ElasticSFC: Auto-Scaling Techniques for Elastic Service Function Chaining in Network Functions Virtualization-based Clouds

Adel Nadjaran Toosi\textsuperscript{1}, Jungmin Son\textsuperscript{c}, Qinghua Chi\textsuperscript{b}, Rajkumar Buyya\textsuperscript{c}

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

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.

Keywords: service function chaining, network functions virtualization, virtualized network functions, software defined networking, auto-scaling, cloud computing
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 [1]. 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) [2]. 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 [1]. 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 [3].

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) [4]. 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 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 [5, 6, 7, 8, 9]. 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 [10, 11].

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 policies, 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 [12]. 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). 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 \( P = \{p_1, p_2, \ldots\} \) as the set of SFC policies in the system. An SFC policy \( p_i \) is defined in the form of \( \{\text{flow} \rightarrow \text{sequence}\} \). A flow is denoted by a 5-tuple: \( \{\text{src}, \text{dst}, \text{srcport}, \text{dstport}, \text{proto}\} \), 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 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 en-
environments [2]. 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 [4].

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 [13]. 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 Figure 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
Figure 1: Service Function Chaining Architecture

The 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

\(^2\)https://www.etsi.org/technologies-clusters/technologies/nfv/open-source-mano
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 manages 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 rules 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.

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

Figure 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.

Figure 2b illustrates the case that VNF instance B is vertically scaled up
Figure 3: Scaling Bandwidth for a Service Chain.
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 Figure 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 [14]. The SDN controller and NFV manager work in coordination to perform the allocation and management of resources required by elastic SFC. Figure 3a depicts the same service chain in Figure 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 Figure 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 hops) 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 Figure 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
5. Proposed Algorithms

In this section, we propose our solution including series of algorithms to build elastic service chains in NFV environments. All 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 ($s_{f_1}$) to the time it leaves the egress SF ($s_{f_n}$). 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 Elastic SFC

Input: $P, T, L, V$

1: function ELASTICSFC($x$)
2:   $\forall, P, V, P = \{\}$
3:   for $p_i$ in $P$ do
4:       $\theta_i \leftarrow$ Average end to end delay of $p_i$
5:       if $\theta_i > T_i - \delta$ then $\triangleright$ Scale Up
6:           for $l_j$ in $L_i$ do
7:               $u_j^b \leftarrow$ Average bandwidth utilization of $l_j$
8:               if $u_j^b > B$ then
9:                   Add $p_i$ to $P$
10:                  Break
11:             end if
12:           end for
13:           for $v_k$ in $V_i$ do
14:               $u_k^h \leftarrow$ Average CPU utilization of $v_k$
15:               if $u_k^h > H$ then
16:                   Add $v_k$ to $V$
17:             end if
18:           end for
19:       else $\triangleright$ Scale Down
20:           $isScaleDown \leftarrow$ true
21:           for $l_j$ in $L_i$ do
22:               $u_j^b \leftarrow$ Average bandwidth utilization of $l_j$
23:               if $u_j^b > B$ then
24:                   $isScaleDown \leftarrow$ false
25:                   $break$ $isScaleDown \leftarrow$ false
26:             end if
27:           end for
28:           if $isScaleDown$ then
29:               Add $p_i$ to $P$
30:           end if
31:           for $v_k$ in $V_i$ do
32:               $u_k^h \leftarrow$ Average utilization of $v_k$
33:               if $u_k^h < H$ then
34:                   Add $v_k$ to $V$
35:             end if
36:           end for
37:       end if
38:   end for
39:  $\forall v_k \in V$, SCALENF($v_k$, false)
40:  $\forall v_k \in V$, SCALENF($v_k$, true)
41:  $\forall p_i \in P$, SCALEBW($L_i$, false)
42:  $\forall p_i \in P$, SCALEBW($L_i$, true)
43: end function
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 $\theta_i$ in the last time interval, and compare it with $T_i$, the maximum end-to-end latency of the agreed on the SLA for the policy minus, $\delta$, a small slack value avoiding SLA violation. If $\theta_i$ is larger than $T_i - \delta$, 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.

Algorithm 2 ScaleBW

Input: $s, d$

1: function ScaleBW($\mathbb{L}_i, up$)
2:     if $up$ then
3:         for $l_j$ in $\mathbb{L}_i$ do
4:             $\omega \leftarrow |l_j|$, $bw \leftarrow \frac{BW}{\omega}$
5:                 for $l_k$ in $l_j$ 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 for
16:             end for
17:     else
18:         for $l_j$ in $\mathbb{L}_i$ do
19:             $\omega \leftarrow |l_j|$, $bw \leftarrow \frac{BW}{\omega}$
20:                 for $l_k$ in $l_j$ do
21:                     ReleaseBW($l_k, bw$)
22:             end for
23:         end if
24: end function

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 $l_j$ in the chain that its average bandwidth utilization in the last time interval is higher than the
Algorithm 3 ScaleNF

Input: $s, d$

1: function ScaleNF($v_k, up$)
2:     if $up$ then
3:         $\tau \leftarrow |v_k|$ ||
4:         $h \leftarrow \frac{H}{\tau}$ ||
5:         $horizontal \leftarrow false$ ||
6:         for $v_k$ in $v_k$ do
7:             if !$CANSCALEUP(v_k, h)$ then $\triangleright$ Scale Up
8:                 $horizontal \leftarrow true$ ||
9:         end if
10:     end for
11:     if !$horizontal$ then
12:         for $v_k$ in $v_k$ do
13:             SCALEUP($v_k, h$)
14:         end for
15:     else $\triangleright$ Scale Out
16:         SCALEOUT($|v_k|)$ $\triangleright$ Add a VNF instance with size of ($|v_k|$) and update bandwidth
17:     end if
18:     else
19:         if $||v_k|| > H$ then
20:             $\tau \leftarrow |v_k|$ ||
21:             $h \leftarrow \frac{H}{\tau}$ ||
22:             for $v_k$ in $v_k$ do
23:                 SCALEDOWN($v_k, h$)
24:             end for
25:         else $\triangleright$ Remove a VNF Instance and update bandwidth
26:         SCALEIN() ||
27:     end if
28:     end if
29: end function
maximum bandwidth threshold $\overline{B}$. Note that $I_j$ can be a list of virtual links if VNF node is horizontally scaled earlier. All policies with an over-utilized virtual link $I_j$ are added into the set $P$ for more bandwidth allocation where function $\text{ScaleBW}$ 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 $V$. In line 40, all over-utilized VNF nodes are scaled by calling $\text{ScaleNF}$ 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 $I_j$ and $v_k$ 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 $I_j$, $u_i^b$, 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., $|P|$. For each policy $i$, the scaling up/down process requires checking both every virtual link in $L_i$ and every VNF node in $V_i$ in the service chain. Therefore, assuming that $|L|$ and $|V|$ 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 $O(|P| \times (L + V))$.

5.2. Bandwidth Scaling Algorithm

The $\text{ScaleBW}$ 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 $\omega$ as the number links in virtual link $I_j$. If we have a horizontal
scale-up for VNFs then \( l_j \) 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 \( |l_j| \). \( bw \) 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 \( l_j \). Firstly, it attempts to allocate \( bw \) for the link \( l_k \) in line 6. This includes bandwidth allocation 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 \( |L_i| \) iteration. The inner loop, then iterates over all links in the virtual link set \( l_j \) for \( |l_j| \) 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 \( O(|L| \times |l_j| \times (E + |N|^2)) \).

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 3-17), first, it attempts to vertically scale up VNF instances in \( v_k \) 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., $|v_k|$. 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 $|v_k|$, that is, the size of VNF instances in VNF node $v_k$ 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 $v_k$. 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 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 [15] environment that extends the CloudSim [16] 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 Figure 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. ElasticSFC 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 [17], VL2 [18] or Bcube [19]. 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 end-user through web server. Based on this process, we gen-
erate 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 hours. 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 Figure 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 seconds for time-out so that the request will be cancelled and marked as SLA violation if it cannot be processed within 10 seconds. The time interval is set to 60 seconds 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

<table>
<thead>
<tr>
<th>VM Type</th>
<th>CPU Capacity (cores*MIPS)</th>
<th># of VMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web server</td>
<td>8*10,000</td>
<td>8</td>
</tr>
<tr>
<td>App server</td>
<td>4*10,000</td>
<td>24</td>
</tr>
<tr>
<td>Database</td>
<td>12*10,000</td>
<td>2</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Source VM</th>
<th>Destination VM</th>
<th>SFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web server</td>
<td>App server</td>
<td>{FW, LB1}</td>
</tr>
<tr>
<td>App server</td>
<td>Database</td>
<td>{LB2, IDS}</td>
</tr>
<tr>
<td>Database</td>
<td>App server</td>
<td>{IDS, LB2}</td>
</tr>
<tr>
<td>App server</td>
<td>Web server</td>
<td>{LB1}</td>
</tr>
</tbody>
</table>
VMs for a SF are removed once the utilization is below the minimum threshold. Scale-NF is similar to the approach presented in [20] 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 [11], 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.

### 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 seconds in any SF or network transmission due to the VM overhead or network congestion.

The SLA violation rate is shown in Figure 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
(a) SLA violation rate.

(b) Average response time of requests.

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

Figure 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.

Figure 7: Allocated CPU and network resources for SFCs over time.
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. Figure 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 seconds 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 seconds 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 Figure 6c. For VM processing, all policies had the same average processing time at 0.03 seconds. 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.

6.3. Analysis of Resource Usage

We measure the allocated CPU and network resources over time. Figure 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 (Figure 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 Figure 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 hours which is due to the continuous SLA violation.

Similarly, the allocated network bandwidth is adapted to the workload with our proposed algorithm (Figure 7b). Initially starting with the minimum bandwidth (0.5MB/sec), it was immediately increased to 2 MB/sec once the SLA violation detected. When the number of requests is decreased after 2 hours, 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 Figure 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.
Table 5: Comparison of existing approaches for VNF placement and auto-scaling.

<table>
<thead>
<tr>
<th>Work</th>
<th>Placement</th>
<th>Migration</th>
<th>SLA</th>
<th>Horizontal</th>
<th>Vertical</th>
<th>Dynamic Flows</th>
<th>SFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>MORSA [5]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Clayman et al. [9]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SOVWin [21]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>VNF-P [6]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Drexler et al. [22]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Wang et al. [7]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Cziva et al. [11]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>FreeFlow [23]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Stratos [8]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Kariz [4]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SLFL [20]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Eramo et al. [2]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>NFV-RT [10]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>DFCA [24]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ElasticSFC</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

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 [14].

The problem of VNF placement, often very related to the traditional VM placement in cloud computing environments [2], has gained considerable attention over the past few years [20]. MORSA [5] is a multi-objective VNF placement approach proposed as part of vConductor framework [25] 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. [9] 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. [21] 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 [6] 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. Dräxler et al. [22] 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. [7] 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. [11] 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. [23] 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 [8] 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. [20] propose an optimization model and heuristic called Kariz to optimize service chain deployment targeting custom throughput. In another work [4], 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 auto-scaling of VNFs. Similarly, Eramo et al. [2] 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. [10] 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. [24] 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, 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.


URL http://arxiv.org/abs/1601.00751


