ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/316736036

Pushing SDN to the End-Host, Network Load Balancing using OpenFlow

Conference Paper · March 2016

CITATIONS
35

READS
1,390

3 authors:

Anees Al-Najjar
Oak Ridge National Laboratory
36 PUBLICATIONS **166** CITATIONS
SEE PROFILE

Siamak Layeghy
The University of Queensland
89 PUBLICATIONS **2,743** CITATIONS
SEE PROFILE

Marius Portmann
Institute of Electrical and Electronics Engineers
208 PUBLICATIONS **4,842** CITATIONS
SEE PROFILE

All content following this page was uploaded by Anees Al-Najjar on 07 May 2017.

The user has requested enhancement of the downloaded file.

Pushing SDN to the End-Host, Network Load Balancing using OpenFlow

Anees Al-Najjar*, Siamak Layeghy† and Marius Portmann‡
School of ITEE, The University of Queensland
Brisbane, Australia
Email: *anees.alnajjar@uq.net.au, †siamak.layeghy@uq.net.au, ‡marius@ieee.org

***Abstract*—The concept of Software Defined Networking (SDN) has been successfully applied to efficiently configure and manage network infrastructure, e.g. in the context of data centres or WANs, and increasingly for ubiquitous communication. In this paper, we explore the idea of pushing SDN to the end-host. In particular, we consider the scenario of load balancing across multiple host network interfaces. We have explored and implemented different SDN-based load balancing approaches based on OpenFlow software switches, and have demonstrated the feasibility and potential of this approach.**

I. Introduction

Software Defined Networking (SDN) is a relatively new paradigm for controlling and managing computer networks. The key idea, in contrast to traditional IP networks, is the decoupling of the control plane, which determines how packets are forwarded, from the data plane, which does the actual forwarding. In SDN, a logically centralised (software) *controller* or *Network Operating System* (NOS), provides an abstraction of the distributed nature of the forwarding elements (switches, routers) to higher layer networking applications, such as routing, traffic engineering etc.

Figure 1 shows the traditional SDN architecture [1]. The bottom layer (*infrastructure layer*) consists of a set of connected forwarding elements, i.e. SDN switches, which represent the data plane and provide basic packet forwarding functionality. In this paper, we extend this traditional view by adding end hosts to this layer.

The middle layer is the *control layer* consists of a centralised SDN controller which implements the functionality of a Network Operating System (NOS) [2]. The NOS deals with and hides the distributed nature of the physical network, and provides the abstraction of a network graph to higher layer services, which reside at the *application layer* of the SDN architecture [3]. The SDN controller configures SDN switches by installing forwarding rules via the so called *southbound interface*. The predominant standard for this is OpenFlow [4], [5], which we will discuss in more detail in Section II.

At the top of the SDN architecture is the application layer, where network applications and services, such as traffic engineering, routing, load balancing, etc. are implemented.

The research on SDN has so far mostly focussed on the management of network infrastructure devices, i.e. forwarding elements, and has been highly successful in practical deployments, in particular in data centres and WANs [6], [7].

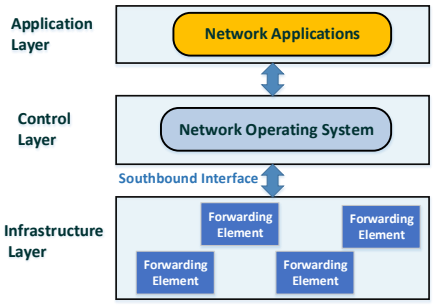

Fig. 1. SDN Architecture

In contrast to the existing body of research, in this paper we are exploring the idea of pushing SDN onto the end-host, which can be a normal computer, smartphone or any networked embedded device or *thing*.

For this, we consider the use case of network load balancing. We assume an end host with multiple network interfaces, e.g. a smartphone with 4G and Wifi, and consider the problem of efficiently balancing the user traffic across the available interfaces. When designing the mechanism, we attempted to meet the following goals, i.e. the mechanism should:

1) be **efficient**, and not introduce a significant amount of overhead on the host.
2) be **transparent** to both the applications as well as the rest of the network. Neither the applications nor the protocol stack of any other nodes in the network should have to be modified.
3) **avoid packet reordering**, and the associated detrimental impact on TCP performance.
4) work with any **OpenFlow compliant** software switch and controller, from version 1.3 onwards.

As we will discuss, the load balancing mechanisms presented in this paper achieve all of these goals.

II. Background - OpenFlow

OpenFlow [5] is currently the dominant southbound interface protocol for SDN, which allows controllers to configure the forwarding behaviour of switches. It provides the interface between the infrastructure layer and the control layer, as shown in Figure 1. The protocol also allows switches to notify the controller about special events, e.g. the receipt of a packet that does not match any installed rules.

OpenFlow allows a controller to install rules (*flow table entries*) in SDN switches via *Flow-Mod* messages. The installed rules give fine grained control over how packets are being forwarded in the network. OpenFlow rules support a basic *match-action* paradigm. Each packet arriving at a switch is *matched* against the installed rules and their corresponding match fields, and the *action* or *action list* of the matching rule is executed. The supported match fields include packet header fields, such as IP source and destination address, MAC source and destination address, VLAN tags, etc.

One of the main actions supported by an OpenFlow switch, *output*, allows forwarding packets via one or more switch ports. OpenFlow also supports a number of *set-field* actions which allow the rewriting of packet headers by the switch. For example, this allows rewriting of IP and MAC addresses to implement address translation. We will make use of this feature in our load balancing mechanisms presented in Section V. A switch can also send a received packet to the controller encapsulated in a *Packet-In* message.

III. Related Works

SDN and OpenFlow have been used for load balancing in previous works, however, mostly for server-side and network infrastructure load balancing. The *Plug-n-Serve* system prensented in [8] is an example of this. It balances web traffic (HTTP request) across a number of web servers and network paths, with the aim of minimising response time.

Another example of OpenFlow-based server load balancing was presented in [9]. The paper addresses the problem of scalability in a data centre context, and uses wildcard flow rules to achieve this. The load balancing decisions are made based on source IP addresses, which is not applicable in our scenario of client-side load balancing.

The Eden system presented it [10] allows end-hosts to implement a wide range of application-aware networking functions, including path-based load balancing. However, the paper does not address load balancing of traffic across multiple host interfaces. Another key contrast to our work is that the Eden system is not transparent to applications or the network infrastructure, and hence cannot as easily be deployed.

In [11], the authors present a multipathing and load balancing solution based on OpenFlow. The proposed method uses OpenFlow only for the network infrastructure and not for the end-host. Furthermore, end-hosts (client and server) need to use Multipath TCP (MPTCP) as the transport protocol to make use of the load balancing feature.

The authors of [12] present an approach that allows the use of multiple network interfaces on end-hosts. Their system is implemented in Android and uses an OpenFlow switch for controlling the network traffic. The paper discusses network handover, interface aggregation and dynamic interface selection. Most of the proposed mechanisms require both ends of the connection to support the special network protocol and stack, which is in contrast to our work. Furthermore, while discussing various aspects of using of multiple host interfaces, [12] does not address the specific problem of load balancing.

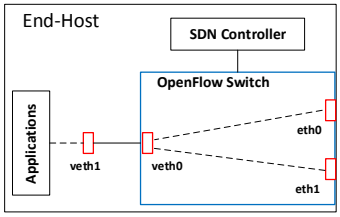

Fig. 2. System Architecture

IV. Proposed System and Mechanism

The basic architecture of our proposed host-based network load balancing mechanism using OpenFlow is shown in Figure 2. In this scenario, the host has only two interfaces, but our approach generalises to any number of interfaces.

The key component that facilitates the load balancing on the host is the OpenFlow switch with its ability to control the flow of network packets. The switch is configured to control the host's external network interfaces, in this case *eth0* and *eth1*. In order to provide the required level of indirection to switch traffic between these interfaces in a way that is transparent to the application, we also configured a pair of (connected) virtual network interfaces (*veth0* and *veth1*). Interface *veth0* is attached to the switch and *veth1* is configured as an *internal gateway* via which all application traffic is sent. This is achieved via the configuration of a corresponding default route in the kernel's routing table.

With this setup, the switch can implement traffic load balancing by choosing the external interface via which packets are sent, in this case either *eth0* or *eth1*, depending on the match-action rules installed on the switch by the controller.[1]

Due to the fact that we have detached the application from directly communicating via an external facing interface, we need to address a couple of issues, i.e. address translation and ARP handling. Address translation is required, since packets leaving the host need to have the correct IP source address as well as MAC source and destination address, depending on which interface was chosen as the egress interface by the load balancer. This needs to be implemented for packets in both the forward and reverse direction. OpenFlow provides a packet header rewriting mechanism, which allows the implementation of address translation for both IP and MAC addresses. This is achieved in OpenFlow by adding a corresponding *set-field* action prior to the *output* action.

The other issue we need to address is ARP handling. In a traditional system, when a host tries to send an IP packet to a particular destination, it would look up the address of the next hop node and the corresponding interface in the routing table. Then, it would issue an ARP request in order to establish the MAC address that belongs to the next hop IP address. Traditionally, the ARP request is broadcast on the local network, and the node with the specified IP

[1] In our current system, the controller is co-located on the host. However, this is not a requirement, and in future work we will explore the idea of delegating control to a remote controller.

address answers with an ARP reply message that contains its MAC address. Due to our introduced level of indirection in the communication, this does not work in our scenario. We therefore implement a Proxy ARP mechanism, in which the switch intercepts any ARP requests from the host and sends them to the controller. The controller then instructs the switch to send an ARP reply message with the required MAC address.

V. Load Balancing Approaches

Since a key requirement of our load balancing approach is the avoidance of packet reordering within a TCP session, we need to guarantee that all packets belonging to the same TCP session are sent via the same host network interface. To achieve this, we perform load balancing at the level of granularity of TCP connections. [2] We consider two basic approaches of SDN-based load balancing on the end host, to which we refer to as the *controller-based* and *switch-based* approach. These methods are discussed in the following.

A. Controller Based Load Balancing - Round Robin

In this approach, load balancing decisions for each individual TCP session are made by the controller. For this, the first packet of every new TCP connection (SYN packet) initiated by a host application is sent to the SDN controller via a Packet-In message, when it reaches the OpenFlow switch. Algorithm 1 shows the processing of packets received from the switch at the controller. The controller checks if the packet is indeed a TCP SYN packet, and then assigns and interface i to the new flow, from the available N interfaces.

This decision could be based on a number of factors, such as the level of congestion, delay characteristics, etc. In our case, we implement a simple Round Robin (RR) mechanism (line 4). Once the interface index i for the new flow is chosen, the controller initiates an OpenFlow rule R, with 3 match fields (lines 6-8) and 4 actions (lines 9-12). The rule will match on TCP\IP packets and the specific source port of the received packet (line 8). In our scenario, the TCP source port will uniquely identify all TCP packets belonging to this session. The *setField* actions in lines 9-11 provide the required address translation, as discussed earlier. The *output* action in line 12 will instruct the switch to forward the packet via the chosen interface i. Finally, line 13 installs the rule R on the switch.

At this point, all packets belonging to this TCP session are sent via the chosen host interface by the switch, without any further involvement of the controller. In the following section, we explore an alternative load balancing mechanism that is switch-based, with only minimal involvement of the controller.

B. Switch Based Load Balancing

An OpenFlow switch provides a very limited set of primitives, and we can not run arbitrary code as we can do on the controller. Therefore, we need a different approach if we want to do load balancing at the switch, with only minimal involvement of the controller.

[2]While our discussions in the paper is focussed on TCP traffic, the same basic load balancing approach can be applied to UDP traffic.

Algorithm 1 Controller-based RR Load Balancing

1: $flowCounter \leftarrow 0$
2: **for each** Packet-In Event with pkt **do**
3: **if** $pkt.flag.SYN == 1$ **and** $pkt.flag.ACK == 0$ **then**
4: $i \leftarrow flowCounter \mod (N)$
5: $flowCounter \leftarrow flowCounter + 1$
6: R.match[0]$\leftarrow eth\_type == IP$
7: R.match[1]$\leftarrow ip\_proto == TCP$
8: R.match[2]$\leftarrow tcp\_src\_port == pkt.tcp.src\_port$
9: R.action[0]$\leftarrow setField(ipv4\_src = IP\_addr[i])$
10: R.action[1]$\leftarrow setField(eth\_src = MAC\_addr[i])$
11: R.action[2]$\leftarrow setField(eth\_dst = GW\_MAC[i])$
12: R.action[3]$\leftarrow output(i)$
13: sendFlowModMessage(R)
14: **end if**
15: **end for**

Since version 1.3, OpenFlow supports the concept of *groups* and *group tables*, which provide abstractions for sets of ports and a level of indirection that allows implementation of features such as multicasting, fast failover, etc. Each group consists of a set of *buckets*, and each of those buckets contains a set of actions which includes the switch port (host interface) via which the packet is to be forwarded. For each packet arriving at a group, one or more buckets are selected and the corresponding actions are performed. OpenFlow supports a number of different group types, *All* for multicast or flooding, *Indirect* to implement simple indirection, *Fast Failover*, which simply selects the first live port, and *Select*, which is the one we are going to use. In the *Select* group type, only a single bucket and corresponding action set is chosen and executed. Possible selection algorithms implemeted by OpenFlow switches include Round Robin and hash based selection. Unfortunately, we cannot use the Round Robin selection method for our load balancing method, since the selection is done on a per-packet basis, rather than per TCP session. This would cause packets belonging to the same TCP session to be spread across different host interfaces, resulting most likely in packet reordering.

In the hash-based selection method, the switch computes a hash function over a tuple of packet information, e.g. the typical address and protocol 5-tuple, and the choice of bucket is based on the value of the hash. For example, in a scenario where we only have two interfaces (and buckets), the least significant bit of the hash value can be used for the selection. The hash based selection guarantees that all packets belonging to the same TCP session are forwarded via the same interface.

However, there is an important difference between the two methods. The hash-based selection implemented at the switch is (pseudo) random and can achieve equal load sharing, however, it cannot implement Round Robin load balancing. The other key difference between the two approaches is that in the switch-based load balancing, the controller is only involved in the initial one-off installation of the group and flow table.

After that, the per TCP session traffic load balancing is handled independently by the switch.

VI. Evaluation

A. Implementation and Experimental Setup

We have implemented a prototype of our host-based traffic load balancing mechanism using OpenFlow. We used Linux (Xubuntu) with kernel version 3.13.0-24 as the operating system for the end-host. The Ryu [13] open-source SDN framework forms the basis for our controller and our load balancing controller component was implemented in Python. We evaluated two different OpenFlow software switches for our prototype implementation, *Open vSwitch* (OVS) and *ofdatapath*. OVS is an open source virtual (software) SDN switch supporting the OpenFlow protocol, and is widely supported and used. OVS is supported and distributed with the Linux kernel (since version 2.6.32). In our implementation, we used OVS version 2.4. *Ofdatapath* is a user-space OpenFlow 1.3 compatible switch, based on the Stanford OpenFlow 1.0 reference switch implementation [14].

We further used the GNS3 [15] tool to emulate our network topology. GNS3 allows the creation of virtual network nodes (hosts, routers, etc.) running a full operating system and network stack, connected via virtual links. The Linux traffic control tool *tc* was used to emulate different link capacities. For our experiments, we considered HTTP traffic, and we utilised the apache benchmark tool *ab* [16] for the generation of HTTP GET requests, and *wbox* [17] was used as the web server. Finally, all our experiments were run on a single Dell PC with a Windows 7 host system, a Core i7 3.6GHz CPU and 16GB of RAM.

B. OpenFlow Switch Baseline Performance

As mentioned above, we considered two OpenFlow software switches for our host-based load balancing mechanism, OVS and ofdatapath. In an initial experiment, we wanted to measure and compare the efficiency of the two versions, and also provide a comparison to a legacy networking stack without any SDN processing, as the baseline.

For this experiment, we used the basic scenario shown in Figure 3, with a host that has a single interface connected to a server via a gateway and another IP router. No capacity limit was imposed on the virtual links.

We considered OVS and ofdatapath as the OpenFlow switch in the configuration shown in Figure 2. However, in this case there is no load balancing and we only use a single host interface, i.e. traffic is simply bridged between *veth0* and *eth0* via the installation of a corresponding rule in the switch. As a reference, we also considered the *No SDN* case, in which the host has a traditional network stack configuration, without any SDN processing.

Figure 4 shows the maximum achievable throughput between the host and the server for these 3 cases. It is obvious that OVS outperforms ofdatapath by a significant factor. This

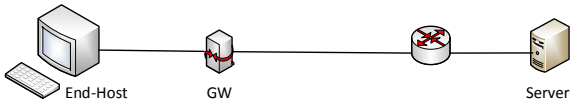

Fig. 3. Scenario for Switch Performance Evaluation

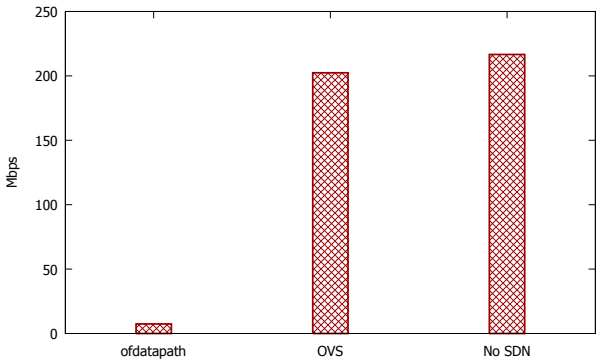

Fig. 4. Switch Performance Results

is not suprising since ofdatapath is a user space implementation while OVS is a kernel implementation of an OpenFlow switch. We further observe that OVS incurs a minor performance penalty compared to the *No SDN* case. This is largely due to the fact that we introduced an extra level of indirection with the virtual interface pair *veth0* and *veth1*, and the extra processing that this requires. Due to the poor performance of ofdatapath, we decided to only consider OVS for our remaining experiments.

C. Uniform Link Capacity

For the next experiment, we considered the scenario shown in Figure 5, with a host equipped with two network interfaces (*eth0* and *eth1*), each connected to a corresponding gateway (*GW1* and *GW2*), both of which are connected to another router, which is in turn connected to a web server. We configure both host interfaces to have a uniform capacity of 10 Mbps. [3]

At the host, we generate 100 HTTP GET request for a file located on the server, and we measure the time for the completion of the 100 downloads.

Each HTTP request results in the establishment of a new TCP connection, which we can spread across the two available interfaces using our SDN-based load balancing methods.

Figure 6 shows the measured time for the 100 file downloads for various file sizes, ranging from 1kB to 100kB. The figure shows 3 graphs. The *single interface* graph serves as a baseline, and shows the download time if only a single host interface is used.

As expected, the time is (roughly) linear with the file size. For a file size of 100 kB, the download time when using a single interface is 8.4 s. The corresponding time for our

[3] All other interfaces and links do not have a capacity limit.

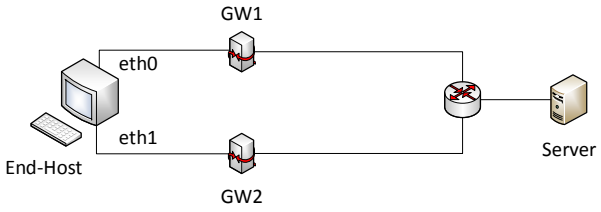

Fig. 5. Load Balancing Experiment Scenario

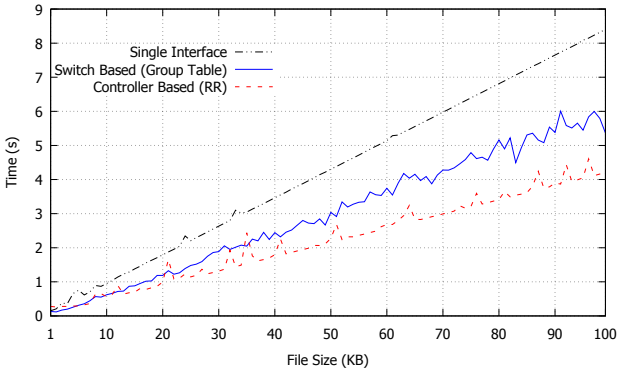

Fig. 6. Load Balancing Results (Uniform Link Capacity)

controller-based load balancing (Round Robin) is 4.2 s, i.e half of the single interface case. This means our controller-based load balancing approach optimally utilises the aggregate link capacity of the two interfaces.

However, we notice that our switch-based load balancing approach does not perform as well, and achieves a speedup of significantly less than a factor of 2. This is due to the fact that we used the *ab* traffic generation tool with a *concurrency level* $C = 2$, which means only two threads or processes are used to generate the requests, without pipelining. This works fine for a Round Robin mechanism, where the interface choice alternates between the two available options. In contrast, the switch-based mechanism relies on a hash function and the interface choice is therefore pseudorandom, and it can happen that the same interface is chosen a number of times in a row. With a concurrency level of 2, this results in one request having to queue while the second interface is idle.

We investigated the impact of the choice of concurrency level on the achieveable load balancing speedup. Figure 7 shows the results of our experiment, where we measured the download time for a range of concureny levels, for a fixed file size of 100kB. It is clear that for the Round Robin load balancer, the maximum efficiency gain is reached for $C = 2$ already. The hash-based load balancer converges towards the optimal gain with an increasing value of C. The Round Robin load balancer therefore has an advantage in scenarios with a small number of parallel requests. In order to exclude this factor in our comparison, we choose a value of $C = 20$ for our following experiments.

We performed further experiments with a higher number

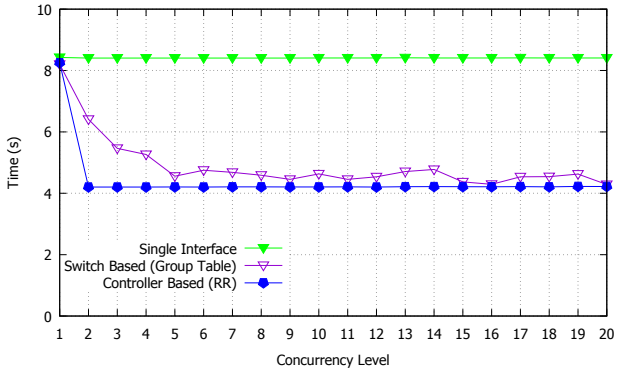

Fig. 7. Varying Concurrency Levels

N of host interfaces, i.e. with $N = 3, 4$ and 5 interfaces. We used a fixed file size of 100kB for these experiments and 100 HTTP request, as previously. Figure 8 summarises the results, and shows the download time for both the controller-based and the switch-based load balancing approaches for $N = 1, 2, 3, 4, 5$. We also included a graph that represents the ideal load balancer as a reference, where the download time decreases with $1/N$.

We can observe that the controller-based load balancer achieves very close to optimal efficiency, with only a maximum difference of 0.5% from the ideal case. The switch-based approach also performs very well, but with a slightly increased gap to the optimal performance, with a maximum difference to the ideal load balancer of around 6%.

D. Non-Uniform Link Capacity

We now consider the same scenario as shown in Figure 5, but this time with non-uniform link capacities, i.e. *eth0* has 10 Mbps and *eth1* has 20 Mbps capacity. Both our controller-based (Round Robin) and switch-based (Group Tables with hash-based selection) load balancing mechanism perform equal load sharing, where each interface is assigned the same amount of traffic. This is obviously not ideal in a case with non-uniform link capacities.

We therefore also implemented versions of our load balancing methods that can deal with this scenario. We have modified our controller-based mechanism and have implemented a Weighted Round Robin (WRR) load balancing mechanism, which assigns traffic (TCP sessions) to interfaces in proportion to the capacity of the corresponding link. Similarly, in the case of the switch-based mechanism, we use a weighting mechanism in the hash-based selection of buckets in OpenFlow Group Tables, supported in OpenFlow 1.3.

Figure 9 shows results of our download time measurement for a 100 HTTP requests. As before, we use file sizes ranging from 1kB to 100kB, and we see that the results are linear (trend) in the file size. As a reference, we also included the download time for the case of a single interface with 10 Mbps capacity. The aggregate capacity of both interfaces is 30 Mbps,

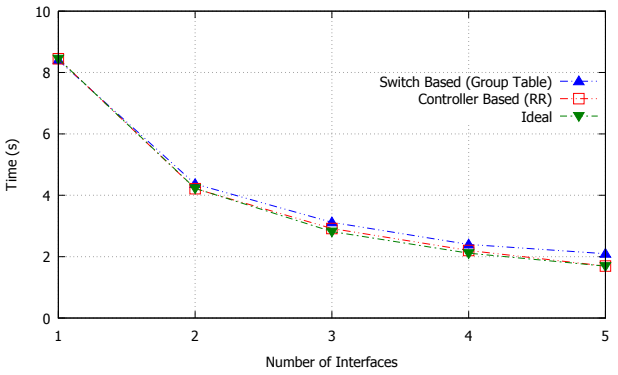

Fig. 8. Varying Number of Interfaces

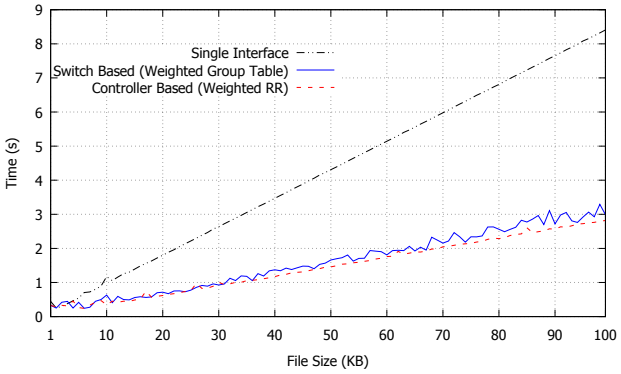

Fig. 9. Load Balancing Results (Non-Uniform Link Capacity)

and we would expect an ideal load balancer to achieve a download time reduction by a factor of 3.

As mentioned previously, the download time for 100 kB files via the single 10 Mbps interface is 8.4 s. The ideal download time for an aggregate capacity of both interfaces is a third of that, i.e. 2.8 s. We can see that the controller-based load balancer with a download time of 2.82 s comes very close to the optimum. The switch-based load balancer, using a weighted hash based approach, achieves a slightly higher time of 2.99 s, but still less than 7% from the optimal value.

We also observe a slightly higer variability of switch-based approach compared to the controller-based version. We attribute that to the pseudorandom interface selection (via a hash function) of the switch-based approach, compared to the deterministic selection of the Weighted Round Robin approach in the controller-based method.

In summary, we can conclude that both SDN-based load balancing approaches across host network interfaces perform very well, in particular the controller-based approach, which achieves near-optimal performance in all the scenarios we considered in our experiments.

VII. Conclusions

In this paper, we have explored and demonstrated the feasibility of efficient end-host network traffic load balancing using SDN and OpenFlow. In contrast to related works, our mechanism is completely transparent to the applications, the network infrastructure as well as other network hosts, and can be deployed by simply installing and configuring an OpenFlow switch (OVS is part of Linux) and a (lightweight) SDN controller on an end-host, which can include a wide range of devices used in the context of ubiquitous communications.

The presented load balancing mechanisms achieve near-optimal performance in the scenarios we considered, and meet all of the goals stated in Section I. We are currently exploring extensions to the load balancing approaches, which can handle highly dynamic and heterogeneous scenarios, with varying link capacities and a wider range of network traffic types. We are also working on implementing the system on Android.

References

[1] Open Networking Foundation, "Software-Defined networking: The new norm for networks," Open Networking Foundation, Palo Alto, CA, USA, White paper, Apr. 2012. [Online]. Available: http://www.opennetworking.org/images/stories/downloads/ sdn-resources/white-papers/wp-sdn-newnorm.pdf

[2] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1384609.1384625

[3] S. Shenker, M. Casado, T. Koponen, N. McKeown *et al.*, "The future of networking, and the past of protocols," *Open Networking Summit*, vol. 20, 2011.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[5] *Open Flow Standard*. [Online]. Available: https://www.opennetworking.org/sdn-resources/ onf-specifications/openflow

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," pp. 19–19, 2010.

[7] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Holzle, S. Stuart, and A. Vahdat, "B4: experience with a globally-deployed software defined wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, 2013.

[8] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *ACM SIGCOMM Demo*, vol. 4, no. 5, p. 6, 2009.

[9] R. Wang, D. Butnariu, J. Rexford *et al.*, "Openflow-based server load balancing gone wild," 2011.

[10] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea, "Enabling end-host network functions," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 2015, pp. 493–507.

[11] R. van der Pol, S. Boele, F. Dijkstra, A. Barczyk, G. van Malenstein, J. H. Chen, and J. Mambretti, "Multipathing with mptcp and openflow," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, Conference Proceedings, pp. 1617–1624.

[12] K.-K. Yap, T.-Y. Huang, M. Kobayashi, Y. Yiakoumis, N. McKeown, S. Katti, and G. Parulkar, "Making use of all the networks around us: a case study in android," in *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*. ACM, 2012, pp. 19–24.

[13] *Ryu sdn framework*. [Online]. Available: http://osrg.github.io/ryu/

[14] *Stanford OpenFlow 1.0 reference switch*. [Online]. Available: http://yuba.stanford.edu/git/gitweb.cgi?p=openflow.git

[15] *Graphical Network Simulator-3*. [Online]. Available: http://www.gns3.com/

[16] *Apache HTTP server benchmarking tool*. [Online]. Available: http://httpd.apache.org/docs/2.2/programs/ab.html

[17] *HTTP testing tool*. [Online]. Available: http://www.hping.org/wbox/

View publication stats