IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 27, 2022, accepted February 16, 2022, date of publication February 28, 2022, date of current version March 9, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3155113

LISP-Based Control Plane for Service Connectivity in Multi-Cluster Cloud Systems

KYOUNGJAE SUN, JANGWON LEE, AND YOUNGHAN KIM, (Member, IEEE)
School of Electronic Engineering, Soongsil University, Dongjak-gu 06335, Republic of Korea

Corresponding author: Younghan Kim (younghak@ssu.ac.kr)

This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) funded by the Korean Government (MSIT) through the Development of Fast and Automatic Service Recovery and Transition Software in Hybrid Cloud Environment under Grant 2020-0-00946.

ABSTRACT In mobile networks, the edge cloud environment has emerged to provide various services to users. When the mobile user moves to another location while connecting the edge cloud service, to maintain an optimal service path, it is necessary to migrate the service to a new edge cloud close to the location where the user is newly connected. Several methods have been proposed to integrate edge cloud service migration technology with existing mobility management protocols in mobile networks, and a location/ID separation protocol (LISP)-based service optimal path management method between edge clouds was recently proposed. However, for real implementation of edge cloud with existing platform specified to containerized infrastructure, where the address of service is allocated randomly and locally, it is hard to discover appropriate service from the external client as well as maintain service connectivity when service location is changed. To solve that, this paper proposed the LISP-based service ID management system which provides uniform access for equivalent services running on different edge clouds. To apply the proposed ID management system on real distributed edge cloud systems, the mobility management and edge cloud networking systems were integrated as shown in the testbed implementation. This shows the possibility of combining the isolated mobility management for mobile users and internal identification of edge services using LISP for reducing the interruption and delay of edge service for mobile users. As a result of the experiments conducted on a real testbed, our proposed system was verified to enable a change in the routing path while maintaining a single service ID between different edge clouds, and the delay time for path reconfiguration is reduced compared to the existing method.

INDEX TERMS Location/ID separation protocol (LISP), Kubernetes, multi-cluster, service connectivity.

I. INTRODUCTION

Distributed edge cloud environments have the advantage of minimizing the delay in accessing services and distributing traffic by deploying virtualized services close to user access locations [1]. To provide these advantages in mobile networks where users are roaming across multiple edge clouds, various methods have been proposed to deploy equivalent services in multiple locations or to move virtualized service proximity to the user's new location. However, when the location of the service is changed, the existing session should be released by changing the IP address of the service, which increases the service downtime for re-discovering the service location and re-establishing the sessions. The traditional method to solve this problem is to adapt IP mobility management to service mobility management, which maintains service IP addresses even when the service location is changed [2]–[5], however the traffic path cannot be optimized by anchoring the address at the edge cloud, where the service is initially instantiated. To solve this problem, several methods have been researched to separate the IP context into the ID and location [6]–[10]. In particular, in our previous studies [9]–[10] the location/ID separation protocol (LISP) [11] was used to assign a unique service regardless of its location. With the enhanced mapping system of the service ID and its locator, our work verified that the nearest service is always accessed while minimizing the service downtime even when the mobile user is rapidly roaming across multiple edge clouds. Despite these efforts, to adapt this method to real implementation, a detailed method is required to assign the same service ID

The associate editor coordinating the review of this manuscript and approving it for publication was Hosam El-Ocla.

and manage them in different edge clouds that operate independently, which should be compatible with the management and orchestration functions running for the edge cloud.

For edge cloud deployment, because resources are more constrained at the edge, a container-based infrastructure with lightweight virtualization technology has been introduced to increase the agility and flexibility of application deployment [12]. For container-based service deployment and execution, resource provisioning, networking, and failure management, Kubernetes [13] is the most widely used orchestration tool. For the configuration of an edge cloud environment, although the physical nodes in a single cluster can be deployed at the edge, independent clusters are generally deployed in each geographic location to provide localization and management distribution. However, according to the basic design principles of Kubernetes networking, to improve the flexibility, the IP address assigned to a pod is non-permanent, and thus the IP address is changed when a pod is reinitialized [14]. For this reason, a fixed endpoint cannot be supported when the service is dynamically instantiated or moved to another edge cloud. To provide consistent service to endpoint nodes, Kubernetes defines a service object that acts as an access point for application containers by allocating a permanent IP address [15]. This IP address can be used as a service ID; however, it is necessary to synchronize unique service IDs across all edge clusters for a specific service to avoid ID duplication. Currently, there are no proposals for allocating the same service ID to different edge clusters. When we use the LISP control plane, a centralized control plane can help in the efficient management of service IDs while avoiding duplication. To do so, the LISP control plane should have a function to communicate for querying and responding to service ID requests in the service instantiation phase at each cluster.

In addition to service IP address synchronization for use as a unique ID, because the IP address is still private, additional operations are required to expose the service to the external internet to provide connectivity for external clients. A common method used in current systems for exposing services to external clients is to expose services using the public IP address of the gateway or load balancer of the cluster, and then translate those addresses into the internal IP addresses of the services [16], [17]. In such an environment, a user can acquire the proper IP address for a target service by using a DNS server that records the service names and corresponding IP addresses. However, when using this service exposure approach, a user must rediscover the service IP to change their connectivity to equivalent services, which can increase the service downtime during the reestablishment of service connections when a user moves or a service failure occurs. Several solutions have been proposed to resolve DNS failures and conduct dynamic server selection [18], [19]. However, DNS-based methods cannot maintain the target IP address of service and break session connectivity when a target service changes. For a seamless change in service connectivity between different clusters with minimal discovery time, it is necessary to minimize the location dependency of the services, which can be achieved by separating the IP contexts into locations and identifications. When we use the LISP approach, this can be solved by implementing the LISP router function in front of each edge cluster to encapsulate a packet that can be forward between the LISP router where the client is connected. Because the LISP control plane manages service ID-Locator mappings, it can replace the DNS system as well as maintain service connectivity even when the service location is changed.

In this study, based on our previous work, we designed and implemented LISP-based service ID management functions for a real multi-cluster cloud system to allocate a single service IP to equivalent services. Instead of using the current system, which allocates service IPs from each cluster orchestration system individually, with the proposed system architecture, a service ID management function located in the centralized control plane manages the IP address pool

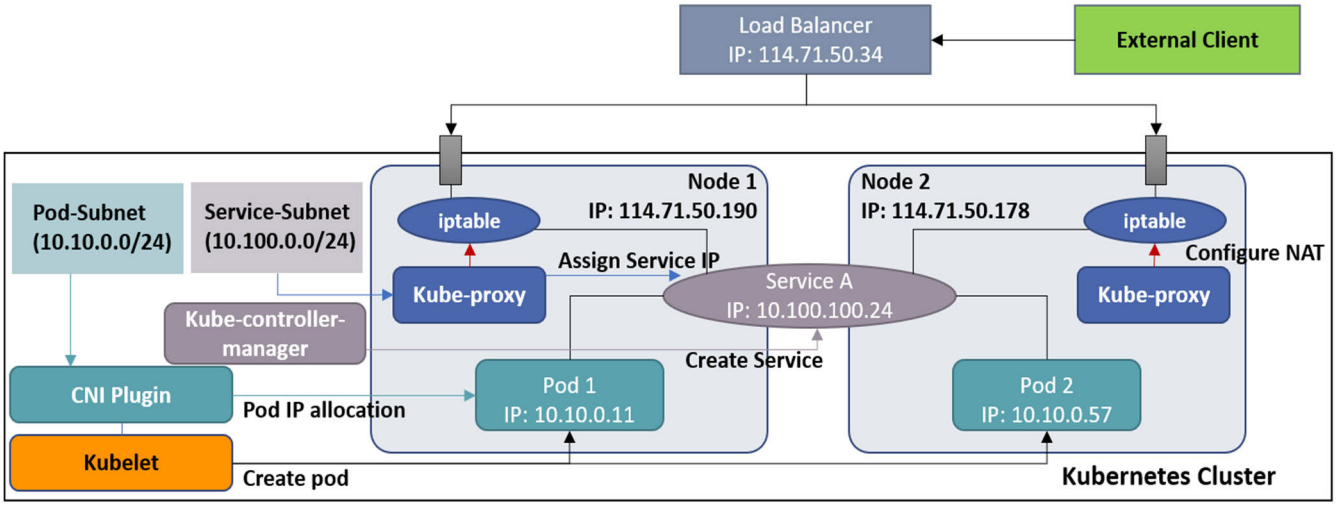

FIGURE 1. Kubernetes networking architecture.

for service IDs that can be allocated across multiple clusters. When a service is requested to be hosted on the cluster, the service ID management agent functions in each cluster orchestrator requests a service ID from the centralized control plane to create a service. In this manner, the same service running on different clusters can be assigned the same service IP, allowing external clients to quickly change their service connection to the same service running on another cluster by changing only their locator without an additional discovery procedure used to change the identifier. For a real implementation, the proposed functions were designed to interwork with Kubernetes functions. In particular, to improve the compatibility, they were designed transparently by considering the current networking models and plugins for internal networking. By implementation and experimentation on the testbed, the proposed the LISP-based service ID management shows that the service could be accessed by mobile users optimally with minor processing delay. The remainder of this paper is organized as follows. In Section 2, we analyze the Kubernetes-based container networking environment and briefly describe the LISP protocol. Section 3 describes the proposed architecture and its operation. In Section 4, we describe an operation verification through the implementation of the proposed method on a testbed. Finally, we provide some concluding remarks in section 5.

II. RELATED WORK

A. KUBERNETES-BASED MULTICLUSTER NETWORKING

Kubernetes configures several adjacent physical nodes in a single cluster and deploys pods, which are defined as groups of one or more containers with shared storage and network resources. Kubernetes defined several control plane functions for managing resources and additional methods. Kube-apiserver is the frontend of the Kubernetes control plane that exposes APIs for management and orchestration, which allows access from other control functions and third-party services for managing clusters. When deploying a pod to a node in a cluster, the Kube-scheduler function selects an appropriate node upon which the pod will operate by considering the resource status and operation policies. These control plane functions are located on the master node in the Kubernetes cluster, and each node in the cluster creates containers for pods using runtime container modules such as Docker. Each worker node utilizes Kubelet, which is an agent for managing the cycle of an actual pod, and Kube-proxy, which is a network proxy that manages network rules for pods within a node.

Figure 1 shows the Kubernetes networking architecture. To provide networking models between pods within a Kubernetes cluster, Kubernetes provides a network proxy function that manages routing rules within the host OS and various container network interface (CNI) plugins [21]. For example, Flannel [22], which is a representative CNI plugin, provides an L3 overlay network between pods. SDN-based networking models are also available, such as those of tungsten fabric [23]. When creating a pod, the CNI plugin allocates an IP address and forwards it using a virtualized routing function or modifying the internal routing table of the host OS kernel. On a host with a Linux OS, the CNI can dynamically configure routing rules by accessing iptables, which provide routing rules and NAT functions in Linux through the kube-proxy function.

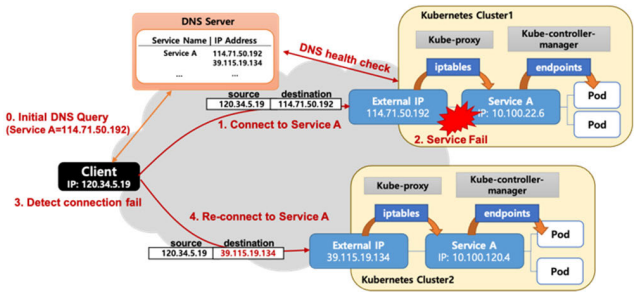

FIGURE 2. Limitations of service access in a multi-cluster environment.

A service object is an abstraction that defines a logical set of pods and a policy by which to access them. Each service object has a unique IP address called ClusterIP. Once a service is created, its ClusterIP does not change, even when pods are dynamically created or released for the service, meaning that one can always access the target service within a cluster. The mapping of the service and its connected pods is managed by the endpoint controller function in the kube-controller-manager, which allocates ClusterIPs within a predefined address pool for internal IPs when a service object is created. Because a ClusterIP is a private IP, an additional procedure is required to expose the ClusterIP as a public IP address that can be accessed by an external client. In Kubernetes, instead of exposing a ClusterIP directly, the public IP address assigned to the interface of the physical node or GW address of the corresponding cluster is exposed. Public IP addresses are generally recorded in a public DNS system and mapped to service names from the outside. This method of exposing services to the external network has limitations in terms of providing consistent reachability when equivalent services are running on different clusters because different IP addresses for the same service are defined as destinations.

Figure 2 presents an example of the limitations of external access in a multi-cluster environment. The problems that can occur in the multi-cluster environment defined in the figure are summarized below. First, when the communicating client moves after acquiring the public IP address of a specific cluster through the initial DNS query process or when it is necessary to change the path because of a service failure in the current cluster, additional procedures are required to discover the IP address of the service located in the other cluster. For this process, the client must detect a connection failure and request the IP address corresponding to the service name from the DNS server. In addition, to keep the DNS records up to date, the DNS server must periodically check all services in all clusters. For the DNS-based service discovery process, additional functions such as dynamic server selection [9],

which can apply DNS requests to respond to mobile users or communication node failures, have been studied. However, because the external client and DNS are not managed by the cluster management function, it is difficult to detect failures and update the statuses quickly. Although setting a small time-to-live (TTL) value for DNS records is also possible, one must be careful because setting a TTL that is too small can lead to vulnerability to DDoS attacks [24]. In addition, it is necessary to translate an IP address multiple times to forward packets to a service. The first translation is required between the public IP address and ClusterIP, and the second translation is required between the ClusterIP and pod IP addresses. These processes may impact the data plane performance by increasing the total number of services or packets.

B. LISP

In LISP [11], an IP address is divided into two types of addresses: endpoint ID (EID) and routing locator (RLOC). The EID is defined as a globally unique IP address assigned to an endpoint node and is used to establish end-to-end connections between pairs of nodes. The IP packets between two EIDs are encapsulated and delivered between LISP routers called xTRs (ingress/egress tunneling routers), where the EIDs are connected using RLOCs, which represent the address of each LISP router. When an EID is connected to the LISP network, it is mapped according to the RLOC of its LISP router. This mapping information is managed using a logically centralized LISP mapping system. In the LISP standard documents [11], control plane messages between the LISP mapping system and LISP routers are defined to update and query EID-RLOC mappings. LISP-based ID/location separation can provide more effective and native support for mobility management compared to existing IP-based infrastructure, which eliminates the problem of non-optimal paths induced by the forwarding of packets through the anchor functions defined in existing IP-based mobility solutions.

Several recent studies have proposed the application of LISP to mobile users and virtualized services running on cloud infrastructure [6]–[9]. In these studies, an EID was assigned to each virtualized service, similar to a mobile user, and mapped to an RLOC, which is defined as the IP address of the LISP router functions deployed on each cloud. Using LISP in cloud infrastructure, because services can be migrated to other clouds without changing their EIDs, previous methods have focused on integrating the service migration process with a LISP control plane operation to optimize traffic paths for migrating services. However, when the equivalent service is already running in the edge cloud, this service has a different identifier as an independent unique entity so it is hard to determine as the same service in network perspective. Therefore, the results may cause such as migrating the service to edge cloud where already the same service is running on or beaking session by re-establishing a connection to the service having a new identifier. To solve that, in our previous study [9], we defined a separated mapping system for effectively managing service EIDs of multiple locators for equivalent services running on multiple edge clouds. In addition, we extended the fields in the user EID mapping table to record the corresponding service EIDs to provide a rapid path configuration during user mobility. However, to provide user mobility support and dynamic path configuration for the services proposed in previous studies in a real environment, it is necessary to consider interworking with cloud management and orchestration systems.

To implement a LISP-based cloud environment, previous studies [6]–[8] used a hypervisor to virtualize physical resources and deploy services as virtual machines (VMs). In such approaches, LISP components are deployed as VMs and executed on each cloud, similar to services [7], [8], or as processes running on the host OS kernel [6]. Although these approaches extend the role of the hypervisor to provide interworking with LISP control management functions, this type of method cannot be applied to the Kubernetes platform because there is no hypervisor. For container-based infrastructures, the author of [25] developed a LISP-enabled CNI plugin for a Kubernetes-based container environment to support service discovery and routing between nodes in a single-cluster environment. However, because this method is only available within a single cluster and cannot expand networking to a multi-cluster environment and an external network, the aforementioned limitations of the Kubernetes environment have not been overcome.

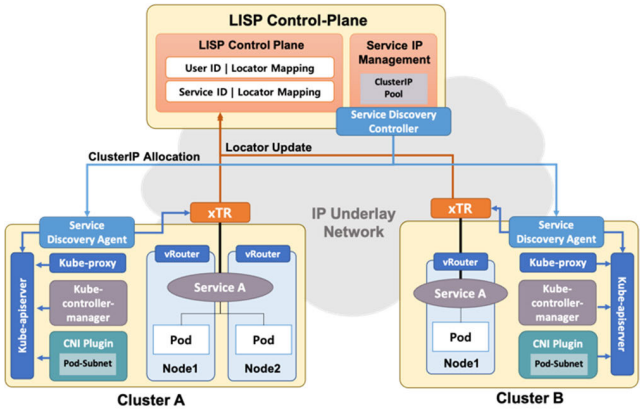

FIGURE 3. Proposed LISP control plane for a Kubernetes multi-cluster environment.

III. LISP-BASED CONTROL PLANE IN MULTICLUSTER CLOUDS

A LISP-based control plane architecture for providing service discovery and external connections for a Kubernetes multi-cluster environment is presented in Figure 3. In this figure, the overall infrastructure is composed of independent Kubernetes clusters and the LISP control plane that integrates and manages service mapping between multiple clusters. In the control plane, a service IP management function is defined in the centralized LISP control plane along with the service-mapping system, which was proposed in our previous study [9]. For communication between the LISP control plane and Kubernetes clusters, a service discovery controller

in the control plane and a service discovery agent in each Kubernetes master node is defined. As a LISP data plane component, we have defined that the LISP xTR function is located in front of each cluster, similar to a gateway router.

A. SERVICE ID AND LOCATOR

Considering the design of the LISP-based distributed cloud system, the first step is to define an ID and locator for the container service. Because the IP address of each pod in Kubernetes is non-permanent according to the dynamic management strategy, it is unsuitable to use service IDs because one cannot guarantee their immutability, which is a requirement for identifiers in LISP. Instead, it is appropriate to define a ClusterIP as a service ID, which is an IP address allocated to a service object that can provide fixed IP access for pods defining the same service. Therefore, the management of the ClusterIP address pool, which was originally independently managed in each cluster, should be moved to the centralized LISP control plane. For the management of ClusterIPs, the service IP management component, which is newly defined, allocates an IP address to a service name requested from Kubernetes when the service is created. Within the ClusterIP address pool, the IP address for a service can be predefined according to the policy or allocated dynamically. In addition, equivalent service objects created in different clusters should be defined using the same name as the service in the template. From a management perspective, it is assumed that service names are consistently maintained between multiple clusters according to the policies of the network and service operators.

In the LISP, an RLOC is defined as a public IP address that can be routed in the IP underlay infrastructure and is assigned to an external interface of the xTR function. As shown in Figure 3, in the proposed architecture, an xTR function is located in front of each cluster, and thus a single cluster can be considered as a single LISP site. In a real implementation, an xTR can be deployed as a container image or host process on the master node, or can be implemented independently, similar to an external load balancer or gateway. In a centralized LISP mapping system, the RLOC address of an xTR is mapped as a locator for a service ID when the xTR updates the mapping of the service IDs by sending a Map-Register message. The LISP control plane has an independent mapping system for each service, and each service-mapping system supports the mapping of multiple RLOC values for a single service ID. When a user queries the location of a service ID, it is necessary to identify an appropriate locator for the equivalent services distributed throughout several locations. Although this is an important challenge, algorithms and optimization methods for this operation are beyond the scope of the present study. Similar to services, users are also assigned globally unique IDs in the LISP-based environment, and it is assumed that a user can determine the endpoint service ID used for a session connection using a predefined method or an additional procedure.

B. CONTROL PLANE OPERATION

For the requests and responses related to service ID allocation between the Kubernetes cluster and the LISP control plane, we define operations and procedures between the service discovery controller and the service discovery agent. The service discovery controller receives a service name from the agent and finds its ClusterIP in the lists formed by [ServiceName:ClusterIP], which are managed by the service IP management function. At this point, if a mapped ClusterIP value for the requested service name does not exist, the controller requests the service IP management function to allocate a new ClusterIP address for the service and requests the service ID mapping system to create a service mapping table for the assigned ClusterIP. Finally, the controller sends a response message to the agent in the Kubernetes cluster that requested the ClusterIP, allowing the creation of a new service object with the received ClusterIP in the cluster. The service

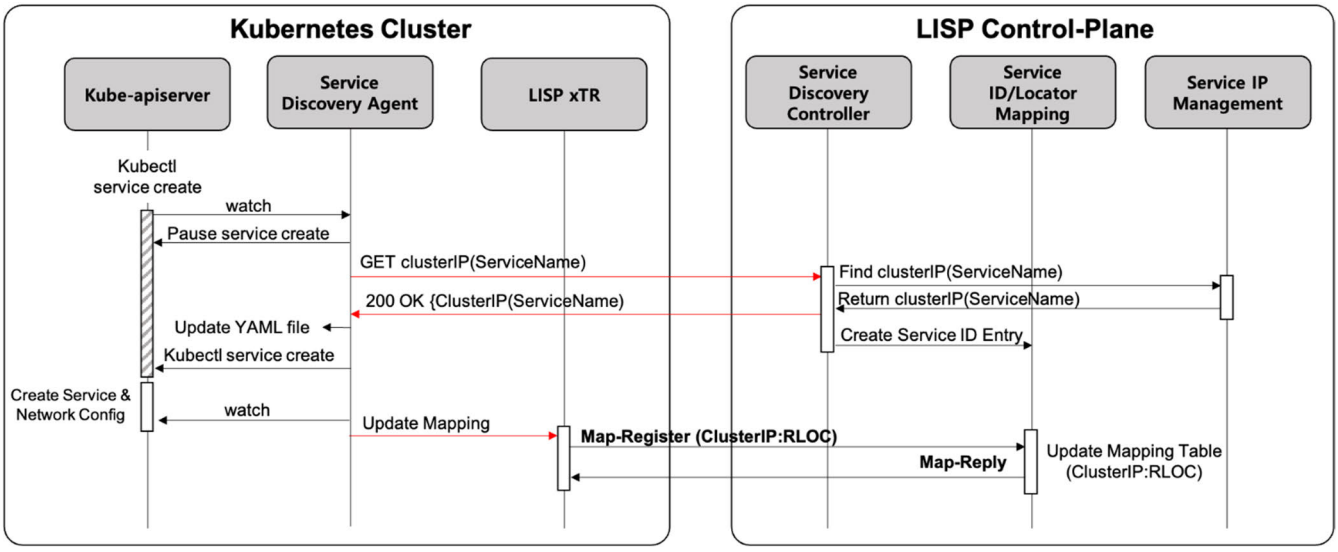

FIGURE 4. Control plane operation procedures.

discovery agent running on the master node of each cluster has the functionality to access the Kubernetes API. When a service creation is requested and the corresponding message arrives at the Kube-apiserver, the agent intercepts this message, reads the service name in the template of the target service, and sends a request message to the controller. When the agent receives a ClusterIP for a requested service name from the controller in the LISP control plane, the agent adds it to the target service template and resumes the service creation process through the Kubernetes API. The service creation process is temporarily paused while exchanging messages between the agent and controller.

After a service is created, a procedure is required to send the active ClusterIP to the xTR to update the mapping information for the service created in the cluster. In a common LISP network, an xTR can detect a node attachment by receiving the L3 attachment processes, such as a router advisement or router solicitation message. However, because a service object is a logical component, it does not generate an attachment procedure, unlike a common node, and an additional process is required to detect an active ClusterIP at xTR. Therefore, the service discovery agent verifies the completion of the service creation through the Kubernetes API and sends a LISP Map-Register message from the xTR of the cluster to the LISP control plane. Figure 4 shows the overall message flows between the Kubernetes cluster and the LISP control plane for creating a service and registering it in the mapping table, including the service discovery controller and agent functions.

C. DATA PLANE OPERATION

Networking in a multi-cluster environment is divided into networking within a single cluster, networking between clusters, and networking with external nodes. Networking within a single cluster is divided into networking between pods within a single node and networking between pods in different nodes. In the proposed architecture, LISP-based networking is not responsible for intra-cluster networking, and it can transparently connect an external network to the intra-cluster networking model provided by the CNI plugin, similar to existing methods. Therefore, intra-cluster networking between the pods and services on a single node or another node in a cluster is managed by the CNI plugin. Inter-cluster networking can be supported in two ways. The first is to implement a networking model based on the service IDs using the LISP control plane, which forwards packets between clusters by encapsulating the header from the xTR of each cluster in the RLOC addresses. When a pod sends a packet to a specific ClusterIP, the packet is first checked for information in the internal routing table configured by the Kube-proxy and CNI in the cluster. If the corresponding ClusterIP is in the entry, the destination service exists in the

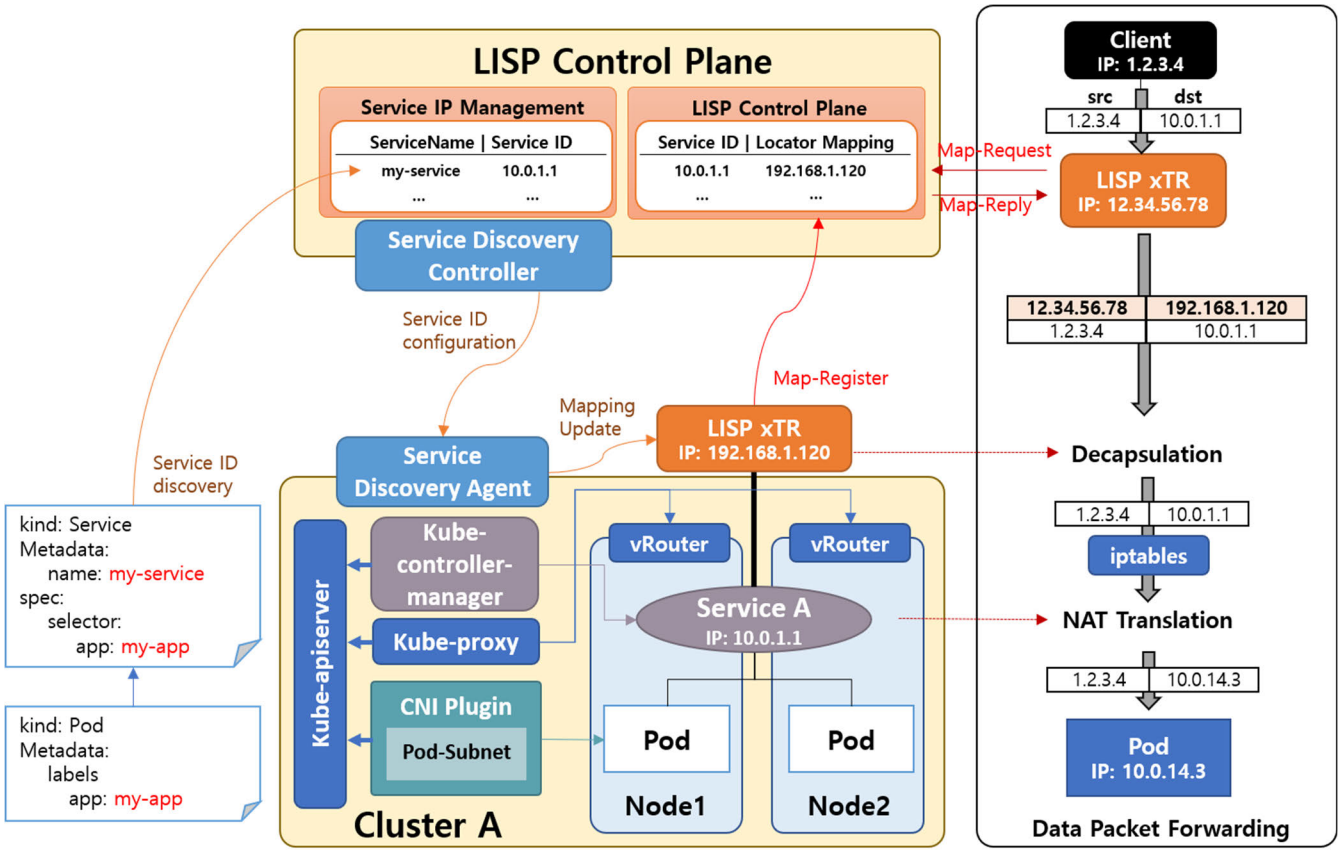

FIGURE 5. Overall operation of the proposed architecture.

same cluster, and thus it is possible to access the service object using the network provided by the CNI Plugin. By contrast, if the destination IP address does not exist in the entry, i.e., the target service does not exist in the same cluster, the packet is forwarded to the xTR of the cluster. Therefore, xTR should be set as the default gateway node in the internal routing table. Once the xTR receives a packet destined for a particular ClusterIP, it sends a Map-Request message to obtain the RLOC of the corresponding endpoint. The second method is to configure an inter-cluster networking model based on the CNI plugin. Networking between different clusters was supported by plugins. For example, the Tungsten Fabric CNI [23] deploys the BGP router function to each cluster and shares subnetwork information using this function. In the case of using a CNI-based networking model between clusters, the LISP xTR of each cluster is not responsible for packet forwarding between clusters and only manages networking between external clients.

Figure 5 presents the overall procedure for delivering packets from an external client to service inside the Kubernetes cluster. When the client sends a packet destined for a service ID, the xTR of the user's LISP site receiving that packet queries the RLOC mapping information for the service ID by sending a Map-Request to the LISP control plane. After receiving the RLOC included in the Map-Reply message from the control plane, the xTR encapsulates the original packet in an RLOC mapped to the endpoint ID and forwards it to the network. Packets destined for RLOC addresses are routable in the underlay network using common routing protocols; therefore, no additional operations are required to optimize the paths between xTRs. Packets arriving at the network interface of a node are delivered to the service object by looking up the internal routing table, translated into the IP address of the active pod mapped by the endpoint controller at the service object, and forwarded to the pod where the target application is running. In a non-LISP environment, with an exposure service using NodePort or gateway IPs, the translation of IP addresses occur at each node interface, and the service object undergoes two NAT operations. By contrast, when using the LISP, an IP address translation occurs only once at the service object, resulting in more efficient data plane operations for packet handling.

IV. IMPLEMENTATION AND ANALYSIS

A. TESTBED ENVIRONMENT

To set up a testbed environment for the proposed architecture, the LISP control plane and Kubernetes multi-cluster environment were constructed, as shown in Figure 6. In this figure, all nodes are created as VMs on a single physical machine (with an Intel Xeon 5220R 48 Core CPU and 238 GB of RAM). Each node and network between nodes were configured using OpenStack. Each cluster consists of one master node and two worker nodes, based on Kubernetes v1.18.10. To support networking within a single cluster and between clusters, Tungsten Fabric [23], which is an SDN-based approach, was installed as a CNI plugin. In a Tungsten-Fabric-based environment, networking inside a cluster is provided by performing virtual routing and forwarding (VRF) between each isolated namespace and providing a dynamic forwarding for VRF using an SDN-based forwarding

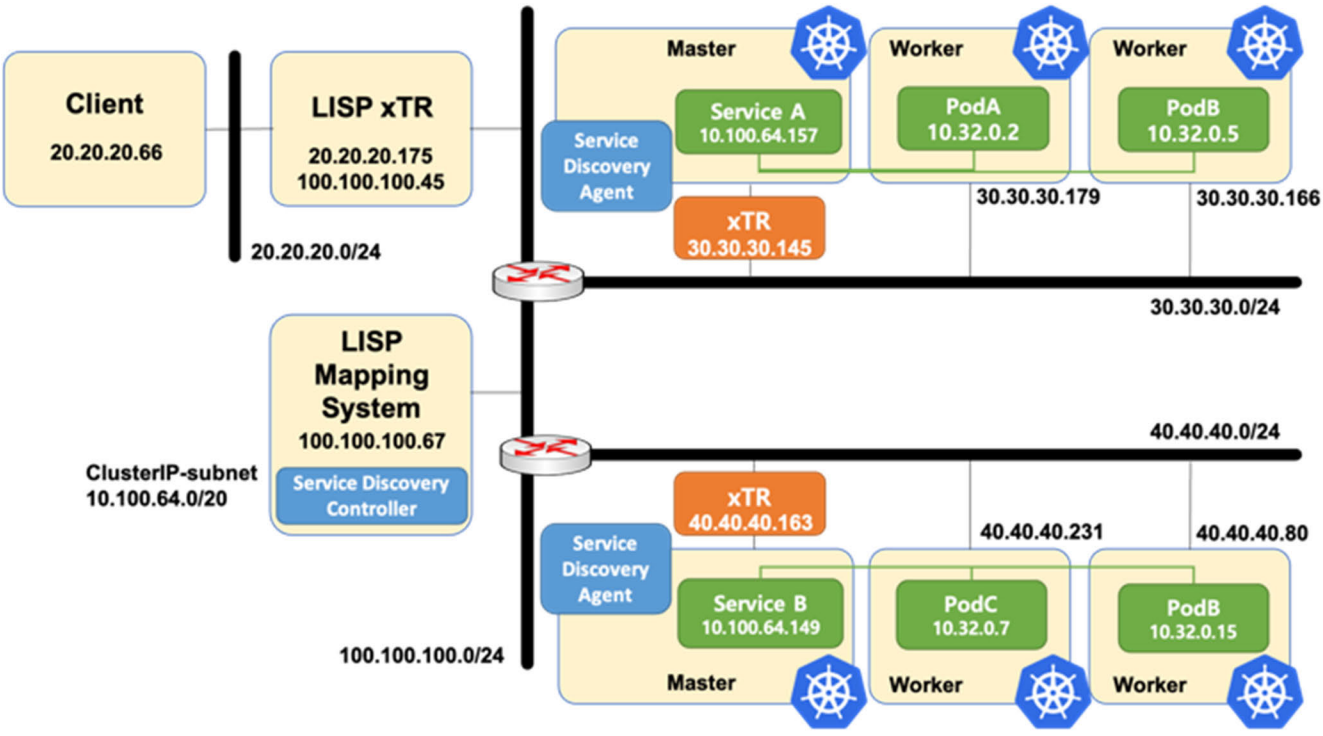

FIGURE 6. Testbed environment.

Pseudo Code 1 Service Discovery Controller in Cluster

```
watch_list_url = "/api/v1/watch/services"
  while watch process:
    if process is CREATE;
      pause CREATE process
      ServiceName = ServiceName(CREATE)
      packet.ServiceID = ServiceName
      packet.destIP = addr(SercviceDiscoveryAgent)
      send packet
      wait response;
  ...
    If reseponse is 200 OK;
      read packet.ClusterIP
      write packet.ClusterIP to
        template.ServiceName(ClusterIP)
      command CREATE(ServiceName) to kubeapi
```

Pseudo Code 2 Service Discovery Agent in the Control Plane

```
wait request;
if message is GET;
  read ServiceID = packet.ServiceName

  if ServiceID is exist in SID table;
    ClusterIP = ServiceID.IPaddr
    packet.CluterIP = ClusterIP
    destIP = srcIP(recievedPacket)
    send packet with 200 OK
  else;
    allocate ClusterIP(ServiceID)
    packet.ClusterIP = ClusterIP(ServiceID)
    destIP = srcIP(receivedPacket)
    send packet with 200 OK
```

policy enforcement. The control plane and xTR functions for providing LISP-based networking were implemented using Lispers.net, which is an open-source LISP implementation using Python [26]. The LISP mapping system, which is combined with the Map-Server and Map-Resolver, runs during each process on the LISP Mapping System VM shown in the figure. The xTR functions of each cluster were implemented on the OS of the master node. The xTR accessed by external clients was deployed on a separate Linux VM, as shown in the figure. The VM resource specifications for each node in the testbed are listed in Table 1. For communication between the service discovery controller and agent functions, we implemented a WebSocket program between the controller and agent, which was written in Python. Pseudo code for each component is written below. Code for service discovery controller includes operation to pause Kubernetes service creation, extract service name from template, send request packet with GET method, and restart service creation process after receiving IP address for the service ID. Code for service discovery agent includes operation to extract service ID from receiving request, determine whether ClusterIP of requested service is existing and response ClusterIP to requested discovery controller.

B. TESTBED EVALUATION

To evaluate the operations within the proposed testbed, we measured the performance of the control and data planes.

TABLE 1. Node specifications in the testbed.

Node	Resource specifications
LISP Control Plane	2 vCPU / 2 GB RAM / 10 GB Storage
Kubernetes Master Nodes	8 vCPU / 16 GB RAM / 160 GB Storage
Kubernetes Worker Nodes	2 vCPU / 4 GB RAM / 40 GB Storage
LISP xTR (Client-side)	2 vCPU / 4 GB RAM / 40 GB Storage
External Client	1 vCPU / 2 GB RAM / 20 GB Storage

In addition, we defined a scenario for evaluating the traffic path changes between multiple clusters and measured the performance under this scenario.

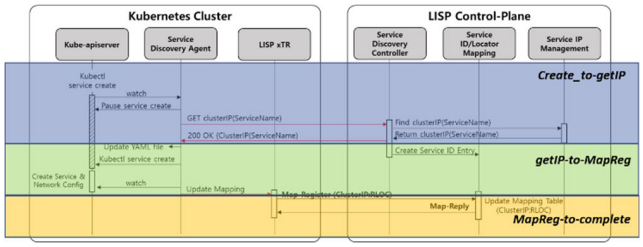

FIGURE 7. Separation into three phases for control plane evaluation.

1) CONTROL PLANE EVALUATION

To evaluate the operation of the control plane, we defined the entire control plane process from the generation of a service creation request to the updating of the mapping information for the generated service in the LISP control plane in three phases and measured the time required for each phase. Figure 7 presents the division of steps in the overall control plane operation procedure. First, *Create_to_getID* includes various operations, from requesting a service creation through the Kubectl command to receiving the service ID from the LISP Mapping System. In the *getID_to_MapReg* phase, ClusterIP is added to the template of the requested service, requests a service creation from the kube-apiserver, and watches for the completion of service creation to notify the xTR. In the last phase, *MapReg_to_complete* is defined as the completed process with a service created by sending the Map-Register from the xTR and updating the mapping table in the LISP mapping system. After completing this phase, the created service can be discovered by external clients.

To calculate the time required for each step, we collected logs for Kubernetes and the LISP control plane, and calculated the processing time using the timestamps in each log. Figure 8 presents a graph of the processing time for each phase. The processing time is divided into the Kubernetes process time, LISP control plane process time, and transmission time for packets between the two components. The total average processing time was approximately 1200 ms, and Kubernetes processing accounted for approximately 70% of the total time. Regarding the evaluation of individual

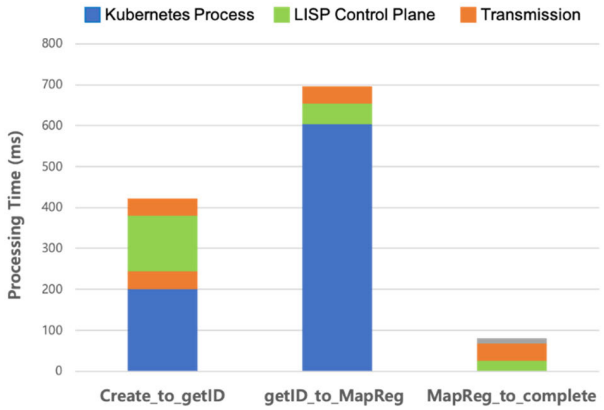

FIGURE 8. Control plane evaluation results.

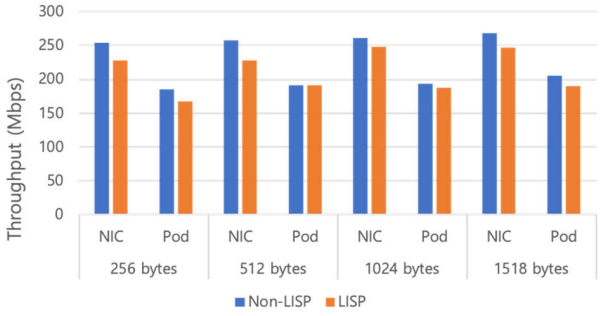

FIGURE 9. Data plane performance results.

phases, *getID_to_MapReg* required the longest processing time because creating a service object after receiving a ClusterIP from the LISP control plane is time-intensive. However, the same value may not be observed every time because the processing time of various configuration options varies, and an image of the relevant pod may or may not already exist. The operation time in the LISP control plane accounts for approximately 13% of the total time, and thus the effects of processing delays incurred through interoperation with the proposed LISP control plane do not significantly impact the overall delay.

2) DATA PLANE EVALUATION

To evaluate the data plane operations in a testbed using a tunnel-based packet transfer between xTR functions, we compared the packet throughput performance to that of the conventional method, which does not use tunneling. Because each xTR function must encapsulate/decapsulate packet headers for a data transfer, the proposed method can increase the processing overhead and impact the performance. To measure the data throughput between an external client and a pod running in a cluster, we used iPerf3 [27] to send/receive packets through a total of 10 threads simultaneously to record the average throughput. To run iPerf3 on a pod, we implemented a container using a network-multitool image [28] from Docker Hub, which supports iPerf3. Figure 9 presents the results for a data throughput between the external client and pod, while increasing the packet size from 256 to 1518 bytes, which is the maximum size that can be transmitted over Ethernet. To clarify any performance degradation, we also measured the path from the external client to xTR, which was implemented on the host's network interface. Compared to general forwarding, which does not use tunneling, the proposed environment that uses a tunneling-based data transfer between xTRs leads to a decrease in throughput of approximately 10%. This result indicates that the additional processing required for the tunneling mechanism, including encapsulation/decapsulation for packet headers in each xTR, leads to a slight performance degradation, which has already been observed in previous studies [29]. As another interesting result, the throughput decreases by approximately 28% for both methods when generating a path between the host interface and pod based on the target service. This means that data throughput is impacted by the process of translating ClusterIPs into actual pod IP addresses depending on the type of CNI plugin or internal networking mechanism. Therefore, it is expected that the performance can be improved by applying different CNI plugins or data plane acceleration technology.

3) PATH RECOVERY EVALUATION

To evaluate the actual path recovery procedure of the proposed architecture, because it is difficult to simulate the mobility of a client deployed as a VM in the testbed, we considered another scenario in which the service path was recovered by another cluster owing to a service failure in the previous cluster. Under this scenario, a service failure is detected by a Kubelet, which is a Kubernetes function running on each node in a cluster. When a failure is detected, a message is sent to the Kube-apiserver such that additional functions can take action according to the failure of the corresponding pods and services. By watching the Kube-apiserver periodically, the service discovery agent can detect failure events and request the xTR to send a message to the LISP mapping system to delete the location of a failed service. Upon receiving a request for mapping deletion from the xTR, the LISP control plane updates the mapping information and informs the xTR of the corresponding client to modify the forwarding path of the packets to a different location of an equivalent service.

Under this scenario, a service interruption time is incurred immediately after a service failure until the location information for a new service is updated in the client's xTR. Therefore, we measured the service interruption time to analyze the LISP-based path-recovery performance. For comparison with the proposed method, a non-LISP method for server reselection was also measured, where the client detects that a service is down and requests another IP address for the service from the DNS server. Under the non-LISP scenario, the bind9 DNS server was deployed in the testbed, and additional functions for detecting service failures and sending DNS queries to the DNS server were implemented in the client's

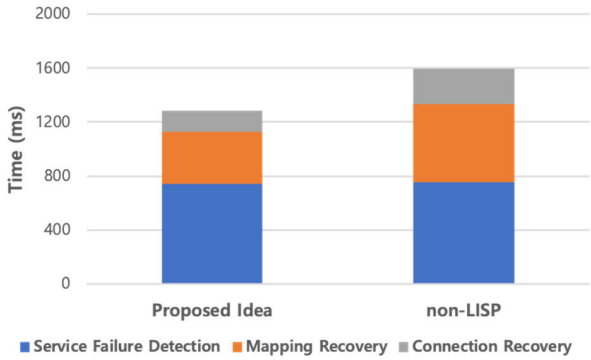

FIGURE 10. Path recovery delay results.

application code. Figure 10 presents the results of measuring the service interruption time during the path recovery process for a service failure in a testbed environment. These values were calculated from timestamps in the Kubernetes log, LISP control planes, and client applications. The average time from service failure to service resumption is approximately 1.2 s, whereas the non-LISP method requires approximately 1.55 s. The delay required for mapping recovery (i.e., changing the destination to the service instance running on another cluster) accounts for approximately 30% of the total delay in the proposed method and 36% of the total delay in the non-LISP method. Although the non-LISP method also recovers service connections quickly, it does not maintain the destination IP address, and thus the previous connection is completely broken. Our performance measurements indicate that a service path can be changed within 1.3 s, proving that service availability in an edge computing environment can be supported by providing a fast path failure recovery.

V. CONCLUSION

In this paper, we proposed LISP-based service management and a discovery mechanism for a multi-cluster cloud environment to provide service connectivity management. With the proposed architecture, by managing an IP address pool for the service objects initialized by each cluster, the LISP-based control plane provides uniform service IDs to external clients, regardless of the location of a service. Therefore, allocating the same ID for equivalent services is possible across multiple clusters. When a packet destined for a service ID address is sent from an external client, it can be delivered to the correct pod through tunneling between LISP xTRs, which is transparent to the current Kubernetes networking model using the CNI plugin. When a user moves to another location while communicating with a service running on an edge cloud, ID-based service discovery is conducted to identify an equivalent service close to the user's new location. Therefore, an external node can connect to a closer service without changing its destination IP address when service location optimization is required according to the user mobility. Using a testbed implementation with additional functions on top of the Kubernetes system, we verified the process of requesting a service ID from the LISP control plane and updating the mapping information in the xTRs. All proposed systems were evaluated in the testbed, and the evaluation results demonstrated that the proposed system can provide external access using the same ID access to equivalent services running on different Kubernetes clusters with little impact in terms of processing delay. Moreover, the analysis results for the path recovery time indicated that it is possible to change the path in less time while providing network availability that cannot be provided by existing distributed Kubernetes environments. From the results, it is verified that our proposed system has a significant benefit in the distributed edge mobile network by minimizing service interruptions caused by the mobility of users between distributed clouds while guarantying low-latency services for distributed edge clouds. In conclusion, considering the service of edge clouds as another type of end-user terminal by assigning an identifier from the common pool of LISP identifiers could enhance the service quality of mobile edge computing.

With our approach to manage equivalent services as a single identifier regardless of their multiple locations, it is expected to adopt an enhanced routing mechanism in the distributed cloud environment. For example, the Compute-First Networking (CFN) concept have been introduced recently, which concerns resource-efficient routing decision for optimizing network management. There are several studies about CFN [30] and studies such as dynamic load balancing mechanism energy-efficient load balancing [31] or maximization of edge utilization with service execution cost by cooperating between decentralized agents [32]. In addition to these efforts, our approach has possibility to support optimal decisions by including resource-related metrics from each edge cloud when they update service ID-location mappings and also routing configuration by responding ID-Location mappings. One possible approach of the CFN based on ID-Location separation has been proposed as the Dynamic Anycast (Dyncast) [33], which allocates anycast IP address to equivalent services and determines routing path by sharing resource-related metrics between edge clouds when the service is requested. In our approach, with a logically centralized LISP control plane, it enables to collect resource metrics efficiently to collect these resources and configure routing paths by sending appropriate locator for requests of service ID. For this, enhancement of LISP protocol to obtain resource-related metrics and control plane operation may be one of interesting future works.

REFERENCES

[1] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proc. Workshop Mobile Edge Commun. (MECOMM SIGCOMM)*, Aug. 2018, pp. 31–36.

[2] S. Kassahun, A. Demessie, and D. Ilie, "A PMIPv6 approach to maintain network connectivity during VM live migration over the internet," in *Proc. IEEE 3rd Int. Conf. Cloud Netw. (CloudNet)*, Krakow, Poland, Oct. 2014, pp. 64–69.

[3] Q. Li, J. Huai, J. Li, T. Wo, and M. Wen, "HyperMIP: Hypervisor controlled mobile IP for virtual machine live migration across networks," in *Proc. 11th IEEE High Assurance Syst. Eng. Symp.*, Nanjing, China, Dec. 2008, pp. 80–88.

[4] J. Ha, J. Park, S. Han, and M. Kim, "Live migration of virtual machines and containers over wide area networks with distributed mobility management," in *Proc. 15th EAI Int. Conf. Mobile Ubiquitous Syst., Comput., Netw. Services*, New York, NY, USA, Nov. 2018, pp. 264–273.
[5] S. Garg, K. Kaur, S. H. Ahmed, A. Bradai, G. Kaddoum, and M. Atiquzzaman, "MobQoS: Mobility-aware and QoS-driven SDN framework for autonomous vehicles," *IEEE Wireless Commun.*, vol. 53, no. 4, pp. 12–20, Aug. 2019.
[6] P. Raad, S. Secci, D. C. Phung, A. Cianfrani, P. Gallard, and G. Pujolle, "Achieving sub-second downtimes in large-scale virtual machine migrations with LISP," *IEEE Trans. Netw. Service Manage.*, vol. 11, no. 2, pp. 133–243, Jun. 2014.
[7] A. Ksentini, T. Taleb, and F. Messaoudi, "A LISP-based implementation of follow me cloud," *IEEE Access*, vol. 2, pp. 1340–1347, 2014.
[8] S. Secci, P. Raad, and P. Gallard, "Linking virtual machine mobility to user mobility," *IEEE Trans. Netw. Service Manage.*, vol. 13, no. 4, pp. 927–940, Dec. 2016.
[9] K. Sun and Y. Kim, "Enhanced LISP mapping system for optimizing service path in edge computing environment," *IEEE Access*, vol. 8, pp. 190571–190599, 2020.
[10] K. Sun and Y. Kim, "LISP-based integrated control plane framework for service function chaining," *IEEE Access*, vol. 9, pp. 52944–52956, 2021.
[11] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis, *The Locator/ID Separation Protocol (LISP)*, document RFC 6830, Jan. 2013.
[12] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with Kubernetes," *Comput. Commun.*, vol. 159, pp. 161–174, Jun. 2020.
[13] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, Sep. 2014.
[14] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an availability manager for microservice applications," 2019, *arXiv:1901.04946*.
[15] R. Kumar and M. Trivedi, "Networking analysis and performance comparison of Kubernetes CNI plugins," in *Advances in Computer, Communication and Computational Sciences*. Singapore: Springer, Oct. 2020, pp. 99–109.
[16] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Honolulu, HI, USA, Apr. 2018, pp. 189–197.
[17] A. Amirante and S. P. Romano, "Container NATs and session-oriented standards: Friends or foe?" *IEEE Internet Comput.*, vol. 23, no. 6, pp. 28–37, Nov. 2019.
[18] Y. Aldwyan and R. O. Sinnott, "Latency-aware failover strategies for containerized web applications in distributed clouds," *Future Gener. Comput. Syst.*, vol. 101, pp. 1081–1095, Dec. 2019.
[19] D. Tomic, D. Zagar, and G. Martinovic, "Implementation and efficiency analysis of composite DNS-metric for dynamic server selection," *Telecommun. Syst.*, vol. 71, no. 1, pp. 1–18, Oct. 2018.
[20] *Kubernetes Components*. Accessed: Feb. 2022. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/
[21] P. Kayal, "Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope: Invited paper," in *Proc. IEEE 6th World Forum Internet Things (WF-IoT)*, New Orleans, LA, USA, Jun. 2020, pp. 1–6.
[22] *Flannel*. Accessed: Aug. 2019. [Online]. Available: https://github.com/coreos/flannel
[23] *Tungsten Fabric Architecture*. Accessed: Feb. 2022. [Online]. Available: https://tungstenfabric.github.io/website/Tungsten-Fabric-Architecture.html
[24] G. Moura, J. Heidemann, R. Schmidt, and W. Hardaker, "Cache me if you can: Effects of DNS time-to-live," in *Proc. Internet Meas. Conf.*, Amsterdam, The Netherlands, 2019, pp. 101–115.
[25] J. Padilla, "OpenOverlayRouter with continers," Bachelor's Thesis, Universitat Politècnica de Catalunya, Barcelona, Spain, 2018.
[26] *Lispers.net*. [Online]. Available: https://github.com/farinacci/lispers.net
[27] *iPerf—The Ultimate Speed Test Tool for TCP, UDP and SCTP*. Accessed: Feb. 2022. [Online]. Available: https://iperf.fr
[28] *Network-MultiTool*. Accessed: Feb. 2022. [Online]. Available: https://github.com/Praqma/Network-MultiTool
[29] N. Bahaman, A. S. Prabuwono, R. Alsaqour, and M. Z. Mas'ud, "Network performance evaluation of tunneling mechanism," *J. Appl. Sci.*, vol. 12, no. 5, pp. 459–465, Feb. 2012.
[30] L. Tian, M. Yang, and S. Wang, "An overview of compute first networking," *Int. J. Web Grid Services*, vol. 17, no. 2, pp. 81–97, Mar. 2021.
[31] R. A. Hasan, M. N. Mohammed, M. A. B. Ameedeen, and E. T. Khalaf, "Dynamic load balancing model based on server status (DLBS) for green computing," *Adv. Sci. Lett.*, vol. 24, no. 10, pp. 7777–7782, Oct. 2018.
[32] Z. Nezami, K. Zamanifar, K. Djemame, and E. Pournaras, "Decentralized edge-to-cloud load balancing: Service placement for the Internet of Things," *IEEE Access*, vol. 9, pp. 64983–65000, 2021.
[33] Y. Li, L. Iannone, D. Trossen, P. Liu, and C. Li, *Dynamic-Anycast Architecture*, Standard draft-li-dyncast-architecture-01, IETF Internet-Draft, Jan. 2022.

KYOUNGJAE SUN received the B.S. and Ph.D. degrees in electronic engineering from Soongsil University. His current research interests include mobility management and edge computing.

JANGWON LEE received the B.S. degree in electronic engineering from Soongsil University, where he is currently pursuing the M.S. degree with the Department of Electronic Engineering. His current research interests include management and orchestration for hybrid cloud architectures with Openstack, Kubernetes, and cloud networking.

YOUNGHAN KIM (Member, IEEE) received the B.S. degree from Seoul National University and the M.Sc. and Ph.D. degrees in electrical engineering from KAIST. He is currently a Full Professor with the Department of Electronic Engineering, Soongsil University. He is also the President of the Korea Information and Communications Society. His current research interests include cloud computing, 5G networking, and next-generation networks.