The Journal of Systems and Software 152 (2019) 108–119

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

ElasticSFC: Auto-scaling techniques for elastic service function chaining in network functions virtualization-based clouds

Adel Nadjaran Toosi[a,*], Jungmin Son[c], Qinghua Chi[b], Rajkumar Buyya[c]

[a] *Faculty of Information Technology, Monash University, Clayton, Australia*
[b] *Wireless Network Research Department, Shanghai Huawei Technologies Co., Ltd., China*
[c] *Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia*

ARTICLE INFO

Article history:
Received 22 October 2018
Revised 20 January 2019
Accepted 25 February 2019
Available online 26 February 2019

Keywords:
Service function chaining
Network functions virtualization
Virtualized network functions
Software defined networking
Auto-scaling
Cloud computing

ABSTRACT

It is anticipated that future networks support network functions, such as firewalls, load balancers and intrusion prevention systems in a fully automated, flexible, and efficient manner. In cloud computing environments, network functions virtualization (NFV) aims to reduce cost and simplify operations of such network services through the virtualization technologies. To enforce network policies in NFV-based cloud environments, network services are composed of virtualized network functions (VNFs) that are chained together as service function chains (SFCs). All network traffic matching a policy must traverse network functions in the chain in a sequence to comply with it. While SFC has drawn considerable attention, relatively little has been given to dynamic auto-scaling of VNF resources in the service chain. Moreover, most of the existing approaches focus only on allocating computing and network resources to VNFs without considering the quality of service requirements of the service chain such as end-to-end latency. Therefore, in this paper, we define a unified framework for building elastic service chains. We propose a dynamic auto-scaling algorithm called ElasticSFC to minimize the cost while meeting the end-to-end latency of the service chain. The experimental results show that our proposed algorithm can reduce the cost of SFC deployment and SLA violation significantly.

1. Introduction

Network Functions Virtualization (NFV) is an emerging trend in networking that concerns with migration of network functions (NFs) such as network address translation (NAT), firewalls, intrusion detection systems (IDS) into virtualized environments to reduce capital expenditure, simplify operations, and speed up service deployment. Traditionally, NFs are embedded on dedicated hardware devices called middleboxes or network appliances. Even though, such middleboxes are designed to handle heavy loads efficiently, they require high up-front investment and do not achieve the elasticity feature of virtualized environments (Pham et al., 2017). Hence, there is a significant tendency among cloud service providers and operators to decouple NFs from their underlying hardware and run them on commodity servers (e.g., x86 servers) (Eramo et al., 2017). This tendency has given birth to NFV technology that converts NFs into virtualized network functions (VNFs) hosted in virtual machines or containers.

Network policies often require those VNFs to be stitched together as Service Function Chains (SFC) to deliver value-added services or certain network functionality (Pham et al., 2017). To supply specific requirements, SFC defines a sequence of service functions (SF) through which traffic (stream of packets) must be steered. Note that here SF, NF and VNF are used synonymously. Recently, the new software defined networking (SDN) technology which decouples the data forwarding and network control planes and enables the network to become centrally manageable is effectively exploited in policy enforcement and appropriate SFC forwarding (Mechtri et al., 2017).

In today's networks, many applications produce a large volume of traffic that is required to be processed by SFC. Cloud service providers' goal is to utilize network and host resources optimally to operate these service chains and provide their associated Quality of Service (QoS) requirements such as end-to-end latency or throughput, while failure to do so results in violation of the service level agreement (SLA) (Ghaznavi et al., 2015). The automated and efficient NFV management and orchestration is one of the key solutions to achieve this goal.

Cloud service providers have access to many prominent mechanisms and techniques to develop efficient NFV management and

[*] Corresponding author.
E-mail address: adel.n.toosi@monash.edu (A. Nadjaran Toosi).

https://doi.org/10.1016/j.jss.2019.02.052
0164-1212/© 2019 Elsevier Inc. All rights reserved.

orchestration systems that minimizes their operational cost while meeting SLA requirements. The elasticity feature of VNFs allows for both *horizontal* and *vertical* scaling of resources in response to variation in service requests and workload. In horizontal scaling, VNF instances can be added or removed, whereas in vertical scaling, host and bandwidth can be allocated or released according to on-demand requests. To the best of our knowledge, this paper is one of the first attempts to use vertical and horizontal auto-scaling of VNFs at the same time to build elastic service chains. The live migration of VNFs is another option that would enable consolidation and replacement of the VNFs to minimize cost and improve performance. For example, a VNF can be migrated to a new host closer to the traffic source to reduce network delay or a host with more residual network bandwidth or computing resources suitable for vertical auto-scaling. Further, dynamic flow scheduling and traffic engineering provides opportunities for flow scheduling to redirects SFC traffic to other paths to gain more network bandwidth or avoid congestion.

With this in mind, in this paper, we intend to minimize the operational cost of the cloud computing service provider while the end-to-end delay requirements of the service function chains are satisfied using dynamic auto-scaling of resources in the network chain. The majority of works in this area focused on the efficient placement of VNFs to reduce operational cost and improve the performance of the chain, for example (Yoshida et al., 2014; Moens and Turck, 2014; Wang et al., 2016; Gember et al., 2013; Clayman et al., 2014). However, they mostly ignore to how provide a strictly guaranteed SLA to satisfy QoS requirements of the users which is very critical to the cloud service providers. There are a few studies, partially similar to our work, that address the auto-scaling and placement of VNFs with respect to end-to-end latency (Li et al., 2016; Cziva et al., 2018).

Moreover, to best of our knowledge none of the existing works proposes a unified method that builds elastic service function chains with simultaneously considering all of auto-scaling (both horizontal and vertical), placement, migration and network traffic engineering together. To overcome the challenges of building such method, we propose an elastic SFC management and orchestration framework and make the following **key contributions**:

1. Definition of an extended architectural framework and principles for building elastic service chains in NFV environments,
2. Proposed ElasticSFC, a novel heuristic algorithm for end-to-end latency-aware dynamic auto-scaling of service function chains using horizontal and vertical scaling of VNFs and dynamic bandwidth allocation. In dynamic bandwidth allocation, we use dynamic flow scheduling and VNF migration to enable efficient utilization of network resources,
3. Evaluation of the ElasticSFC using realistic network policies and workload traces of a web application in our extended CloudSimSDN simulator to support SFC and NFV.

The remainder of this paper is organized as follows: Section 2 provides a brief background on network polices, Service Function Chaining (SFC), and Network Functions Virtualization (NFV). In Section 3, we describe the extended architecture for SFC and fundamental principles to build elastic service chains. We discuss our choices for building elastic service chains to meet QoS requirements of users including horizontal and vertical scaling of service functions, dynamic traffic steering in the network, and virtualized network function migration in Section 4. Section 5 describes our proposed algorithms for dynamic auto-scaling of network service functions. We present performance evaluation and experimental results in Section 6. Section 7 discusses related work and finally we conclude our paper in Section 8.

2. Background

Applications in distributed systems have complex communications patterns sometimes requiring enforcement of network policies to comply with performance and security requirements (Cui et al., 2017). Network policies mandate traffic to traverse a series of network functions (NFs) such as firewalls, proxies, traffic shapers, load balancers, intrusion detection and prevention systems (IDS or IPS). A network policy (shorted as policy here) can be represented as a flow and a list of NFs the flow needs to traverse, for instance, *{Source IP, Source Port, Destination IP, Destination Port, Protocol}* → *{Series of NFs}*. Sample policy for a web application front-end can be configured as:

$$\{*, *, LB1, 80, TCP\} - > \{IDS1, F1\},$$

which implies that all TCP requests from any ports of all clients sent to the port 80 of the public IP of the front-end load balancer (LB1) must traverse through intrusion detection system IDS1 and then firewall F1 before reaching the destination load balancer LB1. The responsibility of an NF is to perform specific treatment of received packets (e.g., dropping all packets with specific headers in a firewall) where it can act at various layers of the protocol stack (e.g., at the network layer or other OSI layers).[1] Traditionally, NFs are typically deployed in the form of “network appliances” or “middleboxes” in which NF software is tightly coupled with proprietary hardware that needs to be manually installed and managed. However, thanks to the advances in the Network Functions Virtualization (NFV), NFs can be realized as virtual elements embedded into Virtual Machines called virtualized network functions (VNFs) that alleviate operational challenges of middleboxes and rapid deployment of NFs.

A network policy can be interpreted as a service function chain (SFC) defining an ordered list of service functions (SFs) enforcing the ordering constraints and steering network flows through them. The term “service function” is used here to denote a “network function” or “virtualized network functions” in the context of this paper.

We define $\mathbf{P} = \{p_1, p_2, \ldots\}$ as the set of SFC policies in the system. An SFC policy p_i is defined in the form of $\{flow \rightarrow sequence\}$. A $flow$ is denoted by a 5-tuple: {*src*, *dst*, *srcport*, *dstport*, *proto*}, where *src* and *dst* are IP addresses of the source and destination hosts and *srcport*, *dstport* are their associated port numbers, respectively. *proto* represents the protocol type that can be either TCP or UDP. *sequence* is a list of SFs, $\{sf_1, sf_2, \ldots, sf_n\}$, that all network flows matching the $flow$ section of policy p_i must traverse in the sequence of $sf_1 \rightarrow sf_2 \rightarrow \ldots \rightarrow sf_n$. We denote sf_1 and sf_n as ingress and egress service functions, respectively.

SFCs might be *unidirectional* or *bidirectional*. Network traffic in the unidirectional SFC needs to be be forwarded through the ordered list of SFs in one direction, i.e., $sf_1 \rightarrow sf_2 \rightarrow \ldots \rightarrow sf_n$, whereas a bidirectional SFC requires the traffic to pass through both directions ($sf_1 \rightarrow sf_2 \rightarrow \ldots \rightarrow sf_n$ and $sf_n \rightarrow \ldots \rightarrow sf_2, \rightarrow sf_1$). Without loss of generality, we assume that SFCs are unidirectional in this paper.

Based on the Internet Engineering Task Force (IETF) model, the main component of SFC architecture includes: 1) a classifier that differentiates the traffic flows against the policy and redirects them to the specific SFC by adding an SFC header, and 2) a Service Function Forwarder (SFF) that uses the information in the SFC header to forward packets received from the network to associated SFs. Traffic from SFs eventually returns to the same SFF that injects back traffic onto the network.

A wide range of research work has been conducted on the design and development of novel frameworks and techniques for

[1] https://tools.ietf.org/pdf/draft-ietf-sfc-architecture-07.pdf.

successful realization of SFC goals. Among these works, considerable amount of attention has been given to VNF routing, placement, and consolidation with the aim of decreasing operational and network communications cost. The VNF placement and consolidation problems in NFV environments are very similar to the well-studied VM placement and consolidation in cloud computing environments (Eramo et al., 2017). However, existing works in the VM management area are not suitable for SFC in their default forms as they are designed without consideration of the sequencing requirement of service chains (Ghaznavi et al., 2015).

The possibility to dynamically scale network functions in service chains at runtime in an automated fashion is an important area of the research that requires particular attention (Szabo et al., 2015). Providing elastic networking services is similar to offering flexible cloud services with the pay-as-you-go cost models. However, it is far from trivial as it requires careful consideration of the chain-ordered set of SFs and configuration of virtualized resources including virtual machines (VMs) and network resources to meet the demand on the service chain.

One of the main benefits offered by the NFV approach is the opportunity to dynamically scale VNFs and the allocated bandwidth between them to meet the performance requirements of the service chain such as throughput or end-to-end latency. As mentioned earlier, considerable attention in the literature has been given to the VNF routing, placement, and consolidation which make the building blocks of elastic service chains. However, holistic auto-scaling approaches that unify all these advancements and consider scaling for both compute and network resources to meet SLA requirement of the service chain are not sufficiently investigated in the current NFV management and orchestration frameworks. Therefore, this work aims to develop dynamic auto-scaling algorithms to support the construction of elastic service chains meeting the end-to-end delay requirement of users in NFV management and orchestration frameworks.

3. Service function chaining architecture

In this section, we describe our extended architecture for SFC and fundamental components to build elastic service chains. With elastic service function chaining, network service functions can be dynamically scaled in order to meet QoS requirements of diversified clients (users).

The proposed architecture, shown in Fig. 1, includes *service function chaining applications* that allow users to composite their network policies and chain together their required network function services via a *dashboard* or *command line (CLI)* or *application program (API)* interfaces. Central to the architecture is *Management and Orchestration (MANO)* module which is in charge of the orchestration and management of resources and realization of service chains. The architecture of MANO is aligned with ETSI NFV architectural framework[2] and has three main functional blocks: NFV Orchestrator (NFVO), VNF manager (VNFM), and Virtualised Infrastructure Manager (VIM). NFVM is in charge of NFV lifecycle management including instantiation, scaling and termination of VNF instances. It maintains the view of the entire virtualization infrastructure and keeps a record of installed VNFs and available resources in the physical infrastructure. VIM controls and manage the compute, storage and network resources within the infrastructure. NFVO is responsible for lifecycle management of network services and policy management for VNF instances.

In this paper, we propose algorithms for building elastic service chains that are dynamically scaled based on the system load and QoS requirements of users. Our proposed algorithms are plugged into a module of MANO called *elastic service controller* that is responsible for auto-scaling of service chains. To build elastic service chains, we use techniques such as flow scheduling, VNF migration, and scaling that are explained in more details in the next section. To perform its duties, MANO communicates with *SDN controller* in a tightly regulated process that ensures proper deployment and functioning of service chains. SDN controller is a logically centralized component of the system with a general view of the network and handles traffic steering according to the requirement of the service function chaining applications. SDN controller uses a protocol such as OpenFlow to set forwarding rule satisfying routing requirement of service chains for the virtual and physical switches in the infrastructure.

In the next section, we focus on the main techniques used by elastic service controller to dynamically scale service functions to meet elasticity requirement of service chains.

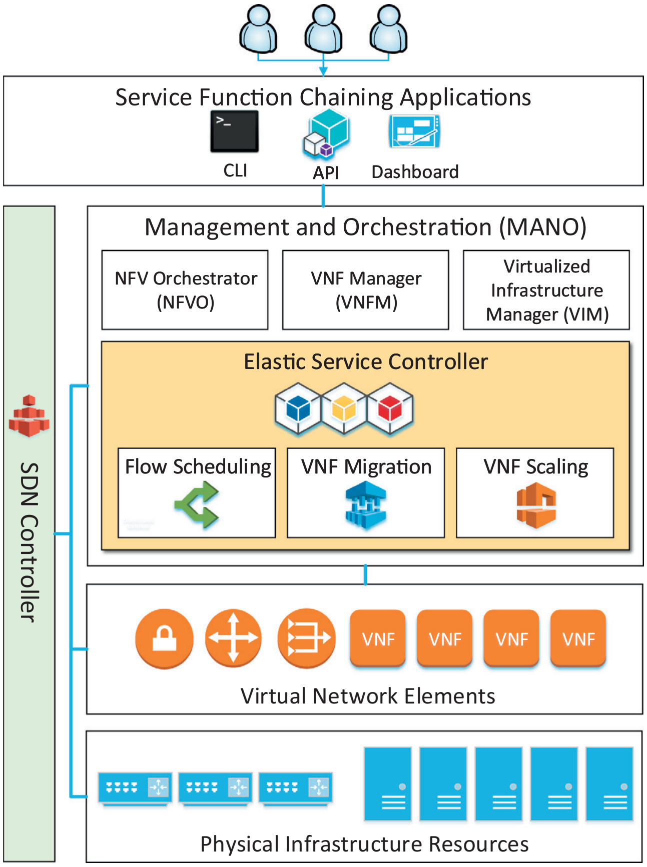
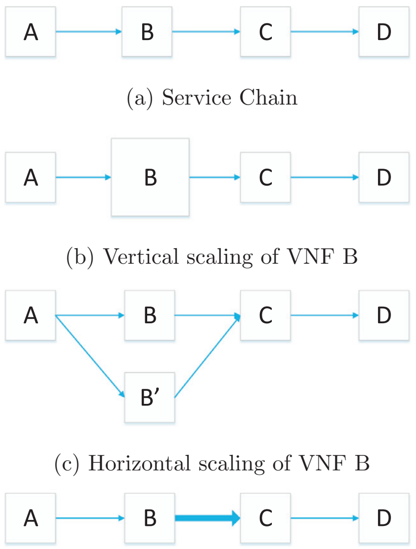

Service Function Chaining Applications
CLI
API
Dashboard
SDN Controller
Management and Orchestration (MANO)
NFV Orchestrator (NFVO)
VNF Manager (VNFM)
Virtualized Infrastructure Manager (VIM)
Elastic Service Controller
Flow Scheduling
VNF Migration
VNF Scaling
VNF
VNF
VNF
VNF
Virtual Network Elements
Physical Infrastructure Resources

Fig. 1. Service function chaining architecture.

4. Elastic service chaining approach

In this section, we explore a range of possible options including scaling up/down and scaling in/out VNFs, dynamic flow scheduling, and VNF migration to allocate required processing power and bandwidth to build elastic service chains meeting QoS requirements. Please note that the majority of software-based network appliances are CPU intensive and unlikely to be memory and disk heavy. Hence, we only focus on the processing power of VNFs and the network bandwidth between them that are essential to the performance of the service chain for the sake of simplicity. We assume that VNFs always have access to adequate memory and storage resources and demand no scaling for these resources. However, our proposed methods can be simply extended to support these

[2] ETSI GS NFV 002 (V1.1.1): “Network Functions Virtualisation (NFV); Architectural Framework”.



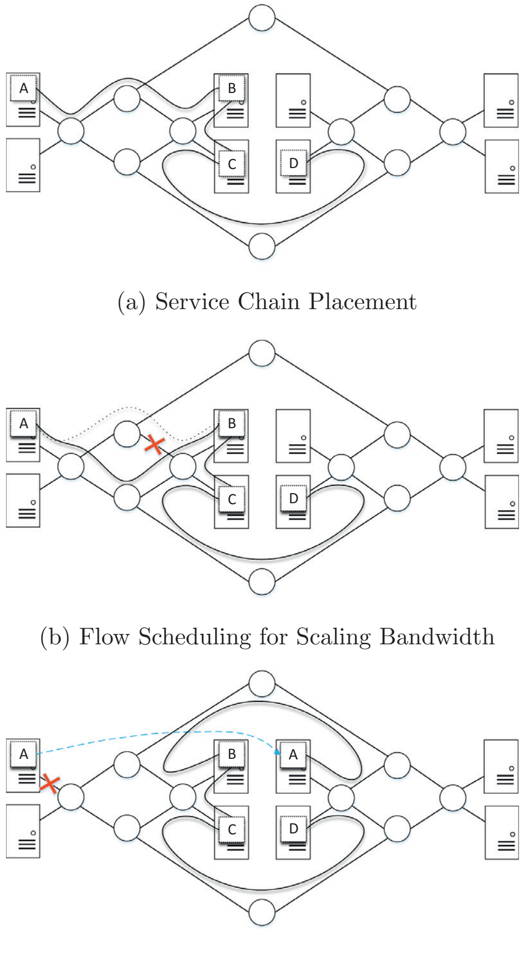
(a) Service Chain

(b) Vertical scaling of VNF B

(c) Horizontal scaling of VNF B

(d) Allocating more bandwidth to B-C connection

Fig. 2. Elasticity mechanism.



(a) Service Chain Placement

(b) Flow Scheduling for Scaling Bandwidth

(c) VNF Migration for Scaling Bandwidth

Fig. 3. Scaling bandwidth for a service chain.

resources. In the following, we discuss challenges related to dynamic scaling of service functions in the NFV framework enforcing service chaining.

Fig. 2a shows a service chain where each box represents one abstract SF. Let us assume that the size of each box shows the processing power of the hosting compute node (virtual machine), and the thickness of connector lines represents the dedicated (available) bandwidth between VNFs. For the sake of simplicity, we assume that each SF is hosted as a VNF in a single VM. Since nodes (SFs) can be part of one or many SFCs and the throughput of the VNF depends on the type (e.g., firewall or proxy) and the load of SF (incoming rate of packets) running in the VNF instance, VNFs can become overloaded and consequently degrade the entire chain performance. Horizontal and vertical scaling of the VNF instances are the promising mechanisms to build elastic SFC honoring the end-to-end latency requirement of the policies.

Fig. 2b illustrates the case that VNF instance B is vertically scaled up by the allocation of more computational resources. Similarly, when utilization of the VNF instance is considerably low, the VNF instance can be scaled down by releasing redundant resources. Vertical scaling of VNF instances to achieve processing capacity required by the traffic might be impossible due to capacity constraints or allocated resources of the physical server. As shown in Fig. 2c horizontal scaling that allows adding/removing (scaling in/out) VNF replica instances is another possible option in this scenario. However, processing (Compute) capacity of VNF instances is not the only source of variation in service function performance. The link capacity for transferring packet (available bandwidth) or even network congestion can become the bottleneck or the source of performance degradation of a service chain. Allocating more bandwidth or finding other network paths to dedicate required bandwidth is necessary under circumstances that the network is the source of the issue.

The most recent architecture for SFC has made use of software-defined networking (SDN) to assist automated deployment of service chains. SDN decouples the control plane from the forwarding plane in traditional network switches and provides a logically centralized management controller along with programming APIs for network management. The SDN controller can be used to dynamically control the SFC topology and perform traffic steering across SFs (Medhat et al., 2017). The SDN controller and NFV manager work in coordination to perform the allocation and management of resources required by elastic SFC. Fig. 3a depicts the same service chain in Fig. 2a in a physical network topology connecting eight physical servers. As it can be seen in the figure, virtual links among SF nodes are mapped into multiple physical links in the network. If a physical link does not have enough dedicated bandwidth to be allocated to the virtual link, dynamic flow scheduling gives us the option to redirect traffic to another network path capable of accommodating updated bandwidth shown in Fig. 3b. This is only the case for network topologies having multiple paths available between a pair of source and destination hosts. If all possible paths between the source and the destination with the required latency (number of hobs) are not suitable to allocate required bandwidth, VNF migration allow migration of either of the two end VNF instances of the virtual link or even both to find a proper placement providing required bandwidth. For example, in Fig. 3c, the physical link connecting the physical server to the edge switch cannot provide the extra bandwidth required by the virtual link connecting A to B. Thus, VNF A has been migrated to another host to gain the required dedicated bandwidth to satisfy performance requirements of the service chain.

5. Proposed algorithms

In this section, we propose our solution including series of algorithms to build elastic service chains in NFV environments. All

Table 1
Notations.

Symbol	Description
$\mathbb{V} = \{\mathbf{v_1}, \mathbf{v_2}, \ldots\}$	List of all VNFs
$\mathbb{V}_i$	List of VNFs of policy i
$\mathcal{V}$	Set of VNFs to scale up
V	Set of VNFs to scale down
$\mathbb{P} = \{p_1, p_2, \ldots\}$	List of policies
$\mathbb{L}_i$	List of virtual links of policy i
$\mathcal{P}$	Set of policies to scale up their bandwidth
P	Set of policies to scale down their bandwidth
θ_i	Average end-end latency of policy i
$\mathbb{T}_i$	End-to-End latency required by SLA for policy i
δ	Slack latency avoiding SLA violation i
$\overline{H}, \underline{H}$	Overloading and underloading thresholds for hosts CPU utilization
$\overline{B}, \underline{B}$	Overloading and underloading threshold for links bandwidth utilization
$\mathbb{L} = \{\mathbf{l_1}, \mathbf{l_2}, \ldots\}$	List of all virtual links
$\mathbb{L}_{c_i}$	List of virtual links used by the chain c_i
u_j^b	The utilized capacity of the virtual link of $\mathbf{l_j}$
u_i^h	The utilization of the host of the VNF node $\mathbf{v_i}$
$\mathbf{l_j}$	List of links in the jth virtual link of the chain
$\mathbf{v_k}$	List of VNF instances in the kth VNF node
$\|\|\mathbf{v_k}\|\|$	Size of VNF instances in $\mathbf{v_k}$
BW	Default unit bandwidth for scaling virtual links

the notations and their description used in this section are given in Table 1.

5.1. Elastic SFC algorithm

Our solution works based on the proposed Algorithm 1, called *ElasticSFC*, and is executed periodically at fixed time intervals to find the list of VNFs and virtual links that must be scaled to meet the end-to-end latency requirement of the policy flow. Note that the service chain end-to-end latency in this work is defined as the time it takes for a packet to traverse the entire service chain from the time it arrives at the ingress SF (sf_1) to the time it leaves the egress SF (sf_n). The variations in the latency caused by auto-scaling algorithms at the service chain level will be translated to network latency changes for the application in an abstract manner.

Algorithm 1 iterates over all policies in the outer loop of the algorithm in lines 3–38. For every policy i, we calculate the average end-to-end delay θ_i in the last time interval, and compare it with $\mathbb{T}_i$, the maximum end-to-end latency of the agreed on the SLA for the policy minus, δ, a small slack value avoiding SLA violation. If θ_i is larger than $\mathbb{T}_i - \delta$, the algorithm addresses the issue by allocating more compute and network resources to service chains in lines 5–18, otherwise, it finds resources that are underutilized and are suitable for scale down process in lines 21–35.

The scaling up process looks into two possible ways of enhancing the performance of the corresponding service chain. 1) In lines 6–12, ElasticSFC algorithm checks that if there exists any virtual link $\mathbf{l_j}$ in the chain that its average bandwidth utilization in the last time interval is higher than the maximum bandwidth threshold $\overline{B}$. Note that $\mathbf{l_j}$ can be a list of virtual links if VNF node is horizontally scaled earlier. All policies with an over-utilized virtual link $\mathbf{l_j}$ are added into the set $\mathcal{P}$ for more bandwidth allocation where function `ScaleBW` is called in line 42. Note that all virtual links of a policy are scaled up even if a single virtual link is over-utilized. 2) In lines 13–17, ElasticSFC algorithm adds all VNF nodes with average CPU utilization over the upper threshold $\overline{H}$ into the set $\mathcal{V}$. In line 40, all over-utilized VNF nodes are scaled by calling `ScaleNF` function. Note that VNF node can be a group of VNF instances if the VNF node is horizontally scale out earlier. This is the reason that both $\mathbf{l_j}$ and $\mathbf{v_k}$ are notated in boldface.

In the scaling down process, the algorithm checks if all virtual links in the service chain are under-utilized, that is, the average bandwidth utilization of every virtual link $\mathbf{l_j}$, u_i^b, must be smaller than the lower threshold $\underline{B}$. If this is the case, the policy is added to P, the set of policies that require to go through the bandwidth scaling down process in line 41, otherwise no bandwidth scaling down is required. A similar process is repeated for VNF nodes of the policy in the loop in lines 31–36. All VNF nodes with average CPU utilization below the lower threshold $\underline{H}$ are added to the set V to be scaled down later in line 39.

The computational complexity of the Elastic SFC algorithm is analyzed as follows. The outer loop iterates over all policies in the system, i.e., $|\mathbb{P}|$. For each polciy i, the scaling up/down process requires checking both every virtual link in $\mathbb{L}_i$ and every VNF node in $\mathbb{V}_i$ in the service chain. Therefore, assuming that $|\mathbb{L}|$ and $|\mathbb{V}|$ are the maximum number of virtual links and the maximum number of VNF nodes in the service chain, the computational complexity of Algorithm 1 is represented as $O(|\mathbb{P}| \times (\mathbb{L} + \mathbb{V}))$.

5.2. Bandwidth scaling algorithm

The ScaleBW algorithm (Algorithm 2) updates bandwidth for a service chain. The algorithm has two parts, one for scaling up the dedicated bandwidth, the other for scaling down. This is decided based on the *up* variable. Note that we assume, service chains are provided by dedicated bandwidth with minimal or no impact on other network traffic. The bandwidth scaling algorithm updates the dedicated bandwidth for each service chain in order to provision network resources elastically. In the scaling up process, lines 2–18, the algorithm allocates the total of *BW* unit of bandwidth to every virtual link. The loop in lines 3–17 iterates over all virtual links in the chain, it calculates ω as the number links in virtual link $\mathbf{l_j}$. If we have a horizontal scale-up for VNFs then $\mathbf{l_j}$ includes all those links connecting previous VNF node to the next one in the chain. The algorithm evenly distributes *BW* between all those links by dividing BW to the number of links noted as $|\mathbf{l_j}|$. *bw* represents the amount of bandwidth needs to be allocated to every link. We assume that when there are multiple links in a virtual link, forwarded packets are distributed among the duplicated VNFs using a *Round-Robin* fashion to equally distribute load across horizontally scaled VNFs. We also assume that a shared storage for sessions is accessible from any individual duplicated VNFs, thus packets can be simply redirected among duplicated VNFs even for the statefull flows. The inner loop in lines 5–16 tries to increase bandwidth by *bw* for every link l_k in the virtual link set $\mathbf{l_j}$. Firstly, it attempts to allocate *bw* for the link l_k in line 6. This includes bandwidth alloca-

Algorithm 1 Elastic SFC.

```
Input: $\mathbb{P}, \mathbb{T}, \mathbb{L}, \mathbb{V}$
 1: function ElasticSFC($x$)
 2:     $\mathcal{V}, \mathcal{P}, V, P = \{\}$
 3:     for $p_i$ in $\mathbb{P}$ do
 4:         $\theta_i \leftarrow$ Average end to end delay of $p_i$
 5:         if $\theta_i > \mathbb{T}_i - \delta$ then                     ▷ Scale Up
 6:             for $\mathbf{l_j}$ in $\mathbb{L}_i$ do
 7:                 $u_j^b \leftarrow$ Average bandwidth utilization of $\mathbf{l_j}$
 8:                 if $u_j^b > \overline{B}$ then
 9:                     Add $p_i$ to $\mathcal{P}$
10:                     Break
11:                 end if
12:             end for
13:             for $\mathbf{v_k}$ in $\mathbb{V}_i$ do
14:                 $u_i^h \leftarrow$ Average CPU utilization of $\mathbf{v_k}$
15:                 if $u_i^h > \overline{H}$ then
16:                     Add $\mathbf{v_k}$ to $\mathcal{V}$
17:                 end if
18:             end for
19:         else                                                   ▷ Scale Down
20:             $isScaleDown \leftarrow true$
21:             for $\mathbf{l_j}$ in $\mathbb{L}_i$ do
22:                 $u_j^b \leftarrow$ Average bandwidth utilization of $\mathbf{l_j}$
23:                 if $u_i^b > \underline{B}$ then
24:                     $isScaleDown \leftarrow false$
25:                     Break
26:                 end if
27:             end for
28:             if $isScaleDown$ then
29:                 Add $p_i$ to $P$
30:             end if
31:             for $\mathbf{v_k}$ in $\mathbb{V}_i$ do
32:                 $u_i^h \leftarrow$ Average utilization of $\mathbf{v_k}$
33:                 if $u_i^h < \underline{H}$ then
34:                     Add $\mathbf{v_k}$ to $V$
35:                 end if
36:             end for
37:         end if
38:     end for
39:     $\forall \mathbf{v}_k \in V$,   ScaleNF($\mathbf{v}_k, false$)
40:     $\forall \mathbf{v}_k \in \mathcal{V}$,   ScaleNF($\mathbf{v}_k, true$)
41:     $\forall p_i \in P$,   ScaleBW($\mathbb{L}_i, false$)
42:     $\forall p_i \in \mathcal{P}$,   ScaleBW($\mathbb{L}_i, true$)
43: end function
```

Algorithm 2 ScaleBW.

```
Input: $s, d$
 1: function ScaleBW($\mathbb{L}_i, up$)
 2:     if $up$ then
 3:         for $\mathbf{l_j}$ in $\mathbb{L}_i$ do
 4:             $\omega \leftarrow |\mathbf{l_j}|$, $bw \leftarrow \frac{BW}{\omega}$
 5:             for $l_k$ in $\mathbf{l_j}$ do
 6:                 if !AllocateBW($l_k, bw$) then
 7:                     $path \leftarrow$ FindAlternate($l_k, bw$)
 8:                     if $path <> None$ then
 9:                         ScheduleFlows($l_k, path$)
10:                         AllocateBW($l_k, bw$)
11:                     else
12:                         MigrateLink($l_k, bw$)
13:                         AllocateBW($l_k, bw$)
14:                     end if
15:                 end if
16:             end for
17:         end for
18:     else
19:         for $\mathbf{l_j}$ in $\mathbb{L}_i$ do
20:             $\omega \leftarrow |\mathbf{l_j}|$, $bw \leftarrow \frac{BW}{\omega}$
21:             for $l_k$ in $\mathbf{l_j}$ do
22:                 ReleaseBW($l_k, bw$)
23:             end for
24:         end for
25:     end if
26: end function
```

tion for all physical links beneath. If that is not possible, the algorithm finds an alternate path capable of accommodating the extra *bw* bandwidth for all physical links of the link, l_k in line 7. If there is such an alternate path, it calls the `ScheduleFlows` function to change the path for the link l_k, and updates bandwidth in line 10. Otherwise, it migrates the entire virtual link l_k by moving both or either of VNF instances at the two ends of the link to find a location that can provide additionally required bandwidth and then updates bandwidth in line 13. Finally, in lines 19–24, the algorithm simply releases bandwidth for all virtual links.

The outer loop of the algorithm iterates over all virtual links in the chain i that requires $|\mathbb{L}_i|$ iteration. The inner loop, then iterates over all links in the virtual link set $\mathbf{l_j}$ for $|\mathbf{l_j}|$ times. The time complexity of finding an alternate path algorithm depends on the topology of the network. If $|N|$ is the number of nodes (vertices) and $|E|$ is the number edges in the network, and Dijkstra's algorithm is used, then the computational complexity based on a min-priority queue implementation is $O(|E| + |N|log|N|)$. Link migration also requires time $O(|N|^2)$. As a result, the overall time complexity of the ScaleBW algorithm is represented as $O(|\mathbb{L}| \times |\mathbf{l_j}| \times (E + |N|^2))$.

5.3. Virtualized network functions scaling algorithm

The ScaleNF (Algorithm 3) is responsible for auto-scaling of a VNF node in the service chain. Similar to ScaleBW, it has two parts to allocate and release resources based on the boolean variable *up*. For allocating more resources (lines –), first, it attempts to vertically scale up VNF instances in $\mathbf{v_k}$ by means of allocating h units of capacity to each VNF instance. h is calculated based on the proportion of the scaling unit capacity H to the number of VNF instances in a VNF node, i.e., $|\mathbf{v_k}|$. If scaling up is not possible due to resource constraints (see line 7), it sets an indicator variable *horizontal* to true, and tries horizontal scaling by adding a VNF instance of size $||\mathbf{v_k}||$, that is, the size of VNF instances in VNF node $\mathbf{v_k}$ in line 1. The `ScaleOut` function is in charge of adding a VNF instance to the VNF node. The function needs to place the newly added VNF instance in the physical servers. It is not our aim to propose a VNF placement algorithm in this paper. Different methods proposed in the literature can be used for the placement of VNF instances. In this paper, we use first fit algorithm for placement of added VNF instances.

Lines 19–27 of the algorithm are dedicated to releasing surplus resources. If size of VNF instances are larger than unit capacity H, the algorithm performs vertical scale down in line 23; otherwise, it removes a VNF instance in line 26.

The time complexity of Algorithm 3 is analyzed as follows. The algorithm has a main loop that iterates over all VNF instances in the VNF node $\mathbf{v_k}$. The `ScaleOut` function complexity is the most compute intensive part of the algorithm that requires at least $O(|M|)$ time for the first fit placement algorithm, where M is the

Algorithm 3 ScaleNF.

```
Input: $s, d$
function ScaleNF($\mathbf{v_k}, up$)
    if $up$ then
        $\tau \leftarrow |\mathbf{v_k}|$
        $h \leftarrow \frac{H}{\tau}$
        $horizontal \leftarrow false$
        for $v_k$ in $\mathbf{v_k}$ do
            if !CanScaleUp($v_k, h$) then                    ▷ Scale Up
                $horizontal \leftarrow true$
            end if
        end for
        if !$horizontal$ then
            for $v_k$ in $\mathbf{v_k}$ do
                ScaleUp($v_k, h$)
            end for
        else                                                  ▷ Scale Out
            ScaleOut($||\mathbf{v_k}||$)    ▷ Add a VNF instance with size of ($||\mathbf{v_k}||$) and update bandwidth
        end if
    else
        if $||\mathbf{v_k}|| > H$ then
            $\tau \leftarrow |\mathbf{v_k}|$
            $h \leftarrow \frac{H}{\tau}$
            for $v_k$ in $\mathbf{v_k}$ do
                ScaleDown($v_k, h$)
            end for
        else
            ScaleIn()         ▷ Remove a VNF Instance and update bandwidth
        end if
    end if
end function
```

number of physical servers in the system. Therefore, the overall time complexity of the algorithm is $O(|v_k| \times N)$.

6. Performance evaluation

In this section, we present the experiment environment and results of the performance evaluation.

6.1. Experimental setup

To evaluate our algorithms, we modeled and simulated a software-defined cloud environment. We implemented the proposed algorithms on CloudSimSDN (Son et al., 2015) environment that extends the CloudSim (Calheiros et al., 2011) to support SDN and NFV features in the simulation. In order to simulate SFC functionalities and auto-scaling policies, we added extra modules to CloudSimSDN that creates SFs based on the exiting Virtual Machine module and the chain of multiple SFs. It also defines SFC policies that specify the source and destination VM, and the chain of SFs that transfers the traffic between the source and the destination VM. The modified CloudSimSDN can detect the network traffic from the source VM to the destination VM specified in the policy and enforce the traffic to travel through the SFC. SFs can be created and placed as like a traditional VMs with additional information to determine the capacity of the SF. SFC policies are specified along with the VMs and SFs so that CloudSimSDN can enforce the SFC for network traffics.

For the performance evaluation, a cloud data center with 8-pod fat-tree topology is created in CloudSimSDN, which consists of 128 computing nodes connected with 32 edge, 32 aggregation, and 16 core switches as shown in Fig. 4. Each pod has 4 edge and 4 aggregation switches, and each edge switch connects 4 hosts. All hosts have 16 cores with 10,000 MIPS each, and the network bandwidth between hosts and switches are set to 200 MBytes/sec. The main reason we chose the common fat-tree topology is that ElasticSFC algorithm uses dynamic flow scheduling to handle the bandwidth requirement of the chain and fat-tree has multiple shortest paths available between any given pair of hosts allowing for such dynamic flow scheduling. EltasticSFC is expected to provide equally comparable results with other network topologies with many equal-cost paths between a given pair of hosts such as leaf-spine (Alizadeh and Edsall, 2013), VL2 (Greenberg et al., 2009) or Bcube (Guo et al., 2009). Note that, we also consider CPU and network performance in the evaluation with the assumption that there is always enough memory and storage resources for VNFs.

6.1.1. Application scenario and workload

We create a 3-tier web application consisting of arbitrary number of web, app, and database (DB) VMs with the detailed specification depicted in Table 2. Once the request from end users arrived at web servers, the request is sent to an app server to retrieve the information from DB. Then, the DB responds to the app server with the relevant information which will be returned to the

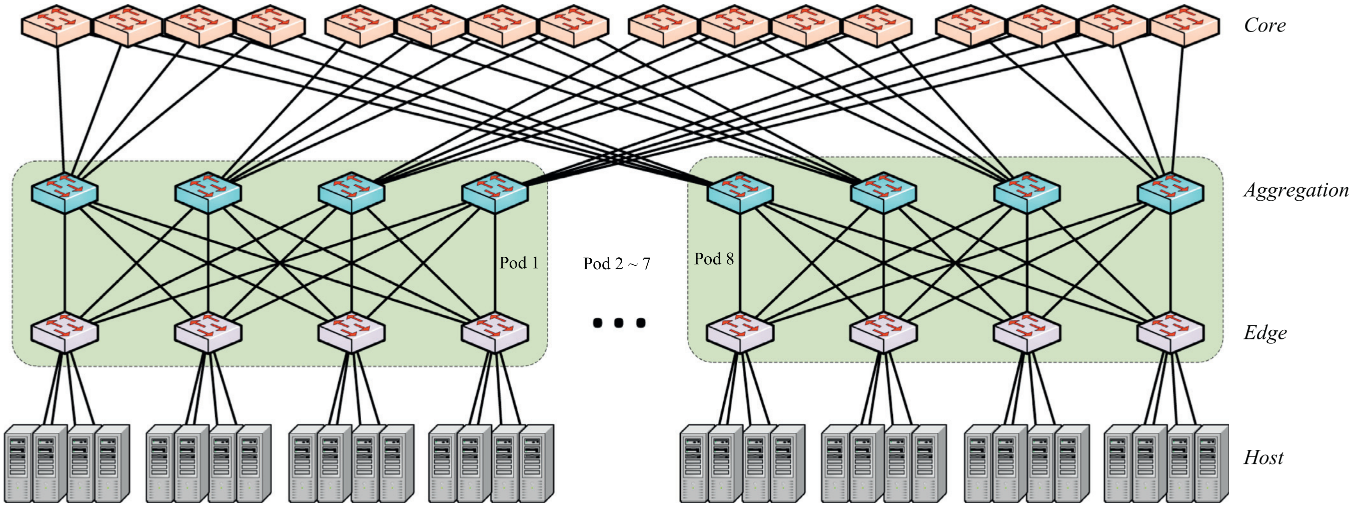


Fig. 4. 8-pod fat-tree topology setup for experiments.

Table 2
VM types for a 3-tier web application.

VM Type	CPU Capacity (cores*MIPS)	# of VMs
Web server	8*10,000	8
App server	4*10,000	24
Database	12*10,000	2

Table 3
SFC policies defined for the 3-tier application in the evaluation.

Source VM	Destination VM	SFC
Web server	App server	{FW, LB1}
App server	Database	{LB2, IDS}
Database	App server	{IDS, LB2}
App server	Web server	{LB1}

end-user through web server. Based on this process, we generate 4 SFC policies which enforces the network traffic to go through a chain of multiple SFs, such as firewall, load balancer, and intrusion detection system. Please note that these policies are designed for performance evaluation purposes and are not necessarily typical in real-world scenarios. The details of SFC policies for the evaluation are shown in Table 3. From the web to app server, the packet goes through the firewall (FW) to filter the request and the load balancer (LB1) to distribute the request across the number of app servers. Similarly, the request from app to DB is also sent through a load balancer (LB2) and intrusion detection system (IDS). For the response from DB to app, network traffic travels through the IDS and LB2 in the opposite direction. From app to web, however, the traffic goes through only LB1 as firewall is not necessary for the response packets. The detailed specification of SFs is explained in the next subsection.

The workload is generated based on the Wikipedia traces in German language for 24 h. Each workload consists of CPU processing in each VM and network traffic between VMs which must travel through the SFC defined in the policy. The number of requests in each hour is depicted in Fig. 5. In total, 29 million requests are generated to be distributed across the multiple servers through the load balancers specified in the SFC.

In the experiment, we set 10 s for time-out so that the request will be cancelled and marked as SLA violation if it cannot be processed within 10 s. The time interval is set to 60 s for the monitoring and scaling checkpoints, which results in running the proposed ElasticSFC algorithm every minute to check if any SFC needs to be scaled.

6.1.2. Baseline policies

The proposed auto-scaling algorithm (*ElasticSFC*) is evaluated by comparison with two baselines which does not implement scaling algorithms. The first baseline, named *NoScale-Min*, is to use the same amount of CPU and network resources as the initial resources given to the ElasticSFC algorithm, but without any scaling. In this baseline, we give the same amount of CPU and network resources to the SFs and the network flows which is same as the initial resource capacity for ElasticSFC policy. As the initial resources allocated to SFCs are not enough for the whole workload, NoScale-Min would result in higher SLA violations especially for high demand.

The second baseline (*NoScale-Max*) is to provide maximum resources enough to process the demand all the time which would guarantee the SLA without any violation. As the SFC allocates more than enough CPU and network resources at the beginning, this baseline is over-provisioning the resources which does not incur any SLA violations without auto-scaling.

In addition to the two static baselines without auto-scaling, we also compare the proposed algorithm with simple auto-scaling approaches which exploit auto-scaling method for only VNFs or network bandwidth. With *Scale-NF* approach, the system automatically adds or removes VMs for SFs based on the fluctuating real-time utilization of a SF. If the utilization of a SF exceeds the predefined threshold, more VMs are created for the SF. Likewise, VMs for a SF are removed once the utilization is below the minimum threshold. Scale-NF is similar to the approach presented in Ghaznavi et al. (2016) which deploys multiple VNF instances in distributed manner to provide network functionality. Scale-NF provides auto-scaling only for VMs, not for network bandwidth which ElasticSFC supports. We also implement *Scale-BW* algorithm as another baseline which scales only network bandwidth between SFs in a policy based on the bandwidth utilization. Similar to Cziva et al. (2018), Scale-BW dynamically allocates bandwidth for over-utilized links in a SF chain to reduce the network transmission time. Note that Scale-BW does not auto-scale VMs.

The initial resources allocated to SF and networks are shown in Table 4. For ElasticSFC and NoScale-Min policies, we allocate 1 VM for each SF with the minimum amount of CPU and network resources. For NoScale-Max policy, multiple VMs are allocated for FW and IDS in order to serve the entire workload. In order to simplify the comparison, we fix the unit MIPS in CPU resources (10,000) whereas the total MIPS changes only depending on the number of cores.

We measure the response time, SLA violation rate, and the amount of allocated CPU and network resources with different policies.

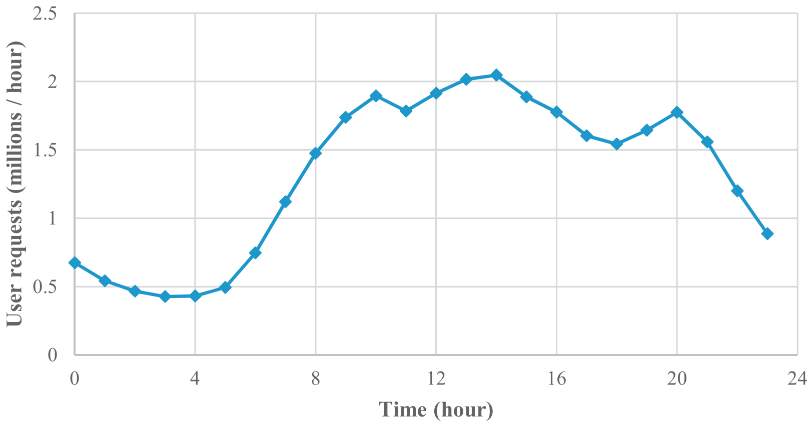


Fig. 5. The number of requests in the workload generated from Wikipedia trace.

Table 4
Initial SF resource allocations in each policy.

SF Type	ElasticSFC NoScale-Min			NoScale-Max		
	CPU (Cores*MIPS)	Number of VMs	Bandwidth (MB/s)	CPU (Cores*MIPS)	Number of VMs	Bandwidth (MB/s)
Firewall (FW)	8*10,000	1	500,000	16*10,000	3	2,000,000
Load balancer (LB1)	2*10,000	1	500,000	10*10,000	1	2,000,000
Load balancer (LB2)	2*10,000	1	500,000	10*10,000	1	2,000,000
IDS	6*10,000	1	500,000	12*10,000	3	2,000,000

6.2. Analysis of SLA violation rate and response time

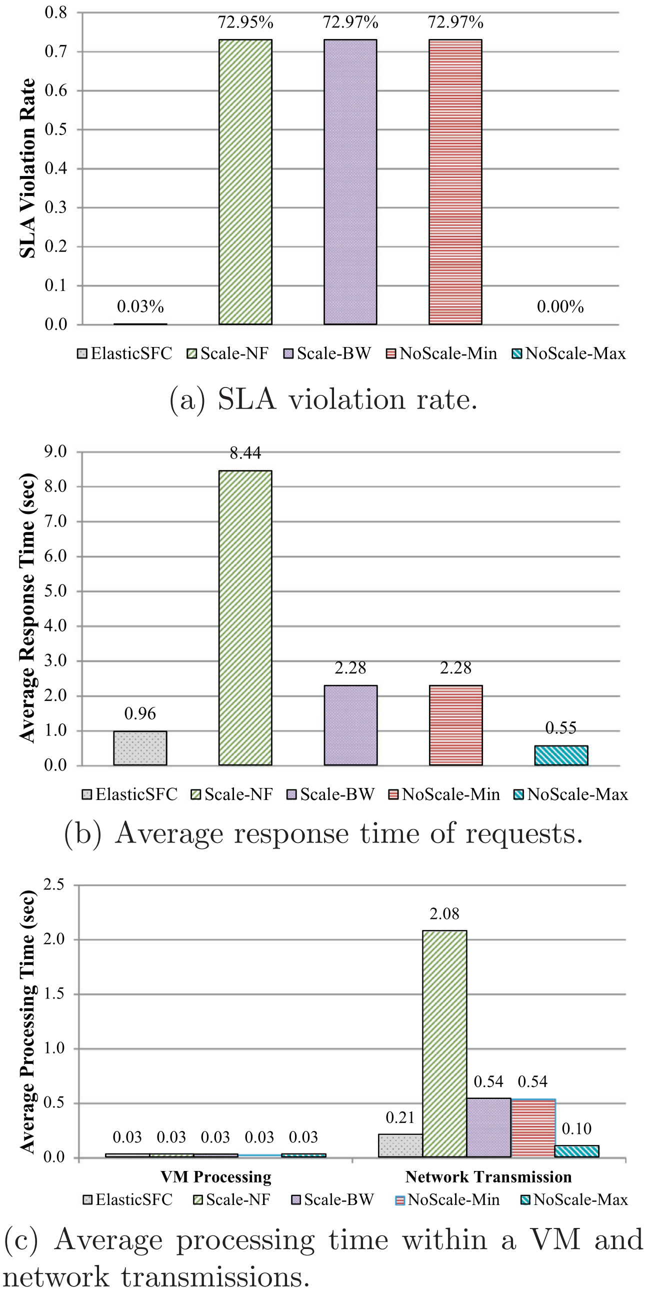
SLA violation rate is calculated based on the number of timed-out requests. We check the processing time inside a SF and network delay between SFs in the chain and mark the request as timed-out if it takes more than 10 s in any SF or network transmission due to the VM overhead or network congestion.

The SLA violation rate is shown in Fig. 6a. With the proposed ElasticSFC algorithm, the SLA violation rate is measured at 0.03%, significantly lower than NoScale-Min. As we discussed earlier, NoScale-Min allocates the minimum amount of CPU resources to SFs and the network resources to the SFC, which resulted in 72.97% of the requests to be timed-out during the experiment. On the other hand, NoScale-Max provides enough resources to process all the requests, which is why no request is timed out.

For the auto-scaling baselines, the SLA violation remains as high as NoScale-Min policy because their auto-scaling is only applied either in VM (Scale-NF) or network bandwidth (Scale-BW). Although Scale-NF policy can add more VMs if the current capacity of VMs are not enough to process incoming requests, it does not provision network bandwidth which becomes bottleneck even after more VMs have been created for a SF. As the network transmission cannot be processed within the deadline, the SLA violation rate of Scale-NF policy reaches at 72.95% which is almost same as NoScale-Min. Similarly, Scale-BW does provide auto-scaling mechanism for VMs, which made the SLA violation as high as the policy without auto-scaling.

The small amount of SLA violations in the proposed ElasticSFC algorithm is caused during the gap time between the high-demand requests and the monitoring checkpoint. The requests are timed out if they are processed before running ElasticSFC algorithm to scale up the SFC. The SLA violation rate can be reduced by decreasing the interval time between the monitoring checkpoints, so that the algorithm runs more often to check if a SFC is overloaded, which however results in the overhead of frequent VM and network scaling. Nonetheless, the result shows that the proposed algorithm can provide elastic resources to adapt to the change of the workload.

We also evaluate the performance of the algorithm by comparing the average response time. Fig. 6b shows the average response time of requests excluding the SLA violated ones. Similar to the SLA violation results, the requests are responded in 0.96 s with ElasticSFC algorithm which is slightly longer than NoScale-Max, but significantly faster than the other baselines. Because of the elastic resource allocation adapted to the workload, ElasticSFC algorithm reduces the amount of resources if the utilization is less than the threshold, which results in more resource saving but increasing the response time slightly. Scale-NF baseline increases the average response time to 8.44 s which is far more than NoScale-Min result. In order to find the reason of this result, we further investigated to measure the average processing time in VMs and network transmissions shown in Fig. 6c. For VM processing, all policies had the same average processing time at 0.03 s. However, for network transmission, Scale-NF takes far more than the other baselines. This shows that the excessive amount of time is wasted for network transmission in Scale-NF, due to more number of requests consuming the limited bandwidth especially after the VMs have been scaled up. The static amount of network resources with more numbers of VMs for a SF creates a bottleneck in network transmission, which results in increasing average response time.

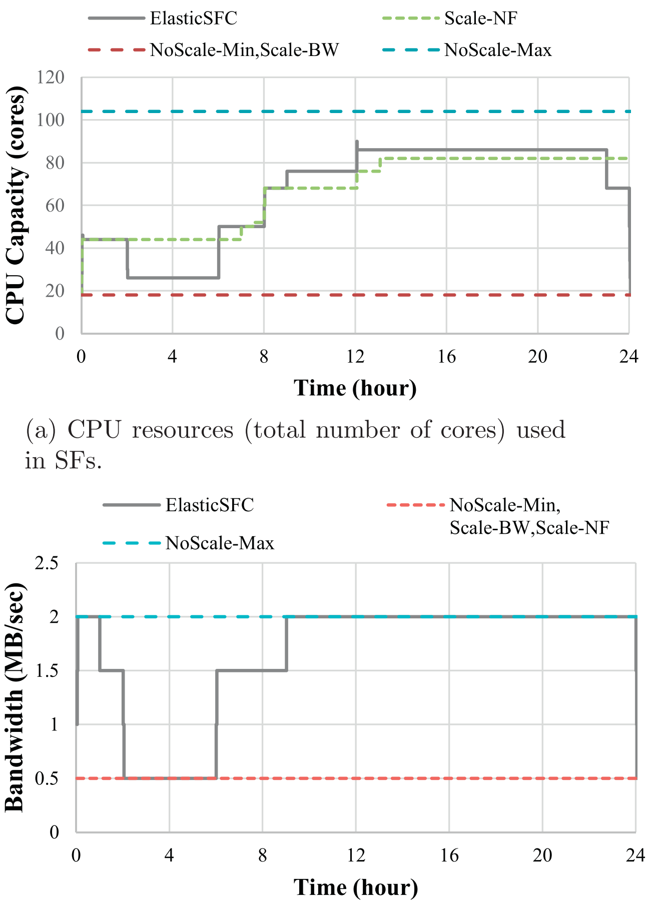


(a) SLA violation rate.

(b) Average response time of requests.

(c) Average processing time within a VM and network transmissions.

Fig. 6. SLA violation rate and average response time in Wikipedia application with different policies.



(a) CPU resources (total number of cores) used in SFs.

(b) Network bandwidth allocated for a SF chain.

Fig. 7. Allocated CPU and network resources for SFCs over time.

6.3. Analysis of resource Usage

We measure the allocated CPU and network resources over time. Fig. 7 shows the number of total CPU cores used in all SFs and the network bandwidth allocated to the SFC from DB to App. For CPU cores (Fig. 7a), ElasticSFC policy started with the initial resource allocation (18 cores), but immediately increased to 44 to adapt to the workload. After that, the algorithm dynamically changes the resource capacity adapting to the workload shown in Fig. 5. The total CPU cores used in NoScale-Min, Scale-BW, and NoScale-Max baseline policies remain at the same amount without any change for the entire experiment, because they do not scale the VMs of service functions elastically. On the other hand, the number of CPU cores with Scale-NF policy dynamically adapts to the workload similar to our ElasticSFC, although there was no scaling down between 4 and 6 h which is due to the continuous SLA violation.

Similarly, the allocated network bandwidth is adapted to the workload with our proposed algorithm (Fig. 7b). Initially starting with the minimum bandwidth (0.5MBytes/sec), it was immediately increased to 2 MB/sec once the SLA violation detected. When the number of requests is decreased after 2 h, the allocated bandwidth is also reduced to 0.5MB/sec. In NoScale-Min and NoScale-Max, the bandwidth allocation is consistent. It is worth mentioning that, in Fig. 7b, the Scale-BW policy also does not have different bandwidth, since CPU capacity is the bottleneck in this case and adding more bandwidth will not improve the end-to-end latency. This observation demonstrate the importance of dynamic auto-scaling across both computing and network resources.

In short, the proposed algorithm dynamically adjusts the amount of CPU and network resources for SFCs elastically adapting to the workload. Although Scale-NF can scale up and down for VMs similar to ElasticSFC, it does not scale network bandwidth which results in continuous SLA violation. The results show that we can achieve the resource saving with the proposed algorithm which does not over-provision the resources all the time, yet it provides adequate amount of resources for the dynamically changing workload.

7. Related work

Network service chaining, also known as service function chaining (SFC) is an automated process used by network operators to set up a chain of connected network services. SFC enables the assembly of the chain of virtual network functions (VNFs) in an NFV environment using instantiation of software-only services running on commodity hardware. To avoid tedious manual steps of the chain setup, the process of service chain provisioning in NFV environments happens through an NFV management and orchestration (MANO) framework. Managing and orchestrating of VNFs in NFV MANO has been a popular research topic and a widely studied problem in the literature (Medhat et al., 2017).

The problem of VNF placement, often very related to the traditional VM placement in cloud computing environments (Eramo et al., 2017), has gained considerable attention over the past few years (Ghaznavi et al., 2016). MORSA (Yoshida et al., 2014) is a multi-objective VNF placement approach proposed as part of vConductor framework (Shen et al., 2014) to minimize the physical machine load and the intra-data center traffic. MORSA is designed to optimize placement for VNFs while it does not take into account other aspects such as auto-scaling of VNFs, consolidation, and service chaining. Similarly, Clayman et al. (2014) proposed MANO framework for the automated placement of VNFs across the resources. They also did not consider scaling and consolidations. SOVWin is a heuristic proposed by Pai et al. (2017) to address the service-chain deployment problem. Different from other works, SOVWin aims at satisfying SLA requirements and maximizing the number of accommodated user requests.

The above works do not consider auto-scaling of service chains. VNF-P is a model proposed by Moens and Turck (2014) for efficient placement of VNFs. They propose an NFV burst scenario in which the base demand for the network function service is handled by physical resources while the extra load is redirected to the virtual service instances. Their method only considers a single service chain. Drãxler et al. (2017) go one step further and propose a mixed integer programming solution and a custom heuristic to address scaling and placement problem of VNFs jointly. Wang et al. (2016) propose online algorithms for dynamic provisioning of virtualized network services. Their approach determines the numbers of VNF instances and their placement on physical servers to optimize operational cost and resource utilization of the system. In a recent work, (Cziva et al., 2018) focused on the VNF placement to optimize end-to-end latency for users. Their optimization method is adapted for a dynamic and ever-changing edge networks. Rajagopalan et al. (2013) proposed a system called FreeFlow based on the Split/Merge abstraction model that enables elasticity for stateful virtual network services. FreeFlow addresses auto-scaling of virtualized middleboxes by removing the configuration complexity of running independent middleboxes. Stratos (Gember et al., 2013) is a MANO framework aims to provide a scalable network-aware strategy based on traffic engineering, elastic scaling, and VM migration for service function chaining. Ghaznavi et al. (2016) propose an optimization model and heuristic called Kariz to optimize service chain deployment

Table 5
Comparison of existing approaches for VNF placement and auto-scaling.

Work	Placement	Migration	SLA	Horizontal	Vertical	Dynamic Flows	SFC
MORSA (Yoshida et al., 2014)	√	✗	✗	✗	✗	✗	✗
Clayman et al. (2014)	√	✗	✗	✗	✗	✗	√
SOVWin (Pai et al., 2017)	√	✗	√	✗	✗	✗	√
VNF-P (Moens and Turck, 2014)	√	✗	✗	√	✗	✗	✗
Drãxler et al. (2017)	√	✗	✗	√	✗	✗	√
Wang et al. (2016)	√	✗	✗	√	✗	✗	√
Cziva et al. (2018)	√	√	√	✗	✗	✗	√
FreeFlow (Rajagopalan et al., 2013)	✗	✗	√	√	✗	√	✗
Stratos (Gember et al., 2013)	✗	√	✗	√	✗	√	√
Kariz (Ghaznavi et al., 2015)	√	✗	√	✗	✗	√	√
SLFL (Ghaznavi et al., 2016)	√	√	✗	√	✗	✗	√
Eramo et al. (2017)	√	√	✗	✗	√	✗	√
NFV-RT (Li et al., 2016)	√	✗	√	✗	√	✗	√
DFCA (Wang et al., 2015)	✗	✗	√	✗	✗	✗	√
ElasticSFC	✗	√	√	√	√	√	√

targeting custom throughput. In another work (Ghaznavi et al., 2015), they propose a heuristic called Simple Lazy Facility Location (SLFL) to minimize the overall operational costs by the efficient consolidation of VNFs using migration and horizontal autoscaling of VNFs. Similarly, (Eramo et al., 2017) study the consolidation, routing, and placement of VNFs in an NFV environment when the vertical auto-scaling of VNFs is the case. They propose a migration policy that decides when and to where migration of VNF instances must be done to minimize the overall cost.

Majority of these solutions are heuristics-based that they do not provide SLA guarantee. Li et al. (2016) propose NFV-RT that dynamically provisions resources in an NFV environment to provide packet-wise timing guarantees to service requests. They formulate the resource provisioning problem with a mathematical model and evaluate their solution using a simulator. Wang et al. (2015) have developed a combinatorial optimization model to address dynamic function composition problem, and proposed a distributed algorithm using Markov approximation method to dynamically decide the appropriate service function instances at runtime. They also use simulation to evaluate their proposed algorithm.

In comparison to other works in the area (Table 5), the innovative contribution of this work is that it proposes a holistic method for auto-scaling of VNFs to build elastic service chains dynamically adapting to the changes of the workload for different network policies. Our model uses migration and horizontal/vertical auto-scaling of VNFs along with dynamic bandwidth allocation and flow scheduling to meet the SLA requirement of a service chain. It does not only dynamically scale NFs within a chain, but also allocates/releases dedicated network bandwidth, updates network paths, and relocates communicating VNFs to meet the latency requirements of the chain while minimizing the overall cost. Even though we use a simple placement algorithm to build our system, VNF placement is not the primary focus of this work and any placement algorithm can be combined with our proposed method.

8. Conclusions and future work

In this paper, we presented a framework to build elastic service chains in NFV-based cloud computing environments. We proposed a set of auto-scaling algorithms to meet end-to-end delay requirements of the service chains while minimizing the overall operational cost. We implemented our algorithms by extending a cloud simulator and conducted realistic experiments using workload traces of Wikipedia application. Our experimental results showed that our proposed method significantly reduces the cost of SFC deployment while reduces SLA violation to 0.03% compared to 72.97% SLA violations of no scaling scenario.

In this work, we assumed that all VNFs are horizontally scalable regardless of their states. A logical extension of this work is to add solutions for auto-scaling stateful NFs and those that are none trivial to scale horizontally. As a future work, we also aim to implement our algorithms in a real networking environment using our micro data center designed for software-defined cloud computing and networking. Our micro data center uses OpenStack as Virtualized Infrastructure Manager and OpenDaylight (ODL) as SDN controller. We will extend ODL SFC Project to evaluate and validate our proposed techniques. Using a real system, we are able to measure latency and link delays for service chains along with impacts of dynamic flow scheduling and VNF migrations on the runtime phase of the SFC lifecycle. We will also develop autonomic SFC composition and management that targets efficient utilization of the data center including placement algorithms and consolidation of VNF instances.

Acknowledgments

This work supported through a collaborative research agreement between the University of Melbourne and Huawei Technologies Co., Ltd (China) as part of the Huawei Innovation Research Program (HIRP). The work is carried out within the Melbourne CLOUDS Laboratory at the University of Melbourne.

References

Alizadeh, M., Edsall, T., 2013. On the data path performance of leaf-spine datacenter fabrics. In: The 21st IEEE Annual Symposium on High-Performance Interconnects, pp. 71–74. doi:10.1109/HOTI.2013.23.

Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A., Buyya, R., 2011. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software 41 (1), 23–50.

Clayman, S., Maini, E., Galis, A., Manzalini, A., Mazzocca, N., 2014. The dynamic placement of virtual network functions. In: Proceedings of the IEEE Network Operations and Management Symposium (NOMS'14), pp. 1–9. doi:10.1109/NOMS.2014.6838412.

Cui, L., Tso, F.P., Pezaros, D.P., Jia, W., Zhao, W., 2017. Plan: joint policy- and network-aware vm management for cloud data centers. IEEE Trans. Parallel Distrib. Syst. 28 (4), 1163–1175. doi:10.1109/TPDS.2016.2604811.

Cziva, R., Anagnostopoulos, C., Pezaros, D.P., 2018. Dynamic, latency-optimal VNF placement at the network edge. In: Proceedings of the 37th Annual IEEE International Conference on Computer Communications (IEEE INFOCOM 2018). Honolulu, HI, USA.

Drãxler, S., Karl, H., Mann, Z., 2017. Joint optimization of scaling and placement of virtual network services. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'17), pp. 365–370. doi:10.1109/CCGRID.2017.25.

Eramo, V., Miucci, E., Ammar, M., Lavacca, F.G., 2017. An approach for service function chain routing and virtual function network instance migration in network function virtualization architectures. IEEE/ACM Trans. Netw. 25 (4), 2008–2025. doi:10.1109/TNET.2017.2668470.

Gember, A., Krishnamurthy, A., John, S.S., Grandl, R., Gao, X., Anand, A., Benson, T., Akella, A., Sekar, V., 2013. Stratos: a network-aware orchestration layer for middleboxes in the cloud. Technical Report. Technical Report.

Ghaznavi, M., Khan, A., Shahriar, N., Alsubhi, K., Ahmed, R., Boutaba, R., 2015. Elastic virtual network function placement. In: Proceedings of the 4th IEEE International Conference on Cloud Networking (CloudNet'15), pp. 255–260. doi:10.1109/CloudNet.2015.7335318.

Ghaznavi, M., Shahriar, N., Ahmed, R., Boutaba, R., 2016. Service Function Chaining Simplified. Technical Report arXiv:1601.00751.

Greenberg, A., Hamilton, J.R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D.A., Patel, P., Sengupta, S., 2009. Vl2: a scalable and flexible data center network. In: Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication. ACM, New York, NY, USA, pp. 51–62. doi:10.1145/1592568.1592576.

Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., Lu, S., 2009. BCube: a high performance, server-centric network architecture for modular data centers. In: Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication. ACM, New York, NY, USA, pp. 63–74. doi:10.1145/1592568.1592577.

Li, Y., Phan, L.T.X., Loo, B.T., 2016. Network functions virtualization with soft real-time guarantees. In: Proceedings of the 35th Annual IEEE International Conference on Computer Communications (IEEE INFOCOM 2016), pp. 1–9. doi:10.1109/INFOCOM.2016.7524563.

Mechtri, M., Ghribi, C., Soualah, O., Zeghlache, D., 2017. NFV orchestration framework addressing SFC challenges. IEEE Commun. Mag. 55 (6), 16–23. doi:10.1109/MCOM.2017.1601055.

Medhat, A.M., Taleb, T., Elmangoush, A., Carella, G.A., Covaci, S., Magedanz, T., 2017. Service function chaining in next generation networks: state of the art and research challenges. IEEE Commun. Mag. 55 (2), 216–223. doi:10.1109/MCOM.2016.1600219RP.

Moens, H., Turck, F.D., 2014. Vnf-p: a model for efficient placement of virtualized network functions. In: Proceedings of the 10th International Conference on Network and Service Management (CNSM) and Workshop, pp. 418–423. doi:10.1109/CNSM.2014.7014205.

Pai, Y.-M., Wen, C.H.P., Tung, L.-P., 2017. Sla-driven ordered variable-width windowing for service-chain deployment in SDN datacenters. In: Proceedings of the International Conference on Information Networking (ICOIN'17), pp. 167–172. doi:10.1109/ICOIN.2017.7899498.

Pham, C., Tran, N.H., Ren, S., Saad, W., Hong, C.S., 2017. Traffic-aware and energy-efficient VNF placement for service chaining: joint sampling and matching approach. IEEE Trans. Serv. Comput. PP (99). 1–1 doi: 10.1109/TSC.2017.2671867.

Rajagopalan, S., Williams, D., Jamjoom, H., Warfield, A., 2013. Split/merge: system support for elastic execution in virtual middleboxes. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation. USENIX Association, Berkeley, CA, USA, pp. 227–240.

Shen, W., Yoshida, M., Kawabata, T., Minato, K., Imajuku, W., 2014. vConductor: an NFV management solution for realizing end-to-end virtual network services. In: Proceedings of the 16th Asia-Pacific Network Operations and Management Symposium, pp. 1–6. doi:10.1109/APNOMS.2014.6996522.

Son, J., Dastjerdi, A.V., Calheiros, R.N., Ji, X., Yoon, Y., Buyya, R., 2015. CloudSimSDN: modeling and simulation of software-defined cloud data centers. In: Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 475–484. doi:10.1109/CCGrid.2015.87.

Szabo, R., Kind, M., Westphal, F.J., Woesner, H., Jocha, D., Csaszar, A., 2015. Elastic network functions: opportunities and challenges. IEEE Netw. 29 (3), 15–21. doi:10.1109/MNET.2015.7113220.

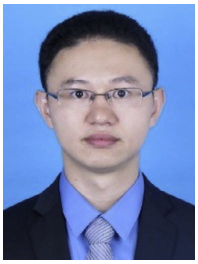
Wang, P., Lan, J., Zhang, X., Hu, Y., Chen, S., 2015. Dynamic function composition for network service chain: model and optimization. Comput. Netw. 92, 408–418. doi:10.1016/j.comnet.2015.07.020.

Wang, X., Wu, C., Le, F., Liu, A., Li, Z., Lau, F., 2016. Online VNF scaling in datacenters. In: Proceedings of the 9th International Conference on Cloud Computing (CLOUD'16), pp. 140–147. doi:10.1109/CLOUD.2016.0028.

Yoshida, M., Shen, W., Kawabata, T., Minato, K., Imajuku, W., 2014. MORSA: a multi-objective resource scheduling algorithm for NFV infrastructure. In: Proceedings of the 16th Asia-Pacific Network Operations and Management Symposium, pp. 1–6. doi:10.1109/APNOMS.2014.6996545.

Adel Nadjaran Toosi is a Lecturer in Faculty of Information Systems, Monash University, Australia. Before joining Monash University, he was a research fellow at the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne. He received his Ph.D. degree from the University of Melbourne in 2015. His research interests include Distributed Systems, Cloud Computing, Software-Defined Networking (SDN), Green Computing, and Soft Computing. He is currently working on the optimization of renewable energy use in micro-data centers. For more details, please visit his homepage: http://adelnadjarantoosi.info.

Jungmin Son is a Research Assistant at the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He completed his Ph.D. degree with CLOUDS Laboratory, received the B.Eng. degree in information and computer engineering from Ajou University, Korea, and the M.Info.Tech. degree from the University of Melbourne in 2006 and 2013 respectively. His research interests include SDN-enabled cloud computing, resource provisioning, scheduling, and energy efficient data centers.



Qinghua Chi is a Senior Engineer in Wireless Network Research Department of Shanghai Huawei Technologies Co., Ltd. He received his Ph.D. degree in control science and engineering from Zhejiang University, Hangzhou, China, in 2015. His current research interests include machine learning and its application in wireless communication network, resource management in 5G network slice, and model predictive control with relevant identification.

Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. Dr. Buyya is recognized as a "Web of Science Highly Cited Researcher" in 2016 and 2017 by Thomson Reuters, a Fellow of IEEE, and Scopus Researcher of the Year 2017 with Excellence in Innovative Research Award by Elsevier for his outstanding contributions to Cloud computing. For further information on Dr. Buyya, please visit his cyberhome: www.buyya.com.