by selecting a set of service functions and by combining/chaining the service functions over the network. Note that the number of possible services drastically increases as the number of functions increases, which makes deployment cost being low by numerical examples in our previous work [13].

In this thesis, we investigate the effectiveness of core/periphery design using the actual implementation of MR services. However, because we cannot design and implement all of the possible (MR) services, we select three services as service scenarios, and design/implement them by using functions for video transfer (Sec. 3.3.2) and for robot operation (Sec. 3.3.2).

Video Transfer

Functions for video transfer provide video capture and output, perform object detection, and distribute video to users. For video transfer, we consider three functions depending on user requests, devices, and real-world environments. First function is the video I/O. When a new camera or device is developed, the performance of the camera capturing the video may not match the performance of users' devices. In that case, the function to change the rate and resolution of the video is needed.

Table 3.1: Service functions for video transfer.

Function	User requests	Processing
Video I/O	Real time High-resolution	Change rates Change resolution
Object detection	Fast but coarse Slow but fine	Choose methods
Video distribution	To one user	UDP-based protocol
	On a large scale	TCP-based protocol

Users can also change the video resolution and rate depend on the network environment they are placed in. Second function is object detection. Users switch object detection methods appropriate to the location of the robot or information the users want. For example, in a shopping service, when a robot is moving through the halls of a shopping mall or window shopping by walking a street, users may select fast but coarse-grained method, and when users want to know the detailed classification of a product in a specific store, they select slow but fine-grained method. Fine-grained object detection is supposed to be used at the user side to provide detailed information about the object based on the user's preference or intention. Also, when new object detection methods are developed, service developers implement additional functions to support them. Third function is the video distribution. A video transfer service which requires real-time video transmission from robots to users may use UDP and mpeg-ts, and other video transfer service may use HLS (HTTP Live Streaming) for transmitting video to multiple users. HLS is a live streaming protocol using HTTP, which allows video delivery/playback to be executed on a web server/client. HLS can distribute video to multiple users at the same time, but HLS transfers the chunk of video with TCP, which makes the delay larger. Table 3.1 summarizes the functions for video processing that are used to realize service scenarios in Section 3.3.3.

We decompose the service into functions, and consider which of those described in Sec. 3.3.1 are core functions and which are peripheral functions, based on the concept of a core/periphery structure in which core functions processes information more efficiently, while periphery functions have various configurations for flexibly adapting to environmental changes.

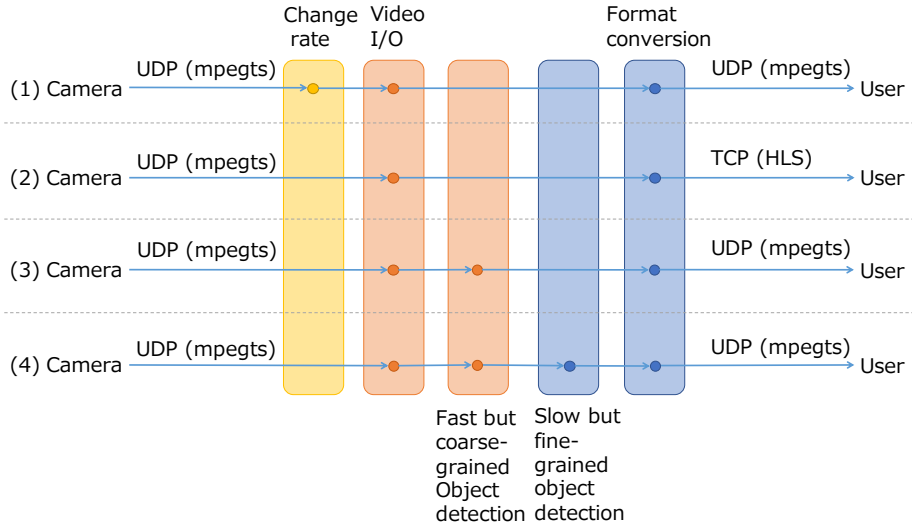

Figure 3.2: Examples of processing in video transfer. (1) Video providers change video frame or bit rates. (2) Video providers distribute video to multiple users. (3) Object detection with only a standard part is executed. (4) Object detection with a new part is executed.

Figure 3.2 shows examples of video transfer processing. When video providers want to change the video frame or bit rates to adapt to the amount of available resources, the video is processed before input. Users too can change the frame or bit rate. In that case, video is processed after output. Protocols and the video format can be changed at the video providers' request. For example, video providers use the UDP-based transfer protocol to send video to a single user, and TCP-based protocols such as HTTP otherwise. When users want to know what is in the video, object detection is executed. There are various object detection methods, such as YOLOv3 [41], which is fast and widely used, and Mask R-CNN [42], which provides more detail but is slow. Users can adopt their preferred method. Orange functions in Fig. 3.2 are common core functions, while light orange and blue functions are peripheral functions.

Figure 3.3 shows the core/periphery structure for video transfer, with orange fields indicating core functions, light orange fields indicating camera-side periphery functions, and blue fields indicating user-side periphery functions. Video is sent from the camera, whose frame and bit rates are adjusted based on provider requests as a peripheral function. The video then passes through

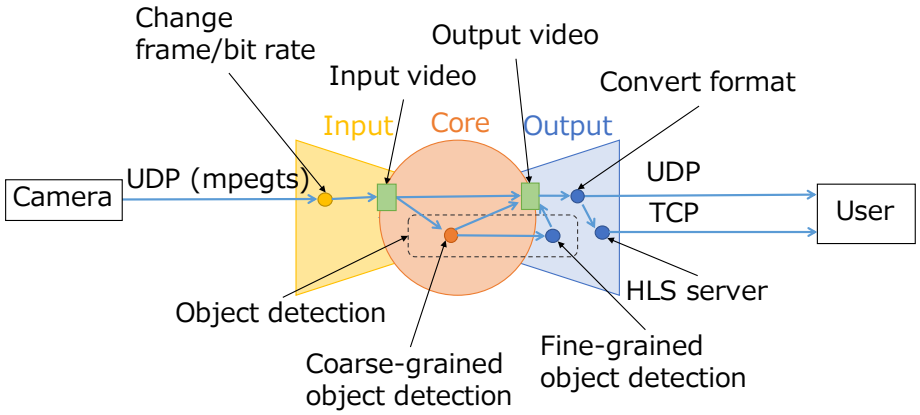

Figure 3.3: Video transfer based on a core/periphery structure.

core functions, including those for inputting video, outputting video, and the standard part of object detection. Finally, the video format and distribution protocol are selected and sent to users. By utilizing the flexibility of a core/periphery structure, all developers have to do is remake or add peripheral functions for adapting to different user requests, changes in the real-world environment where devices are placed, or device evolution.

Robot Operation

Robot operations provide functions for recognizing user actions, sending messages from users, accessing APIs, adjusting robot speeds to avoid obstacles, and collecting and aggregating information obtained from robots. For robot operations, we consider three functions depending on user requests, devices, and real-world environments. First function is command interfaces based on the users' devices, which includes either separately or in combination of gamepads, gestures, and gaze. Users select how to operate remote devices depending on the users' device type and its specification. In the future, as new command interfaces or devices are developed, new functions to use the new devices are developed and provided to users. Second function is the selection of a remote device to operate. Users select the remote devices e.g. robots and drones to operate based on the remote environment or users' requests. The APIs used in the service are switched accordingly. When new remote devices are developed, users can use the new remote devices. Third function

Table 3.2: Variations of service for robot operations.

Function	User requests/ Real-world environments	Variations of service
Command interface	Use gestures Use gamepad Use gaze	Change interface
Device to control	Operate robots Operate drones	Switch/Add APIs
Adjust robot speed	Some obstacles Slippy	Move robot slowly
	No obstacles	Move robot speedily

is related to adapting changes in the real environment in which the robot is located. For example, when the robot is located in a crowded area, it moves slower, and when it is in a large area, it moves faster. Table 3.2 summarizes the functions for robot operation variation of service on different user requests/real-world environments for requests for robot operations.

We decompose the service into functions and determined core functions as in Sec. 3.3.2. Figure 3.4 shows examples of processing for robot operations. When users operate a robot with gamepads or gestures, their device recognizes instructions and send messages based on the selected method. When users operate different devices such drones, service behavior after receiving user messages will change to access the robot or drone's API. Furthermore, robot speeds are adjusted based on the surrounding environment. When there are no obstacles or crowds, users can speedily move robots. Otherwise, robots slow down to avoid collisions.

The function for send messages in robot operations is a common function, and therefore should be divided as a core function, rather than the whole service being performed as an all-in-one function. Figure 3.5 shows the core/periphery structure for robot operations. Functions for sending instruction messages from users and aggregating information obtained from robots are common, so they are core functions. Functions for adapting to user requests and changes in the real-world environment, such as how to input user instructions, are peripheral functions. Functions for accessing robot APIs, collecting information such as the current robot position and adjusting movement speed

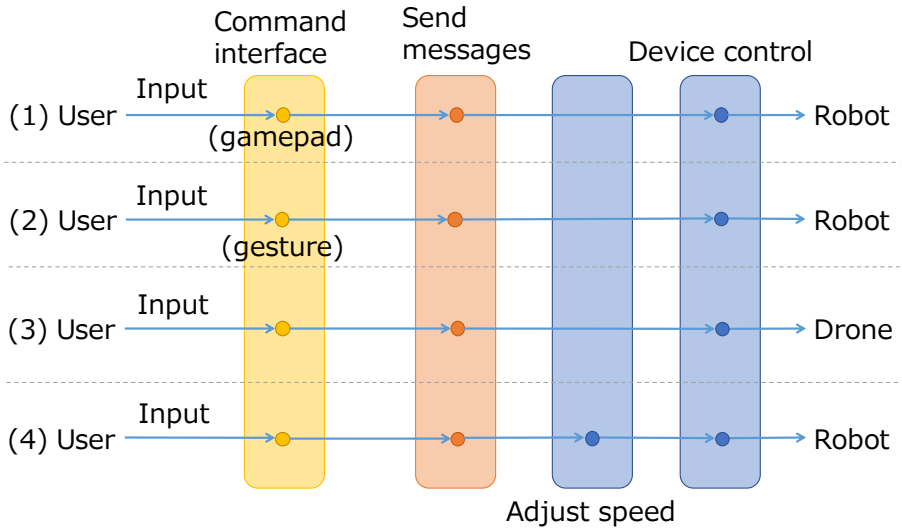

Figure 3.4: Examples of processing for robot operations.(1) User operates a robot with gamepad. (2) User operates a robot with gestures. (3) User operates a drone. (4) Robot speed adjusted based on the environment.

are peripheral functions, because they change according to device type and real-world environment. Flexibility of a core/periphery structure allows developers to simply remake or add peripheral functions to adapt to varying user requests, environmental changes, and device evolution.

3.3.3 Service Scenarios

We prepare two service scenarios for implementation. Note that we use a robot, Pepper [43], for the implementation and did not use the Drone devices. The applicability to the Drone and other devices is discussed in Section 3.4.1. In the first, we modify robot behavior according to its real-world environment. This scenario realizes communication between robots' peripheral functions for adjusting speed and core functions related to robots, object detection, and messaging. In the second scenario, we modify behavior of a user application based on the user's real-world environment. This scenario realizes communication between user-side peripheral functions for displaying information and core functions related to users, information aggregation, and messaging.

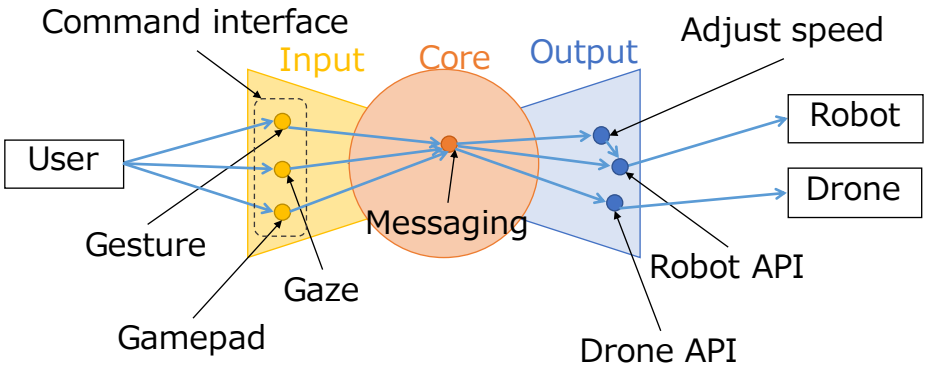

Figure 3.5: Robot operation based on a core/periphery structure.

Behavior Based on the Real-world Robot Environment

The following describes a scenario in which robots modify their behavior based their real-world environment. Functions for robot operation and core/periphery functions are as follows:

- Core: Functions for transmitting instructions from the user to the robot and functions for object detection.
- Periphery: Functions for obtaining information near the robot, adjusting the robot movement speed, and aggregating information sent from multiple robots.

Figure 3.6 shows this scenario. There are users with MR headsets, robots, cameras, and edge servers on robot side. Orange functions are core functions. Blue functions are peripheral functions on robot side, and light orange functions are peripheral functions on user side. Users send instructions to robots, moving their bodies and heads by gamepads, gestures, and gaze. Robots can detect nearby obstacles and stop using sensors. Video captured by robot-mounted cameras are sent to the edge servers, which perform object detection to recognize objects and persons around the robots. Object detection, a core function, needs to be performed in real time and requires powerful servers. These functions should thus be deployed on edge servers, not on end devices. The results of object detection are returned to robots. For example, when robots know that there are many people around them, they can reduce speed to avoid collisions. Moreover, information from robots can be

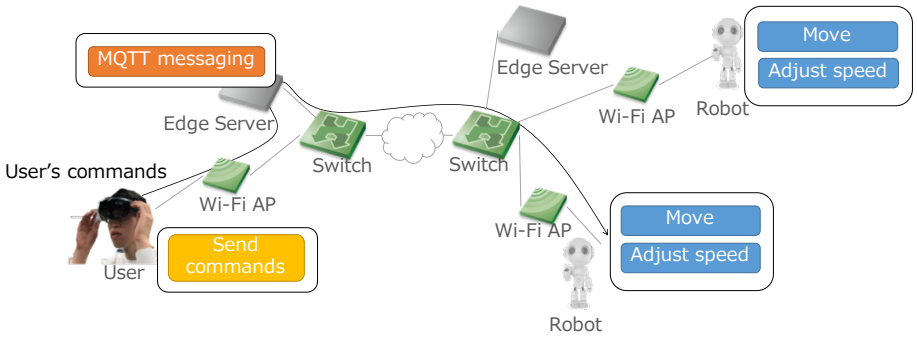

Figure 3.6: Robot operation

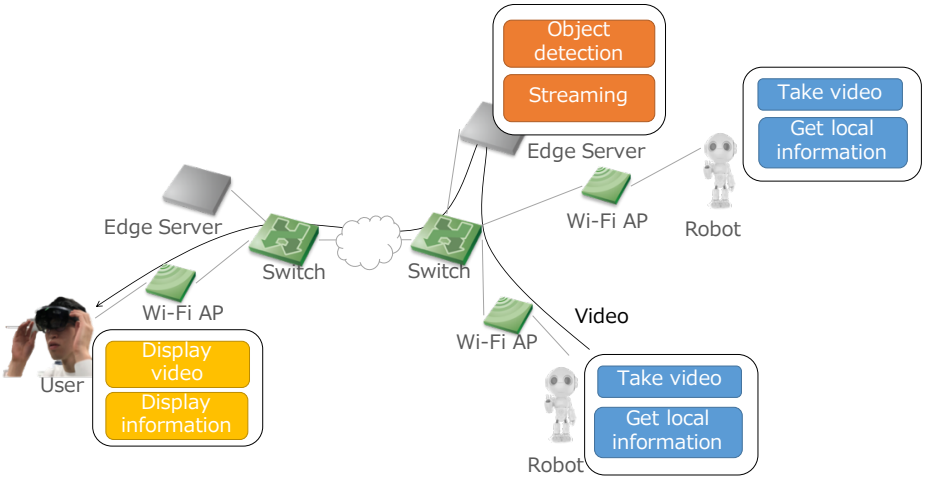

Figure 3.7: Video processing

aggregated on edge servers and shared with other robots for collision avoidance and the like.

Behavior Based on the Real-world User Environment

The following describes a scenario in which behavior is based on the real-world user environment. Core/periphery functions are as follows:

- Core: Functions for sending user instructions to robots and for aggregating information from multiple robots.
- Periphery: Functions for displaying video, detailed object information, and information about each robot.

Figure 3.7 shows this scenario. As a core function, store and robot information such as product information or communication status is collected at user-side edge severs. Users select which robot to operate only by communicating with an edge server while viewing aggregated information about stores and robots. Video sent from cameras is roughly classified by object type on robot-side edge servers. These functions perform real-time image processing and information aggregation, and thus are inappropriate for execution on end devices. To improve responsiveness, core functions should be performed on edge servers instead of the cloud. Then, detailed object detection is performed as a peripheral function on a user-side edge server. User devices collect personal information such as user tastes, what the user already owns, and purchase history, and this information is aggregated on an edge server. Using this personal information, the system can display content most appropriate for the user. For example, the application can recommend commodities based on previous frequent purchases, or can warn users of impending expiration dates for food.

3.4 Implementation and Evaluation of a Service Based on a Core/Periphery Structure

This section describes implementation details and evaluates the service based on Section 3.3.3.

3.4.1 Implementation of a Service Based on a Core/Periphery Structure

This section describes the implementation of our service. Although there is techniques such as inheritance in object-oriented languages as an implementation that is based on the concept of a core/periphery structure, in this thesis, we implemented the core and peripheral functions as separate programs to locate core functions on edge servers and peripheral functions on robots or MR headsets. We implemented the service using only HoloLens devices on the user side and only robots on the remote side.

Video Transfer

Video from cameras is sent to a robot-side edge server. Video is captured using OpenCV [44], then object detection is performed using a PyTorch implementation of YOLO v3 [41]. For video processing, mask R-CNN (Region-based Convolutional Neural Networks) [42], an algorithm that surrounds detected objects with a rectangle and recognizes the object type for each pixel and colors it accordingly, can be used. The processed video is transmitted via UDP using ffmpeg [45] to HoloLens [46], an MR headset worn by users, for display. HoloLens is a standalone head-mounted computer made by Microsoft that displays holograms and recognizes user gaze and gestures to provide a MR experience.

Robot Operation

HoloLens controller information is transmitted via Message Queuing Telemetry Transport (MQTT), a publish/subscribe-type protocol developed for frequent message exchange between IoT devices. Users use an Xbox controller that can connect to HoloLens. The available operations are as follows;

- Xbox controller: moving forward, backward, left, and right, with left stick, rotation with L button and R button, and resetting robot's neck rotation with X button
- gesture: moving forward, backward, left, and right with dragging, and rotation with holding
- gaze: robot's neck rotation synchronized with HoloLens direction

Because we use the MQTT for the core function, some of the robot operations here are easily extended to other devices such as Drone [47]. Note again that we selected HoloLens and Pepper to evaluate the amount of the source code and to measure the application-level delay. The MQTT broker receives controller commands via HoloLens and sends them to a program running on the robot. The robot is a Pepper [43] running a program developed using the programming tool Choregraphe. This program converts messages from the MQTT broker to the Pepper API.

Figure 3.8 shows a screenshot of the HoloLens application. Users can see video with the object detection results and a map made by Pepper displayed at the top left. The green dot represents Pepper's position.

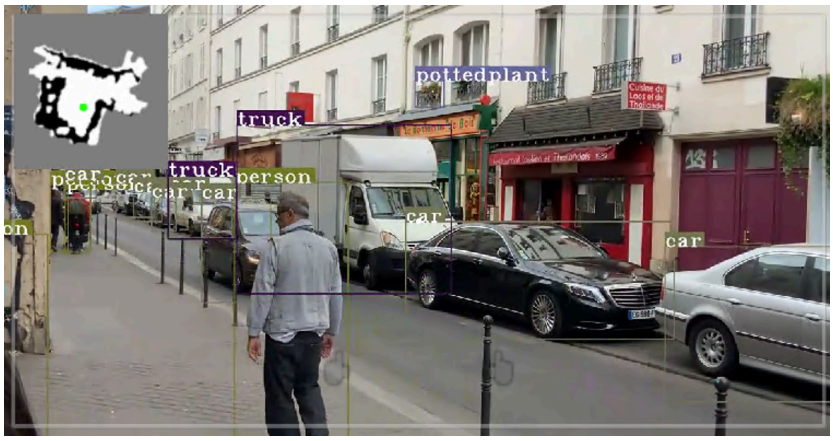

Figure 3.8: Screenshot of the HoloLens application.

3.4.2 Evaluation Metrics and Measurement

This subsection describes the evaluation metrics, namely implementation cost and service responsiveness, and how we measure those metrics.

Implementation Cost

Using the implemented service, we show that adopting a core/periphery structure lowers implementation costs.

We evaluate the number of lines of source code as the implementation cost, comparing source code size when the service is designed based on a core/periphery structure with the case where the service is not designed based on a core/periphery structure and implemented on an end device. Actually, the number of program code lines highly depends on a programming manner. However, because we implemented functions with the same programming manner, and program code mainly consists of essential codes to prepare/handle API calls for each device, we use the number

of program code lines for comparison. Comparisons between other programming manners may be possible with more implementations of MR services, but is left for our future work.

While knowledge and preparation of the development environment is also part of the implementation cost, such factors are difficult to numerically evaluate. Section 3.5 describes these and other lessons regarding service implementation.

Service Responsiveness

Because sending user instructions via an edge server can increase application-level delay compared with the case of directly sending instructions to robots, we measure and evaluate application-level delay as a penalty for using edge servers.

We measure times from when the HoloLens application publishes a message to return of robot sensor data to HoloLens directly, and through the edge server. Then, we compare these times to evaluate the effect of allocating core functions on an edge server. We regularly sent messages about 20 times from the HoloLens application and saved each message return time as $t_1, t_2, \ldots, t_{20}$. We also record times when Pepper returned sensor data as $t'_1, t'_2, \ldots, t'_{20}$ in the HoloLens application. Then, we calculate the average of $t'_1 - t_1, t'_2 - t_2, \ldots, t'_{20} - t_{20}$ as the application-level delay.

Application-level delay is a one-way delay. However, since there are different system clocks between HoloLens and Pepper, accurate comparison of one-way delay is difficult. We therefore measure round-trip delay.

We construct a MEC environment using OpenStack located in Osaka.

3.4.3 Results

Implementation Cost

Figure 3.10 shows the relation between the number of device types at remote sites and the number of lines of source code for the connection establishment part (Fig. 3.10(a)) and for the messaging part (Fig. 3.10(b)). We omit the complete source code due to space limitations, but it is available at our GitHub repository [48]. The "Direct" represents the design not based on a core/periphery structure, and Core/Periphery represents the design based on a core/periphery structure. Solid lines

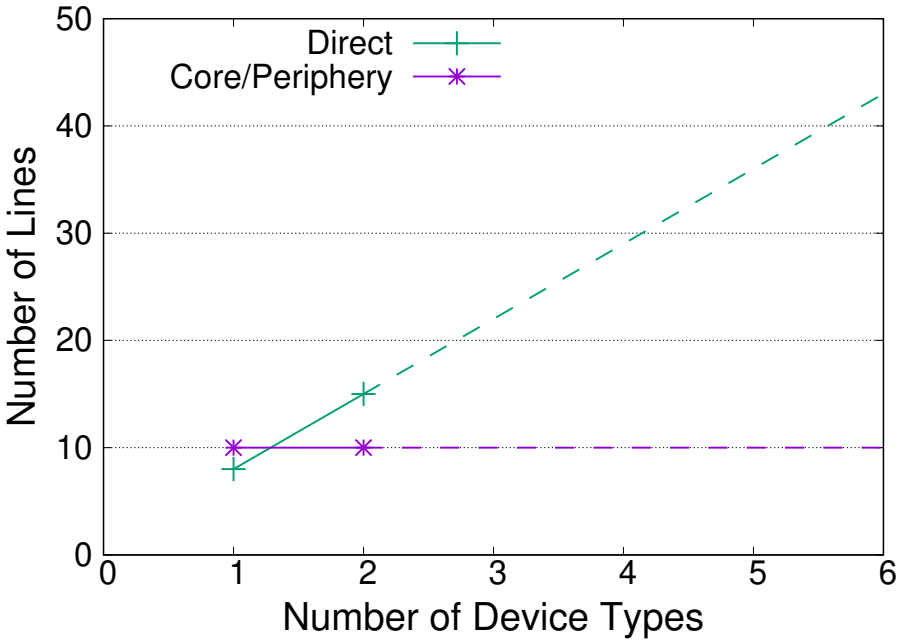

Figure 3.9: Connection establishment part

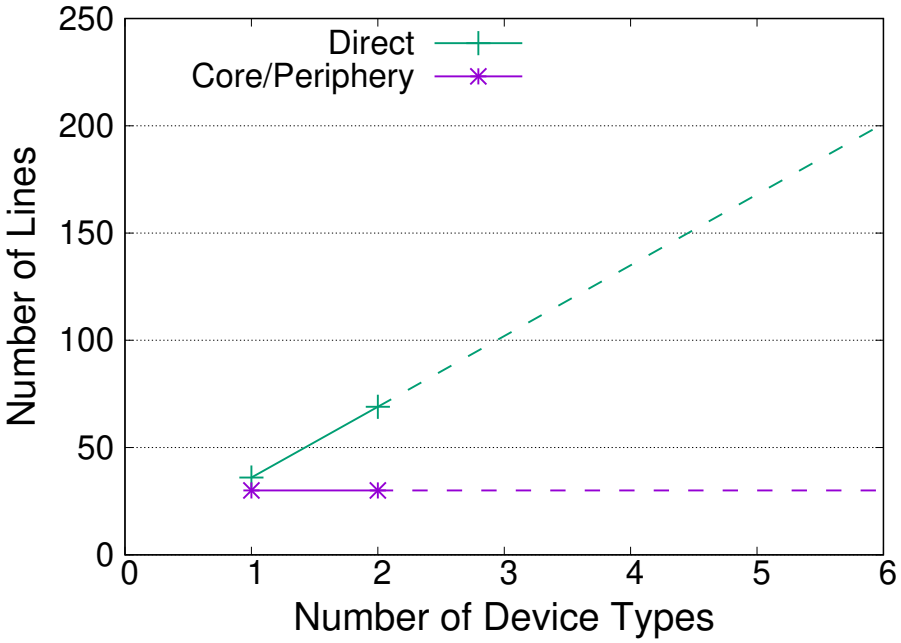

Figure 3.10: Messaging part

in the figure represent the number of lines for two robot types, a Pepper and a presumed robot, and dashed lines represent the number of lines when using more than two robot types. We have not implemented the application with more than two robots, but predict that the number of lines will linearly increase because applications not based on a core/periphery structure require source code for establishing connections and messaging for each device API, resulting in a constant additional number of IF statements for each.

Figure 3.10 shows the effect of a design based on a core/periphery structure increases as the number of device types increases. Note that when a single type of device or a single type of service is implemented, the design based on a core/periphery structure is less effective. When additional type of remote devices are deployed for the service, developers need to prepare service functions, i.e., write code, to establish connections and operations for the remote devices. Writing the code is necessary for both the design based on a core/periphery structure and the design not based on a core/periphery structure. However, in the application designed not based on a core/periphery structure, the amount of source code increases linearly against the increase of remote device since the service functions are dependent each other. In the application designed based on a core/periphery structure, developers can reuse these functions as core functions, and the amount of code is constant or increases marginally.

We considered both variation of devices at remote sites and variation of user-side devices. Both in services based on a core/periphery structure and those not based on this structure, developers must add source code for obtaining controller information, because this is a peripheral function. However, increasing the number of controller types also increases the number of source code parts to be added on the remote side, an effect that is mitigated by designing services based on a core/periphery structure. Therefore, developers can implement applications more easily by adopting a core/periphery structure.

Service Responsiveness

Table 3.3 shows average, maximum, and minimum values for application-level delay, along with ping round-trip time (RTT) when the HoloLens application directly connects to the Pepper and

Table 3.3: Results of experiments measuring penalty of using an edge server.

	Direct	Core on Edge
Average [ms]	21	52
Max [ms]	24	263
Min [ms]	0	18
Ping RTT [ms]	-	1

when the HoloLens application connects to Pepper via edge servers. As shown in Table 3.3, the application-level delay in the case of Direct was 21 ms, and that in the case of Core on Edge was 52 ms. The difference between the two shows that the penalty for locating the core function, MQTT, on the edge server is 31 ms. The application-level delay in the case of Core on Edge is 52 ms, and the 52 ms delay is tolerable because humans' reaction time is around 190 ms for light stimuli. The results show that application-level delay when using MQTT on an edge server is about 52 ms. A 52 ms delay is tolerable because humans' reaction time is around 190 ms for light stimuli [49–53]. In combination with the results presented in Section 4.3.1, therefore, a service design based on a core/periphery structure reduces implementation costs without significantly deteriorating service responsiveness. Note that application-level delay when core functions are placed on the cloud[1] was 626 ms, which exceeds the tolerable delay due to the round-trip time. However, the penalty of separating service functions into core and periphery is the same as the "Core on Edge".

3.5 Lessons from Service Implementation

This section presents lessons learned from service implementation based on a core/periphery structure, including factors that cannot be numerically represented.

First, developers do not need to consider device APIs and specifications. When a service is not based on a core/periphery structure, functions are not divided and user-side devices directly establish connections with remote devices. Developers need to know the APIs of many remote devices to write many parts of source code, including how to establish connections, how to move

[1] AWS (Amazon Web Services) cloud host in Ohio, USA.

devices at remote sites, and the parameter settings such as the sensitivity to user operations, which are depend on the speed and other features of the remote device. By dividing functions based on a core/periphery structure, user-side developers need to know only user-side device APIs, and do not need to consider remote device APIs.

Furthermore, adopting a core/periphery structure absorbs differences in development environments. We implement the service using Unity. A 32-bit version of Unity is required to directly use the Pepper API from a user-side application, but 32-bit versions of Unity are no longer being developed. To develop for Pepper, therefore, we must use an old version of Unity and modify the source code as appropriate. When developing for devices that require an old development environment and those that require new ones, we need to know the APIs provided by both. When designing services based on a core/periphery structure, however, developers need to prepare an environment for the user-side device only, because core functions absorb differences in device specifications.

Second, we consider the implementation cost for sharing information among robots. A non-core/periphery service structure does not have edge servers. To share information such as positions, robots must establish connections with each other. Therefore, each time a new robot appears, developers must modify source code to allow other robots to connect with the new one. By designing services based on a core/periphery structure, since robots send information to only edge servers, where that information is aggregated, source code does not need to be changed even when new robots appear.

Third, we derive guidelines for service function placement in a core/periphery structure. Taking advantage of a core/periphery structure allows appropriate division of service functions and deployment of those functions to different servers or devices. If no functions are divided and deployed in the cloud or on end devices, new services must be entirely recreated to adapt to various user requests or device evolution. Furthermore, allocating core functions on edge servers and peripheral functions on end devices is the most effective in terms of service responsiveness and implementation cost, because it is possible to form feedback loops by short-distance communication between end devices and edge servers located near those devices and to adapt to real-world environmental changes.

3.6 Conclusion

We revealed implementation cost and actual penalty of services using a core/periphery structure, which is a known model for flexible behavior in biological systems, and evaluated it in terms of implementation cost and service responsiveness.

To utilize the flexibility of a core/periphery structure, we regard core functions as those with unchanging behaviors even when there are changes in user requests or the real-world environments, and peripheral functions as those whose behaviors can change under such circumstances. We implemented a service and evaluated the effects of a design based on a core/periphery structure under an experimental laboratory environment. These experiments showed that the penalty due to MQTT on an edge server is about 31ms. Taking advantage of a core/periphery structure allowed us to appropriately divide service functions and locate functions in a MEC environment, thus reducing implementation cost for adding new functions with little penalty.

In future work, we will evaluate implementation costs for object detection and feedback to robots, and for sharing information among robots. There is also a need for implementation and evaluation of the service using robots other than Pepper. Service design based on a core/periphery structure is more efficient when there are various devices, but in this thesis we implement and evaluate a service using only one kind of headset and robot.

Chapter 4

Design, Implementation and Evaluation of a Network-oriented Service with Environmental Adaptability based on Core/Periphery Structure

4.1 Introduction

In Chapter 3, we focused on a shopping experience service using mixed reality (MR) and robots as a use case to realize a service scenario based on [13], implemented the service with an actual device, and showed 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 chapter, we evaluate in the following two aspects more pragmatically than previous works. The first one is the service scenario to use in our experiment. In our previous works [13] and [14, 15], we considered a service scenario that includes only information processing; however, commonly used applications today not only process information obtained from devices, they also share information among such devices. We focus on the information sharing in this chapter. We consider a service scenario that includes information processing and information sharing among

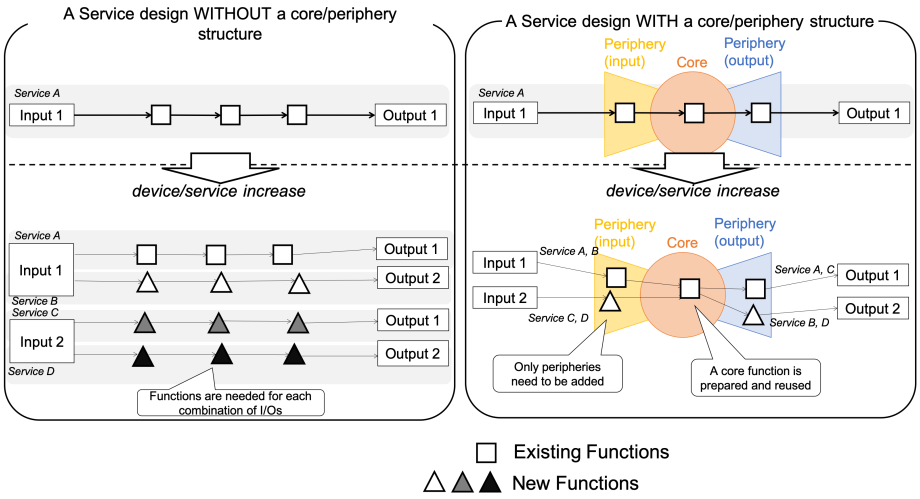

Figure 4.1: Effectiveness of service design using core/periphery structure. Consider adding Input 2 and Output 2 to a service that has Input 1 and Output 1. By providing core functions, fewer additional functions are needed to support various inputs and outputs.

remote robots and users and evaluate our service design in terms of the complexity of the source code and overhead for information sharing. To investigate the amount of penalties on sharing information, we implement a service and measured the penalty through experiments on actual devices. The second one is the metric to represent the implementation cost. In our previous works [13] and [14, 15], 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 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., a modern software development. We use the cyclomatic complexity [18], which is the number of independent paths from the start to the end of the program as the metric to evaluate the implementation cost.

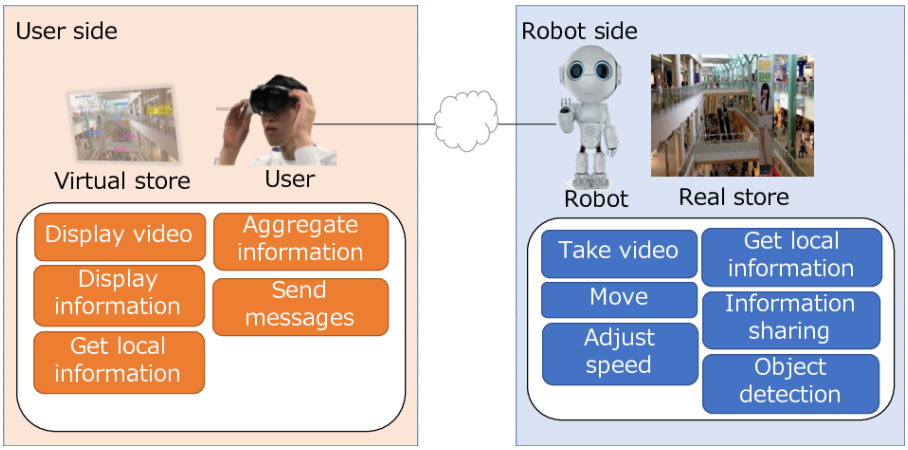

Figure 4.2: Our service with added functions

4.2 Service Scenario

4.2.1 Basic Service

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

4.2.2 Additional Functions to Improve the Service

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

- 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 shown in Fig. 4.2. To realize these services, it is necessary to conduct object detection, store and share information using the results of the object detection, and

adjust the manner in which the robot moves. When realizing such services, these functions are combined as necessary.

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

4.3 Service Design and Implementation

4.3.1 Design Scenario

Using the following three design scenarios, we evaluated the effectiveness of the service design using a core/periphery structure.

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 Sec. **??**, we there call the structures such as microservices [36] and service-oriented architecture (SOA) [37] “No Core” because they are considered as service structures with only peripheral functions.

Note that a monolithic structure in which all functions are tightly coupled is another candidate for the design scenarios. Because functions are tightly coupled, we can interpret the monolithic design as the design consisting only of core functions. Monolithic designs are more difficult to maintain and scale than designs with microservices because monolithic services are larger and

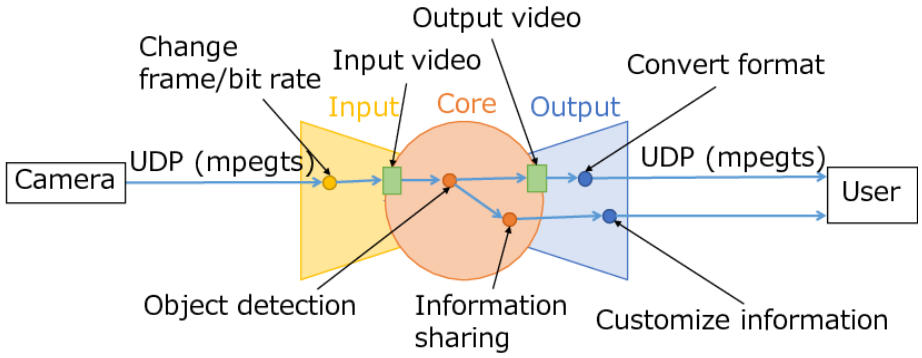

Figure 4.3: The core/periphery structure for video processing and information storage

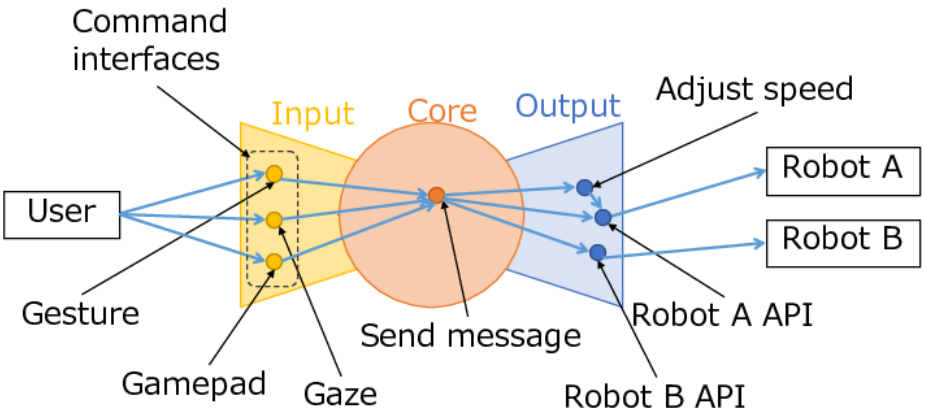

Figure 4.4: The core/periphery structure for robot operation

more complex than microservices [36]. Since its development cost is greater than that of a structure consisting of only peripheral functions, we do not compare with the monolithic design.

4.3.2 Service Design based on a core/periphery structure

We designed our service scenarios in Sec. 4.2.2 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, core functions, and other functions that can be connected peripherally to the core functions. Therefore, when environmental changes occur, only peripheral functions need to be changed within a short period of time.

Because the object detection function is commonly used in our service scenario, both it and the tightly connected information storing function are core components, and peripheral functions provide how the information is used, as described in 4.3.3 and 4.3.3. For the robot operation, the core function is sending messages from the user to the robot, and the peripheral function adjusts the speed of the robot and processing the messages based on each API. Figures 4.3 and 4.4 respectively show the core/periphery structure for video processing and information storage, as well as the core/periphery structure for a robot operation.

4.3.3 Implementation

Object Detection and Information Sharing (Core Function)

Video images from cameras are sent to the robot-side edge server. The video images are captured using OpenCV [44], and object detection is then applied using a PyTorch implementation of YOLO v3 [41]. Using FFmpeg, the processed video is transmitted through UDP to HoloLens, an MR headset worn by users, for display purposes. HoloLens is a standalone head-mounted computer made 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 are acquired as remote information and stored as shared information among the robots.

Messaging (Core Function)

HoloLens controller information is transmitted through message queuing telemetry transport (MQTT), a publish/subscribe-type protocol developed for frequent message exchanges between IoT devices. Users use an Xbox controller that can connect to HoloLens. We develop an MQTT messaging system on a user-side edge server using Mosquitto, an open-source message broker, and Node-RED, a

programming tool for event-driven applications. The MQTT broker receives controller commands through HoloLens and sends them to a program running on the Pepper robot [43].

Speed Adjustment (Peripheral Function)

The basic service provides a function to move at a constant speed. To enable a change in the way the robot moves according to the surrounding environment, we add a function to adjust the speed of the robot such that it does not bump into people or obstacles, using the information of the surroundings obtained by the object detection function. The results of the object detection are returned from the edge server to the robot, and the robot uses the information to adjust its speed, such as slowing down in crowded areas. The core function is to perform object detection on the images sent from the robot and return the results, and the peripheral function is to reduce the speed when there are many people in the area. The peripheral function processes the results, and thus the robot can choose to avoid obstacles other than people, depending on where it is. The edge server stores a list of objects and their sizes, and the robot refers only to the information of the object to be avoided.

Displaying Information to Users (Peripheral Function)

The basic service displays information about an object using object detection with a learned model. To display detailed information about the surrounding objects based on the user's attention, we add a function to detect gaze and display information using a database of products. 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, and a recommendation is made for a product of the same type that the user has not yet been shown. 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.

4.4 Evaluation

Using the following three design scenarios described in 4.3.1, we evaluated the effectiveness of the service design using a core/periphery structure in terms of the implementation cost and overhead

for information sharing.

4.4.1 Implementation Cost

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

In [14, 15], we have measured the implementation cost of core and periphery functions by the amount of source code. However, the amount of source code does not fully capture the effort required to adapt to the environment. This is 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, and the complexity is especially important when multiple people develop the service, i.e., a modern software development.

We assume that all implementations under the No Core scenario are written in the program of the user-side device, based on the implementation of the program that directly connects the HoloLens MR headset with the Pepper robot. Therefore, when modifying the program to control other robots, it is necessary to write the API program for each robot. When users have $m + 1$ types of devices, it is necessary to describe the process for n types of robots for the $m + 1$th device. In addition, when the number of robot types becomes $n + 1$, it is necessary to describe the process for the $n + 1$th robot in each program of the m types of user devices. The more we add to the program, the more complex the program becomes, and the more complex the program is, the longer it takes to read and write to add the source code. Thus, the implementation cost is considered to be not only 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 use the cyclomatic complexity [18] to evaluate the implementation cost. Cyclomatic complexity is the number of independent paths from the start to the end of the program. The higher the number of branches, the higher the value, and the harder it is to read.

We show extracts of source code to establish connection with robot. Source Code 4.1 shows a part of the source code of the HoloLens application under the No Core scenario. To access each

robot directly from the HoloLens application, we need to write all processes for all robots' APIs within a single program. This example represents the case where $n = 2$ and $m = 1$. The second and ninth lines are branches according to the environment (in this case, the type of device), and become more complex as n and m increase.

Listing 4.1: 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
}else if(!string.IsNullOrEmpty(RobotIP)){
    session_robot = RobotSession.Create(tcpPrefix + RobotIP + portSuffix);
    if (!_session.IsConnected){
        Debug.Log("Failed to establish connection");
            return;
    }
}
```

Source Code 4.2 shows a part of the source code of the HoloLens application, and Source Code 4.3 shows a part of the source code of the Pepper application with a core/periphery structure. Because the messaging function with MQTT is provided as a core function, we only need to write the process for connecting to the MQTT broker from each device.

Listing 4.2: 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);
}
```

Listing 4.3: Establish connection from Robot to MQTT broker.

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

Under the No Core scenario, all source code for accessing the robots is written into the HoloLens program, and thus the more robots are used, the more branches are written into the program. Thus, the cyclomatic complexity for the entire service becomes $O(m \times n)$. Under the Core on Robot/Edge Server scenarios, we do not need to consider branching, and thus the value does not change and remains at 1. Therefore, as the number of robot types increases, the difference in the cyclomatic complexity between the No Core and Core on Robot/Edge Server design scenarios becomes more significant.

4.4.2 Overhead for Information Sharing

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, we measured the number of messages sent for information sharing in the three design scenarios: No Core, Core on Robot, and Core on Edge Server, and 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 4.6 shows configuration of the experimental environment in our laboratory. We constructed MEC environment using OpenStack version 3.8.1 [54]. The user-side edge server is an OpenStack virtual machine (192.168.10.73). 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 [55]. Messages from the user to the robot, e.g., controller

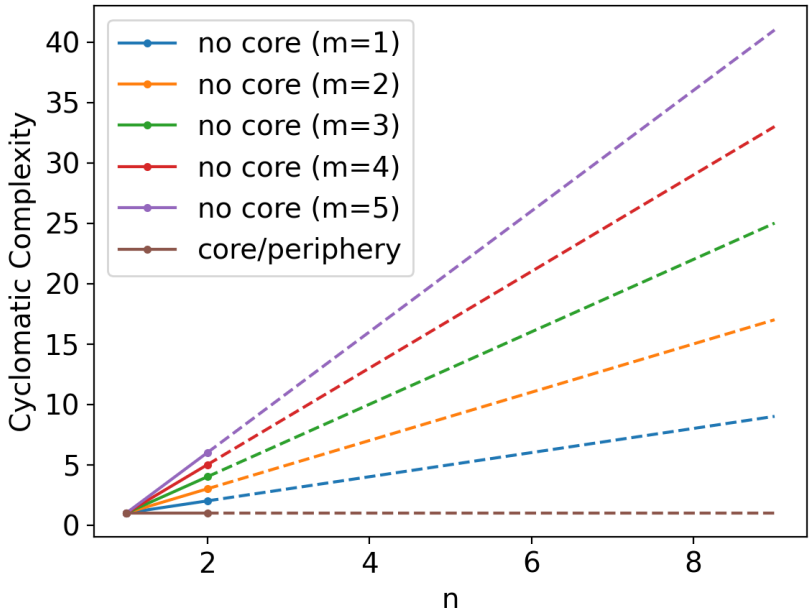

Figure 4.5: Cyclomatic complexity of application source code when the number of robots n is increased.

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. Note that the Pepper has an embedded camera with 320x240 resolution, but it is too low to enjoy the video streaming at the HoloLens. Thus, in this experiment, we attached an external camera on the head of the Pepper, and the camera is connected to Aja HELO [56] via SDI cable to perform the H.264 encoding for the video streaming. Video from camera is input to the HELO at 60 fps, 1920x1280, and output at 30 fps, 1080x720. The video encoded by HELO is sent to the robot side edge server using UDP in mpegts format. Once the processing is completed, the video is output to standard output as raw data, then encoded into mpegts using FFmpeg, and sent to the HoloLens worn by the user.

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

1. The object detection function is executed until each robot obtains new information. The information is a list of objects seen by the camera used in our application. Under the No core and Core on Robot scenarios, this phase is not applied because the robot conducts the object recognition function directly. 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 information held by the robot.

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. 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.

3. Information is provided from the information sharing function to the user when the user gazes at a specific object. Under the No Core and Core on Robot scenario, the robot sends information to the user. Under the Core on Edge Server scenario, the edge server sends information to the user.

Under the No Core scenario, the information-sharing function is implemented in a specific way based on the type of robot applied. To keep the information stored in the information-sharing function on all robots up to date, each robot shares information through flooding when there is an update in the surrounding information held by the robot. 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, and thus it is necessary to break down and transmit each

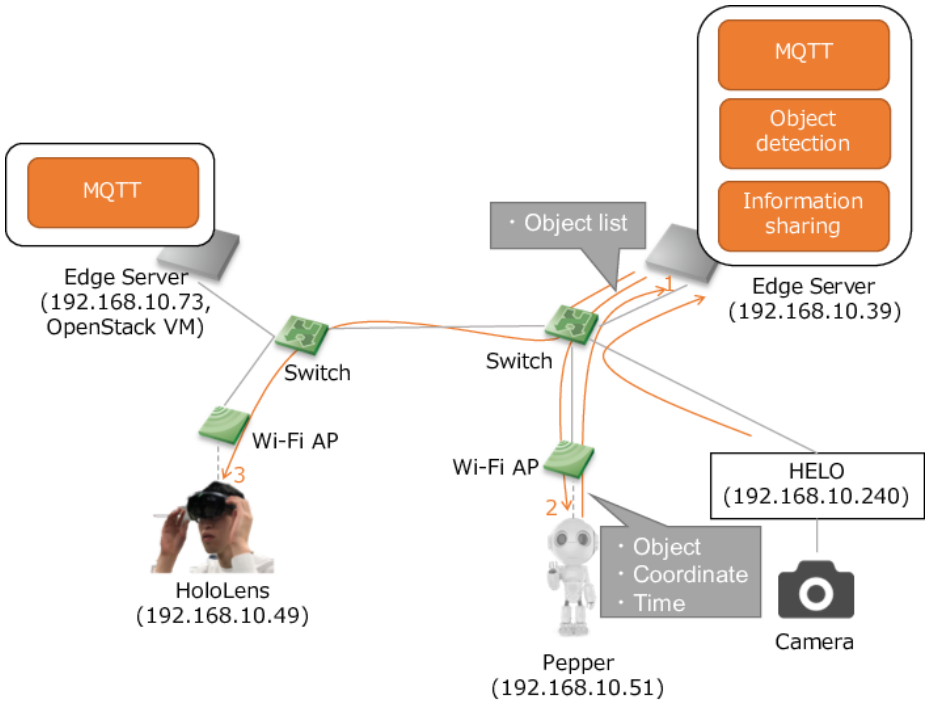

Figure 4.6: Experimental environment to measure messages for information sharing. The numbers correspond to each phase.

type of information, such as the name and coordinates of the object. Therefore, $O(k \times n^2)$ messages are sent, where k is 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 we need to keep the information stored in the information sharing function of all robots up to date in the same way as that under the No Core scenario, and thus we assume that when each n robot sends information to the $n-1$ robots other than itself, $O(n^2)$ messages are sent.

Figure 4.7 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 coefficient. For Phase 1, overhead occurs only in scenario Core on Edge Server. Because the information-sharing function is implemented as a

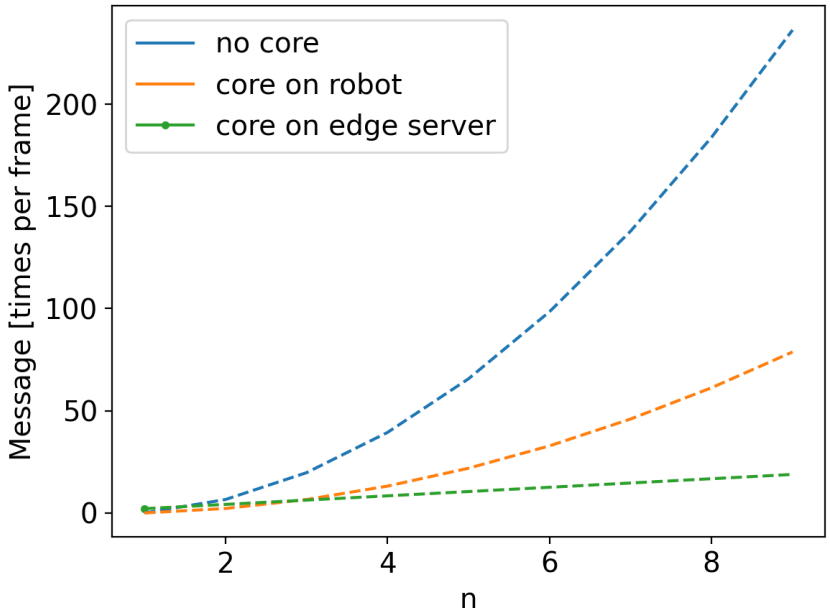

Figure 4.7: Number of messages for information sharing when the number of robots n is increased

core function, the method of storing and sharing information is unified, and all the 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 in length (790 frames) captured at 29.97 fps and revealed that the edge server sent 854 messages. A total of 1.08 messages were sent every time the information was updated. For Phase 3, 11 gazes occurred in the same video as 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. From these results, the number of messages in the Core on Edge Server scenario is $2.094 \times n$ times per frame.

In extrapolating the results of our experiment shown in Fig.4.7, under the No Core scenario, the messages occur at a rate of $k \times 1.094 \times n \times (n-1)$ times per frame (where k is the number of information types, which in this implementation is three), and at $1.094 \times n \times (n-1)$ times per frame under the Core on Robot scenario. In the case of a single robot, the number of messages under

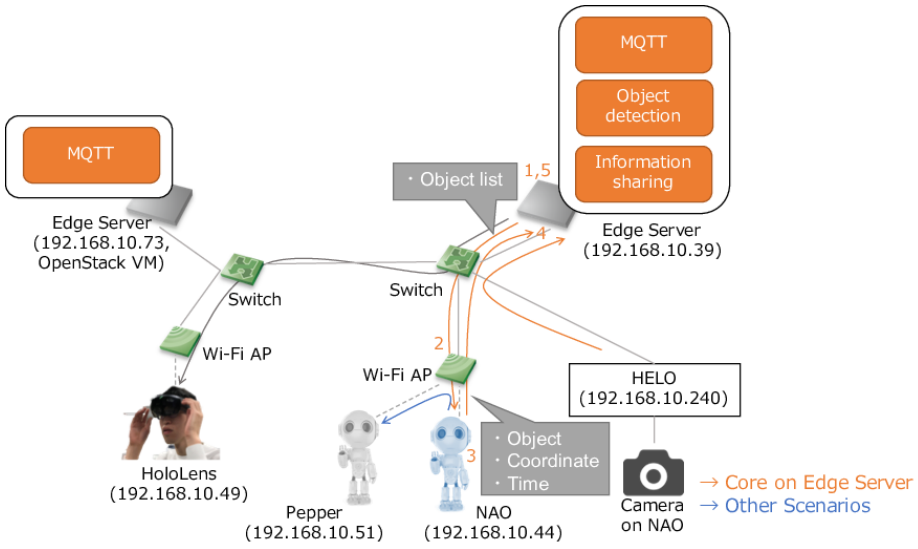

Figure 4.8: Experimental environment to measure penalty due to the extra communication path. The numbers correspond to each phase.

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.

4.4.3 Penalty in scenario Core on Edge Server

Dividing functions and placing them in different devices creates extra communication paths and penalty of service responsiveness. To investigate this penalty, we have conducted another experiment which measures the penalty of communicating through an edge server when robots share information.

In this experiment, we evaluate the application-level delay using a service scenario of information sharing among robots. Figure 4.8 shows our experimental environment. The robot NAO (192.168.10.44) [57] is connected as 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 user's attention. Among these delays, the penalty for using the edge server to provide information to the user via is about the same as that

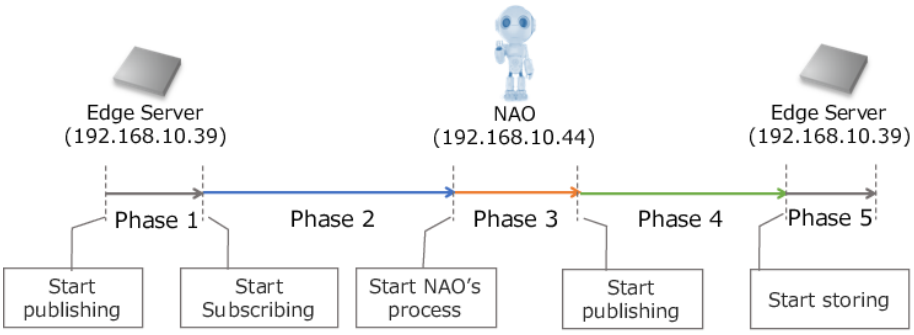

Figure 4.9: Each phase to measure the delay

measured in [15] because the extra path in the Core on Edge Server scenario is between the robot-side switch and the robot-side edge server. Thus we measured the time as application-level delay it takes for the information obtained by the object detection function to be reflected in the information sharing function under the Core on Edge Server scenario, represented by the orange arrows in Fig. 4.6. To understand the application-level delay more clearly, we divided the service process into five phases. Figure 4.9 shows the five phases, starting from the time the new information was acquired at the edge server, and ending with the time the information is stored in the edge server.

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 gets 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 is 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 are extra processing compared to other design scenarios. Therefore, the time taken in these phases is a penalty in the Core on Edge Server scenario. Phase 2 is Pepper/NAO's MQTT subscription process. Since there are different system clocks among edge servers and robots, accurate measurement of the time it takes for Phase 2 is difficult. 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 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.

Result

We show the result of our experiment using the robot NAO. TABLE 4.1 shows the average, maximum, and minimum values for total time, the time for each phase and the penalty by time incurred by Core on Edge Server scenario. The total measured time, 104 [ms] on average, is the application-level delay for information sharing among robots. The penalty under the scenario Core on Edge Server is 99 [ms] on average, and can be up to 536 [ms]. The application-level delay penalty in

Table 4.1: 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

users' operation of the robot was 31 [ms] on average [15]. In [15], since the penalty is for user-robot communication, the extra path is between the edge server at the user side and the switch at the user side. In this experiment, the extra communication path is between the robot and the robot-side edge server, which 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 those phases are communicating with the MQTT broker on the same edge server without communication delay. The reason why the delay of Phase 3 changes drastically is that 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. In addition, the robot sequentially adds and sends information for every item in the list of objects. Thus, the more objects there are in one frame, or the longer it takes to send other objects, the larger the maximum value becomes, because waiting time to send the object at the end of the list becomes large. Therefore, we evaluate the penalty due to the extra communication path compared to 4 [ms], which is the average time taken for Phase 3.

Figure 4.10 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. Since Wi-Fi communication is used in Phases 2 and 4, the delay varies due to congestion, obstacles, and the distance from the access point. This result reveals that the penalty is tolerable because interaction delay tolerance is 100 [ms] [58]. The latency of wireless communication can be expected to improve with the ultra-low latency that 5G offers.

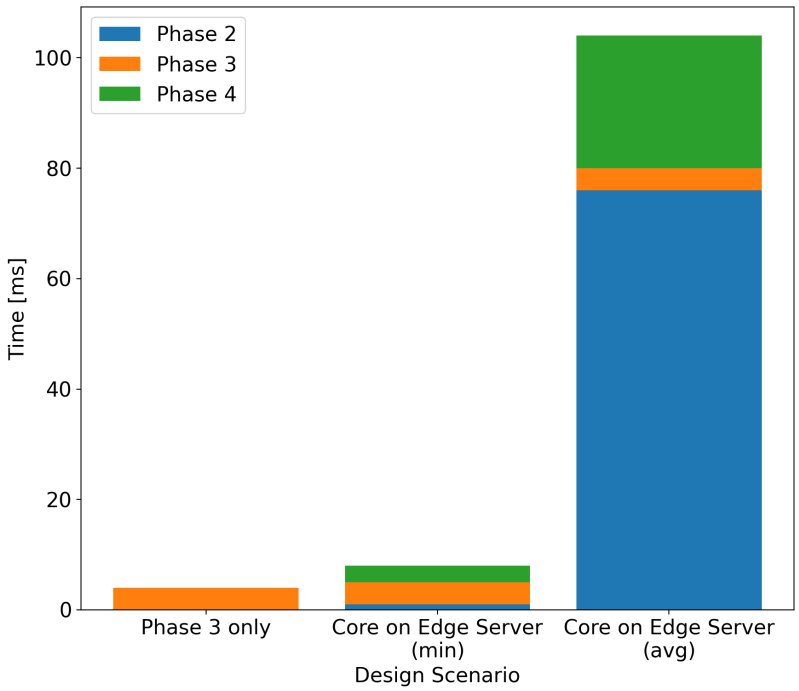

Figure 4.10: Minimum and average time for Core on Edge Server scenario

4.4.4 Hierarchical Core/Periphery Structure

In our implementation, 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 regard 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, in a large-scale service configuration where multiple edge servers exist, we can find a hierarchical core/periphery structure. Considering the same situation as shown in Fig. 4.7, placing the core functions in the edge servers of a higher hierarchy is effective when aggregating the information of four or more edge servers. However, this figure assumes that 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 needed when information is shared among an extremely large number of edge servers.

4.5 Conclusion

In this chapter, we introduced a core/periphery structure to actualize a flexible and adaptive service composition in an MEC environment, which is a known model for flexible behavior in biological systems, and designed and implemented a network-oriented service based on this structure. We designed and implemented multiple service scenarios and evaluated them using two metrics: cyclomatic complexity and overhead for information sharing. We demonstrated that the source code does not become complex when we add functions to access different devices using the core/periphery structure. Furthermore, we measured the penalty through experiments on actual devices and showed that it is tolerable.

Although this chapter focuses 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.

Chapter 5

Evolvable Design of Network-oriented Services based on Core/Periphery Structure

5.1 Introduction

To accommodate large numbers of services at low cost, the service design needs to be adaptable to user requirements and environmental changes. We have been investigating a core/periphery structure [8, 9] that allows service components to effectively adapt to each user request and environmental variation. Information processing units in a core/periphery structure are classified as core or peripheral units. In a core/periphery structure in biological systems, a periphery processes various inputs and outputs and reuses a core, which processes information efficiently. When designing services that require efficient processing of a various input/output data based on environmental changes, designing inspired by the biological core/periphery structure is expected to enable the service adaptable to various inputs and outputs. Our previous work has shown that an information processing platform using a core/periphery structure is adaptable to environmental changes at a small cost by reusing the core and recreating only the periphery. However, when large-scale environmental changes arise, it remains necessary to change the core/periphery roles of functions, and

reconfiguration of only the peripheral functions may not be sufficient to adapt to such changes.

In this chapter, our aim is to achieve an evolvable service structure based on a core/periphery structure. Here, we refer to service structures that can change the system at low cost with maintaining the ability to provide unknown services, as evolvable service structures. We assume that there are many service functions created by software developers, and that the service functions are connected through the interfaces are connected to each other. When environmental changes occur, the functions commonly used to make up service chains change or new functions are required to be added. Adapting to environmental changes requires the addition of interfaces between service functions or the development of new service functions. Therefore, we propose a service structure that can efficiently accommodate various service chains with low development cost by controlling the density of service functions. Efficient accommodation provides the service with short chain length using only the minimum service functions. For example, if the network of service functions is provided in a full mesh, the shortest chain can be configured, but the number of interfaces between functions to maintain that density when adding new functions is large. This makes the development cost significant and makes it impossible to maintain the services in the future. If all functions are designed sparsely, the cost when adding new functions is small, but the chain length is long since it requires extra functions to accommodate service chains.

5.2 Design Problem of Evolvable Service Structure

5.2.1 Design Problem

In this thesis, we assume that there are many service functions created by software developers, and that the service functions are connected through the interfaces are connected to each other. When peripheral functions are added by developers, some new service chains using them will be generated. We add appropriate interfaces between service functions so that accommodate service chains in the future. Since it is difficult to predict what service chain will be required in the future, we use only the information available at the time.

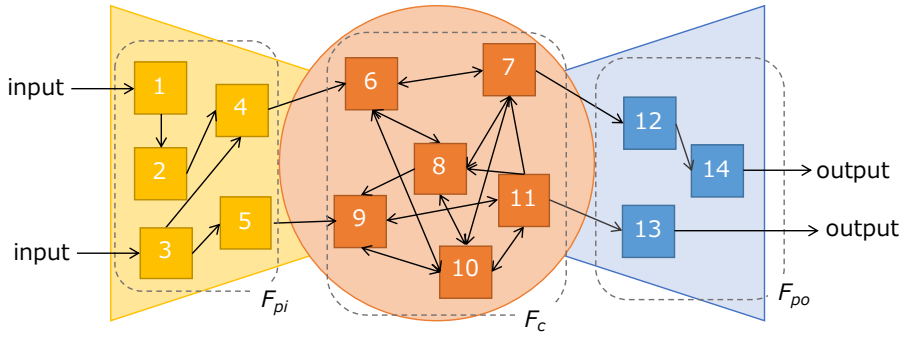

Figure 5.1: An example of service functions network G_t

Service Structure and Service Chain

We consider a network consisting of service functions and interfaces connecting them. The network of service functions at time t is represented by a directed graph $G_t(\{F_{pit}, F_{ct}, F_{pot}\}, L_t)$ consisting of a set of input-side peripheral functions F_{pit}, a set of output-side peripheral functions F_{pot}, a set of core functions F_{ct}, a set of links L_t. L_t represents interfaces among the service functions. Service functions are developed by several different developers, each working independently from the others. When an interface exists between service functions, the interface is always available. The service providers attempt to connect the service functions they use based on their input data and output data they want to acquire. The directed graph connecting those service functions from the input side to the output side is a service chain. We call a service chain sc_t using an interface other than an existing interface links a known service chain. For instance, given a service function network G_t in Fig. 5.1, the chain shown in Fig. 5.2 is a known chain and it is a subgraph of G_t. A known service chain is represented by a directed graph $G'_t(\{F'_{pit}, F'_{pot}, F'_{ct}\}, L'_t)$. G'_t is a subgraph of G_t since $F'_{pit}, F'_{pot}, F'_{ct}$, and L'_t are subsets of F_{pit}, F_{pot}, F_{ct} ,and L_t respectively. Here, we assume that the functions in F_{pi} is not used after the functions in F_{po} and the functions in F_c and F_{pi} is not used after the functions in F_{po}.

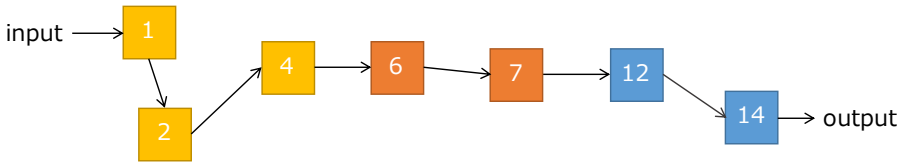

Figure 5.2: An example of G'_t

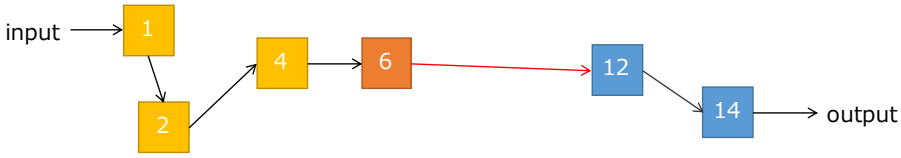

Figure 5.3: An example of service chain that is accommodated by using other service functions and interfaces

Service Accommodation

This section describes the accommodation of unknown service chains. In this thesis, we assume that a service chain can be accommodated when there is a combination of interfaces that allows the service functions that consist the service chain to be used in a given order. All of G'_t and some of service chains that are not subgraphs of G_t are accommodated. For example, the chain in Fig. 5.3 is unknown, but can be accommodated since by using service function 7 between function 6 and 12. However, in this case, service functions and communications that are not originally required are used. The link $(6, 12)$ allows to provide a shorter chain and efficient service. We call the minimum number of interfaces required to accommodate a given service chain sc_t the chain length l_{sct}. Service chains which have a combination of unreachable service functions, as shown in Fig. 5.4 are not accommodated.

5.2.2 Problem Description

Development Cost

To increase the probability of accommodation of the service chain and to provide services with short chain length, additional interfaces between service functions are required. Based on the information

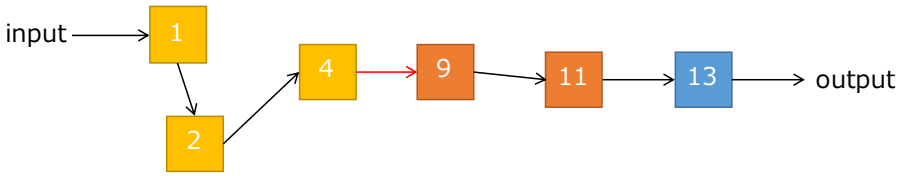

Figure 5.4: An example of service chains that is not accommodated

available at time t, we determine interfaces L_{t+1} and add the link. We define the cost C required to accommodate a service chain by the number of links to be added.

$$C = |L_{t+1} \setminus L_t| \tag{5.1}$$

Service Accommodation Ratio

Let $ac_{sc} = 1$ when the service chain sc can be accommodated in G_t and ac_{sc} otherwise. For the set of service chains SC arising at time t, the accommodation ratio $AC(C)$ when using the development cost C is

$$AC(C) = \frac{\sum_{sc \in SC} ac_{sc}}{|SC|} \tag{5.2}$$

Our goal is to maximize $AC(C)$ for the same C when t is increased.

We list our representations at Table 5.1

5.2.3 Approach based on Density Control

When considering the realization of an evolvable service function network, it is difficult to solve the problem described in Sec. 5.2.2 as an optimization problem because it is difficult to express unknown service requirements in mathematical form. Thus, we enable service function networks to evolve by maintaining a core/periphery structure of appropriate size and density. Our method can determine the structure of the network at the next time based only on the topology of the service function network at the current time. Therefore, it is expected to be able to continue to evolve at a stable and low cost, regardless of the inability to predict environmental changes.

In [59], for a given two blocks in the human brain, when (connection strength within block m)

Table 5.1: Representations in Chapter 5

Name	Description
F_{pi}	Set of input-side peripheral functions
F_{po}	Set of output-side peripheral functions
F_p	$F_{in} \cup F_{out}$
F_c	Set of core functions
$coresize$	Ratio of cores to the total network
L	Set of links
$d_{\{pi,po,c\}}$	Density within the function block
$d_{(c,\{pi,po\})}$	Density between the function blocks
C	Development cost
sc	A service chain
l_{sc}	Chain length

$>$ (connection strength between blocks m, n) $>$ (connection strength within block n), block m is a core block and n is a peripheral block. Gu et al. [59] shows that in terms of the organization of brain function, as the brain develops, modules become more separated, core-periphery pairs increase, and the strength of the connections within blocks the strength of the connections between blocks are negatively correlated, i.e., the more separated they are, the more indicating a state of well-developed function. In the service function relationship, a negative correlation between the density between blocks and the density within a block facilitates development of the function of each block because of its small dependencies with other blocks.

It is important that the density between the blocks be between the density of the core block and the density of the peripheral blocks. Gu et al. [59] explains that, when the density between blocks is greatest, they are not separated and are a monolithic structure, and when the density between blocks is smallest, there is no interrelationship between them. That is, the density between the core block and the peripheral block is required to be sufficiently smaller than the density within the core block to facilitate development, but sufficiently greater than the density within the peripheral blocks to accommodate service chains.

In addition, this approach enables to distinguish the role of core/periphery and determine L_{t+1} from only the network topology information regardless of the content of the service function or the

frequency of use. Determining the structure of the service function by distinguishing the roles of the core and periphery based on the frequency of use of the service and the content of the service function is conceivable, but it is difficult to obtain the information for all service functions and predict future information.

On a service functions network, the role of the service function can change to accommodate unknown service chains over time. Therefore, we control the structure of service functions so that some of the peripheries functions are newly regarded as core functions while keeping the inequalities.

5.3 Density Control of Service Functions Network

This section describes the operation of determining L_{t+1}.

5.3.1 Service Functions Structure based on Core/Periphery Structure

The connection between function blocks is represented by an adjacency matrix A of blocks m, n. $A_{ij} = 1$ when the interfaces $i \in m$ to $j \in n$ are available, and $A_{ij} = 0$ when they are not. When $m = n$, that is, within F_{pi} ,F_{po}, F_c, we define d_m, the density within function block m as

$$d_m = \frac{\sum_{i \in m, j \in m, i \neq j} A_{ij}}{|m| \cdot |m| - 1} \tag{5.3}$$

We define $d_{m,n}$, the density between blocks as

$$d_{m,n} = \frac{\sum_{i \in m, j \in n} A_{ij}}{|m| \cdot |n|} \tag{5.4}$$

Hereinafter, we refer the density within F_{pi}, F_{po}, F_c as d_{pi}, d_{po}, d_c respectively, and density between the core block and peripheral block as $d_{c,pi}, d_{c,po}$.

Algorithm 5 Determine G_{t+1}

Input: $G_t = (F_{pi}, F_c, F_{po}, L_t)$
Output: G_{t+1}

while $d_c - d_{c,pi} \leq th_{c,cp}$ **or** $d_c - d_{c,po} \leq th_{c,cp}$ **or** $d_c < dmin_c$ **do**
  $n1, n2 \leftarrow$ random nodes in F_c
  add edge $(n1, n2)$
  add edge $(n2, n1)$
end while
while $coresize < coresize_{max}$ **or** $d_c - d_{c,cpi} \leq th_{c,cp}$ **or** $d_c - d_{c,cpo} \leq th_{c,cp}$ **do**
  $fp \leftarrow$ a random node in adjacent nodes of F_c
  $c \leftarrow$ a random node in F_c
  add edge (fp, c)
  add edge (c, fp)
end while
while $d_{c,pi} - d_{pi} \leq th_{cp,p}$ **or** $d_c - d_{c,pi} > th_{c,cp}$ **do**
  $fp \leftarrow$ a random node in F_{pi}
  $c \leftarrow$ a random node in F_c
  add edge (fp, c)
end while
while $d_{c,po} - d_{po} \leq th_{cp,p}$ **or** $d_c - d_{c,po} > th_{c,cp}$ **do**
  $fp \leftarrow$ a random node in F_{po}
  $c \leftarrow$ a random node in F_c
  add edge (c, fp)
end while
while $d_{c,pi} - d_{pi} > th_{cp,p}$ **or** $d_{pi} < dmin_p$ **do**
  $n1, n2 \leftarrow$ a random nodes in F_{pi}
  add edge $(n1, n2)$
end while
while $d_{c,po} - d_{po} > th_{cp,p}$ **or** $d_{po} < dmin_p$ **do**
  $n1, n2 \leftarrow$ a random nodes in F_{po}
  add edge $(n1, n2)$
end while

5.3.2 Adding Links to Accommodate Unknown Service Chains

We focus on the service function $fp \in F_p$. f is connected to one or more of the service functions in F_c. Let $L_{fp,c}$ be the set of links connecting fp and F_c, and $L_{fp,p}$ be the set of links connecting fp and the block where fp belongs.

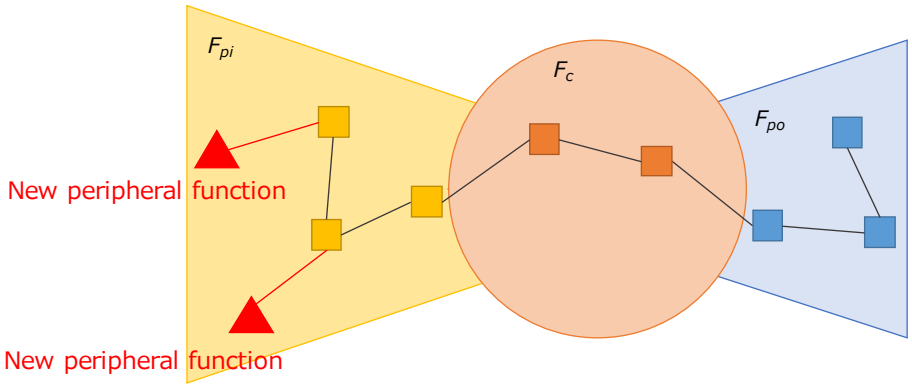

Figure 5.5: An example of service functions before application of the method

G_{t+1} is determined by Algorithm 5. In our simulations, we control the density in the following order, but you can get the same results if you switch the order. Figure 5.5 shows an example of the structure of the service functions before the method is applied, assuming an increase in the peripheral functions on the input side. In line 1-5, we control d_c. We select two functions at random in F_c and add links between them. As we explained in Sec. 5.2.3, to make the density of core blocks sufficiently greater than the density between blocks, we add links until the following conditions are met:

- $d_c - d_{c,pi} > th_{c,cp}$
- $d_c - d_{c,po} > th_{c,cp}$
- $d_c \geq dmin_c$

Figure 5.6 shows the service functions structure when this process is finished.

In line 6-11, we control the core size. We randomly select $fp \in F_p$ and nodes in F_c and add the links between them so that fp can be considered $fp \in F_c$. F_c including fp become the new core block. To keep core size below the certain value, we transit to the next state when at least one of following conditions is met:

- $coresize \geq coresize_{max}$
- $d_c - d_{c,cpi} > th_{c,cp}$ and $d_c - d_{c,cpo} > th_{c,cp}$

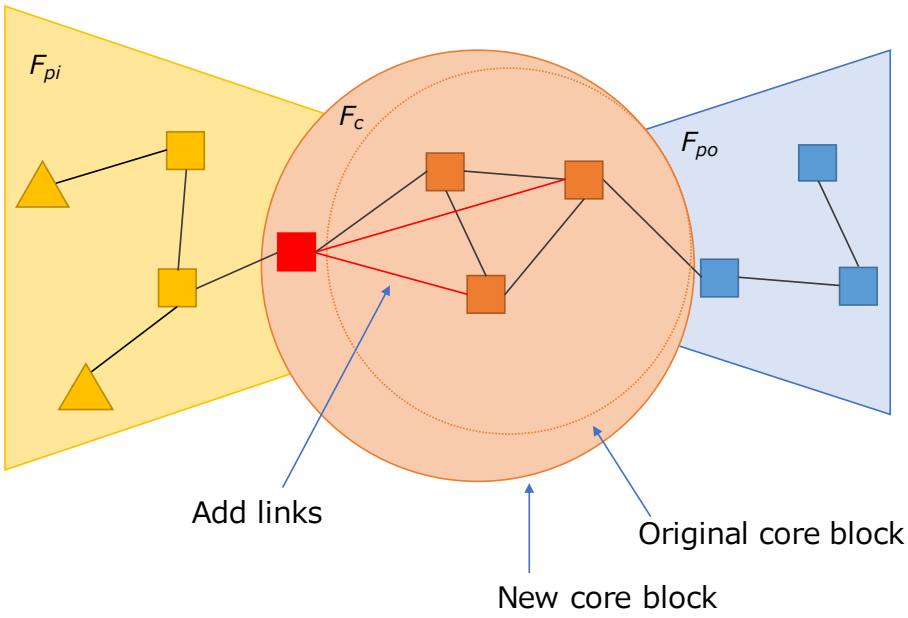

Figure 5.6: An example of service functions after adding a core function

In line 12-21, we control $d_{c,pi}$ and $d_{c,po}$. We randomly select nodes in F_p and nodes in new F_c and add the links between them. As we explained in Sec. 5.2.3, to make the density between blocks sufficiently greater than the density within peripheral blocks, we add links until the following conditions are met. We set the second and fourth conditions to avoid adding an unlimited number of links.

- $d_{c,pi} - d_{pi} > th_{cp,p}$
- $d_c - d_{c,pi} \leq th_{c,cp}$
- $d_{c,po} - d_{po} > th_{cp,p}$
- $d_c - d_{c,po} \leq th_{c,cp}$

In line 22-29, we control d_{pi} and dpo. We randomly select peripheral nodes in the same block and add the links between them. To make the density within peripheral blocks sufficiently great to be the function blocks, we add links until the following conditions are met. We set the second and fourth conditions to avoid adding an unlimited number of links.

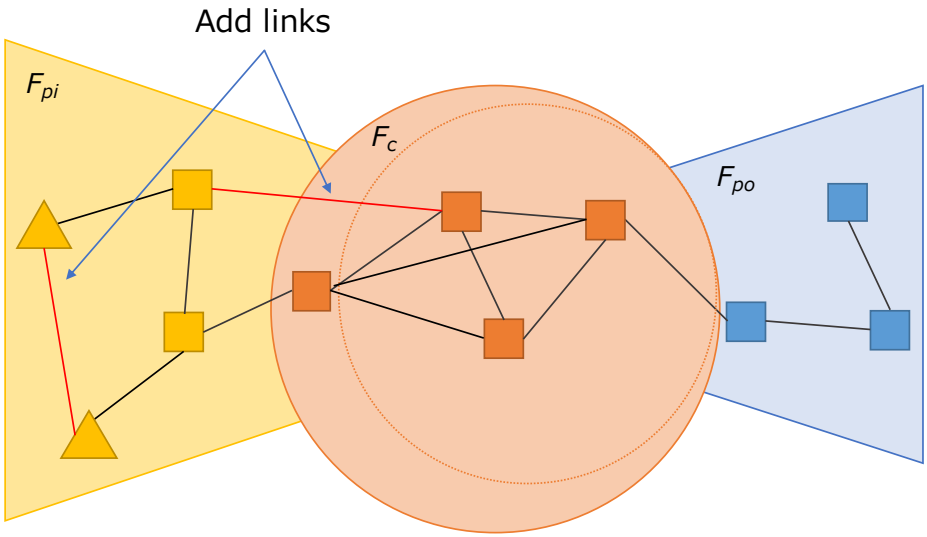

Figure 5.7: An example of service functions after adding links between peripheral functions

- $d_{c,pi} - d_{pi} \leq th_{cp,p}$
- $d_{pi} \geq dmin_p$
- $d_{c,po} - d_{po} \leq th_{cp,p}$
- $d_{po} \geq dmin_p$

Figure 5.7 shows the service functions structure when this process is finished.

Our method randomly selects which nodes to add links between. The nodes to which links are added can be determined based on information such as frequency of use or degree. Since service chains that will arise in the future are difficult to predict, adding links randomly leads to accommodating a variety of service chains.

5.4 Evaluation

5.4.1 Comparative Methods

The methods used in our evaluation are as follows:

- Density Control: our proposal explained in 5.3.2.
- Random: this method randomly selects nodes and adds links between them with the same cost as our With Density Control method.
- Low-cost Accommodation: this method accommodates service chains occurring at t with as low cost as possible. Specifically, the following operations are executed.
    1. We calculate how much the accommodation ratio is increased by each link between all nodes. Note that the links which are able to add are that from F_{pi} to F_c, F_{pi} to F_{po} or F_c to F_{po}.
    2. We add the link with the largest value calculated in 1.
    3. We repeat 1. and 2. until there are no more links to increase the accommodation ratio.
- Shortest-Path Accommodation: this method connects all nodes of the service chains so that each service chain at t is a subgraph of G_t.

The simulation program executes in the following order at each step t.

1. Initial state at t. The function network consists of a core block F_c, an input-side peripheral block F_{pi}, and an output-side peripheral block F_{po}.
2. Environmental Change at t. We add peripheral functions with probability p. We generate random number r. When $\frac{2}{p} < r \leq p$, a function is randomly selected from the F_{pi} and otherwise a function is randomly selected from the F_{po}. Then, the new function is connected to the selected node as a leaf.
3. Each method determines G_{t+1}, and calculate their development cost and accommodation ratio $AC(C)$.
4. $G_t \leftarrow G_{t+1}$ for each methods.

5.4.2 Evaluation Metrics

We evaluate the methods with following three metrics.

Accommodation Ratio

First, we calculate the accommodation ratio of the service chains explained in Sec.5.2.2. We assume that the Random and Shortest-Path Accommodation methods can use the same cost as our Density Control method at each step.

We created 30 service chains for each step in the following way.

1. We select one input node and one output node at random. The nodes in F_p that are leaves are candidates for input and output nodes of service chains, respectively.

2. We randomly select nodes to generate the service chain from all nodes other than the input/output nodes selected in 1. Let the number of nodes at step t N_t. The number of nodes selected here is set to be between 1 and $\frac{N_i}{3}$. Note that they are connected in the order of the nodes in F_{pi}, nodes in F_c, and node in F_{po}.

3. The chain connecting the input node, the node selected in 2, and the output node in that order is a service chain.

Here, 15 of the service chains are selected based on the core/periphery classification at $t = 0$, and the remaining 15 are selected based on the core/periphery classification when the algorithm is applied in section 5.3.2. This is to focus on new service chains that use core functions that were not previously used as core functions.

Development Cost

The Low-Cost Accommodation method tries to accommodate the service chains with as little cost as possible, so its cost is less than Density Control. Therefore, we calculate the development cost of each method to achieve its objectives at each step to evaluate how much more cost the Density Control and Shortest-Path Accommodation methods require compared to the Low-Cost Accommodation method to accommodate the same service chains. We calculate the cost of accommodating all service chains generated at each t in the shortest distance, with unlimited cost available for the Shortest-Path Accommodation method.

Chain Length

In Low-Cost Accommodation method, $AC(C)$ is expected to be higher, but it has the disadvantage of requiring extra paths to be taken to accommodation the service chain. Longer chain lengths mean the use of service functions and communication paths that are not required to provide the service, which leads to a degradation of the responsiveness of the service.

Therefore, we calculate the chain length l_{sc} with our Density Control method and it with Low-Cost Accommodation method respectively. For a service chain sc which consists of N_{sc} functions, the minimum value is $N_{sc} - 1$ when the given service chain is a subgraph of G_t. In this simulation, we calculate $\frac{l_{sc}}{N_{sc}-1}$.

5.4.3 Simulation Results

Table 5.2: Parameters Setting for evaluation

Parameter	Description	Setting
$coresize_{min}$	Minimum ratio of cores to the total network	0.3
$coresize_{max}$	Max ratio of cores to the total network	0.4
p	Probability of new peripheral functions to be created	0.4
$th_{c,cp}$	Threshold of $d_c - d_{c,piorpo}$	0.3
$th_{cp,p}$	Threshold of $d_{c,piorpo} - d_{piorpo}$	0.2
$dmin_{cp}$	Minimum value of $d_{c,piorpo}$	0.3
$dmin_p$	Minimum value of d_{piorpo}	0.2
$dmin_c$	Minimum value of d_c	0.7
$dmax_c$	Maximum value of d_c	0.8

We give the network represented in Fig. 5.8 and parameter shown in Table 5.2. We executed the methods until $t = 40$ respectively for 20 times.

Service Accommodation

The results of the accommodation ratio calculations are shown in Fig. 5.9, and 5.10. The accommodation ratio with a case of an example of evolution paths each service function network followed is shown in Fig. 5.9. The horizontal axis represents steps and the vertical axis represents $AC(C)$.

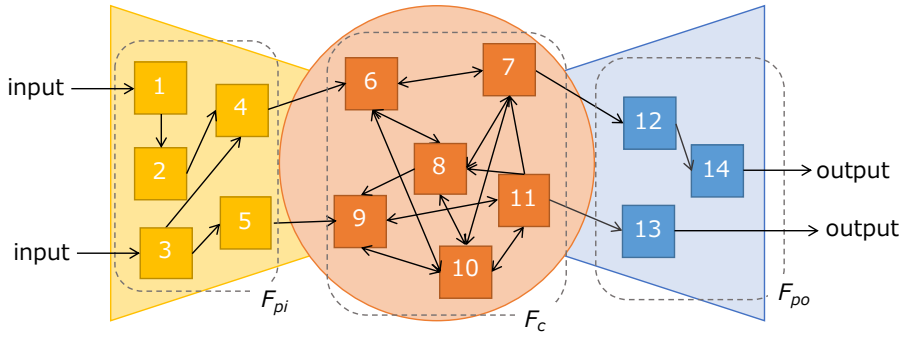

Figure 5.8: The service function network at $t = 0$

Note that Random and Shortest-Path Accommodation methods can use the same cost as our Density Control method at each step.

As the steps pass, the accommodation ratio approaches nearly 1 when Density Control is applied but that with Low-Cost Accommodation method decreases. This is because there are less nodes that are bi-directionally linked (i.e., less functions that are used mutually) and it cannot accommodate the service chains that do not use new core functions. In Density Control method, the number of combinations of functions that can be used mutually is increased by densely connecting peripheral functions as new core functions. As a result, it is now possible to accommodate service chains that use as core a function that was not previously used as core.

The accommodation ratio is not stable when connecting the nodes in the service chain arriving at each step using the same cost as our Density Control method. As the size of the network increases over the course of the steps, the long service chains increases. Since Shortest-Path Accommodation method uses more costs to accommodate longer service chains, the accommodation ratio is lower when more long service chains arise.

Figure 5.10 shows the accommodation ratio for 100 executions of each method, that is, the accommodation ratio for following 100 evolutionary paths. We excerpted the 30th step and later, when the effect of the initial network state is small. When Density Control method is applied, there are some service chains that cannot be accommodated because not all nodes are reachable, but the accommodation ratio is close to 1 for almost all evolutionary paths. When other methods

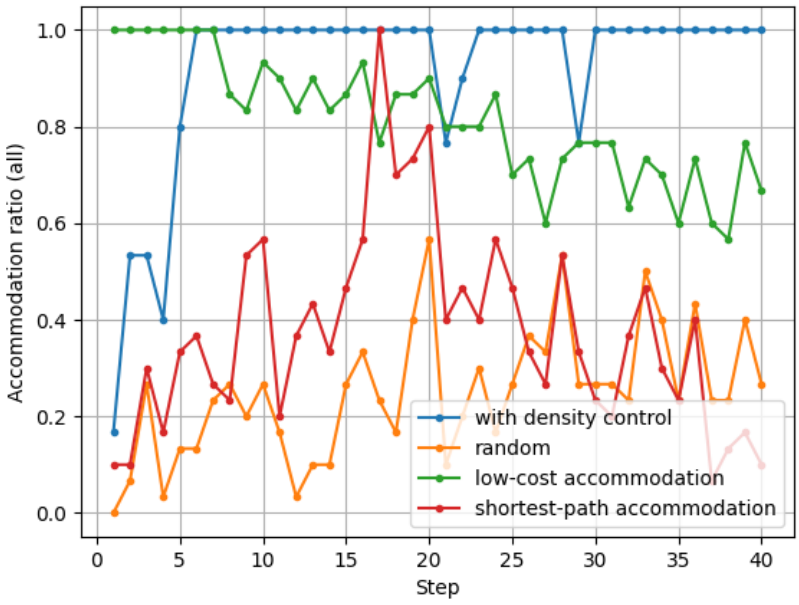

Figure 5.9: An example of accommodation ratio for all service chains

are applied, the variance in accommodation ratios is large. Density Control method also has many random factors, but has a stable higher accommodation ratio than the Random method for the same cost. This indicates that controlling the density of service functions leads to accommodating many service chains.

Development Cost

The results of calculating the cost at each step are shown in Fig. 5.11, and 5.12. The horizontal axis represents a step and the vertical axis represents the cost.

Density Control method adds about 10 times more links than the Low-Cost Accommodation method with our settings. However, Low-Cost Accommodation method calculates accommodation ratios for all possible links. The larger the service functions network, the more difficult it becomes to find the optimal link to add from vast combination of nodes.

The cost for Shortest-Path Accommodation method shown in Fig. 5.11, is the cost required to

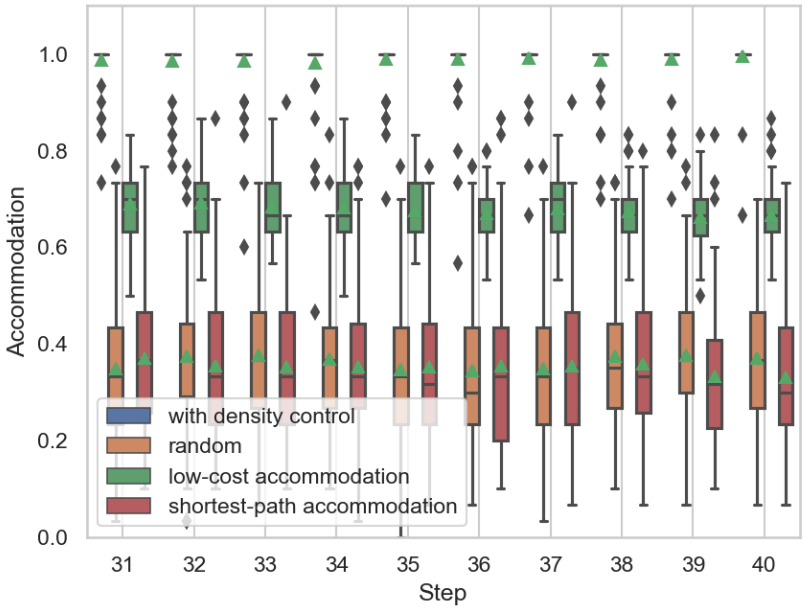

Figure 5.10: Accommodation ratio for all service chains

accommodate all service chains, including those not accommodated shown in Fig.5.9, 5.10. The cost for Shortest-Path Accommodation method increases significantly because the links added at t are rarely reused in the future. Development cost for Low-Cost Accommodation and Shortest-Path Accommodation method are highly dependent on the parameter settings such as the length and number of service chains arriving at each t and it is difficult to predict what service chains will occur in the future. However, development cost for Density Control method is not dependent on the content of the service chain since it is based only on network topology.

Path Length to Accommodate Services

The length of the chains with Density Control and Low-cost Accommodation is shown in Fig. 5.13, 5.14, respectively. Density Control method requires only 1 to 2 times the shortest length of the service chains since we control the density between F_c and F_p. Low-Cost Accommodation method requires about 3 to 5 times the shortest length of the service chains. It adds links between

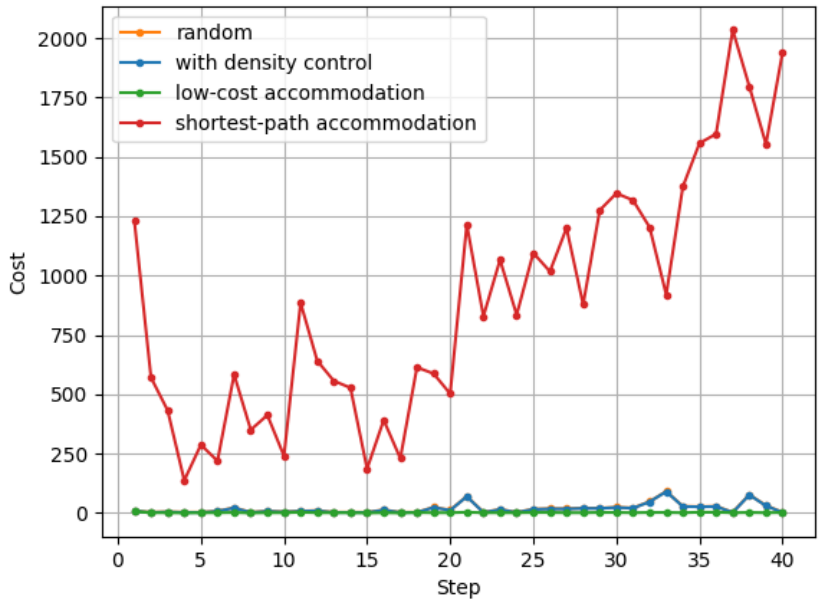

Figure 5.11: An example of development cost

peripheral functions because it determine the links based on the accommodation ratio. As a result, the peripheral function becomes closer to being connected in a ring-like shape. If the Low-Cost Accommodation method is modified to have a shorter chain length, it performs the same process as the Shortest-Path method, and the cost become higher as shown in Fig. 5.11, 5.12.

5.4.4 Evolvability of Service Design

Our simulation results shown in Sec. 5.4.3 revealed that our proposed method allows the service functions network to evolve a structure that can accommodate more unknown service chains. Density Control method achieves stable and high service chain accommodation ratios in multiple evolution paths. In addition, the development cost used to apply Density Control method is independent of the number or length of future service chains. This provides an advantage for changing the service functions structure in the future for a long period of time, because other methods require

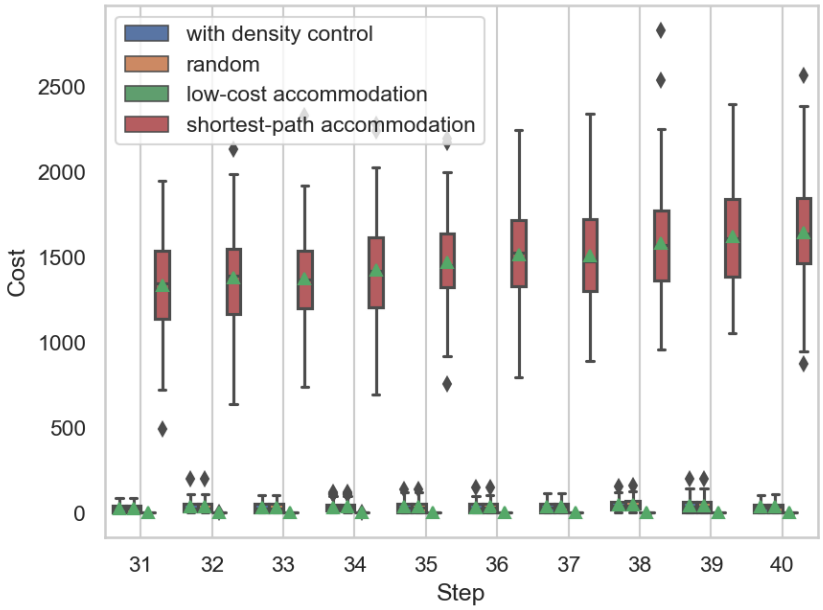

Figure 5.12: Development cost

different costs to accommodate depending on the number or length of service chains and it is difficult to predict service chains that will arise in the future. We expect that this advantage becomes more significant as the size of the service functions network and service chains become larger.

5.5 Conclusion

To accommodate large numbers of services at low cost, the service design needs to be adaptable to user requirements and environmental changes. When large-scale environmental changes arise, it remains necessary to change the core/periphery roles of functions, and reconfiguration of only the peripheral functions may not be sufficient to adapt to such changes. In this chapter, we proposed a evolvable service structure that can efficiently accommodate various service chains with low development cost by controlling the density of service functions. Our proposed method efficiently accommodates a many service chains for cases following multiple evolutionary paths. Future work

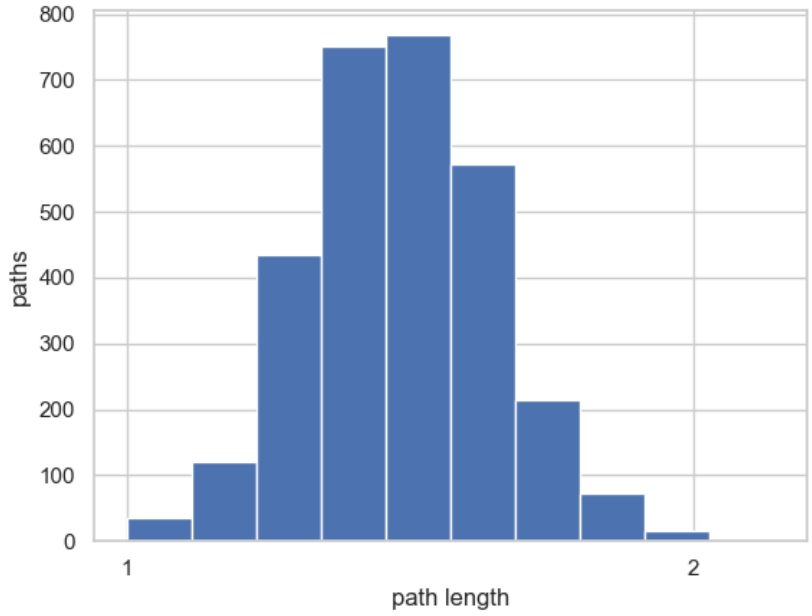

Figure 5.13: Chain length (Density Control)

includes addressing the problem of where to place service functions and providing a service structure for the unavailability of some service functions.

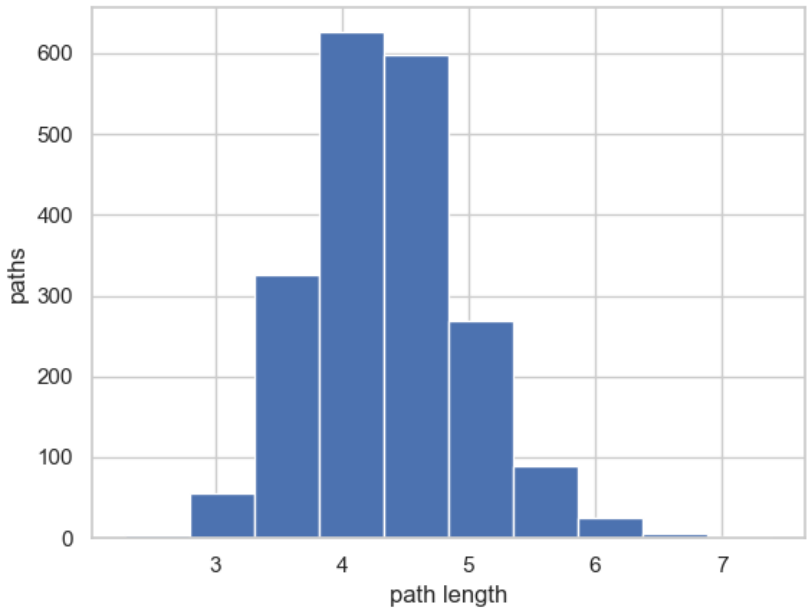

Figure 5.14: Chain length (Low-Cost Accommodation)

Chapter 6

Conclusion

Many new network-oriented services have been developed in recent years, and these services are expected to be virtualized in multi-access edge computing (MEC) environments, which are being standardized along with 5G. To accommodate large numbers of services at low cost, the service design needs to be adaptable to user requirements and environmental changes. Module-based design have been widely introduced in the software design, but the concept of modularization alone is not sufficient because development costs will change depending on how the service is divided. Module-based design enables to develop different services by combining the modules, but it is difficult to properly divide the service into modules at the design phase of the service. When design decisions that are expedient in the short term, the costs of maintaining and adapting this system in future increase, and it is known as technical debt. Therefore, a service design requires not only modularization, but also efficient adaptation to environmental changes.

We first investigated the design principles and the placement policies that reduce the cost of designing and developing VNFs for accommodating new service requests. As for the design policy, we introduce a Core/Periphery-Based Design (CPBD) that utilizes the core/periphery concept for developing VNFs. In CPBD, “core” VNFs are developed in advance and repeatedly used to accommodate future service requests. While “core” VNFs are common to current and future service requests, “periphery” VNFs are developed and customized for each service request. Next, we

investigate the placement policies of VNFs for CPBD to fully utilize the nature of their core/periphery structure. In addition, we examined the Center-Located Core/Periphery placement (CLCP) policy and the Geographically-Distributed Core/Periphery placement (GDCP) policy, and evaluate the long-term cost of the NFV system under resource restrictions to run VNFs. Our results show that CPBD reduces the long-term cost of design and development of VNFs by 23% compared to the design with no core VNFs. Moreover, in the case of no resource restrictions, both CLCP and GDCP reduce the long-term costs of placing and connecting VNFs by 15% compared to the existing VNF placement algorithm. With resource constraints, GDCP reduces the long-term costs over CLCP by 11%.

Second, we introduced a core/periphery structure for service components, which is known as a model for flexible behavior in biological systems, and design and implement a network-oriented mixed reality service based on this structure. To utilize the flexibility of a core/periphery structure, we regarded core functions as those whose behaviors remain unchanged even when there are changes in user requests or the environment. In contrast, peripheral functions are those whose behaviors can change under such circumstances. Experiments revealed that implementation costs are reduced while retaining increases in service response time to less than 31 ms. These results showed that taking advantage of a core/periphery structure allows appropriate division of service functions and placement of functions in a MEC environment, with only small penalties on latency and at a low implementation cost.

Third, we evaluated the core/periphery-based service structure in the following two aspects more pragmatically. The first one is the service scenario to use in our experiment. We focused 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 evaluate our service design in terms of the complexity of the source code and overhead for information sharing. To investigate the amount of penalties on sharing information, we implement a service and measured the penalty through experiments on actual devices. The second one is the metric to represent the implementation cost. We introduced 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., a modern software development. We used the cyclomatic complexity, which is

the number of independent paths from the start to the end of the program as the metric to evaluate the implementation cost. Our experiment showed that an information processing platform using a core/periphery structure is adaptable to environmental changes at a small cost by reusing the core and recreating only the periphery.

Finally, we proposed an evolvable structure of service functions network based on a core/periphery structure, that is, a network that can change the system at low cost with maintaining the ability to provide unknown services, as evolvable service structures. We proposed a method that can efficiently accommodate various service chains with low development cost by controlling the density of service functions. Our simulation revealed that our proposed method efficiently accommodates a many service chains for cases following multiple evolutionary paths.

In summary, we introduced a core/periphery structure for service components, which is known as a model for flexible behavior in biological systems, and revealed that taking advantage of a core/periphery structure allows appropriate division of service functions and placement of functions in a MEC environment. By reusing core functions and modifying only the peripheral functions, the system adapts to environmental changes with low cost. Also, we proposed a method to evolve the service functions network based on a core/periphery structure. Against large environmental changes, our method enables the service functions network to accommodate service chains in which a service function that was not previously used as a core is used as a core by controlling the density of service functions.

Future work includes addressing the problem of where to place service functions. Dividing functions and placing them in different devices creates extra communication paths and penalty of service responsiveness. While we have shown that the penalty for dividing services into core and peripheral functions and placing them in different device is tolerable. However, in service structures that evolve to adapt to large environmental changes, the peripheral functions sometimes change their role to core functions. We consider that migration of the functions or distributed deployment of copies are possible. Determining how to deploy them based on deployment costs and service responsiveness will help to implement our evolvable service structure. Also, providing a service structure for the unavailability of some service functions. We considered only the case where the service functions increase, but there are cases where some of service functions become unavailable.

Bibliography

[1] S. Tachi, "Telexistence: Enabling humans to be virtually ubiquitous," *IEEE Computer Graphics and Applications*, vol. 36, no. 1, pp. 8–14, Jan. 2016.

[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, no. 4, pp. 2359–2391, Jun. 2017.

[3] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing a key technology towards 5G," Sep. 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, no. 3, pp. 1657–1681, May 2017.

[5] A. MacCormack and D. J. Sturtevant, "Technical debt and system architecture: The impact of coupling on defect-related activity," *Journal of Systems and Software*, vol. 120, pp. 170–182, Oct. 2016.

[6] Baldwin, C.Y. and Clark, K.B., "Managing in an age of modularity," *Harvard Business Review*, vol. 75, pp. 84–93, Sep. 1997.

[7] A. Albers, N. Bursac, H. Scherer, C. Birk, J. Powelske, and S. Muschik, "Model-based systems engineering in modular design," *Design Science*, vol. 5, pp. 1–33, 2019.

[8] 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, Sep. 2013.

[9] V. Miele, R. Ramos-Jiliberto, and D. P. Vázquez, "Core–periphery dynamics in a plant–pollinator network," *Journal of Animal Ecology*, vol. 89, no. 7, pp. 1670–1677, Mar. 2020. [Online]. Available: https://besjournals.onlinelibrary.wiley.com/doi/abs/10.1111/1365-2656.13217

[10] IPLOOK, "Network Exposure Function(NEF) — Our High-performing Core Network," https://www.iplook.com/products/5gc-nef Accessed 25 April 2022.

[11] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, 2018, pp. 1–10.

[12] G. Liu, J. Wang, Y. Tian, Z. Yang, and Z. Wu, "Mobility-aware dynamic service placement for edge computing," *EAI Endorsed Transactions on Internet of Things*, vol. 5, p. 163922, 07 2018.

[13] Y. Tsukui, S. Arakawa, S. Takagi, and M. Murata, "Design and placements of virtualized network functions for dynamically changing service requests based on a core/periphery structure," *IEEE Access*, vol. 8, pp. 166 294 –166 303, Sep 2020.

[14] S. Takagi, S. Arakawa, and M. Murata, "On the implementation and evaluation of a network-oriented mixed reality service based on core/periphery structure," *IEICE Technical Report (NS2019-218)*, pp. 221–226, Mar. 2020.

[15] ——, "Design, implementation and evaluation of core/periphery-based network-oriented mixed reality services," *Journal of Internet Services and Applications*, vol. 12, no. 1, pp. 1–10, Feb. 2022.

[16] ——, "Implementation and evaluation of a network-oriented service with environmental adaptability based on core/periphery structure," *IEICE Technical Report (NS2020-158)*, pp. 208–213, Mar. 2021.

[17] ——, "Design, implementation and evaluation of a network-oriented service with environmental adaptability based on core/periphery structure," *submitted for publication*, Dec. 2022.

[18] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[19] S. Takagi, S. Arakawa, and M. Murata, "Evolvable design of network-oriented services based on core/periphery structure," *submitted for publication*, Jan. 2023.

[20] S. Kim, S. Park, Y. Kim, S. Kim, and K. Lee, "VNF-EQ: Dynamic placement of virtual network functions for energy efficiency and QoS guarantee in NFV," *Cluster Computing*, vol. 20, no. 3, pp. 2107–2117, Sep. 2017.

[21] Y. Nam, S. Song, and J.-M. Chung, "Clustered NFV service chaining optimization in mobile edge clouds," *IEEE Communications Letters*, vol. 21, no. 2, pp. 350–353, Oct. 2017.

[22] Q. Sun, P. Lu, W. Lu, and Z. Zhu, "Forecast-assisted NFV service chain deployment based on affiliation-aware vNF placement," in *Proceedings of IEEE Global Communications Conference (GLOBECOM)*, Dec. 2016, pp. 1–6.

[23] M. Richards, *Software Architecture Patterns*. O'Reilly Media, Inc., Feb. 2015.

[24] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Management Science*, vol. 52, no. 7, pp. 1015–1030, Jul. 2006.

[25] A. MacCormack, "The architecture of complex systems: Do "core-periphery" structures dominate?" *Academy of Management Proceedings*, vol. 2010, no. 1, pp. 1–6, Nov. 2010.

[26] H. P. Breivold, I. Crnkovic, and P. J. Eriksson, "Analyzing software evolvability," in *32nd Annual IEEE International Computer Software and Applications Conference*. IEEE, Aug. 2008, pp. 327–330.

[27] W. Hasselbrin, "Microservices for scalability: Keynote talk abstract," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM, Mar. 2016, pp. 133–134.

[28] 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*, vol. 20, no. 2, Feb. 2019.

[29] M. Wajahat, B. Balasubramanian, A. Gandhi, G. Jung, and S. P. Narayanan, "A model-driven graybox approach to rehoming service chains," in *Proceedings of IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sep. 2018, pp. 116–122.

[30] V. Latora and M. Marchiori, "Efficient behavior of small-world networks," *Physical Review E 87, 19801*, no. 19, Oct. 2001.

[31] M. E.J. Newman, "Modularity and community structure in networks," *Proceedings of the national academy of sciences*, vol. 103, no. 23, pp. 8577–8582, Jun. 2006.

[32] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, Oct. 2008.

[33] ANA, "ANA Avatar," https://ana-avatar.com, Accessed 12 November 2021, 2018.

[34] 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 *Proceedings of 2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, Mar. 2019, pp. 1241–1242.

[35] R. Ii, T. Hudson, A. Seeger, H. Weber, J. Juliano, and A. Helser, "VRPN: A device-independent, network-transparent VR peripheral system," in *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, Jun. 2001, pp. 55–61.

[36] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[37] M. Papazoglou and W. van den Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The VLDB Journal*, no. 16, pp. 389–415, Mar. 2007.

[38] 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, no. 3, pp. 42–50, Sep. 2019.

[39] 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, no. 3, pp. 28–41, Sep. 2019.

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

[41] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," *CoRR*, vol. abs/1804.02767, Apr. 2018. [Online]. Available: http://arxiv.org/abs/1804.02767

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

[43] Softbank Robotics, "Pepper press kit," https://www.softbankrobotics.com/emea/sites/default/files/press-kit/Pepper-press-kit_0.pdf Accessed 13 July 2021.

[44] OpenCVteam, "OpenCV," https://opencv.org Accessed 13 July 2021.

[45] "FFmpeg," https://www.ffmpeg.org/ Accessed 13 July 2021.

[46] "Microsoft HoloLens," https://www.microsoft.com/ja-jp/hololens Accessed 13 July 2021.

[47] A. Mukherjee, N. Dey, and D. De, "Edgedrone: QoS aware MQTT middleware for mobile edge computing in opportunistic internet of drone things," *Computer Communications*, vol.

152, pp. 93–108, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366419315750

[48] "Hololens-peppercontroller," https://github.com/s-takagi15/HoloLens-Peppercontroller, Aug. 2020.

[49] F. Galton, "On instruments for (1) testing perception of differences of tint and for (2) determining reaction time." *Journal of the Anthropological Institute*, vol. 19, pp. 27–29, 1899.

[50] K. von Fieandt, A. Huhtala, P. Kullberg, and K. Saarl, "Personal tempo and phenomenal time at different age levels," *Reports from the Psychological Institute*, vol. 2, 1956.

[51] A. T. Welford, "Motor performance. in j. e. birren and k. w. schaie (eds.), handbook of the psychology of aging," *Van Nostrand Reinhold, New York*, pp. 450–496, 1977.

[52] ——, "Choice reaction time: Basic concepts." *A. T. Welford (Ed.), Reaction Times. Academic Press, New York*, pp. 73–128, 1980.

[53] J. T. Brebner and A. T. Welford, "Introduction: an historical background sketch," *A. T. Welford (Ed.), Reaction Times. Academic Press, New York*, pp. 1–23, 1980.

[54] Open source cloud computing infrastructure - OpenStack, "OpenStack," https://www.openstack.org/ Accessed 13 July 2021.

[55] E. Foundation, "Eclipse Mosquitto," https://mosquitto.org Accessed 13 July 2021.

[56] AJA Video Systems, "HELO," https://www.aja-jp.com/products/helo Accessed 13 July 2021.

[57] Softbank Robotics, "NAO the versatile humanoid robot," https://www.softbankrobotics.com/emea/sites/default/files/press-kit/NAO-press-kit-EN.pdf Accessed 31 July 2021.

[58] ETSI, "5G; Extended Reality (XR) in 5G," *3GPP TR 26.928 version 16.1.0 Release 16*, Jan. 2021.

[59] S. Gu, C. H. Xia, R. Ciric, T. M. Moore, R. C. Gur, R. E. Gur, T. D. Satterthwaite, and D. S. Bassett, "Unifying the notions of modularity and core–periphery structure in functional brain networks during youth," *Cerebral Cortex*, vol. 30, no. 3, pp. 1087–1102, Aug. 2019. [Online]. Available: https://doi.org/10.1093/cercor/bhz150