Lost in Network Address Translation: Lessons from Scaling the World's Simplest Middlebox

Vladimir Olteanu
U. Politehnica of Bucharest

Felipe Huici
NEC Europe Ltd.

Costin Raiciu
U. Politehnica of Bucharest

Abstract

To understand whether the promise of Network Function Virtualization can be accomplished in practice, we set out to create a software version of the simplest middlebox that keeps per flow state: the NAT.

While there is a lot of literature in the wide area of SDN in general and in scaling middleboxes, we find that by aiming to create a NAT good enough to compete with hardware appliances requires a lot more care than we had thought when we started our work. In particular, limitations of OpenFlow switches force us to rethink load balancing in a way that does not involve the centralized controller at all. The result is a solution that can sustain, on six low-end commodity boxes, a throughput of 40Gbps with 64B packets, on par with industrial offerings but at a third of the cost.

To reach this performance, we designed and implemented our NAT from scratch to be migration friendly and optimized for common cases (inbound traffic, many mappings). Our experience shows that OpenFlow-based load balancing is very limited in the context of NATs (and by relation NFV), and that scalability can only be ensured by keeping the controller out of the data plane.

1. INTRODUCTION AND MOTIVATION

Hardware middleboxes are as numerous as switches and routers in enterprise networks [19] and can handle packet processing speeds in the tens of millions. However, they come with steep prices and are very difficult to scale or upgrade: such operations require hardware purchases. Running network processing on commodity hardware is the obvious solution: software processing can sustain reasonable packet-level speeds for simple processing by bypassing the network stack [9, 18], and its biggest selling point is the ease with which software can be scaled or upgraded. Scalability in particular is crucial, as it allows the network to dynamically dimension its resources in response to load, leading to energy savings and smaller up-front costs. Running middleboxes on commodity hardware has been termed Network Function Virtualisation (NFV) and is more than just hype: all of the major network operators have gotten together to specify an architecture for Network Function Virtualization [20] and to allow chaining different functions on the same traffic flow [10].

HotMiddlebox '15 August 21, 2015, London, United Kingdom
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3540-9/15/08...$15.00
DOI: http://dx.doi.org/10.1145/2785989.2785994

There is already a growing body of research on how we should approach NFV: the basic recipe is to use hardware load balancing (e.g. OpenFlow) to split traffic to a number of commodity servers, as proposed by Flowstream [7]. To implement middlebox functionality, the simplest choice is to use existing apps running over Linux; however major gains can be made in both performance and ease of deployment if we restrict the programming language for middleboxes as proposed by ClickOS [12] and FlowOS [3].

One basic question, though, remains unanswered: *can we implement scalable network functions on commodity hardware that achieve packet rates similar to hardware appliances?* In this paper we set out to answer this question for network address translators, or NATs. Our goal is to build a scalable software-based NAT with comparable performance to hardware middleboxes.

We are acutely aware that NATs are far from exciting. However they also are the simplest and most popular form of middlebox that maintains per-flow state. NAT functionality is embedded in almost all network appliances including routers, application security appliances, service gateways in cellular networks, etc. Scaling a NAT is therefore the lowest-common denominator in scaling more complex appliances: if we can't build a fast, scalable NAT, there is little hope of building more complex functionality that performs well. Finally, the depletion of the IPv4 address space means these boxes are in high demand, as more network operators are contemplating deploying carrier-grade NATs.

Designing a scalable NAT seems an easy task at first sight, as there is a large body of literature on Software Defined Networking that we can rely upon including [6, 16, 17, 21]. Existing works do not focus on specific network functions or hardware, and thus fail to take into account crucial limitations of OpenFlow switches or NAT requirements (e.g., require that both traffic directions hit the same box). In fact, these restrictions heavily restrict the set of feasible solutions to the point where existing works do not apply.

Instead, we took a clean-slate approach at designing a fast, scalable NAT for our specific OpenFlow switch. We have developed novel data structures that allow fine-grained locking to enable cheap migration, while offering high throughput. Finally, we present **daisy migrate**, a state migration that keeps the centralized controller off the data path.

Our experimental results show it is possible to perform network address translation at 40Gbps with 64B packets on six low-end commodity servers coupled to a 10Gbps OpenFlow switch. Additionally, our NAT can seamlessly scale up and down as load fluctuates, and can be easily extended by adding more servers: e.g., adding 9 more servers would allow us to process 100Gbps with 64B packets. This implies that NFV seems indeed feasible: at around thirty thousand dollars, this software NAT is cheaper than existing CGN appliances[1].

This paper begins by discussing in §2 the design choices we made while building our NAT, most of which are heavily influenced by the hardware limitations of OpenFlow switches. Next, we describe our implementation in §3 and our evaluation in §4. We make a quick suvey of existing literature in §5 and discuss the lessons learnt in §6.

2. DESIGNING A CARRIER-GRADE NAT

Our carrier-grade NAT is presented in Figure 1, and consists of an IBM G8264 10Gbps OpenFlow switch connected to six commodity servers that perform the actual translation, and another server that controls both the switch and the machines. The controller connects to the servers and the switch via a standard Gigabit switch, not shown in the picture. To build a scalable software NAT, we need the following building blocks:

1. A fast NAT implementation for a single machine. Although arbitrary performance can be achieved by scaling out (at least in principle), having a fast NAT implementation is key to ensure the resulting solution is economically feasible.
2. A load balancing algorithm that spreads the traffic over the existing NAT instances.
3. A migration algorithm that allows the controller to add more machines to the NAT when load grows, and turn off machines when load shrinks to save energy / costs.

For the first part, there are plenty of implementations for NATs that we can rely on: Linux supports NAT in its `iptables` suite, and the Click modular router [11] also has configurations for NATs. To decide which ones to choose, we profiled both, finding that native Linux can only handle around 500K packets per second, which is rather low. If we run our NAT in Click [11] in user-mode over netmap [18] we can achieve a little over two million packets per second in the most demanding scenario on a single core (see §4), and close to 7 million packets per box; this is a good basis for our scalable NAT.

The load balancing algorithm is the key ingredient of a scalable middlebox, and we have found that its design heavily depends on the capabilities of the OpenFlow switch. Our IBM switch supports OpenFlow 1.0, which allows prefix matching for IP addresses, and only exact matches or do not cares for the other fields.

A strawman load-balancing algorithm can rely on the **reactive** behaviour of the OpenFlow switch: when seeing a packet from an unknown connection, the packet is forwarded to the controller, which can then select an appropriate NAT

[1]For instance, the A10 Thunder 4430 CGN can process at most 38Gbps of traffic and costs ninety thousand dollars [1].

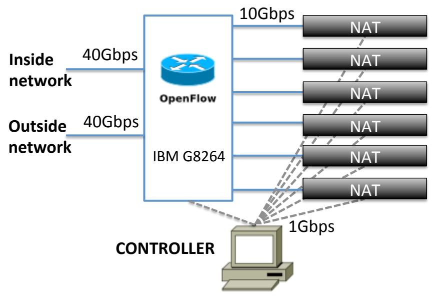

Figure 1: Software Carrier-Grade NAT: an OpenFlow Switch, six commodity servers and a controller.

box (say N_0) to process the packet and install a rule that matches future packets from this connection, rewrites their destination MAC address and forwards them to the appropriate NAT box. N_0 creates a new mapping for the flow and then translates the packet, changing the source address and port of IP_n and P_n, and forwards the packet back to the switch. The new packet belongs to a flow not known by the switch, so this packet will also be sent to the controller. The controller now knows the translation and installs another rule that forwards traffic from the server to N_0.

There are at least three problems with this strawman load-balancing algorithm:

- The OpenFlow switch has to maintain two rules per connection. Our IBM G8264 switch supports around 80 thousand rules, which would imply a maximum of 40 thousand connections, which is very small.
- The control plane of the OpenFlow switch can only process around 200 packets (i.e. at most 100 new connections per second), which is insufficient.
- Even if a switch existed with no scalability limitations, our software controller would become a bottleneck since it sees every new connection twice.

To avoid stressing the control plane of the OpenFlow switch, we must **proactively** insert load-balancing rules into the switch to spread the traffic to the NAT boxes. At first sight, a hash-based load-balancing algorithm would be ideal to spread traffic coming from the inside network to the NAT boxes; indeed, hash-based load balancing is supported in OpenFlow 1.1. Unfortunately, this does not solve the problem of directing outside traffic through the appropriate NAT box: we still need one rule to be actively inserted for each connection's incoming traffic, and we are back to the problems above. In fact, all load balancing solutions that need the exact mapping to work correctly have this problem.

To avoid this problem, our distributed NAT uses multiple external IP addresses, assigning one or more to each individual NAT box. This allows us to route traffic from the Internet to the NAT box with traditional destination-based forwarding and requires one rule per IP instead of one rule per connection. In principle, we could have multiple NAT boxes share the same external IP address and we could partition the port space between these machines using prefix matching on the ports, but OpenFlow does not support this feature.

To load balance traffic from the inside network to the NAT boxes, we rely on prefix matching on the IP source address

```
migrate(IP,dest){
   copy(half_of_the_free_ports, dest);
   start_daisy_chaining();
   copy(remainig_ports, dest);

   for (mapping m in mappings){
      freeze(m);
      copy(get_state(m),dest);
      unfreeze(m);
      send_buffered_packets(dest);
   }
}
```

Figure 2: Daisy-migrate algorithm: pseudocode run by NAT source when migration is executed.

of traffic, assigning each subnet to a fixed external address. This technique ensures the requirements[2] for NAT translation are obeyed [2, 8].

To summarize, the limitations of OpenFlow switches have forced us to assign at least one external IP address to each NAT box. This is also a simplifies the implementation of the NAT: there is no need to synchronize global state (i.e. the free ports) during normal operation. The NATs are completely independent, which makes them more robust and easy to scale.

Scaling NATs. The standard way to scale up and down processing is to use multiple virtual machines that are spread onto many physical machines when load is high, and consolidated onto fewer servers when load drops. We have run our NAT in a Linux VM and migrated the whole VM. Unfortunately, at 4Gbps speeds the resulting downtime from stop-and-copy migration results in a dip in throughput of around 1.5Gbps. Also, the migration takes a few seconds.

To reduce the migration time, we only migrate the NAT state instead of the whole VM, cutting the time needed to copy the VM's memory unrelated to the NAT.

How should NAT state be copied across? VM migration uses a few rounds of pre-copying, where "dirty" memory is transferred to the new NAT until the amount of memory dirtied between rounds plateaus. A stop-and-copy phase follows, where the VM is stopped, the remaining "dirty" memory is copied across, the new VM is started and traffic is shifted to it [4].

Applying the same mechanisms to migrating NATs is suboptimal. First, a large number of connections may be active and their associated state will be altered constantly, which means pre-copy will have limited effect; this is duly confirmed by experiments we ran on the MAWI traces [13]. Simply applying stop-and-copy to a large number of connections simultaneously disrupts traffic massively, resulting in unavoidable packet loss.

Our solution avoids the long stop-and-copy phase by applying stop-and-copy at smaller granularity, migrating connections one at a time or in small groups. To do so, we run the two NATs in parallel while the migration is taking place, which implies we must split the available ports across the two NATs. The pseudocode of our algorithm is given in Figure 2, and it starts by giving half of the available ports to the new NAT. From this point onwards, the new NAT

[2]We are required to use paired IP pooling; in a given session, a client must receive the same external IP address for all its traffic.

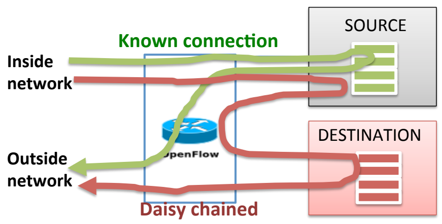

Figure 3: Migrating a NAT using daisy chaining allows fine-grained migration without per-flow rules at the switch.

can create mappings using the same external IP address but different ports.

To avoid the need for per connection rules at the switch, all traffic first hits the first NAT while the migration is taking place, as shown in Figure 3. The first NAT only translates traffic for which it has a mapping; other traffic will be processed by the second NAT, and the first NAT forwards it by rewriting the destination MAC address of the packet. We call this state *daisy chaining*, and it is enabled as soon as the new switch has available ports. From this point on, the old NAT creates no new mappings, so it can send all the available ports to the new NAT.

Next, the algorithm migrates small groups of connections independently. First the connections are frozen: the NAT buffers all packets destined for them. Next, the mappings are copied across. When these mappings are acked by the destination NAT, the source NAT also sends the buffered packets to the new NAT. When all mappings have been transferred across, the migration routine finishes, and the controller updates the OpenFlow entry to forward traffic to the new NAT. Note that the move may reorder a few packets, but this effect does not impact performance in our tests.

3. IMPLEMENTATION

We have implemented a custom Click element that deals with packet translation and state migration and a controller that installs rules and orchestrates migration.

The controller. The controller is based on Trema [14]. Its two responsibilities are directing traffic and overseeing migration. When starting up, the controller starts a single NAT instance and installs all the flow entries needed to direct traffic from the inside network to this instance, as well as return traffic from the Internet. When an administrator decides to migrate some flows (e.g. to scale out), the controller instructs the source and destination Click instances as to the desired outcome and waits for them to finish before assigning the flows to the destination by replacing the relevant OpenFlow rules.

The Click instances. The bulk of the work is performed by Click instances running on commodity x86 machines. We have chosen to run Click in usermode and to use netmap for packet I/O. This enables Click to achieve speeds comparable to running Click in the Linux kernel, while benefitting from the advantages of user-level programming: robustness, easy development.

Our Click configuration uses a custom NAT element that we have implemented from scratch for high performance and migration at the same time. We have decided against using the IPRewriter element from Click because its implemen-

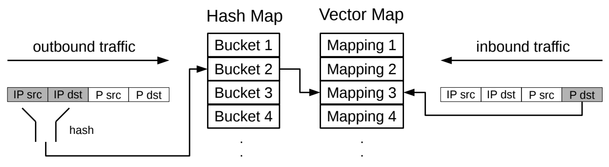

Figure 4: Data structures optimized for NAT and migration.

tation makes it difficult to implement our fine-grained migration algorithm. In particular, IPRewriter uses the Click hashtable data structure, and the only way to ensure safety during migration would be to use a big lock. Writing our element from scratch allows us to optimize it from ground up for both performance and migration.

Packet translation. Every external IP that is managed by a CarrierGradeNAT element has its own data structures that store all associated states. These data structures are further segregated by layer 4 protocol: TCP, UDP or ICMP. For the sake of brevity, our discussion of how packets are translated will be focused on TCP packets within the context of a single external IP, unless otherwise noted. A list of free ports is also maintained, so that external ports are never allocated to multiple internal endpoints at the same time.

All connections that share the same internal IP and port (along with their states) are gathered under the umbrella of a single bidirectional mapping of an internal (IP, port) tuple to an external port and vice-versa. Because mappings are independent of outside hosts' IPs and ports[3], it is possible for multiple TCP connections to share the same mapping. TCP mappings contain data about their corresponding connections. For the purpose of looking up these mappings, two data structures are used (see Figure 4):

- A 2-choice hash map is used to lookup mappings based on the internal (IP, port) tuple. A hashtable lookup is performed for all packets originating from the inside network, and the key is a hash function applied to the packets' source IP and port.
- A vector map is used for looking up mappings based on the external port.

Our hash function implements two-choice hashing, minimizing collisions for outgoing traffic when there are many existent mappings. Packets originating from the outside network are only matched against their destination port, which requires a single array access and is faster than a hashtable lookup. This optimization ensures the more voluminous inbound traffic is processed faster than outbound traffic.

If no packets match a mapping for a certain amount of time, the mapping is deleted and the external port is freed up. UDP and ICMP Query mappings have simple inactivity timers, which are only refreshed when an outbound packet is translated. Inbound packets do not refresh this timer for security reasons; an external attacker would otherwise be able to keep the mapping alive indefinitely.

Each TCP connection associated with a mapping has its own inactivity timer, whose timeout depends on the connection's state. As far as the NAT is concerned, a connection falls into one of three categories: partially open, established or closing (see [8]). A TCP mapping is deleted when all of its connections have expired. The timers are implemented using expiry queues. There is one queue for each timeout value. Expired connections and mappings are culled periodically from the front of each queue.

[3]This is because of endpoint-independent mapping [8].

Experiment	`iptables`	`IPRewriter`	`CGN`
1 conn OUT (empty table)	0.49	2.67	2.6
1 conn OUT (full table)	0.46	2.64	2.3
65K conn OUT	0.4	1.9	2.1
1 conn IN (full table)	0.49	2.89	3.1
65K conn IN	0.39	1.93	2.65

Table 1: Basic throughput comparison on a single machine for different solutions

# configs	1	2	3	4
1 conn OUT	2.5	5	7.4	9.7
65K conn OUT	2	3.9	5.55	6.8

Table 2: Throughput on a single machine when using multiple cores

State migration is performed by a separate thread. Our hashmap implementation features individually-lockable buckets, which allows us to migrate one connection at a time without disrupting other traffic.

4. EVALUATION

We begin our evaluation by analyzing the performance of our distributed NAT (see Figure 1) using a single physical machine. We start a single Click configuration running our NAT and use two other servers (not shown) to emulate the inside and outside networks. In each test, the client or the server generates UDP traffic using the `pktgen` utility provided by the netmap suite[4], and we measure the throughput at the destination, and the results are shown in Table 1.

The performance depends on the number of active mappings at the NAT, and on the direction of the traffic. We first test performance with a single UDP connection, which is the best case since it has great code and memory access locality. The worst case is when all 65 thousand entries are in use and traffic is sent in a round-robin fashion among these. The results show that `iptables` gives poor performance (0.4-0.5Mpps) and that Click-based NATs provide 4 to 5 times more throughput. Our NAT implementation gives similar performance to the IPRewriter NAT when there are few active connections. When there are many active connections, our NAT provides 10% more throughput for outgoing traffic and 35% more for incoming traffic, despite implementing locks that enable migration.

Figure 5 shows how throughput evolves when packet sizes vary. On a single core we can translate 10Gbps line-rate when packets are larger than 1000B.

Finally, we ran multiple NAT instances on the same box. Click forces us to assign one NIC exclusively to one Click configuration, which uses a single core; in the future we plan to modify the Click `FromDevice` element to receive from a single NIC queue, rather than all queues, which would allow multiple Click configurations to use the same NIC. To understand the performance we can achieve if we utilize all available cores, we installed four 10Gbps NICs in the same

[4]`pktgen` can generate minimum-sized packets at 10Gbps line rate (i.e. 14.88 million packets per second)

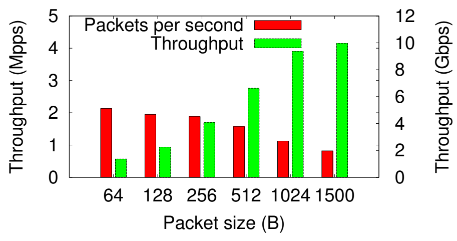

Figure 5: Throughput for a single NAT Click instance as a function of packet size.

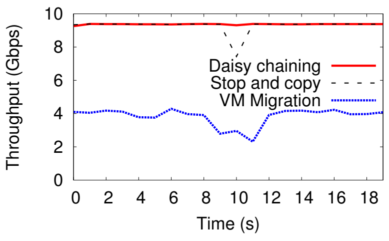

Figure 6: Effects of migrating a NAT that is translating a single high-speed TCP connection.

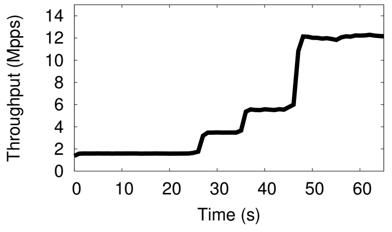

Figure 7: NAT scaling out under heavy load: capacity can be added in under one second.

server and ran multiple Click configurations, each assigned a separate NICs and CPU core to run on. Table 2 shows the throughput as we add more Click configurations. In the worst case, the total throughput nears 6.8Mpps, which is a good base for our scale-out solution.

Latency. We ran pings through our otherwise idle NAT to understand the latency it adds compared to a wire between the two machines, where the average ping latency is $30\mu s$. `iptables` increases the ping time by $34\mu s$, while our CGN adds 46 μs. The extra delay is negligible, and can be reduced if we if we run Click in the kernel.

Migration. To analyze the performance of our migration algorithm, we use TCP traffic instead of UDP because it is more sensitive to packet loss and thus a more reliable benchmark for migration. We use iperf to create 1-100 TCP connections between our client and server machines, and after a few seconds we migrate the NAT to another machine. We compare Xen VM migration, a variant of stop-and-copy migration we have implemented, and our daisy-chaining approach. Figure 6 shows the instantaneous throughput achieved by a single TCP connection when migrated with the three solutions.

First, we notice that the throughput of `iperf` through the Xen VM is rather poor; this is partly due to overheads in the Xen networking subsystem, and partly due to the fact that there is no netmap support for the Xen guest netfront driver. VM migration takes a few seconds and severely impacts the performance of the connection, even in the pre-copy phase due to CPU contention. The stop-and-copy implementation performs much better, but there is still a noticeable dip of around 2.5Gbps in the second the migration takes place. This is because stop-and-copy does not buffer packets during the migration, which also includes copying across the free ports. Finally, daisy chaining migrates the state in a few milliseconds and has limited effect on performance: the raw data shows a drop of 40Mbps.

The results for running 10 and 100 parallel connections are similar for daisy chaining, with negligible impact for performance. The throughput penalty for stop-and-copy is around 2Gbps at 10 connections and 0.4Gbps at 100 connections.

Scaling out. We now investigate how our NAT can scale out when under heavy load. We again use minimum-sized UDP packets belonging to 65K different mappings to stress our NAT. Figure 7 shows total throughput in million packets per second achieved by the NAT.

We use six external IP addresses, initially assigned to the same physical host. After 25s, we add one machine and rebalance load to the two machines by migrating half of the IPs and their associated mappings; the results show a doubling of throughput. After another ten seconds a new box is added which receives one IP from each of the existing two machines. Finally, at 45s we add three more machines and migrate IPs and mappings such that every machine handles a single IP address. The figure shows almost perfect scalability, within the boundaries of an OpenFlow switch. These results, corroborated with our single box results that show performance close to 7 million packets per second, give us confidence that our NAT can handle packets speeds of 40+ Mpps, close to industry standards.

5. RELATED WORK

Flowstream platforms [7] have been proposed as the natural architecture for building scalable middleboxes. Our NAT borrows several high-level ideas from the paper, most notably having the traffic hit multiple machines in sequence when needed (which is the basis of our migration algorithm).

OpenNF [6] and Split/Merge [17] are two migration frameworks that aim to simplify scaling middleboxes. They propose a novel API that middleboxes must use to benefit from migration support: Split/Merge proposes a memory-based hash-table API, while OpenNF proposes a getState / putState API to allow the external controller to control state migration. These frameworks require major code changes to middleboxes, and use the centralized controller in the data path which severely affects their performance.

Ananta [15] and Duet [5] are cloud-scale load balancers that also provide NAT functionality. In Ananta, traffic originating outside the cloud is spread across and processed by multiple software-based multiplexers using ECMP before making its way to the end-hosts. Duet leverages some features of the hardware switches to acheive the same functionality. Both of these solutions rely on the cooperation of the machines used by the cloud tenants: each machine must run a Host Agent, that performs some of the packet processing. Port allocations are performed by a central controller (the Ananta Manager or the Duet Controller, respectively). Our NAT achieves higher packet processing rates in software than the above solutions, while also offering support for state migration, despite the fact that network address translation is more expensive than load balancing.

6. LESSONS LEARNED

Lesson 1: NFV needs tailor-made software. Simply running existing tools such as the IPTABLES suite on Linux results in poor performance; to achieve our target 40Mpps for the NAT we would need 80 servers instead of the 6 in our setup. Moreover, the Operating System kernel adds many overheads that are unnecessary for packet processing and bypassing it via tools like netmap or DPDK [9] is an easy way to boost performance.

Lesson 2: OpenFlow-based load balancing is very restrictive in practice. Most existing SDN solutions rely on fine-grained load balancing that can be achieved when rules are installed reactively, after the controller receives "packet-in" events from the OpenFlow switch [6, 16, 17, 21]. For a NAT, the controller would need to process two such events for every single connection, as suggested by SIMPLE [16]. The limitations in both the control plane software and the number of rules supported by switches render this strategy inappropriate for our NAT. Instead, we have exploited the inherent parallelism of multiple NATs translating with different external IP addresses to proactively install load balancing rules in the switch.

Lesson 3: The controller is best kept out of the data path. A centralized controller makes it possible to support complex functionality, such as loss-free and reordering-free migration as proposed by OpenNF [6]. To achieve this, the controller will receive a large number of packets during migration events, becoming a performance bottleneck and a central point of failure.

Our experiments show that the performance of TCP flows during daisy-chaining migration is not affected by the small amounts of reordering and packet loss our solution introduces. Our controller only installs rules in the switch, and never sees any data traffic. Even if it fails, the impact on the system is minor: if it fails during normal operation, nothing really happens to the traffic since all the load balancing rules are in place. If the controller fails during migration, the machines involved will finish transferring the state across, but the system will be left in a daisy-chaining state, with the relevant packets hitting both the source and the destination machines; however, it will function correctly. In both cases, the controller can simply be restarted by a watchdog and it is trivial to have it learn the state of the system by querying the switch and the NAT servers.

Lesson 4: Universal migration frameworks. When implementing migration support we are trading implementation complexity for performance. If we had used OpenNF or Split/Merge, it would have been easy to add scaling support, but at the expense of performance. Both OpenNF and Split/Merge rely on a centralized controller on the datapath during migration, severely limiting performance. Furthermore, the Split/Merge API makes it difficult to create data structures optimized for inbound traffic.

Acknowledgements

This work was partly funded by Trilogy 2, a research project funded by the European Commission in its Seventh Framework program (FP7 317756).

7. REFERENCES

[1] A10 Networks. A10 Networks Introduces ... https://www.a10networks.com/press-releases/a10-networks-introduces-industrys-first-100-gigabit-ethernet-adc-layer-4-7-services-four-new?id=1830530.

[2] F. Audet and C. Jennings. Network address translation (nat) behavioral requirements for unicast udp. BCP 127, RFC Editor, January 2007. http://www.rfc-editor.org/rfc/rfc4787.txt.

[3] M. Bezahaf, A. Alim, and L. Mathy. Flowos: A flow-based platform for middleboxes. In *HotMiddlebox*, 2013.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[5] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *SIGCOMM*, 2014.

[6] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*, 2014.

[7] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy. Flow processing and the rise of commodity network hardware. *SIGCOMM Comput. Commun. Rev.*, 39(2):20–26, Mar 2009.

[8] S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Sriruresh. BCP 412: NAT Behavioral Requirements for TCP, October 2008.

[9] Intel Corporation. DPDK: Data Plane Development Kit. http://dpdk.org/.

[10] Internet Engineering Task Force. Working Group on Service Function Chaining. https://datatracker.ietf.org/wg/sfc/documents/.

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F.Kaashoek. The Click modular router. *ACM Trans. Computer Systems*, 18(1), 2000.

[12] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI*, 2014.

[13] MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/.

[14] NEC Corporation. Trema. http://trema.github.io/trema/.

[15] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

[16] Z. A. Qazi, C. C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *SIGCOMM*, 2013.

[17] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI*, 2013.

[18] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proc. USENIX Annual Technical Conference*, 2012.

[19] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM*, 2012.

[20] E. N. F. Virtualisation. http://www.etsi.org/technologies-clusters/technologies/nfv.

[21] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *HotICE*, 2011.