Implementing NFV System with OpenStack

Jinlin Chen, Yiren Chen, Shi-Chun Tsai, Yi-Bing Lin

National Chiao Tung University, Hsinchu, Taiwan
{jinlin075, yiren, sctsai, liny}@cs.nctu.edu.tw

***Abstract*—Network Function Virtualization (NFV) is a promising approach for service providers to reduce CAPEX and OPEX and has drawn a lot of attention from vendors. Service Function Chaining (SFC) provides the ability to define a sequence of network services. This paper identifies two important issues (placement and scaling) on deploying efficiently SFC with a cloud platform. Based on the OpenStack Tacker architecture, we explore possible solutions and implement the missing features in Tacker for SFC, etc. We propose two decision algorithms for service location and policy of scaling. The experiments show the efficiency of our approach compared to the default deployment methods provided in OpenStack. We also discuss and highlight future work to improve utilization of cloud platform for deploying service function chaining.**

Keywords: Service Function Chaining; Software Defined Networking; OpenStack; Cloud Computing

I. Introduction

Network Function Virtualization (NFV) is one of the hottest technologies in recent years. Network virtualization is used to separate network functions from the designated hardware. Besides operating on a variety of standard platforms, it can be deployed dynamically on demand [1]. Hence hardware utilization is increased to reduce capital expenditures (CAPEX). Hardware devices may be spread through service over different locations. For the service providers, it is an important issue how to deploy the virtual network functions to provide efficient service function chaining (SFC) [2].

With recent advancement in virtualization technology, it has drawn a lot of interests in implementing NFV with cloud platform (such as OpenStack) and Software Defined Network (SDN). SDN separates the control plane and the data plane. Usually, there is one SDN controller in the control plane, and the data plane consists of many network devices. Programmers can implement network functions by developing their applications and embed them into the SDN controller to control the packet forwarding in the data plane. Network devices use specific protocols such as OpenFlow [3] to communicate with the controller via the control plane. Network devices on the data plane simply forward packets according to the flow entries defined in their flow tables managed by the SDN controller. In a traditional network environment, the network administrator has to manage each network device and arrange routing rules one by one. On the other hand, SDN devices can be managed automatically by developing applications for the SDN controller.

This work was supported in parts by the Ministry of Science and Technology of Taiwan under contract MOST 104-2622-8-009-001 and MOST 104-2221-E-009 -046.

There are several open source projects for NFV development, including OPNFV [4], OPEN SOURCE MANO [5], etc. However, these developments are still in the early stages. We adopt OpenStack Tacker [6], which is an official OpenStack project building a Generic VNF Manager (VNFM) and a NFV Orchestrator (NFVO) to manage Network Services and VNFs on an NFV infrastructure platform. Its eventual goal is to implement NFV and SFC by adopting ESTI MANO architectural frameworks[7]. Since its announcement in 2015, Tacker has shown great promise in realizing the MANO framework.

Although palatable, several hurdles remain and need to be cleared. First, the current Tacker architecture is still in a primitive status. Many functions mentioned in the proposal are unavailable when we developed our systems. Second, it is unclear how the resources are allocated and orchestrated. Since the OpenStack project was proposed before MANO architecture, the strategy of assigning VMs is not designed specifically for deploying service chains. To develop an efficient SFC based on open sources, we need to understand the details of related mechanisms. In certain occasions, it needs to modify the original designs of the open platforms.

In this paper, we explore a solution by using OpenStack Tacker as our platform to implement service function chaining. The goal of this work is to identify important issues while developing NFV and SFC with cloud platform. Besides implementing service function chaining mechanism, we add extra features and slightly modify the original architecture of OpenStack Tacker. Thus we can design, implement and experiment our system and algorithms for service placement and scaling. The rest of the paper is organized as follows. In Section II , we describe our experiment environment and overall architecture. In Section III, we show the mechanism of our service function chain. The details of the algorithms will be discussed Section IV. The experimental results are shown in Section V. Finally, we conclude with Section VI.

II. Architecture

Figure 1 shows a modified Tacker system architecture, which consists of 4 modules: VNFM (VNF Manager), VNFD (VNF Descriptor), Deployment and SFC (Service Function Chaining). The NFV system is implemented by using services

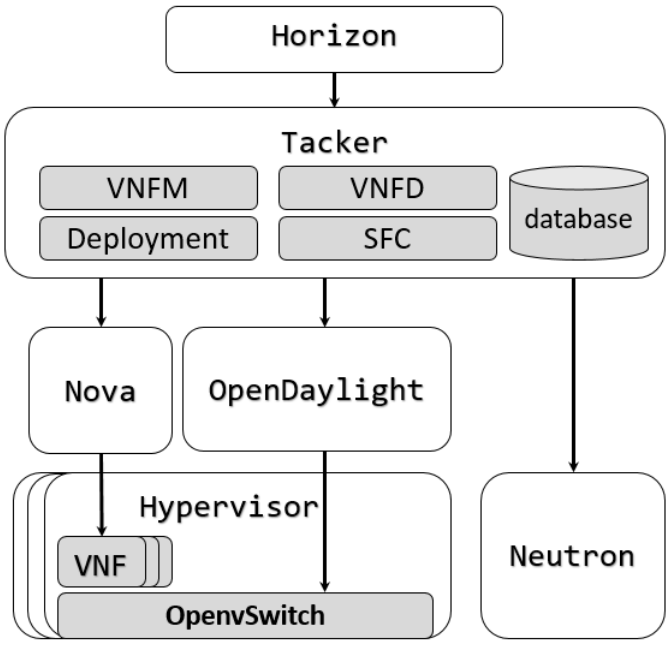

Fig. 1. A modified Tacker Architecture

from OpenStack Nova, Neutron and Horizon, and OpenDaylight Controller [8]. There are 4 steps for deploying VNFs: (1) Update VNFD, (2) Deploy VNF, (3) Update VNF Config and (4) Auto Scaling.

We first design a webpage with OpenStack Horizon for uploading VNFD to the database of Tacker. Tacker adopts TOSCA [9], a descriptive language, to define the attributes of virtualized network functions. A file with TOSCA format is called a virtual network function descriptor (VNFD). Such files are processed by VNFM, and are stored in the database. Besides the standard attributes of VNF (i.e., network, image and service type), we extend the definition of VNFD by including the deployment information of SFC, such that we can describe the chaining order of VNFs in a single VNFD file.

Through the Horizon webpage, we can select a VNFD, with which Tacker will deploy the VNF accordingly by our Deployment module. The original design of Tacker deploys VNF with OpenStack Heat [10]. Instead, we apply directly the APIs of OpenStack Nova to obtain more flexibility for placing and scaling VNFs. Therefore, we modify some of the original codes in Tacker, such that Tacker can manage VNFs with our algorithms through Nova API. Nova is in charge of maintaining the virtual machines of OpenStack. With Advanced Message Queuing Protocol (AMQP), Nova controls Hypervisor to create or delete virtual machines. Once each VM for VNF is deployed, the module for SFC will link the VMs in the order described in VNFD. The SFC module connects the VNFs in a predefined order via OpenStack Neutron and OpenDaylight Controller. We address the details in Section III.

After the SFC module is executed, we need to install the service of each VNF on the VM, such as setting the policy of fire wall. We design a Tacker agent for each VNF. The agent communicates with Tacker via AMQP. The installation depends on the type of service. Since our system provides auto scaling, the number of VNFs for a service may change from time to time. If the file of a service is large, it may cause problems: for example, the files become inconsistent in different VNFs and it may waste disk space. To resolve these problems, we design a synchronizing mechanism with Network File System (NFS) for managing large VNF files. In this case Tacker agent will mount on the NFS Server for synchronization and disk space saving.

While the services are functioning, Tacker agent will regularly report the VNF status including the usage of CPU and memory. VNFM will determine when to scale in or out VNFs. Our experiments use VNFs for Firewall, Loadbalance and Web server to verify and measure the efficiency of our algorithms.

III. Service Function Chaining

We implement Service Function Chaining with OpenDaylight Controller. After OpenStack Tacker deploys the VNFs, the SFC module is in charge of linking the VNFs in a specific order as defined in VNFD. SFC module acquires information of network topology and status from OpenStack Neutron and decides the flow rules to be installed. With RESTful API, SFC module posts flow rules on OpenDaylight. Accordingly, OpenDaylight will set up the flow rules on the OpenvSwitches under its control to route the packets of each VNF properly and complete the functionality of a service chain. Some VNFs, such as load balancer, are capable of directing the packet flow and they do not need flow rules. For this type of VNFs, we need to set up proper attributes to direct the packets to the next VNFs. While deploying a service chain, we can install the flow rules at the same time, since the VNFs need some initialization before functioning.

Once the chain has started to serve, redirecting may cause some problems, such as when scaling in and out. More specifically, when scaling out, we need to make sure if the VNFs have been initialized properly before we can direct packets to the newly added VNFs. In our implementation, the Tacker agent will notify Tacker when VNFs have been initialized, and then Tacker will install corresponding flow rules. Similarly, for scaling in, we cannot remove targeted VNFs right the way. We need to first remove the corresponding flow rules and wait until the VNF has finished earlier requests. Once the requests to the VNF have been completed, the Tacker agent will notify Tacker to remove the corresponding VNF.

IV. Algorithms

We deign two algorithms for placement and scaling, in order to provide an efficient and stable service. Besides reducing the potential network bottleneck and overall delay, we can rapidly deploy extra VNFs when the request surges.

A. Placement

The placement problem can be addressed as a load-balancing problem [11]: there are N items (each of a given weight p_i), which are to be assigned to M machines; the goal is to assign each item to one machine so to minimize the total traffic among the machines. In our case each item is a chain, which consists of several VNFs. Therefore, our item may be assigned to more than one machine, if necessary.

Algorithm 1: Placement algorithm

Input : Complete Graph $G(V, E)$, n:
V is the set of compute nodes and the network node;
n is the number of VNFs in a chain
Output: Updated Graph $G(V, E)$

```
root ← G.root
while n > 0 do
    u ← null /* u is the current best candidate */
    Weight(root, u) ← ∞
    for for each neighbor v of root do
        if v and u have the same capacity then
            if Weight(root, v) < Weight(root, u) then
                u ← v
            end
        end
        else if v has a larger capacity than u then
            u ← v
    end
    if u has empty capacity then
        return null
    end
    Increase Weight(root, u) by 1
    n ← n− (the number of VNFs hosted at u)
    root ← u
end
return G(V, E)
```

A server that hosts the VNFs in OpenStack is called a Compute Node, which is equipped with an OpenvSwitch and communicates with other nodes via tunnels (e.g. GRE, Vxlan, etc). The Network Node of OpenStack is in charge of virtual network service, which is equipped with an OpenvSwitch and communicates with other nodes via tunnels. The network functions of VMs are handled by the Network Node. Our placement algorithm finds suitable Computes Nodes for VNFs in order to reduce the delay between two Compute Nodes. Berger [12] has observed that tunneling between nodes needs non-trivial computing from CPU. Thus placing a service chain in a single node whenever possible will save the CPU cost and thus the latency.

Algorithm 1 shows the details of the greedy placement approach. The inputs include two parts: a graph $G(V, E)$ indicating the logical topology of the Network Node and Compute Nodes; and an integer n indicating the number of VNFs to be deployed. The root of the graph is the Network Node, since when accessing the VNFs it first needs the Network Node to translate the address. $G(V, E)$ is actually treated as an undirected complete graph, where each pair of Nodes connect to each other. Each edge has a weight indicating the number of chains passing by. Initially, the weight is 0. Each edge weight is increased by 1, whenever there is a new chain using this edge. The larger the the edge weight, the more service chains pass through. While deploying the VNFs, we choose the edges with the smaller weights. The output of the algorithm is the graph with updated edge weights.

The algorithm conducts a greedy search starting at the Network Node ($root$) and looking for a Compute Node (u) with the smallest edge weight between them and the largest remaining capacity to accommodate the VNFs. The capacity of a Computer Node indicates the number of VNF that can be hosted on it.

If a single Compute Node is sufficient to host the VNFs, then we are done; else the algorithm continues the search until it finds enough Nodes for all the VNFs. With the edge weights, we can balance the load over the Compute Nodes. According to our experiments, it shows this approach can improve the efficiency up to 50% compared to the default method provided by OpenStack Heat. Finally, when a chain ends its service, we need to remove and recycle the VNFs and update the edge weights of $G(V, E)$.

B. Scaling

Algorithm 2: Scaling algorithm

Input : Two non-negative integers m and n:
n is the number of VNF for a single service in the chain;
m is the number of busy VNF
Output: An integer x: x is the number of VNF deployed or terminated

```
load factor ← m/n
if load factor = 1 then
    if there are enough resource to host another n VNFs then
        return n
    end
else if the resource of the system is sufficient then
    if n > 4 then
        if load factor < 1/4 then
            return -n/2
        end
    end
else if the resource of the system is not sufficient then
    if n > 2 then
        if load factor < 1/2 then
            return -1
        end
    end
return 0
```

The scaling algorithm adjusts the deployment of VNFs. When the request increases, the algorithm adds new VNFs to

the chain to mitigate the load. On the other hand, the algorithm will reduce the VNFs when load is under a threshold.

Algorithm 2 shows the details of the scaling algorithm, which takes two integer parameters: m and n, where m indicates the number of busy VNFs and n is the number of total VNFs of a single function and m/n defines the load factor of the function. When the latency of a service increases, we say the VM hosting VNF becomes busy, however it will depend on the property of the VNF. For a Web Server, if its CPU rate reaches 90%, then it leads to longer latency and it is busy for sure. Based on the load factor the scaling algorithm decides the scaling policy and the number of VFNs to be adjusted. Then our system use Nova API to make the adjustment.

More specifically, the algorithm has two stages: it first determines if the service is overloaded or not. Overloading means that each VNF of a function in a chain is busy. If overloading and the server has sufficient capacity, then we double the number of VNFs. It takes some time to deploy the VNFs. If current request is not satisfied, new request will recur. Also we observe that when the CPU has sufficient cores, it takes about the same time to launch multiple VMs as for a single VM. Even though our strategy may oversupply VMs, it takes only little time to release the excess VNFs.

If the service is not overloaded, then the algorithm determines if the VNFs are oversupplied. We implement a function *isSufficient()* to check if the physical machines have enough resource or not. If the resource is sufficient and the load factor is smaller than 1/4 with $n > 4$, then the algorithm will reduce the size by half and this will maintain the load factor around 1/2. The requirement of $n > 4$ is to avoid frequently increasing and decreasing the number of VNFs. When the resource is insufficient, the algorithm will try to release a redundant VNF for the other chains in need. In this case, if the load factor is smaller than 1/2 and $n > 2$, then the algorithm will reduce the number of VNFs by 1. Similarly, the requirement of $n > 2$ is to avoid unnecessary excessive increasing and decreasing of VNFs. The performance of the scaling algorithm can be justified with amortized analysis as follows.

Theorem 1. *The amortized cost for each operation for a sequence of N scaling out operations is $O(1)$.*

Proof. Scaling out a service in a chain depends on the load factor m/n. Let S be the VNF set of a specific service and $S.n$ and $S.m$ indicates the number of VNFs in S and the number of busy VNFs, respectively. Define a potential function $\Phi(S) = 2S.m - S.n$. Immediately after doubling the size of S, we have $S.m = S.n/2$ and $\Phi(S) = 0$. Immediately before doubling, $S.m = S.n$, and thus $\Phi(S) = S.m$. For scaling out only, we always have $S.m \geq S.n/2$. For the cost of the i-th scaling out, let m_i be the number of busy VNFs and n_i be the number of VNFs in S, respectively, and Φ_i be the potential after the i-th scaling out. Let $\hat{c}_i$ and c_i be the amortized cost and actual cost, respectively after the i-th scaling out. Initially, $m_0 = n_0 = \Phi_0 = 0$.

By the potential method, we have $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$. If the i-th scaling does not double S, then $n_i = n_{i-1}$ and m_i increases at most 1. Thus $\hat{c}_i \leq 3$ is bounded by a constant. If the doubling is executed, then $n_i = 2n_{i-1}$ and $n_{i-1} = m_{i-1} = m_i - 1$. So $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = m_i + (2m_i - n_i) - (2m_{i-1} - n_{i-1}) = 3$. This completes the proof. □

The following is similar to the argument for dynamic table [13].

Theorem 2. *The amortized cost for each operation for a sequence of N scaling out and scaling in operations is $O(1)$.*

Proof. Similarly, let S be the VNFs of a specific service. Here we define the potential function as follows:

$$\Phi_i(S) = \begin{cases} 2m_i - n_i, \text{if load factor} \geq 1/2, \\ n_i/2 - m_i, \text{if load factor} < 1/2. \end{cases}$$

With a similar argument as above, we can show that the amortized cost is bounded by a constant. □

V. Experiments

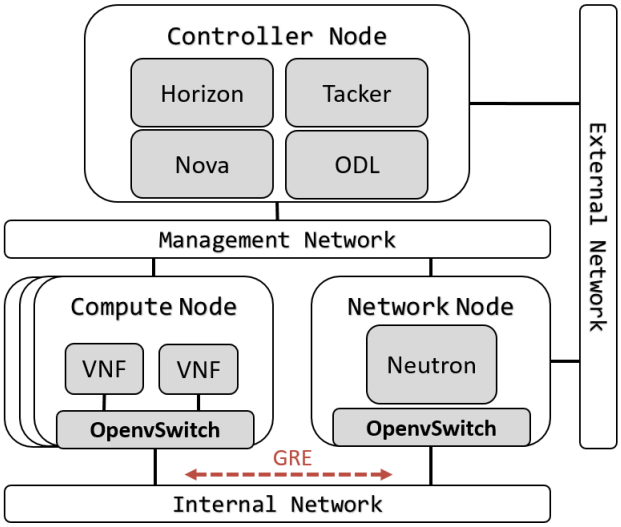

Fig. 2. Experiment Architecture

As shown in Figure 2, we use VMWare workstation[14] to set up our experiment environment, which consists of 3 Compute Nodes and 3 networks. The functionality of each component is illustrated as follows:

1) The Controller Node is used for the control plane of OpenStack, which consists of services for identity, dashboard, network server. Tacker and SDN controller are also deployed on this node.
2) Network Node takes care of the network of OpenStack. With Neutron API, users can set up various of virtual network environments. Through OpenvSwitch, VMs in Compute Nodes can connect to public network via the External Network.
3) The hypervisor is run on the Compute Node to receive request from Controller Node to launch VMs. The virtual NIC of VM is connected to the OpenvSwitch of

Compute Node, which has the flow rules to direct the packets to the intended destination. We have 3 Compute Nodes in our experiments.

4) The Networks are built with virtual bridges, installed with VMware Workstation, which consists of 3 parts for external network, management network and internal network. The external network connects to public network. The management network handles the exchange of management information of OpenStack [15]. The internal network is used by VMs for communication. We also install GRE tunnels among OpenvSwitches of Compute Nodes and Network Node.

For Controller Node and Network Node, we each allocate 1 CPU core and 4GB RAM. For each Compute Node, we allocate 4 CPU cores and 8GM RAM. We create VNFs for firewall, load balance and web server. Each VNF has 512MB RAM and 1 virtual CPU. Since the endpoint of our service chain is web server, we use ApacheBench (ab) [16] to evaluate our experiments. ApacheBench is specifically used to test the performance of various Web Servers, and it provides several parameters to tune the system for experiments. Since ApacheBench also provides the service of Web Server and thus it can measure the performance of a chain that includes Web Server. Therefore, it is a very handy tool for evaluating our placement and scaling algorithms.

A. Placement experiment

We compare the efficiency of our placement policy with the default deploying method of Heat. We test two scenarios with each service chain consisting of firewall, load balancer and web server. We use ApacheBench to measure the performance of the web server. Along the service chain, we also measure the latency of each service. In this experiment, we did not activate the scaling algorithm.

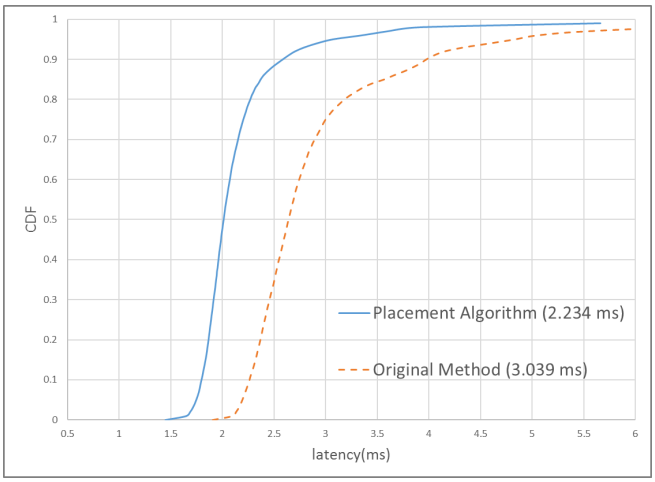

Fig. 3. Latency with placement and without placement algorithm (1 request each time)

Figure 3 illustrates the cumulative probability of the first scenario, where we establish a service chain and repeatedly send a request to it for a million times. We calculate the average latency, which is 3.039ms for the default method and 2.234 ms with our placement algorithm. It improves about 26.5%.

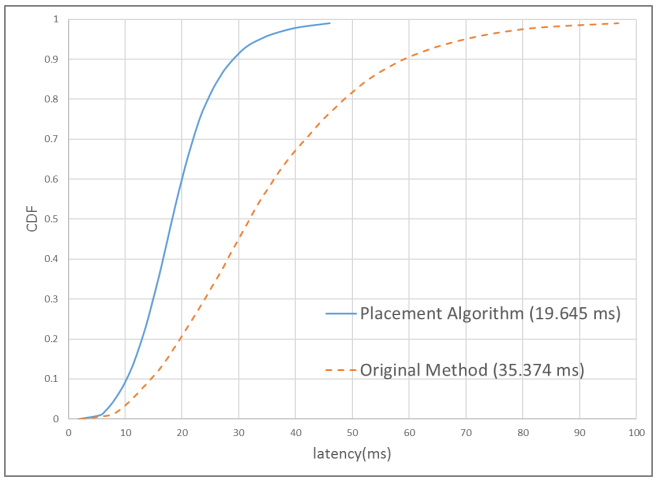

Fig. 4. Latency with placement and without placement algorithm(50 requests each time)

When the number of requests increases, network may congest easily. Our performance improvement becomes more significant. Figure 4 illustrates the result of the second scenario, where we set up 4 service chains and repeatedly send 50 requests to each chain for a million times. The default method has average latency 35.374ms. While the average latency with our placement algorithm is 19.645ms. It is an improvement of 44.5%. For the worst case, it drops from 96.76ms to 46.012ms. It shows that our placement algorithm saves a lot of computation for GRE tunnel encapsulation/decapsulation and thus improves the performance.

B. Scaling experiment

There are two parts for this experiment: one for scaling out and the other for scaling in. For comparison, we design a naive scaling method, which scales out VNFs one by one until the service chain is not overloaded. In this experiment, we deploy a service chain, which consists of one load balancer and one web server. During the experiment, the system will detect if it needs to scale the web server. Initially, ApacheBench repeatedly sends requests for one second and measures the average latency. Then it increases a large number of requests to the web server at the 10th second and 50th second, respectively.

As shown in Figure 5, both methods can add new VNF in 30 seconds and reduce the latency. The first time when the request increases, both methods increase one VNF and there is no significant difference. In the second increase of the requests, our scaling algorithm doubles the number of VNFs, and the latency drops in a much shorter time. While the naive scaling method takes about twice as much time to reduce the latency to the same level. From the figure, we see that at the 80th second, both approaches reduce the latency with the same efficiency. The naive method takes about 30 second to launch

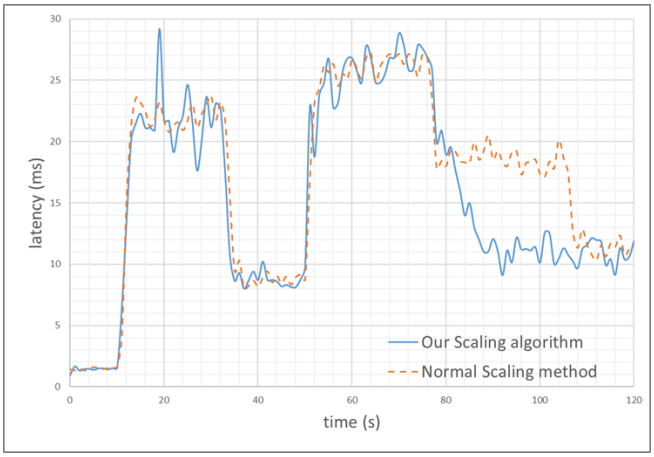

Fig. 5. Latency with scaling algorithm and normal scaling method

another VNF and the latency will be reduced to the level as with our scaling algorithm at the 110th second.

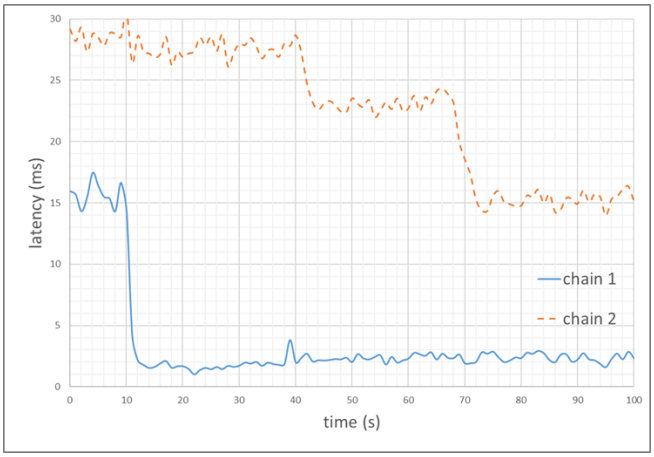

Fig. 6. Latency with two chains of VNFs

For scaling in, we test by releasing resource from a non-busy service chain and creating VNFs for an overloaded chain. As shown in Figure 6, we have 2 service chains and both provide load balancer and web servers. The first chain has 5 web servers and is stable, While chain 2 has only one web server and is overloaded. For the moment there is no extra resource to create new VNF and the second chain is overloaded. At the 10th second, we clean up the requests of chain 1 and scale in by releasing a VNF, such that chain 2 can reuse it. At the 42nd second, chain 2 launches another web server to mitigate the load. It takes about 32 seconds, which is about 4-5 seconds more than scaling out. The extra time is caused by the scaling in of chain 1. At the 70th second, it takes about 28 seconds to launch the second web server for chain 2. This is faster because of the fact that chain 1 has already released the required resource for chain 2.

From the above experiments, we observe that it takes about 25s to add a new VNF, while it takes only 5s to remove a VNF. It means removing VNF is much faster than creating one. By taking advantage of this observation, we can always scale out more VNFs and remove the excess one if necessary. This will help speed up the scaling out.

VI. Conclusion

In this paper, we show how to implement the mechanism for service function chaining with OpenStack Tacker and OpenDaylight. To complete the functionality, we implement the missing features as claimed in Tacker. We also design algorithms for service placement and scaling in/out. Compared with the straightforward methods, our algorithms reduce the latency significantly. For scaling out, our algorithm mitigates the load more effectively. The experimental results show that our approach can improve the efficiency significantly for service function chaining. Our results provide valuable experiences for delivering VNF solution with cloud platforms, especially OpenStack.

To have a complete VNF system, mechanism for efficient and auto live migration will be necessary. It will depend on the migration mechanism of cloud platform. We leave it as a future work.

References

[1] "Network functions industry specification group. "network function virtualization (nfv): An introduction, benefits, enablers, challenges and call for action"," p. 1–16, in SDN and OpenFlow World Congress, 2012.
[2] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary, "Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions.," in *IM* (R. Badonnel, J. Xiao, S. Ata, F. D. Turck, V. Groza, and C. R. P. dos Santos, eds.), pp. 98–106, IEEE, 2015.
[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.
[4] "Opnfv: Open platform for nfv." https://www.opnfv.org/.
[5] "Open source mano." https://www.osm.etsi.org/.
[6] "Openstack tacker." https://wiki.openstack.org/wiki/Tacker.
[7] E. G. MAN001, "Management and orchestration (mano)," 2014.
[8] "OpenDaylight." https://www.opendaylight.org/.
[9] "Tosca template for nfv." http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html.
[10] "Openstack heat." https://wiki.openstack.org/wiki/Heat.
[11] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. Cambridge University Press, 2011.
[12] T. Berger, "Analysis of Current VPN Technologies.," in *ARES*, pp. 108–115, IEEE Computer Society, 2006.
[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 3rd ed., 2009.
[14] "Vmware." http://www.vmware.com.
[15] "Openstack." https://wiki.openstack.org/wiki/Main_Page.
[16] "Apache apachebench." https://httpd.apache.org/docs/2.4/programs/ab.html.