Scaling middleboxes with OpenFlow

Vladimir Olteanu
Universitatea Politehnica Bucuresti

Costin Raiciu
Universitatea Politehnica Bucuresti

1. INTRODUCTION

Middleboxes are widely deployed to perform various functionality including policy enforcement and performance optimization. They often constitute a single point of failure; high availability solutions are usually expensive and typically require the purchase of a hot spare. Further, middleboxes can often be performance bottlenecks: when networks grow beyond what a single middlebox can handle, administrators are forced to partition them. Upgrading usually means buying a replacement box, which is expensive and slow.

Programmable switches (such as OpenFlow) coupled with x86 machines have been proposed as the natural architecture to create scalable middleboxes that are also easy to deploy and update [3]. The basic recipe is very simple: a collection of x86 servers are connected to an OpenFlow switch, which is in turn "on-path" for the traffic. The servers implement distributedly the functionality of a single middlebox, such as NAT or firewall. The programmable switch is a key ingredient, splitting load between the machines. When load changes new machines can be added to the mix or existing machines removed, without disrupting traffic.

There are two basic primitives needed to make such distributed middleboxes practical. First, traffic must be load balanced to the servers according to their capacity. Secondly, when load changes and machines are added or removed, per-flow state must be moved accordingly. In this work we focus on load-balancing.

OpenFlow switches process traffic according to a limited number of rules on packet headers that are chosen and installed by an out-of-band controller [5]. In theory, per-flow rules could be used for load-balancing, but this solution is constrained by the rule memory available in switches, currently reaching a few thousands.

In this work we propose a novel load-balancing algorithm for such middleboxes. It handles arbitrary traffic distributions and server capacities using an average of 2-3 rule pairs per server. On top, we implemented a reactive algorithm that checks current traffic distributions and re-balances if the servers are unevenly loaded.

2. LOAD BALANCING ALGORITHM

A good scheme for splitting incoming traffic across several machines would need to provide good balance and require as few rules as possible.

One common way of splitting traffic is hashing each packet's 5-tuple. For instance, Equal Cost Multipath [8] splits flow across equal cost routes to the destination by choosing an outgoing route with a hash of each packet's 5-tuple. Version 1.0 of the OpenFlow standard [1], and consequently all commercially available OpenFlow switches, do not support hashing. The newly-released version 1.1 of the standard [2] allows for a "switch-computed selection algorithm", leaving decisions regarding its actual functionality up to the vendor; it is unclear what hashing functionality the switches will support.

5-tuple based hashing ensures that all packets belonging to a TCP flow will be forwarded to the same box, thus allowing stateful middleboxes. However, even if it were implemented, it is not sufficient for most middleboxes due to additional application-level constraints.

Take network address translation as an example, with a host initiating a TCP connection from private IP address X and port x. The current best common practices document require NATs to perform endpoint-independent mapping, i. e. to allocate the same external IP Y and port y regardless of the destination address and port of the connection [4] (as long as these are not repeated). To implement this behaviour with our architecture we should load balance based on the source IP and port, rather than the 5-tuple. Next, consider scaling a datacenter firewall with hundreds of thousands of rules; a natural way to distribute this on many machines is to split rules based on destination addresses (and possibly port numbers). This requires load balancing based on the destination address rather than the 5-tuple. In short, load-balancing needs to move beyond the 5-tuple and be applicable to different fields in the headers.

To implement load balancing with a small number of rules we use the IP prefix matching support in OpenFlow switches. The IPv4 address space can be represented using a binary tree, with 0.0.0.0/0 as the root, 0.0.0.0/1 and 128.0.0.0/1 as its children etc. Splitting a node always yields two children whose network mask is longer by one bit. The first child will have the newly-acquired bit set to 0, while the other one will have it set to 1. The address space is fully covered by the leaf set of any arbitrarily-constructed tree.

Our algorithm needs accurate information regarding each leaf's load. It is run periodically by the controller and uses data provided by the machines processing the traffic. The machines use the time between two consecutive runs to sample the traffic and gather said data.

In a nutshell, our algorithm greedily tries to assign as much traffic to the least loaded server that can fit

```
default_computer = pop(computers)
default_computer.load = root.load
while not overloaded(default_computer)
    computer = pop(computers)
    leaf = pop(leaves)
    if fit(leaf, computer)
        assign(leaf, computer)
        default_computer.load -= leaf.load
    else
        (left, right) = split(leaf)
        push(leaves, left)
        push(leaves, right)
    push(computer, computers)
```

Figure 1: Load balancing algorithm pseudocode.

it. The algorithm starts off with the root of the IP address space and selects a "default" server, which is assigned every unallocated prefix (which can be summarized by a low priority set of rules that match any prefix (0.0.0.0/0)). It greedily attempts to assign the largest leaf to the computer that can accommodate the most traffic. If the leaf is too large to fit, it is split. The algorithm stops when the "default" computer is no longer overloaded.

We define imbalance as the proportionally most "overworked" computer's load over the load it was supposed to have. The algorithm attempts to keep the imbalance under a user-configured maximum.

Figure 1 shows a simplified and unoptimized version of the algorithm. `pop` always returns the computer with the most unused capacity for processing traffic, or the leaf with the most traffic. A computer is considered to be overloaded when its load exceeds the load it was supposed to have by at least the same percentage that is indicated by the maximum imbalance.

In order to reduce the number of assignments, once the largest leaf's load falls below a certain threshold (dubbed the chunk threshold), the algorithm stops looking for leaves that will "fit" and starts looking for leaves that will not overload the computer.

Related Work A load-balancing algorithm making use of prefix matching was proposed by Wang et al. [7] assuming traffic is uniformly distributed across IP addresses, and that the sum of server capacities is a power of two; these assumptions rarely hold in practice [6].

3. IMPLEMENTATION

The load-balancer runs on a controller box and initially assumes traffic is uniformly distributed across the IP address space and assigns it to servers. As time passes the controller learns the real traffic distribution; if the imbalance grows the algorithm is rerun and the resulting rules are installed on the switch.

The controller is based on NOX. It supports any stateless packet inspection/processing scheme, such as firewalls. Both controller and servers run vanilla Linux 2.6.

To estimate traffic accurately we have implemented a Linux kernel module that runs on each server and estimates traffic distribution within its assigned prefixes. The implementation uses a high priority Netfilter hook that samples 1 in 100 packets that are forwarded by the machine and stores per-prefix counters in a radix tree. The controller monitors the traffic distribution by periodically polling servers.

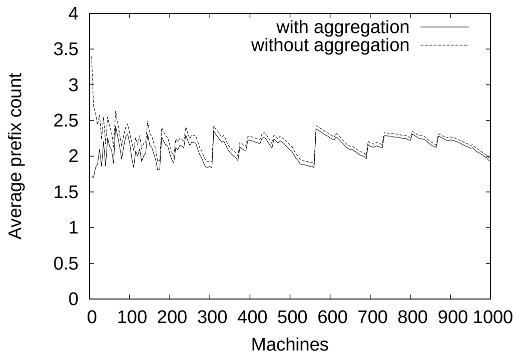

Figure 2: Average number of prefixes.

4. EXPERIMENTS

Early experiment results hint at a linear correlation between the number of machines and the number of allocated prefixes, yielding less than 3 prefixes per machine if the maximum imbalance is set to 1.1. As each prefix translates into 2 OpenFlow rules and typical switches have around 50 ingress ports, we can expect to use less than 300 rules for a full-blown deployment, which is well within a switch's capabilities.

Figure 2 shows how the average number of prefixes varies with the number of machines. We have used five different synthetic functions to simulate traffic distribution: 2 Gaussian bell curves, one constant function and 2 random distributions. Each of them yielded similar results and the graph was plotted using their average.

The algorithm's running time is less than 7ms when assigning prefixes to 1000 servers.

5. REFERENCES

[1] Openflow switch specification, version 1.0.0.
[2] Openflow switch specification, version 1.1.0.
[3] 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.*, 2009.
[4] S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT Behavioral Requirements for TCP, 2008.
[5] 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.*, 2008.
[6] J. Wallerich, H. Dreger, A. Feldmann, B. Krishnamurthy, and W. Willinger. A methodology for studying persistency aspects of internet flows. *SIGCOMM Comput. Commun. Rev.*, 2005.
[7] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. Hot-ICE'11.
[8] Z. C. Zheng, Z. Wang, and E. Zegura. Performance of hashing-based schemes for internet load balancing, 1999.