Possibilities for Load Balancing in SDN using OpenFlow's Group Tables

M.L. Pors - 4008847 - M.L.Pors@student.tudelft.nl

CS4055 - High Performance Data Networking
Delft University of Technology

January 16, 2016

Abstract

Load Balancing is a principle which can increase the performance of networks, making it interesting to investigate whether this is also deploy able in Sofware Defined Networks. One possible implementation for this makes use of OpenFlow's Group Tables. In this research, the usage of group tables is implemented for a static fat tree like topology, where there is also elaborated on the possibilities of generalizing the code for larger topologies. Measurements elaborate on the correct working of the code and the performance which can be achieved using the created groups.

1 Introduction

Load Balancing is a principle which can increase the performance of networks. Firstly, dividing the load over multiple links can increase the bandwidth and is a countermeasure to resolve congestion. Secondly, the usage of multiple paths makes the network more robust. Given that the multiple paths are disjoint, the failure of one link (and thus one path) does not kill the connection.

With the introduction of Software Defined Networking, research has been done whether using programmable switches can also implement load balancing and whether this improves the performance of the network (which is expected). [1, 2, 3] show results for load balancing in SDN with topologies with multiple switches, where [4] gives an elaboration on dynamic load balancing in a network with a single switch where a host wants to connect to a server. From these researches one can observe that load balancing is also possible with SDN.

The above named researches do not elaborate on the implementation which establishes the load balancing. For this study, I'm going to investigate whether the usage of group tables is sufficient to achieve load balancing in a network with multiple switches. To this extend, I looked for existing implementations using group tables. More information about my findings are presented in Section 2. The implementation relies on OpenFlow 1.3 and uses a RYU controller. An elaboration on my implementation approach, as well as the used topology, can be found in Section 3. After this, I will focus on the performance of the implementation. The report will end with Section 6, where the conclusions and future options are presented.

2 Related work

As stated, [1, 2, 3, 4] do not elaborate on the implementation of their solutions. To gain more knowledge about RYU and OpenFlow, but mostly about the usage of Group Tables (and SDN implementations) I relied on the following projects:

- The wiki page of [5] explains how one can achieve load balancing in a network with one switch where one host connects to different servers in a round robin fashion. This wiki provides the code and a step-by-step simulation.
- The page of [6] gives code and a step-by-step simulation for a network with multiple switches, where a multipath solution is given for a connection between host 1 and host 2.

In both projects, the load balancing is achieved by creating a group table. The group tables are both of the `SELECT` group type, which means that per flow one bucket is selected. The distribution of the chosen buckets should depend on the weights which are given to the different buckets. More information about group tables can be found in the OpenFlow Switch Specification [7].

In the Server project, every bucket has the same weight, which should resolve in a sort of round robin (equal) division of the load. There should be noted that in these buckets, every action has a different destination, because every server which the host can connect to has another address. In the Multipath project, the load should have a distribution of 30:70, following from the weights of the buckets. This project also shows a useful way to perform `iperf` and find out what the true distribution of the load during this performance was.

3 Implementation approach

In both [5, 6], the group table and flows are set up at the beginning of the network configuration, after receiving the `EventOFPSwitchFeatures` event. For my implementation, I received code from Niels van Adrichem which implements a controller for networks with multiple switches. Here, when a switch is added a forwardings matrix is calculated which is used to retrieve paths between different hosts. Flows are only added when there is communication between the hosts. My goal was to implement group tables also at this moment. Because group tables need multiple paths, the forwardings matrix which only provides information to retrieve one path, was skipped. In my implementation, hardcoded paths between the hosts are returned as a starting point.

Returned paths and topology generalization

Although as a starting point, hardcoded paths between the hosts are returned, I used a topology which can easily be generalized, such that eventually my solution and observations could be used for this general topology. The used topology, shown in Fig. 1, represents a small example of a fat tree like topology, which are for example used in datacenters.

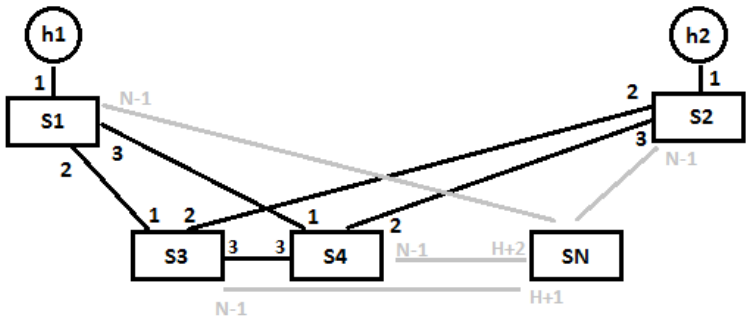

Figure 1: Multipath fat tree like topology

Each host is connected to each possible intermediate switch. When having a total of n switches where h switches are connected to hosts, we have $n - h$ intermediate switches. This means that there are $n - h$ possible paths between two hosts. As an addition, the intermediate switches could be connected

to each other, to establish even more possible paths between the hosts. In Fig. 1, the black lines are the edges which exist in my implementation. The numbers represent the ports on the switches. Following the conventions used in Niels his code (next hop is a tuple (switch, out_port)) the returned paths from host 1 to host 2 and from host 2 to host 1 are given as:

$$\begin{bmatrix}(3,2) & (2,2)\\(4,3) & (2,2)\end{bmatrix} \qquad \begin{bmatrix}(3,2) & (1,1)\\(4,3) & (1,1)\end{bmatrix}$$

When we follow the allocation of switches and ports when extending the network (the gray additions in Fig. 1), we could easily come up with a general form of returned paths between two hosts. In this generalization, hosts are always connect via the first port of a switch and the switches which are connected to host (with a maximum of one) are numbered first. This way, returned paths only depend on the final destination ($dest$) and we get:

$$\begin{bmatrix}(h+1,2) & (dest,dest)\\(h+2,3) & (dest,dest)\\\vdots & \vdots\\(n,n-1) & (dest,dest)\end{bmatrix}$$

With this, one could easily develop my approach to a general one where no hardcoded paths are returned but where the paths are calculated using the information about the number of hosts (h), the total number of switches (n) and the destination ($dest$).

Note that in this scenario I don't take the links between intermediate switches into account. Using these links would result in even more possible paths and could be desirable to use when we have a network where there are (possible) link failures, because then a particular switch can be reached using a detour via any other intermediate switch.

Group Table creation

In the implementation, the group table and other needed flows to secure a connection are added in the `install_path` method, which we only reach when there is no matching flow in the switch of the sending node. In `install_path` we have three important steps:

1. Create the group table for the sender's switch which contains $n-h$ buckets with an even weight to forward the data to each intermediate switch, using the first tuple of each path (or row).
2. Make sure the flow table of the sender's switch knows of this group table.
3. For reach intermediate switch, add a flow to forward the incoming data to the destination over the port with the same number, which used the second tuple of each path (or row).

Two things can be noted about this approach when we think about scaling the network and generalizing the code. First, in principle every flow starting at host a (thus switch a) can use the same group entry, because for every destination we want to distribute the load over all intermediate switches. Second, the flows towards destination b added in the intermediate flows could be used for every package which needs to be forwarded towards b, no matter where it came from. Because in the code the added groups and flows match using only the destination, this code is also usable in larger networks. One could also argue that the group and flow tables could easily be set up at the configuration phase of the network / controller, just as it is done in [5, 6].

4 Measurement scenario

To measure whether the explained approach and implementation works, I'm going to deploy the code of the controller (`LoadBalancingMultiSwitch.py`) on the topology shown in Section 3 (`HPDNTopology.py`). Using a variety of `dump` commands on the different switches one can check whether the groups and flows are created and added correctly.

Expectation is that the group tables and flows do not exist as long as there is communication between the two hosts. After for example `h1 ping -c 10 h2`, we should see added groups and flows in all switches.

Secondly it is interesting to look at achieved bandwitdh and distribution of the load. For this I follow the approach of [6]. With the usage of the command `dump-ports` at switch 1, one can see how many packages leave via the different ports. Other ways to investigate is to use Wireshark and trace the interfaces `s1-eth2` and `s1-eth3` when still focusing on switch 1, or trace for example the interfaces `s3-eth2` and `s4-eth2` when focusing on the intermediate switches.

Expectation for the load distribution is in this scenario 50:50, following the weights of the buckets. This result would be in line with the results presented in [6]. For the results on bandwidth, one can expect that the achieved bandwidth is larger than the maximum bandwidth of the links (which I will state to be 100Mb), because we can distribute our load over multiple links in parallel.

Versions

For this research, I relied on OpenFlow 1.3, Open VSwitch 2.4.0 and RYU 3.27. As will be elaborated on in Section 5, it is important to note these versions.. Namely, there is much development in the SDN field, which can alter the results of the working software. Also, as an example, the usage of groups (and thus also the command `dump/groups`) in Open VSwitch is only supported from version 2.1.0.

5 Measurement results

Implementation tests

```
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=26.083s, table=0, n_packets=58, n_bytes=2958, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=26.082s, table=0, n_packets=3, n_bytes=230, priority=0 actions=CONTROLLER:65535
mininet> sh ovs-ofctl -O OpenFlow13 dump-groups s1
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
mininet> h1 ping -c 5 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=40.8 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=50.0 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=49.7 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=57.5 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=56.6 ms

--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4008ms
rtt min/avg/max/mdev = 40.835/50.952/57.513/5.998 ms
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=53.251s, table=0, n_packets=118, n_bytes=6018, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=6.742s, table=0, n_packets=0, n_bytes=0, dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=53.250s, table=0, n_packets=10, n_bytes=804, priority=0 actions=CONTROLLER:65535
mininet> sh ovs-ofctl -O OpenFlow13 dump-groups s1
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=10,type=select,bucket=weight:50,actions=output:2,bucket=weight:50,actions=output:3
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s2
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=68.993s, table=0, n_packets=151, n_bytes=7701, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=21.991s, table=0, n_packets=0, n_bytes=0, dl_dst=00:00:00:00:00:02 actions=output:1
 cookie=0x0, duration=68.992s, table=0, n_packets=12, n_bytes=964, priority=0 actions=CONTROLLER:65535
mininet> sh ovs-ofctl -O OpenFlow13 dump-groups s2
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=20,type=select,bucket=weight:50,actions=output:3,bucket=weight:50,actions=output:2
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s3
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=82.432s, table=0, n_packets=274, n_bytes=13974, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=35.912s, table=0, n_packets=0, n_bytes=0, dl_dst=00:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=35.875s, table=0, n_packets=0, n_bytes=0, dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=82.425s, table=0, n_packets=0, n_bytes=0, priority=0 actions=CONTROLLER:65535
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s4
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=88.531s, table=0, n_packets=294, n_bytes=14994, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=42.501s, table=0, n_packets=0, n_bytes=0, dl_dst=00:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=42.467s, table=0, n_packets=0, n_bytes=0, dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=88.480s, table=0, n_packets=0, n_bytes=0, priority=0 actions=CONTROLLER:65535
```

Figure 2: Correctness measurement for the creation of groups and flows

Fig. 2 shows the behavior of the knowledge of the switches. As expected, before performing a `ping` between the hosts, switch 1 only has a known connection with the controller. After the `ping` we see that there are complete group and flow tables for all four switches. Both switch 1 and 2 have one

group containing two buckets with the same weight, which should resolve to an evenly distribution of traffic over switch 3 and 4. Switch 3 and 4 on the other hand know what to do with traffic with a destination of switch 1 and 2, which means they are able to cope with traffic coming (and going) to both sides. Also, the flows show that there will be no communication between switch 3 and 4 directly, because both switches do not have a flow which have `output:3`.

Bandwidth and load balancing tests

As stated, I follow [6] to figure out whether we can achieve an improved bandwidth and load balancing using the implementation. Fig. 3 shows the performed `iperf` commands performed on host 1 and 2. Fig. 4 and Fig. 5 show the overview of the received and sent packages per port of switch 1, before and after the `iperf`, where in Fig. 4 only 5 pings were done between the two hosts.

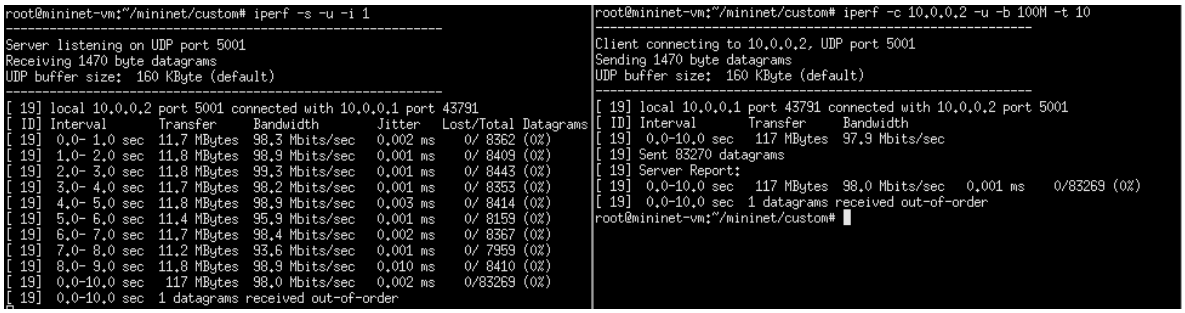

Figure 3: `iperf` overview for host 1 and 2

```
mininet> sh ovs-ofctl -O OpenFlow13 dump-ports s1
OFPST_PORT reply (OF1.3) (xid=0x2): 4 ports
  port LOCAL: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=0, bytes=0, drop=0, errs=0, coll=0
           duration=36.452s
  port  1: rx pkts=21, bytes=1730, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=53, bytes=3055, drop=0, errs=0, coll=0
           duration=36.491s
  port  2: rx pkts=41, bytes=2082, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=49, bytes=2803, drop=0, errs=0, coll=0
           duration=36.492s
  port  3: rx pkts=43, bytes=2381, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=47, bytes=2543, drop=0, errs=0, coll=0
           duration=36.491s
```

Figure 4: Switch 1 Port overview before performing the `iperf`

```
mininet> sh ovs-ofctl -O OpenFlow13 dump-ports s1
OFPST_PORT reply (OF1.3) (xid=0x2): 4 ports
  port LOCAL: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=0, bytes=0, drop=0, errs=0, coll=0
           duration=280.287s
  port  1: rx pkts=83294, bytes=125907566, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=326, bytes=18421, drop=0, errs=0, coll=0
           duration=280.326s
  port  2: rx pkts=313, bytes=15936, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=83590, bytes=125922325, drop=0, errs=0, coll=0
           duration=280.327s
  port  3: rx pkts=315, bytes=17714, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=320, bytes=16448, drop=0, errs=0, coll=0
           duration=280.326s
```

Figure 5: Switch 1 Port overview after performing the `iperf`

The results, clear as they are, contradict our expectations. First, from Fig. 3 we can observe that the bandwidth (were the maximum bandwidth per link is set to 100Mb) does not exceed the 100Mb. Secondly using the other two figures, one can clearly see that the load is not distributed over the two intermediate switches. The number of received packages (`rx_pkts`) at port 1 is closely the same to the number of send packages (`tx_pkts`) over port 2, while `tx_pkts` of port 3 stays very low. Thus, all traffic seems to travel via switch 3.

To investigate whether these results come only with my implementation, more measurements were done - of which the exact result are not literally shown in order to keep this report compact. Firstly, also a parallel `iperf -P 8` was executed to see whether in that case the load would be distributed over the multiple paths, but this was not the case. Secondly, I did the measurement using the code of [6] to see whether here I would end up with a 30:70 result. But again, the packages were only sent over one port. In other words, only one bucket of the group table is selected, every time.

Another project, [8] provided an multipath load balancing example which uses group tables but no controller. Following the steps of this project, the `dump-ports` results of the switch where the group table exist show that not a single package goes over the port which is stated as action in the second bucket in the created group.

The results of my measurements contradict the expectations stated in Section 4 but also contradict the results which were presented in [6], which do show effective load balancing using group tables. When searching for an explanation, [9] provided a statement that the Open VSwitch only uses the destination MAC to select the bucket in a group (note that this page originates from 2014). Because in my group table, each bucket has the same destination, this could be an explanation why each time the same bucket is selected. It would also explain why running [5] does work as expected, because here all servers have another destination adress.

From [9] it is not directly clear which version of Open VSwitch operates this way and from [6] it is not directly clear which version of Open VSwitch is used to get the results which match with our expectations stated in Section 4. [8] does provide changes for the code of Open VSwitch (there used with 2.1.0 version) to create stochastic selection of buckets. Implementation of these changes in the 2.4.0 version of Open VSwitch did not give different results on my measurements. I did not try to install the patch named in [9] due to time limitations.

6 Conclusions

In this research I investigated the possibilities for load balancing achieved with the implementation of group tables using the RYU controller and OpenFlow 1.3. Although the implementation contained a static, small topology, Section 3 shows how the implementation approach can easily be generalized to larger fat tree like topologies.

The correctness of the code to add the groups and flows to the switches was shown in the first measurement. Measurements investigating bandwidth and load distribution contradict with the expectations that bandwidth could increase the maximum bandwidth of the links and the load would be evenly distributed over the two intermediate switches. These results also contradict the results which were achieved using a sort-like implementation in [6], which do show that group tables can distribute load over multiple links.

From further investigation and [9], we can suspect the Open VSwitch to be the troublemaker, but more research is needed to say which version does support 'correct' of use of group tables. Namely, the OpenFlow Switch Specification [7] does state that the selection algorithm should implement equal load sharing and can optionally be based on bucket weights when the group type `SELECT` is used. Thus, possibility is that the selection algorithm is not correctly implemented in Open VSwitch 2.4.0 used in this research. It is not known which version of Open VSwitch is used in [6].

Another way to implement load balancing is to work with matches which are more strict and not (directly) to work with group tables. In this research the match only contains the destination address, both for flows as for groups (although this seemed enough for the [6] project). Disadvantage of that approach is that it probably will lead to less structured/clear code as well as flow tables.

References

[1] M. Bredel, Z. Bozakov, A. Barczyk, and H. Newman, "Flow-based load balancing in multipathed layer-2 networks using OpenFlow and Multipath-TCP," *HotSDN'14*, 2014.

[2] S. Govindraj and A. e. a. Jayaraman, "OpenFlow: Load balancing in enterprise networks using Floodlight controller," *'Capstone paper' for the Interdisciplinary Telecommunications degree, University of Colorado*, 2014.

[3] W. Yahya, A. Basuki, and J.-R. Jiang, "The Extended Dijkstra's-based load balancing for OpenFlow network," *International Journal of Electrical and Computer Engineering*, vol. 5, no. 2, 2015.

[4] S. G. N and R. S, "Dynamic load balancing using software defined networks," *International Journal of Computer Applications*, 2015.

[5] "Server Load Balancing using ryu," https://github.com/OpenState-SDN/ryu/wiki/Server-Load-Balancing, accessed: 2016-01-15.

[6] "Multipath Transmission using RYU," http://csie.nqu.edu.tw/smallko/sdn/ryu_multipath_13.htm, accessed: 2016-01-15.

[7] O. N. Foundation, *OpenFlow Switch Specification*, 2012.

[8] "Stochastic switching using Open vSwitch in Mininet," https://github.com/saeenali/openvswitch/wiki/Stochastic-Switching-using-Open-vSwitch-in-Mininet, accessed: 2016-01-15.

[9] "OpenFlow 1.3 Groups with type select discussing," http://openvswitch.org/pipermail/discuss/2014-August/014785.html, accessed: 2016-01-15.