Framework of Network Service Orchestrator for Responsive Service Lifecycle Management

Keisuke Kuroki, Masaki Fukushima and Michiaki Hayashi
Integrated Core Network Control and Management Laboratory
KDDI R&D Laboratories, Inc.
2-1-15 Ohara, Fujimino-shi, Saitama, JAPAN
E-mail: {ke-kuroki, fukusima, mc-hayashi}@kddilabs.jp

***Abstract*—NFV is expected to reduce costs and provide an agile new service. To achieve OPEX reduction and providing agility of service lifecycle management, we need to improve the responsiveness of service lifecycle management by reducing the turnaround time of creating, repairing and deleting a network service. Although orchestrator is defined in NFV architecture, the architecture is relatively new and a reference model of NFV orchestrator is partially immature. In this paper, we propose a framework and the implementation of network service orchestrator for responsive service lifecycle management. In addition, we show that our orchestrator can reduce the time of creation, repair and deletion of a virtualized EPC network service as an example by experiments.**

Keywords—Orchestrator; Network Function Virtualization; Network-designing.

I. INTRODUCTION

Network Function Virtualization (NFV) [1] has emerged as a cutting-edge industry trend led by many telecommunication operators and vendors. NFV is expected to reduce the CAPEX of operators by fostering virtualization of various network functions running on Commercial-Off-The-Shelf (COTS) hardware. Furthermore, softwarization of the telecommunication infrastructure is bringing new opportunities for operators to automate many aspects of their operations through the Application Programming Interface (API), which was not possible with the conventional physical infrastructure. Such automation opens up the possibility of reducing OPEX by eliminating not only time-consuming but also error-prone service provisioning processes performed by human operators. In addition, automation enables operators to achieve value-added agile service provisioning in an incredibly short time to market.

From the viewpoint of OPEX reduction and service agility, a key performance indicator of an NFV system is responsiveness of service lifecycle management (i.e., how long does it take to create/repair/delete a service on the NFV infrastructure?). Service lifecycle management is a fundamental functionality provided by the NFV architecture and, amongst its Management and Orchestration (MANO) Framework [2], orchestrator is the primary component that plays the central role in the service lifecycle management. The orchestrator receives a service creation request from its operator or Operation Support System (OSS), finds available resources, designs the optimal virtual network satisfying the request, and deploys/configures the virtual network on the actual infrastructure. Upon reception of failure alerts, the orchestrator automatically takes countermeasures according to the preconfigured repair policy. When the service is discontinued, the orchestrator promptly releases the resources occupied by the service in order to make them available to upcoming new service creation requests.

Despite the significance of the orchestrator in the NFV architecture, there have been few attempts to answer the research question: to what extent can an NFV orchestrator reduce the time of service lifecycle management? There have been various studies on orchestrator. One line of inquiry examines Software Defined Networking (SDN) controllers that orchestrate multi-domain network resources. These studies consider orchestration of only network resources and do not consider orchestration of Virtual Network Functions (VNFs), which is a fundamental point of difference between SDN and NFV. There are a few studies on NFV orchestrator. However, since the NFV architecture itself is relatively new and partially immature, these studies mainly focus on architectural aspects of NFV orchestrator and do not provide results of quantitative performance evaluation.

In order to answer the aforementioned question, we propose a design and implementation of NFV orchestrator and show the evaluation results of the time of service lifecycle management by experiments. In addition to the standard service description including VNFs and virtual links, our orchestrator can take an optimization policy as part of a service creation request. Our optimization engine is generic in order for its users to define their own optimization algorithm. We describe the interface between the orchestrator and the optimization engine. Our evaluation results show that we can reduce the time of service lifecycle management with our orchestrator design.

The organization of this paper is as follows: In Section-II, we review related works. In Section-III, we explain the architecture of our orchestrator to reduce the time of service lifecycle management. In Section-IV, we evaluate the time of creation, repair and deletion time (i.e., responsiveness of service lifecycle management) and clarify what issues remain. Finally, concluding remarks are given in Section V.

II. Related Work

According to NFV reference architecture, an orchestrator plays the role of managing NFV Infrastructure (NFVI) and VNF resources, and activating network services. To achieve responsive service lifecycle management, it is more desirable for orchestrator to execute automatic network-designing (i.e., calculation of the optimal location, path and so forth) and be able to create a whole network (i.e., deployment of VNF and networking) without manual operation. Furthermore, evaluation of the architecture of the orchestrator is needed.

Qian et al. [3] achieved end-to-end orchestration for cloud and Wide Area Network (WAN). In their architecture, SDN controllers, Data Center (DC) controller and WAN controller, are used. They created three virtual networks across two DCs using the two controllers. However, if we want to create a network services with NFV technology, we have to design not only networking but also optimal distribution of VM and virtualized network functions.

Shen et al. [4] proposed an NFV management system called vConductor, which conforms to ETSI NFV end-to-end architecture, particularly its MANO specification, and has an information model based on the TeleManagement Forum (TMF) Shared Information/Data (SID) model. Although VM and VNF can be managed by vConductor, service optimization and performance-management are not demonstrated.

Hershey et al. [5] proposed a new architecture for automated mission service orchestration and distribution. The architecture has three key components: application, algorithm and data filter orchestration. With utilization of the three orchestrations that play different roles respectively, a new algorithm is introduced for dynamic rules learning that correlates historical events in order to update dataflow and orchestrate mission service. However, the optimal distribution of service components and the performance evaluation are not demonstrated.

Thus, a proposal of the architecture for orchestrator that can create the whole network supported by performance evaluation results in an orchestrator is required. In this paper, we show the evaluation results and make an index of an orchestrator.

III. Architechture

In this section, we propose the architecture of orchestrator, which can manage the lifecycle (creation/repair/deletion) of services in a responsive and optimal manner. First, we explain the overall architecture of our proposed orchestrator in Section III-A. Then, we describe how our orchestrator automates the network designing phase of lifecycle management through an example of service creation in Section III-B. Finally, we explain how our orchestrator automates the operation phase of the service lifecycle management in Section III-C.

A. Overall Architecture of Orchestrator

We design the architecture of orchestrator as shown in Fig. 1. Our orchestrator consists of the following five components:

- Graphical User Interface (GUI)
- Information database
- Network-designing engine
- Controller driver
- Operation automation engine

The GUI enables the creation of a network for users that are not network specialists, and the users can draw a logical network through the GUI instinctively. The information database includes resource statuses (i.e., CPU utilization of server, amount of traffic of link, and so forth), and the database is used for calculation of the optimal network structure. In addition, our orchestrator can create multiple network services and each configuration of users is kept in the information database. Thus, our orchestrator knows each configuration of users and can also delete network services easily. The network-designing engine has algorithms for calculating the optimal network-structure (i.e., optimal location, optimal path and so forth). By this, the designing of a network is automated and our orchestrator can consider each optimal network for users. In other words, our orchestrator determines the optimal network structure before the creating network by calculating the optimal resource distribution.

After determining the optimal network structure, our orchestrator instructs some controllers to create the network through the controller driver. Our orchestrator can instruct three controllers: OpenStack [6], OpenDaylight [7] and Evolved packet core (EPC) controller. OpenStack is open source software and an infrastructure as a service (IaaS) framework. This has become the de-facto standard for IaaS cloud platforms and is utilized for education, experiment and so forth [8, 9]. We use OpenStack to deploy VMs and utilize Neutron, which is also an OpenStack element for creating paths between VMs. OpenDaylight is also open source software and a platform for network programmability to enable SDN. We create paths with Open vSwitch (OVS) [10], and we use OpenDaylight as the controller of OVS. The EPC controller is developed by KDDI Labs. In this paper, we handle three nodes of EPC as a Virtualized Network Function (VNF) (i.e., Mobility Management Entity (MME), Serving GateWay (S-GW) and

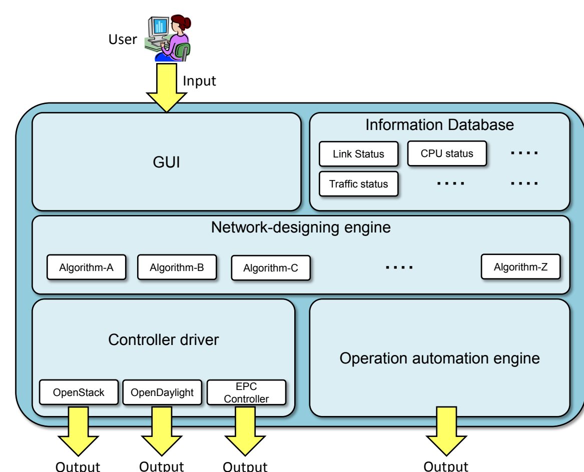


Figure 1. Architecture of orchestrator.

Packet data network GateWay (P-GW)). The EPC controller can redirect signaling for executing auto-scaling. For example, if MME has signaling congestion from base stations and a monitoring system detects the congestion, the monitoring system gives notice of the congestion and related information (i.e., the number of session) to the EPC controller. Then, the EPC controller calculates the number of CPU cores that is needed to deal with the sessions properly, and then the EPC controller allocates CPU cores as additional resources to the new virtualized MME that is created beforehand. Finally, the EPC controller sets configuration to the base stations to access the new virtualized MME preferentially and instructs signaling redirection to MME with an overload start message. Thus, the EPC controller handles signaling reduction.

Our orchestrator can handle three controllers to create a network. However, it is insufficient for automatic creation of the whole network to handle the three controllers. For example, OpenStack can only create paths between VMs and that cannot create paths to a gateway router to access the Internet. Therefore, our orchestrator has an operation automation engine for open source software.

B. *Automation of Design Phase*

In order to obtain responsiveness in service lifecycle management, our orchestrator automates the network designing phase of service lifecycle management. In general, an orchestrator should take two different aspects into account when designing a virtual network for a service. One is *what* VNFs and *what* amounts of resources are required to fulfill the service request, which is usually specified by its user as a *service description* and has been studied in the literature. The other is *how* to and *where* to allocate the required VNFs and resources in the network, which is also inevitable for operators to achieve optimal resource usage. We postulate that the latter is also crucial in orchestrator design and propose architecture that can flexibly adopt various optimization policies.

Fig. 2 shows the proposed data model of a service-creation request. The request consists of a service description and an optimization policy. In the service description, the network topology is described in the network component object with related parameters. Instead of specifying a location, the user selects the optimization policy implemented beforehand in the network-designing engine or the user can define another policy. The user can make the service creation request through the orchestrator's GUI. The network-designing engine searches available resources and calculates the optimal network structure automatically based on the service creation request. For example, if the user selects delay policy to minimize in a certain section delay, the network-designing engine automatically searches servers that have sufficient resource, links that have spare bandwidth and server location to satisfy the user policy. In this way, we achieve an automated designing network.

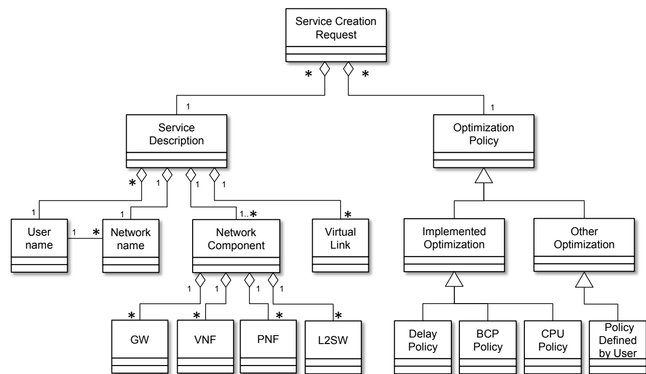


Figure 2. Modeling of service-creation request

In NFV reference architecture, there is interface named Os-Ma between the Operations Support System (OSS) and the orchestrator. We consider that the proposed modeling of service-creation request exchanged can be utilized for the Os-Ma interface.

C. *Automation of Operation Phase*

In order to manage the service lifecycle across the whole network consistently, automation of the operation phase is also required in addition to automation of the design phase discussed in Section III-B. Our orchestrator has a high-level operation engine as well as low-level driver components for each domain of virtualized infrastructure. By leveraging the existing open source software such as OpenStack, we focus on the high-level operation engine. Some of the low-level drivers, however, are also newly designed because they are missing from the current open source software.

We have to solve two problems to automate the operation of a virtualized network. One is to provide a method of avoiding interference between network services because multiple users are supposed to create multiple network services with one orchestrator. Another is to provide connectivity with existing facilities.

To avoid interference between network services, our orchestrator provides isolated virtual-routers and virtual-links. In Fig. 3, there are two tenants (i.e., network services) on

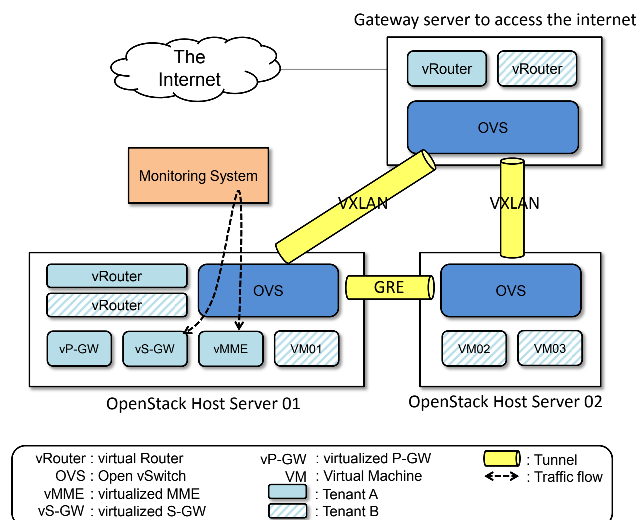


Figure 3. Example of network structure

shared network infrastructure. Our orchestrator provides virtual-routers by using Linux *namespace* mechanism, which can create separate routing tables and *iptables* (i.e., network policy) for every tenant within the same Linux box. Thus, we can allow overlapping IP addresses between network services by separating network domains. In order to provide virtual-routers in OpenStack host servers, we develop a network domain driver for OpenStack Havana, particularly its networking component called Neutron, which has a user-separation mechanism based on Linux *namespace*. With respect to host servers that are not under the control of OpenStack, we develop a driver that set up a virtual-router for every network service by directly using Linux *namespace* to avoid interference between tenants.

Besides, our orchestrator provides virtual links by creating tunneling paths between host servers. In order to create virtual links between VMs deployed on OpenStack host server (e.g., tenant V in Fig. 3), we use our OpenStack driver to make Neutron create Generic Routing Encapsulation (GRE) [11, 12] tunnel paths and allocate an identifier automatically. In order to create a tunnel path whose endpoint is not under the control of OpenStack, we develop a driver to directly control OVS to create Virtual eXtensive Local Area Network (VXLAN) [13] tunnel path allocate a VXLAN Network Identifier (VNI). As described above, our orchestrator prevents interference between network services by providing virtual-routers and virtual-links automatically.

To provide connectivity of existing facilities, our orchestrator uses OpenFlow 1.0 [14, 15] for routing between OVSs. The EPC controller can redirect signaling based on information of the session counts. To count the number of sessions, packets between vMME and vS-GW have to be transferred via a monitoring system, even if those VNFs are deployed on the same physical server. To perform packet transfer via a way point, we use OpenDaylight running as an OpenFlow controller.

Thus, our orchestrator has the above functions in an operation automation engine. We achieve automatic operation by contriving to create a network service consistently with automatic network-designing and automatic operation.

IV. Evaluation

We implement and evaluate the orchestrator proposed in Section III. In Section IV-A, we describe our implementation and evaluation results regarding to what extent we can reduce the time of service lifecycle management. In Section IV-B, we analyze a breakdown of the time and clarify what issues remain.

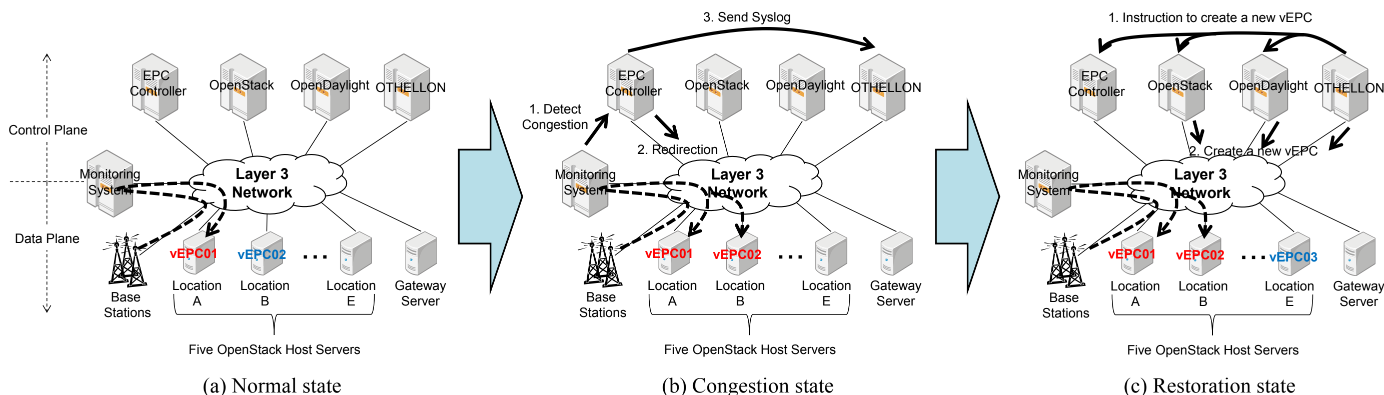
A. Performance Evaluation

Table I shows the parameters we use system in evaluation.

TABLE I. System Parameters in Evaluation

Node	Software	OS
Orchestrator	-	Ubuntu 12.04
VNFM	EPC controller	CentOS 6.4
VNF	OpenEPC	CentOS 6.4
VIM	OpenStack (Havana)	Ubuntu 12.04
VIM	OpenDaylight (Hydrogen)	Ubuntu 12.04
NFVI (network)	Open vSwitch (1.10.2)	Ubuntu 12.04
NFVI (compute)	KVM	Ubuntu 12.04

We evaluate the EPC network service as an example, and we handle vMME, vS-GW and vP-GW as VNF. Fig. 4 shows a repairing scenario for signaling congestion. The vEPC in the figure means a set of vMME, vS-GW and vP-GW. In the normal state shown in Fig. 4-(a), we have two sets of vEPC, and vEPC01 runs as an Active and vEPC02 runs as a Spare. The monitoring system monitors traffic between a base station and vMME and counts the number of sessions. In the monitoring system, a threshold for the number of sessions is set and sends a message to the EPC controller as shown in Fig. 4-(b), if a monitoring system detects signaling congestion. When the EPC controller receives a message relating congestion from the monitoring system, the EPC controller calculates the necessary number of CPU cores to process the signaling properly and allocates the calculated CPU core to Spare-vEPC dynamically as a new and additional Active-vEPC. Then, the EPC controller redirects the signaling with an overload start message and the signaling congestion is solved. Thus, though the signaling congestion is improved by the redirect-function of the EPC controller, we cannot address second congestion due to exhausting the spare system in this situation. Hence, the EPC controller sends a syslog related addressing congestion to our orchestrator after redirection processing. After receiving the syslog from the EPC controller, our orchestrator creates a new spare system to address second congestion with instructing controllers (i.e., OpenStack, OpenDaylight and EPC controller) as shown in Fig.



(a) Normal state (b) Congestion state (c) Restoration state

Figure 4. Operational scenario of repairing EPC network

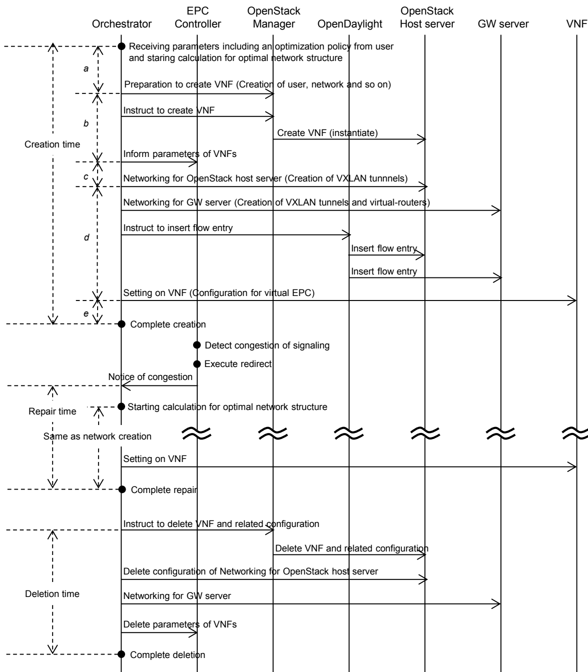


Figure 5. Operational sequence for creation, repair and deletion

3-(c). This repairing scenario can be repeated until computing and networking resources are exhausted.

Fig. 5 shows the operational sequence for creation, repair and deletion of vEPC networks in the case of using our orchestrator. At the beginning, a user defines some parameters of vEPC networks (e.g., IP address, VM memory volume, the prefigured bandwidth and so forth) and selects an algorithm that suits the user through the GUI. Our orchestrator begins to create new vEPC networks consequently. Network creation is separated into the following five steps:

a) Seaching resouce for VNF and Link

b) Deployment of VNF

c) Updating database of the EPC controller

d) Networking

e) Setting on VNF

In this evaluation, we create two sets of vEPC network (i.e., Active and Spare) in the beginning. In other words, we create six VNFs (i.e., two vMMEs, two vS-GWs and two vP-GW). In step-*a*, our orchestrator searches computing and networking resources to deploy six VNFs and the network-designing engine calculates the optimal network structure based on the optimization policy selected by user. In step-*b*, our orchestrator instructs OpenStack to prepare for creation of VNF (e.g., instruction of creation of new project and IP address subnets and so forth) and then instructs the creation of VNF. This instruction of VNF creation is repeated several times that is equal to the number of VNF. OpenStack deploys VNFs on OpenStack host servers. Consequently, VNFs begin to boot. In step-*c*, our orchestrator updates the database of the EPC controller to inform parameters related to new VNFs. By this, the EPC controller can recognize Active-vEPC and Spare-vEPC and become able to execute redirection, when the EPC controller detects signaling congestion. In step-*d*, our orchestrator creates VXLAN tunnels between OpenStack host servers and the GW server and instructs OpenDaylight to insert flow entries. Finally, our orchestrator sets the configuration on VNF in step-*e*. In this step, the primary work of our orchestrator is to revise the application file related EPC. Thus, our orchestrator creates two sets of vEPC network with 5 steps.

If the EPC controller executes redirection of signaling to the monitoring system's detecting signaling congestion and sends syslog related the congestion to our orchestrator, our orchestrator begins to create a new vEPC network as a new Spare. In the repair phase, the operation sequence to create a new vEPC network is the same as the first creation. However, the number of created VNF is three (i.e., vMME, vS-GW and vP-GW). Thus, repair is executed automatically.

If a user wants to delete EPC networks, the user can instruct the deletion of the EPC network through the GUI. In the deletion, our orchestrator instructs OpenStack to delete VNFs and the configuration related VNFs (i.e., deletion of VNF, project and network subnets and so forth). Consequently, the deletion of configuration such as VXLAN tunnel and information kept in the database of the EPC controller is executed. Thus, the user can easily delete their own network.

We measured three kinds of time: creation time, repair time and deletion time. Creation time is defined as the duration from orchestrator's starting calculation for optimal network structure to finishing setting on VNF. Repair time is defined as the duration from the receipt of syslog related congestion by the EPC controller to finishing setting on VNFs. Deletion time is defined as the duration from orchestrator's sending the instruction to delete VNF to the deletion of information kept in the database of the EPC controller. We measured the creation time, repair time and deletion time 10 times respectively, and the average times were 438, 391 and 138 milliseconds respectively. These results indicate that we can reduce workload related network creation including network-designing and operation, which result in reducing the time of service lifecycle management compared to the conventional solutions where human operator designs a network and deploy it with OpenStack manually.

B. Discussion

It is important to shorten the turn-around time in the lifecycle management of a service. Fig. 6 shows a sequence diagram of creating a network service. Our orchestrator has the advantage of built-in network-designing engine and operation automation engine. By extending scope of work in an orchestrator over network-designing and automatic creation of whole network, we can create a network service within eight minutes. That indicates the possibility that we can achieve responsive service lifecycle management by using our orchestrator. In order to further improve the responsiveness of our orchestrator, we pose the following two research issues.

Firstly, we need to mitigate the time waiting for VM boot up. Fig. 7 shows a breakdown of time for new network creation in the case of first creation and repair creation. Amongst each steps in the creation process, step *e* is dominant, which the time that the orchestrator is waiting for VM to start up. To reduce

the time to create and repair a network, we can boot up VMs in advance of service creation and pool them for future request. There is a trade-off between resource consumption by pre-booted VMs and the agility attained by VM pooling. We can further overcome this trade-off by adopting some container technology such as Docker [16] in order to decouple VM images and VNF application software into different components.

Secondly, we need to expand the applicability of automation to more workflow patterns. To this end, we need to address automatic creation of workflow. Currently, our orchestrator can execute only pre-defined repair patterns. In the case of a failure rarely experienced, we need to address the failure manually. However, orchestrator would be able to address the failure upon subsequent occurrences by automatically learning the workflow pattern.

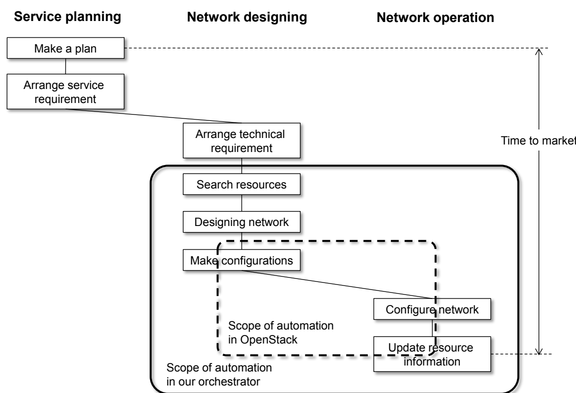


Figure 6. Scope of our orchestrator in creating a network service

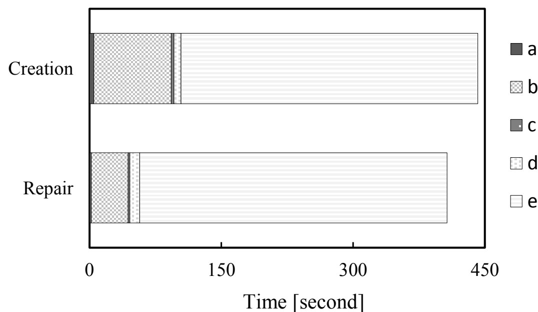


Figure 7. Breakdown of time for new network creation

V. CONCLUSION

In this paper, we present the architecture of our orchestrator to achieve responsive service lifecycle management. Though OpenStack can allocate resources and deploy virtual machine automatically, the allocation and deployment are not always optimal. If we want to create optimal network without an orchestrator, we have to design network manually, which takes time. Our orchestrator can design a network service by searching resources and automatically perform basic service lifecycle management (i.e., creating/repairing/deleting network services). According to our experiments, we can execute the service lifecycle management in less than ten minutes, which is smaller than conventional management by at least two or three orders of magnitude. We observe that a key to success of responsive lifecycle management is to fully exploit the functionality provided by highly softwarized infrastructure of today. To this end, we designed and implemented a full-fledged orchestrator by integrating existing open source controllers such as OpenStack in a consistent and unified orchestrator architecture. In future work, we will integrate performance monitoring components into our orchestrator in order to react to dynamic change in network traffic.

ACKNOWLEDGMENT

We are grateful to Takeshi Usui for giving advice about the EPC controller and support for our experiments. We are grateful to Yasunori Maruyama also for productive discussions, support for our experiments and programming assistance.

REFERENCES

[1] ETSI – NFV, http://www.etsi.org/technologies-clusters/technologies/nfv [retrieved: September, 2014]

[2] ETSI GS NFV 002 v1.1.1 “Network Functions Virtualisation (NFV); Architecture Framework,” October 2013.

[3] H. Qian, X. Huang and C. Chen, “SWAN: End-to-End Orchestration for Cloud Network and WAN,” Cloud Networking (CloudNet), 2013 IEEE 2nd International Conference on, November 2013, pp. 236-242.

[4] W. Shen et al., “vConductor: An NFV Management Solution for Realizing End-to-End Virtual Network Services,” Asia-Pacific Network Operations and Management Symposium (APNOMS), September 2014.

[5] P. Hershey, D. Clark and D. Wisniewski, “System Architecture for Dynamic Mission Service Orchestrator (DSMO),” Systems Coference (SysCon), 2014 8th Annual IEEE, March 2014, pp. 259-265.

[6] OpenStack Open Source Cloud Computing Software, http://www.openstack.org [retrieved: September, 2014]

[7] OpenDaylight A Linux Foundation Collaborative Project, http://www.opendaylight.org [retrieved: September, 2014]

[8] G. von Laszewski, J. Diaz, F. Wang and G.C. Fox, “Comparison of Multiple Cloud Frameworks,” Cloud Computing (CLOUD), 2012 IEEE 5th International Confernce on, June 2012, pp. 734-741.

[9] S. Bonner et al., “Using OpenStack To Improve Student Experience in as H.E. Enviroment,” Science and Information Conference (SAI), October 2013, pp. 888-893.

[10] Open vSwitch, http://openvswitch.org [retrieved: September, 2014]

[11] “Generic Routing Encapsulation (GRE),” IETF RFC 2784, March 2000.

[12] “Key and Sequence Number Extensions to GRE,” IETF RFC 2890, September 2000.

[13] “Virtual eXtensive Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks,” IETF RFC 7348, August 2014.

[14] N. McKeown et al., “OpenFlow: enabling innovation in campus networks,” ACM SIGCOMM Computer Communication Review, Vol. 38, issue 2, April 2008, pp. 69-74.

[15] “OpenFlow Switch Specification Version 1.0.0,” Open Networking Foundation, December 2009.

[16] Docker, http://www.docker.com [retrieved: September, 2014].