An Overview of A Load Balancer Architecture for VNF chains Horizontal Scaling

Jiefei Ma, Imperial College, UK
Windhya Rankothge, SLIIT, Sri Lanka
Christian Makaya, IBM, US
Mariceli Morales, UPF, Spain
Franck Le, IBM, US
Jorge Lobo, ICREA & UPF, Spain

***Abstract*—We present an architectural design and a reference implementation for horizontal scaling of virtual network function chains. Our solution does not require any changes to network functions and is able to handle stateful network functions for which states may depend on both directions of the traffic. We use connection-aware traffic load balancers based on hashing function to maintain mappings between connections and the dynamically changing network function chains. Our references implementation uses OpenFlow switches to route traffic to the assigned network function instances according to the load balancer decisions. We conducted extensive simulations to test the feasibility of the architecture and evaluate the performance of our implementation.**

I. Introduction

In this paper we propose a generic framework to horizontally scale virtual network functions (VNFs). The architecture relies on two load balancers: a *master* to handle traffic from the source to the destination of a connection, and a *slave* to handle the traffic in the reverse direction. These two load balancers rely on hashing functions to map connections to network function (NF) chains, and ensure that both outgoing and incoming packets of each connection traverse the same sequence of NFs instances. In addition, we introduce algorithms for the load balancers to simultaneously add and/or delete NFs chains, and evenly distribute the traffic among the NFs chains especially considering that connections may be of different sizes and change characteristics over time.

We built a prototype with a mix of virtual and physical components to run our experiments. To forward the traffic to the assigned NF instances, the load balancers add an identifier to each packet. The identifier is used by the switches for packet forwarding. In our implementation, we rely on OpenFlow switches for the data plane and use VLAN tags as identifiers. Our prototype relies on the open source PF_RING network socket [1] to capture incoming packets and add the VLAN tags. The throughput of the cards under this configuration was around 465 Mbps while the load balancer throughput was about 366 Mbps. The results show that it takes less than 3 seconds for the traffic to be rebalanced when adding or removing a NF function instance. The experiments were run using standard installations of Snort and Bro intrusion detection systems [2], [3], but we will report only Snort experiments.

The rest of the paper is organized as follows. The next section describes an operational scenario with the load balancers. Section III describes our testbed. The evaluation and experimental results presented in Section IV. Related work is discussed in Section V. Final remarks are presented in Section VI.

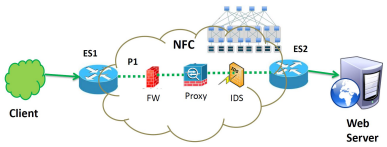

Fig. 1: Original deployment of NF chain $P1$

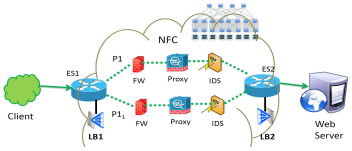

Fig. 2: Load balancers deployment

II. Operational Scenario

To illustrate the problem, we consider the following scenario (Fig. 1): the network administrator of a web service wants the web traffic to be processed by a VNFs chain hosted in a Network Function Cloud Center (NFC). The chain consists of a firewall, a proxy, and an intrusion detection system (IDS). Horizontal scaling will replicate the chain as illustrated in Fig. 2 with two load balancers, $LB1$ and $LB2$ to distribute the traffic going to the server and returning to the client.

The load balancers are new VNFs dedicated to the management functionalities of the NFC and they do not need to be deployed at the edges. In fact, load balancers can be deployed surrounding only the portion of the chain that needs to scale. It is also possible to deploy chained pairs of load balancers if there are differences in the scaling needs of the different NFs resulting in deployments as the one depicted in Fig. 3.

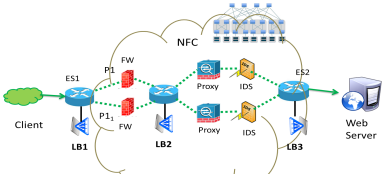

Fig. 3: Load Balancers Chains

978-3-903176-14-0

The management system (MS) of the NFC upon receiving the service request for a new NF chain ([4], [5], [6]), it will allocate and deploy resources as indicated in Figure 1 with the caveat that of adding to the chain the master/slave pair of load balancers as depicted in Figure 2. The routing in $ES1$ will be set such as all the traffic coming from the client to be passed through the chain, $P1$ as well as the traffic coming back out from the chain going back to the client will be bridged to $LB1$. Similar routing settings are done in $ES2$ for $LB2$.

Hence, the MS knows how packets to be sent through $P1$ are identified by $ES1$ and $ES2$ and this must be known by the load balancers as well. Therefore, the MS makes an asynchronous call to the master with the pair of IDs, one for the master to be used for the traffic coming from the client, and to be routed to the firewall, one for the slave to do the equivalent work with the traffic coming through $ES2$ from the server to be sent to the IDS. The master synchronously calls the slave to pass the IDs and makes sure that both know about the IDs. From this moment on, all packets going from $LB1$ and $LB2$ to $ES1$ and $ES2$ are modified for the switches to route appropriately: Client - $ES1$ - $LB1$ - $ES1$ - $\cdots P1 \cdots$ - $ES2$ - $LB2$ - $ES2$ - Server.

The MS will monitor and analyze the traffic in the NFC and decide when the NF chain must scale, and it will allocate and deploy resources for the new child chain as depicted in Figure 2. When the deployment is finished (the child path yet to be used), the MS will asynchronously call the master with the new pair of IDs, the master will call synchronously the slave to pass the new IDs and they will decide according to their own load balancing policy how to modify the packets before passing them to the switches. The MS can asynchronously request statistics from the master and the master will collect data from the slave and pass the results to the MS about the distribution of traffic among the chains. The MS can also request the master to re-balance the load. The master will coordinate the re-balance process.

To maintain sessions affinity during dynamic re-balancing and changes of NF chains, the load balancers maintain a table of `active_Sessions`. We assume that a session (flow) can be identified and mapped into a session `key` regardless of the direction of the flow. We can use the tuple (source IP, source port, destination IP, destination port) to identify a session in the outgoing flow and the tuple (destination IP, destination port, source IP, source port) to identify the same session in the returning flow. When the network adapter in each load balancer receives a packet `p`, extracts its session `p.key` and passes it to the load balancer. The load balancer returns a chain ID and the network adapter modifies the packet accordingly before putting the packet back into the network. To respect sessions affinity, the load balancers need to implement a consistent session function `map()` that maps any packet `p` of a give session to the same ID. Hence, in addition to having active sessions, the `active_Sessions` table also stores the IDs assigned to the sessions. Algorithm 1 describes a general schema of how this computation can be done.

The decision of removing a chain instance is also made by the MS. Similar to adding a new path, the MS will call asynchronously the master with the pair of IDs assigned to the path the MS wants to remove. The master synchronously will call the slave to communicate the IDs and a cool-down process will start in both the master and the slave. They will stop sending new sessions to the path being removed but they should keep using the path for active sessions already assigned to the path. The MS polls the master and the slave and ask whether the path is active (i.e., if there are still active sessions assigned to the path). When both, the master and the slave, respond that the path is not active the MS can re-use the resources of the chain for other tasks. The master and slave must access the `active_Sessions` table to decide if a path is still active avoiding race conditions affecting updates to this table. Note that the re-balancing that will affect the behavior of `get_newID` when adding or removing instances, or at any time that the MS requests it, can be done concurrently since it does not read the `active_Sessions` table. Session management overrides balancing. The challenge now is how to make sure that IDs will be assigned consistently in both the master and the slave, differentiating each session. This is achieved using a consistent hashing function which is described in detailed in the full version of the paper [7].

Algorithm 1 Session-aware `map(p:packet)`

```
if p.key is an active session then
    Update_session(active_Sessions[p.key])
    RETURN active_Sessions[p.key].id
else
    instanceID = get_newID(p.key)
    Activate_session(active_Sessions, p.key,
    instanceId)
    RETURN instanceID
end if
```

III. Reference Implementation

To test our proposal we needed a reference implementation of a full operational testbed. We had to decide how to implement two components of the architecture. We needed a: (1) network where we could reroute traffic dynamically and (2) a concrete implementation of a network adapter. For (1), we have used OpenFlow [8]. For (2), the load balancer will take a session to be all the packets with the same set of pairs (source IP:port, destination IP:port) in the headers that pass through the load balancer, for which any two consecutive packets with the same set are not separated by more than a fixed time window τ. The architecture of the deployment is depicted in Fig. 4. We have three physical machines: a PowerEdge R430 Dell Server with 12 cores, 32 Gb of memory and four 1 Gb Ethernet cards, and two ALDA+ PCs both having a dual core 6320 Intel CPU and 4 Gb of memory. The PCs have three network cards two of which were able to support the open source PF_RING network sockets [1].

In Fig. 4, the cloud inside the Dell server represents a Mininet deployment. It has 5 switches and 6 servers. The servers named n_i will run NFs. All the switches, the n_i

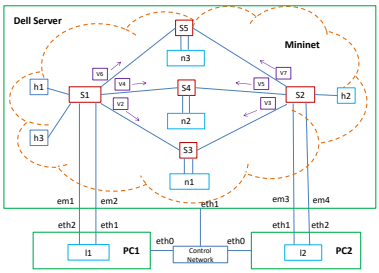

Fig. 4: Testbed Architecture

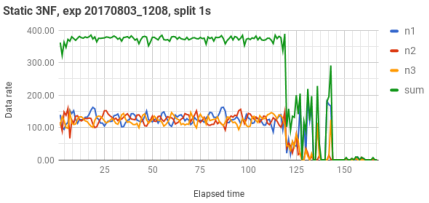

Fig. 5: Throughput with static number of NFs

servers and PC1 and PC2 are assumed to be under the NFC management. The h_i servers are assumed to be outside the NFC and will be sending and receiving traffic. Two ports of Switches s_1 and s_2 are directly associated to the physical cards of the Dell server to send and receive traffic from the load balancers running on the PCs. The third network cards in PC1 and PC2 and an USB network interface in the Dell server (eth1) are used for the control traffic of the MS. The load balancer l_1 running in PC1 will act as a master of the slave load balancer l_2 running in PC2. Server h_2 will be running a web server and clients will connect from h_1 and h_3.

The NFC management uses a unique pair of VLAN tags for each NF chain and it will install OpenFlow rules in the appropriate switches to route the traffic inside the NFC [4], [9]. In our testbed, we will use three chains with a single NF in it: one chain will use the pair of VLAN tags $(2, 3)$, the second chain will use the pair $(4, 5)$ and the last chain will use the pair $(5, 6)$.

When a master is brought up, it is informed of the address of its slave through the control plane channel and does a handshake with the slave. At any moment the MS can send pairs of IDs (VLAN tags) to the master and it will pass the information to the slave and they will automatically start using the new IDs for balancing. Similarly, pair of IDs can be removed at any moment through the master. We have implemented a polling mechanism to ask the load balancers what IDs are active since old sessions can linger for some time after removing a pair of tags. The load balancer network adapters take traffic coming from interface eth2, extract the (source IP:port, destination IP:port) pair and gets a ID from the load balancer for this tuple. The network adapter takes that ID and adds it to the packet as a VLAN tag and puts the packet in the network card in eth1.

IV. Experiments

We have conducted two sets of simulations: (1) test the performance of the load balancers independent of the VNFs and (2) test the performance of the load balancers with traffic that was slowdown by the processing of VNFs. Due to the space limits, in this paper, we are reporting the results of the latter. A comprehensive description of the results can be found in the full version of the paper [7].

The first test was a clean throughput test over the testbed. All the Mininet links were set to have bandwidths of 1 Gbps and we set Linux bridges between the interfaces of n_1 as well as between network interface cards eth1 and eth2 in both PC1 and PC2. We also set OpenFlow rules in s_1, s_2 and s_3 so that any traffic between h_1 or h_3 and h_2 will follow the path (s_1, PC1, s_1, s_3, n_1, s_3, s_2, PC2, s_2, h_2). Using iperf and the proper parameters in httperf, we consistently got throughput over 900 Mbps. Next, we replace the Linux bridges in PC1 and PC2 with bridges using the vanilla version of PF_RING and PcapPluplus to parse the packet arriving into eth2 and putting it back into eth1. In this case the average throughput was an average of 465 Mbps.

A. Simulations with Snort

For these experiments we ran Snort in n_1, n_2 and n_3. We configured Snort in in-line mode with simple rules to detect potential SQL injections attacks, and injected URLs that contained SQL statements as they would appear in some SQL injection attacks using curl calls. Alerts were collected in Snort logs to verify that all the injections were detected, and that injection traffic was evenly distributed among the NFs.[1]

We measured the throughput of Snort by running experiments with a single NF and no curl calls. First, we ran experiments using httperf generated traffic consisting of 8000 http connections at a rate of 75 connections per second, each connection fetching a 0.75MB file. The average throughput went down to 162 Mbps. Second, we generated httperf traffic using 3000 connections at a rate of 40 connections per second, transferring a 1 MB file per connection with the same average results. We reduced the number of connections because we were getting to the limits of file descriptors needed to run the simulation in the Dell server. Next, We ran experiments injecting 600 and 1000 URLs every 1, 3, and 5 seconds, reaching up to 80,000 URL injections in some of the runs. We got consistent decrease in throughput (from about 162 to 152 Mbps) when we had more than 5,000 URL injections. The plots looked more jagged than without URL injections – see Fig. 6. We suspect this is caused by the short-lived curl sessions. The intervals to calculate the average values were adjusted to be over [20,80] because the traffic started to decrease earlier than in the previous experiments.

a) Warm-up: The warm-up experiments tested the time it took for the traffic to be balanced after a new NF has been added. We started with one instance of Snort running, then after 35 seconds had elapsed, we sent a pair of tags to the

[1]We also ran simulations with Bro, but given its detection and not prevention nature, processing traffic with Bro did not affect network throughput.

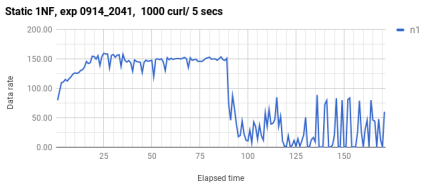

Fig. 6: Snort throughput with 20K SQL injections

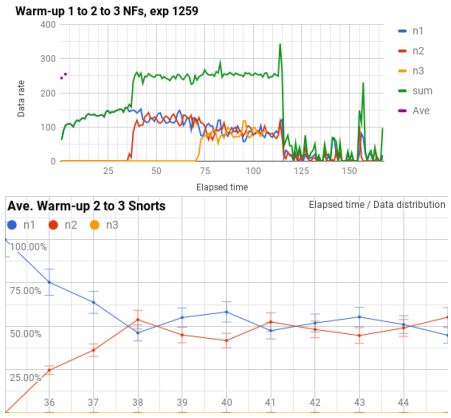

Fig. 7: Dynamic increases of Snort instances

master load balancer to add a second Snort instance, and then, 35 seconds later, a second pair to add a third instance. We generated traffic using httperf, with 8000 connections generated at a rate of 75 connections per second, each connection transferring a 0.75MB file from server to client. This traffic, generated using httperf from host h_1, was accompanied with an average of 10K curl calls generated from h_3 with SQL-like injection attacks. URL injections were limited to 10K to avoid exhausting file descriptors. The Fig. 7 shows the results.

The top graph shows the throughput over time for one of the five experiment runs. We measured the throughput in the intervals [10, 35], [37, 70], and [75, 110] averaging 148, 247, and 277 Mbps respectively. The second graph plots the time it took, in average, for the traffic to re-balance after the master receives the request to bring up the path to the second instance of Snort after 35 seconds have elapsed. The graph zooms-in to the 35 second mark. It takes about 3 second for the traffic to be balanced - 10% error bars are shown in the plot.

b) Cool-down: The cool-down experiments tested the time it took for the traffic to be balanced after a NF has been removed. We ran a set of cool-down simulations starting with 3 instances of Snort running. After 35 seconds had elapsed, we sent a request to the master to remove one of the instances, and 35 seconds later we sent another request to remove a second instance. The traffic load was as with the combined warm-up. Typical results are illustrated in Fig. 8. The average throughput in the intervals [10,35], [40,70], and [75,110] were 270, 244 and 152 Mbps showing a symmetric behavior from the warm-up scenario. The cool-down time was about four seconds - indicated by the moment when the 10% error bars of the Snort instance stop overlapping with error bars of the

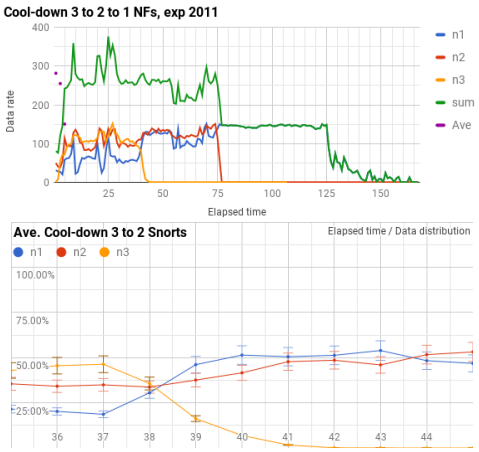

Fig. 8: Dynamic decreases of Snort instances

reminding instances.

V. Related Work

Perhaps the first realistic demonstration of the use of SDN for load balancing NF traffic appeared in [4]. The paper discusses a new controller called SIMPLE to balance the work load across a fixed set of NFs, not for scaling. Most of the existing scaling that handle session affinity depend on state migration techniques: moving the relevant state from one instance to another. Frameworks like Split/Merge [10] and OpenNF [11] facilitate fine-grained transfers of internal NF state to support fast and safe reallocation of flows across NF instances. First drawback of these two approaches is that NF code must be modified prior to support such export/import operations. Second is, that they require a large number of rule sets in the switches and it is not clear whether the methods as presented will scale. E2, described in [12], is a VNFs scheduling framework that supports traffic affinity based on NF placement and dynamic scaling while trying to minimize traffic across switches. It introduces a high-performance and flexible data plane where Click [13] modules (i.e., classifier and TCP re-constructor, etc.) are embedded to accelerate packet processing. However, [12] does not describe how bi-directional flow sessions are handled.

VI. Final Remarks

We have presented a load balancing methodology for the management of horizontal scaling of NF chains that does not require changes to the NF code. We also developed a prototype reference implementations to illustrate the feasibility of the proposed solution and conducted extensive simulations to assess the performance. Our prototype relies on the open source PF_RING network socket [1] to capture incoming packets and add the VLAN tags. The throughput of the cards under this configuration was around 465 Mbps while the load balancer throughput was about 366 Mbps. The results show that it takes less than 3 seconds for the traffic to be rebalanced when adding or removing a NF function instance.

REFERENCES

[1] L. Deri *et al.*, "Improving passive packet capture: Beyond device polling," in *Proceedings of SANE*, vol. 2004. Amsterdam, Netherlands, 2004, pp. 85–93.

[2] C. Gerg and K. J. Cox, "Managing security with snort and ids tools," 2004.

[3] V. Paxson, "Bro: a System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999. [Online]. Available: http://www.icir.org/vern/papers/bro-CN99.pdf

[4] Z. Qazi, C. Tu, L. Chiang, and at el, "Simple-fying middlebox policy enforcement using sdn," in *ACM SIGCOMM '13*, 2013.

[5] W. Rankothge, F. Le, A. Russo, and J. Lobo, "Optimizing resource allocation for virtualized network functions in a cloud center using genetic algorithms," *IEEE Transactions on Network and Service Management*, vol. 14, no. 2, pp. 343–356, 2017.

[6] W. Rankothge, F. Le, and at el., "Experimental results on the use of genetic algorithms for scaling virtualized network functions," in *IEEE SDN/NFV 2015*.

[7] J. Ma, W. Rankothge, and at el, "A comprehensive study on load balancers for vnf chains horizontal scaling," in *eprint arXiv:submit/2423624*.

[8] "Openflow 1.4 specifications," https://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

[9] W. Rankothge, J. Ma, F. Le, A. Russo, and J. Lobo, "Towards making network function virtualization a cloud computing service," in *IEEE IM 2015*.

[10] S. Rajagopalan, D. Williams, and at el, "A. split/merge: System support for elastic execution in virtual middleboxes," in *USENIX NSDI 2013*.

[11] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 163–174.

[12] P. Shoumik, L. Chang, H. Sangjin, and at el, "E2: a framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[13] J. Martins, M. Ahmed, and at el, "Clickos and the art of network function virtualization," in *NSDI '14*, 2014.

ACKNOWLEDGMENTS

This work was supported by the Spanish Ministry of Economy and Competitiveness under grants MDM-2015-0502 and TIN2016-81032-P.