**Hash based ECMP load balancing**

Author: Nisha Kumari

**EXECUTIVE SUMMARY**

Data center networks every so often use compactly interconnected topologies to deliver high bandwidth for internal data exchange. In such network, it is precarious to employ effective load balancing schemes so that all the available bandwidth resources can be utilized. To make proficient use of network resources, satisfy user demands, and in order to adapt to network conditions, network operators do traffic engineering (TE), i.e., managing traffic how it is getting routed across the network. Traffic engineering tries to optimize both network efficiency and the performance to the current network conditions. Equal Cost Multipath (ECMP) is one of the traffic engineering methods that enable the usage of multiple equal cost paths from the source node to the destination node in the network. The advantage is that the traffic can be split more uniformly to the whole network avoiding congestion and increasing bandwidth consumption.

This white paper proposes that congestion in network can be reduced using Equal Cost MultiPath routing complementing with hashed forwarding. Due to less congestion it would also help in reducing the time taken in data transfer. Also with use of this algorithm network performance of shuffle phase data transfers in Hadoop MapReduce can be enhanced.

ECMP load balancing refers to distributing traffic more evenly by installing entries for multiple best paths to the switch's forwarding layer and using load balancing algorithm to identify flows and distribute them to different paths. If number of flows is large this algorithm helps in distributing the traffic along with other unused paths in order to reduce the congestion. ECMP implementation enables us to evenly spread flows across the network leading to best utilization of multiple links towards the destination. It reduces the time taken in data transfer

up to large extent due to less congestion. This algorithm helps us in improving network performance in SDN enabled networks.

We have implemented ECMP module with POX controller using mininet fattree topology (with four and nine core switches K=4, 6) and studied effect of ECMP for iperf flow generator and Hadoop Map Reduce shuffle phase.

**AIM**

The key goals of this experiment are:

1. Implement and validate ECMP load balancing algorithm in SDN enabled environment
2. Study effect of ECMP on Hadoop Map Reduce(MR) shuffle phase

**Background**

**What is ECMP Forwarding?**

ECMP forwarding is a mechanism that helps us in getting multiple equal cost paths to the same destination and route traffic over those multiple paths. It is a simple and feasible load-balancing routing scheme for networks. [4] It helps us in getting multiple best paths to the destination if possible and spread out traffic through all paths if needed.

This algorithm

* Helps to distribute traffic uniformly across all the links in network reducing congestion
* Load balance flows to the same destination over multiple equal-cost links.
* Make use of the available bandwidth on links to the same destination rather than leave some links unused.
* Dynamically shift traffic to another equal cost path to the same destination within seconds after detection of link failure without severe loss of traffic.
* Reduces time taken in data transfer due to less congestion

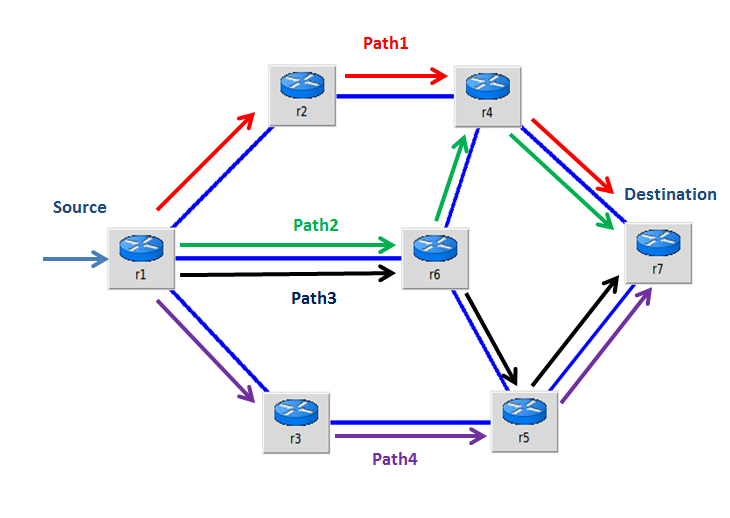
****

Figure : ECMP Load balancing

An example of ECMP load balancing is shown in **Figure 1**. Here four equal cost paths are available to route the packet to destination r7 from source r1. With use of ECMP algorithm all four paths are getting utilized to route packets. This helps in distributing traffic evenly across the network and reduces congestion. Additionally, these four ECMP paths are backups/failover routes for each other. If one of the paths fails, traffic could be split between the other three paths after failure detection. SDN facilitates quick implementation of the above algorithm scheme.

**SDN Introduction**

Software Defined Networking (SDN) is an evolving paradigm in computer networking that consents a logically centralized software program to control the behaviour of an entire network.

The goal of SDN is to efficiently program the network with software running on a logically centralized controller. These days network switches and routers program their forwarding locally which means that network devices make their own decisions internally about how to forward traffic. Traffic-forwarding and routing decisions are learned by distributed control-plane protocols including [spanning-tree](http://searchnetworking.techtarget.com/definition/spanning-tree-protocol), [OSPF](http://searchenterprisewan.techtarget.com/definition/OSPF) and [BGP](http://searchtelecom.techtarget.com/definition/BGP). But these networking protocols have restrained flexibility. To make them work with all network devices contributing in the forwarding domain we have to follow the same rules as defined by the protocol standard. That leaves little room for creativity or unusual business requirements.

SDN takes the [control plane](http://searchsdn.techtarget.com/definition/control-plane-CP) (how a network device will forward traffic) and separates it from the data or forwarding plane (a network device forwarding traffic based on the control-plane policy). With SDN, the separated [control plane](http://searchsdn.techtarget.com/news/4500244596/ICYMI-ONOS-Blackbird-focuses-on-SDN-control-plane-performance) resides on a logically [centralized controller](http://searchsdn.techtarget.com/definition/SDN-controller-software-defined-networking-controller) that sees and knows all about the network, including where the hosts connect to the network and what the network topology connecting all of the hosts together looks like. The SDN controller allows network engineers to implement unique and flexible forwarding policies limited only by the ability of the software running on it. It manages flow control to enable intelligent networking. SDN controllers are based on [protocols](http://searchnetworking.techtarget.com/definition/protocol), such as [OpenFlow](http://whatis.techtarget.com/definition/OpenFlow), that allow [servers](http://whatis.techtarget.com/definition/server) to tell [switches](http://searchtelecom.techtarget.com/definition/switch) where to send [packets](http://searchnetworking.techtarget.com/definition/packet).

The controller is the core of an SDN network. It lies between network devices at one end and applications at the other end. Any communications between applications and devices can be controlled by SDN controller. The controller also uses protocols such as OpenFlow to configure network devices and choose the optimal network [path](http://searchnetworking.techtarget.com/definition/path) for application traffic. In effect, the SDN controller serves as a sort of operating system ([OS](http://searchcio-midmarket.techtarget.com/definition/operating-system)) for the network. By taking the control plane off the network [hardware](http://searchcio-midmarket.techtarget.com/definition/hardware) and running it as [software](http://searchsoa.techtarget.com/definition/software) instead, the controller facilitates automated network management and makes it easier to integrate and administer business applications.

**ECMP Forwarding Implementation in SDN Environment**

ECMP implementation ensures, if there are numerous best paths to get to a specific MAC address (i.e. multiple paths have the same cost metric), the controller installs rules in switches in such a way so that all the paths to reach that MAC address are utilized instead of overloading a single path. This is accomplished by identifying rules that match on more fields in conjunction with the destination MAC address. To implement this we have used Pox, a Python based SDN controller. The switches in this implementation run Open vSwitch which implements OpenFlow, a protocol to enable SDN controllers to determine the path of network packets across a network of switches. This algorithm finds all possible shortest paths to reach the destination and perform modulus operation on the hash value with number of shortest possible routes and select the final path based on the output. Once final path is decided controller installs the rule on all switches of selected path and forwards the packet to destination host.

This implementation has been categorized into two steps.

## Multiple best path selection algorithm

When the first packet of a flow arrives on source switch it sends notification to controller as switch does not have rules that match with it. The controller then checks to see what could be the possible ways to reach the packet’s destination MAC address. First it tries to figure out the destination switch to which destination host is connected. We are populating a dictionary where key is host’s MAC address and value is a tuple consisting of switch and port with which host is connected. The Controller looks up in this dictionary to get the destination switch

* **If destination MAC address entry is updated:** It gets the respective switch and the port on which destination host is connected. If source switch and destination switch both are same then controller simply installs the rule to forward the packet on out port as there is only one possible path to reach the destination. If source switch and destination switch both are different we try to get all possible shortest paths to reach the destination using Dijkstra algorithm. We did slight modification in Dijkstra algorithm to get multiple shortest paths between source and destination instead of a single path. Topology information is represented as non-directional graph and stored as an adjacency list. All edges have equal cost. This algorithm computes the weight of the path where weight of the path is sum of weight of constituent edges and returns paths with minimum weightage. This gives us all possible shortest path to reach the destination.
* **If destination MAC address entry is not updated:** In that case it send the packet out on all ports except the one it came in and learns the location of the hosts.

We populate dictionary with this entry so that next time when packet for that destination arrives instead of flooding all ports we can just send to the appropriate port. This is based on backward learning concept. Next packet in the flow will follow the above mentioned logic and ECMP forwarding rule will be installed for the flow.

## Hashed Routing

When forwarding a packet the switch must decide which path to use. Fundamentally, <n> paths exist in the forward and reverse path direction for given source and destination.

In hashed routing a modulo <n> hash of various header information is performed. Header fields used for hash calculation includes source IP address, destination IP address, protocol, Source port and Destination port. Hash function returns ECMP-style 5-tuple hash for packets, otherwise 0. We perform modulus operation on the hash value with total number of best possible routes. The output of this modulus indicates which of the paths to use.

Ideally, the header information and hash algorithm are chosen such that an arbitrary traffic pattern will be equally distributed on the paths while ensuring that a single flow remains on the same path for the life of the flow in order to preserve in-order packet delivery.

Please refer Appendix A for flow chart displaying whole implementation briefly.

Experimental Setup

To study the effect of ECMP load balancing algorithm for iperf flow generator and Hadoop Map Reduce shuffle phase we have to do following setups.

**Topology Setup**

We are creating topology using mininet. Mininet is a network emulator for rapid prototyping of SDN. It allows us to launch a virtual network with switches, hosts and an SDN controller all with a single command providing network isolation to the end hosts from the host machine. We have created fattree topology (As shown in **Fig 2**) using the mininet python API with following configuration.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Topology | Nodes | Links Bandwidth | CPU | Memory |
| Fat Tree | 16 | 100 mbps - core switches and aggregation switches  100 mbps - aggregation switches and edge switches  100 mbps - edge switches and end hosts | 64-bit | 16 GB |

Table : Fattree Topology Configuration

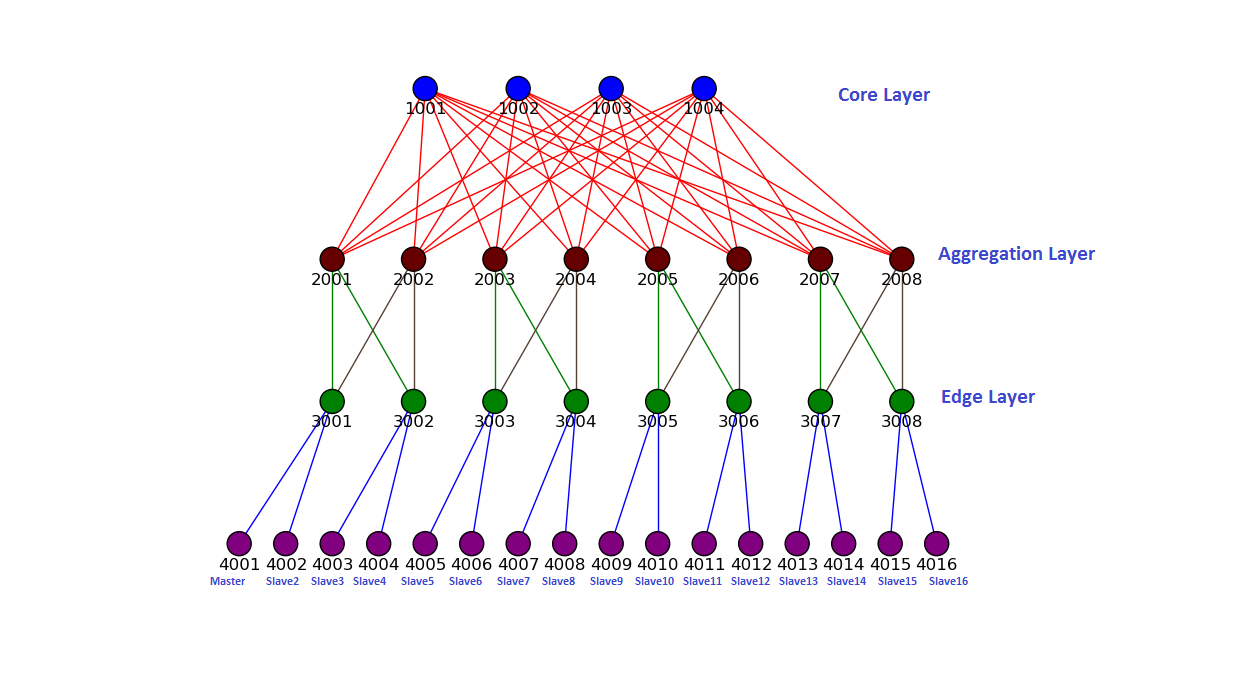
****

Figure : Fattree Topology

**Flow Forwarding & Monitoring Setup**

For flow forwarding we are using ECMP controller module. The syntax to start the controller is as follows:

./pox.py ecmp openflow.discovery

Where ecmp is the python module having flow forwarding and monitoring logic and openflow discovery is a POX module which is responsible for learning the network topology and discovering created links.

In ecmp module we are registering ConnectionUp, ConnectionDown and other link events. When this module is loaded into POX, it constructs entire topology graph using various events. Once topology graph is constructed we pass it to modified Dijkstra algorithm along with source and destination to discover all possible shortest paths to reach the destination.

There is another module Flowmonitor which helps in monitoring and measuring the flows. It measures data periodically by sending flow stats events to edge switches in network. Using response to flow stats message, we can calculate per flow rate.

Both these components are dependent on POX discovery module.

Please refer Appendix C for detailed description of flow monitoring module.

**iperf data generation Setup**

We are generating data using iperf to spawn traffic across the network. iperf is a traffic generation tool for performing network throughput measurements. With help of this tool we can do experiment with different TCP and UDP parameters to see how they put impact on network performance.

We are starting multiple hosts running in server mode as the traffic receiver and then have been starting multiple iperf processes running in client mode on other hosts as the traffic sender.

Once fattree topology has been created and POX controller is up we are starting data transfer by running the python script file.

sudo python <python script file>

In python script file we have to specify which hosts need to start iperf in server mode and which all need to start iperf in client mode. Also we have to specify iperf server port numbers and amount of data to be transferred. We are plotting forwarding path installed for each flow transfer with help of networkx and matplotlib module.

**Hadoop MapReduce Shuffle phase Setup**

Hadoop MapReduce is an Open source framework. It helps us to write applications that stores and process huge amounts of data in a large cluster of commodity hardware. We are measuring the time taken in shuffle phase in a Hadoop MapReduce job that runs [1] over a SDN-enabled technology. To run Hadoop over an SDN-enabled topology we swapped default Mininet hosts with docker containers to obtain isolation of processes running on end hosts. We used docker -an open-source virtualization application which provides lightweight containers that run processes in isolation with each other to create sixteen lightweight containers as end hosts. To run Hadoop TeraSort on end hosts, we created one docker image having Hadoop installed on Ubuntu. While creating docker containers as end hosts in Mininet topology, we are starting each host container from the Hadoop docker image.

We are running Hadoop TeraSort as a Hadoop MapReduce job for flow generation in an SDN-enabled topology. TeraSort is a Hadoop benchmarking tool which can sort any amount of data quickly. First we are generating data using TeraGen and once data has been generated we are sorting those data using TeraSort. For detail steps please refer to Appendix B.

We can specify the hosts from the topology that we want to use to run Hadoop job by adding the hosts in /usr/local/hadoop/etc/hadoop/slaves file. As we are starting each host from the Hadoop docker image, each host gets the Hadoop configuration from the image. We may change the Hadoop configuration files in all hosts according to our needs. After formatting the Hadoop distributed file system via NameNode that are running on master node, we are starting the MapReduce daemons, and run Hadoop TeraSort. [1]

**Observation and Conclusion**

We have implemented ECMP algorithm to route packets which helped us in balancing load by spreading traffic across the networks. By this implementation we have optimized network by avoiding the situation where the network is congested in some links while other parts of the network are underutilized. We have tested and validated it with iperf flow generator and Hadoop Map Reduce.

## With iperf Generator

We have generated network traffic across various hosts in fattree topology and compared the time taken in flow transfer with OpenNetMon – a POX OpenFlow controller module and ECMP controller module. We have started iperf processes on hosts ‘**10.0.0.1’**, **’10.0.0.2**’, ‘**10.0.0.9’**,‘**10.0.0.10**’ in server mode and **’10.0.0.8**’,**’10.0.0.7**’,’**10.0.0.16’**,**’10.0.0.15**’ in client mode. Servers haven been started on ports ‘**5001**’, ‘**5002**’, ‘**5003**’ and ‘**5004**’. We have performed the test for 1GB, 5GB and 10GB data transfer.

Observations

As ECMP is utilizing more than one core switch for routing packets its taking less time as compared to using only the spanning tree in case of OpenNetMon. But with OpenNetMon controller module it is utilizing only one core switch for all flow which results in congestion and network traffic (as shown in **Table 2**).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Source Host** | **Destination Host** | **Data Transferred** | **Without ECMP Forwarding** | | **With ECMP Forwarding** | |
| **Time taken(in min)** | **Transfer rate(Mbits/sec)** | **Time taken(in min)** | **Transfer rate(Mbits/sec)** |
| 10.0.0.1 | 10.0.0.8 | 5GB | 11.41 | 62.7 | 7.47 | 95.8 |
| 10.0.0.2 | 10.0.0.7 | 5GB | 14.97 | 47.8 | 7.49 | 95.5 |
| 10.0.0.9 | 10.0.0.15 | 5GB | 14.98 | 47.6 | 9.18 | 94.7 |
| 10.0.0.10 | 10.0.0.16 | 5GB | 15.39 | 46.5 | 9.23 | 93.2 |

Table 2: Time taken in 5GB data transfer from source host to destination host

Table 2 displays the data collected for 5GB data transfer from source host to destination host with iperf generator. It displays time taken as well as transfer rate for data flow with ECMP forwarding and without ECMP forwarding. With ECMP transfer rate is high resulting in less transfer time.

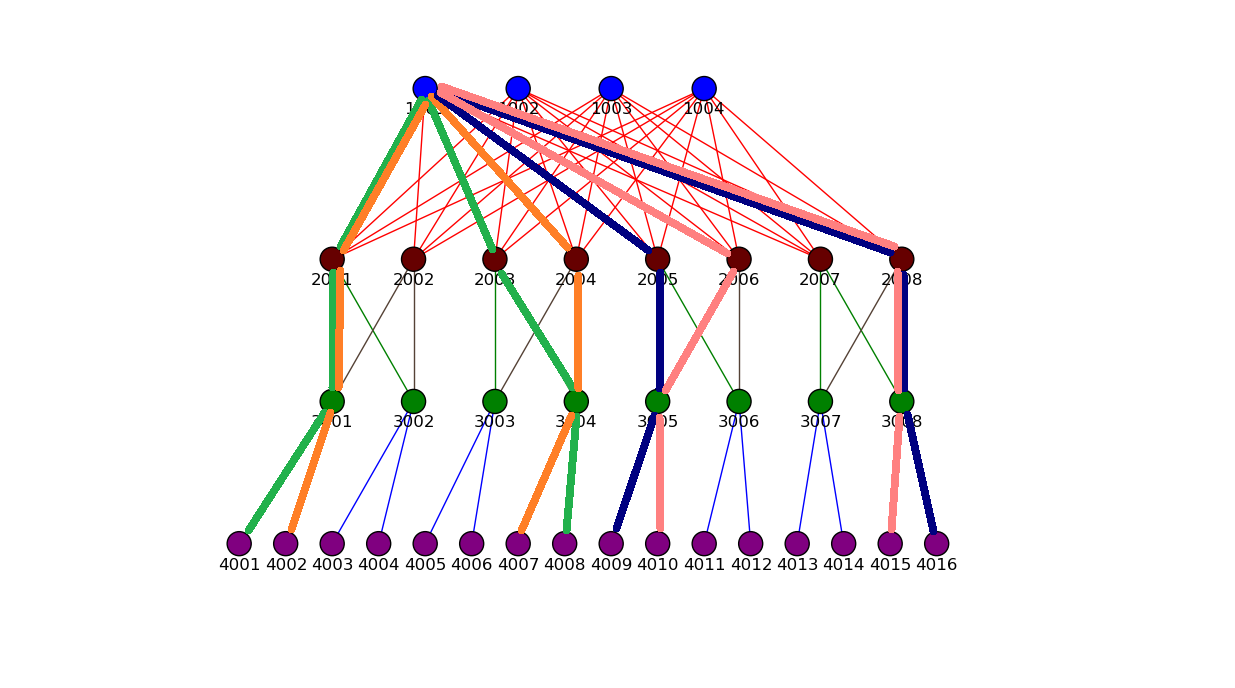
****

Figure : Routes getting traversed for flows with OpenNetMon

In above figure routes getting traversed for flows with OpenNetMon controller has been highlighted. Here single core switch is getting utilized for all flows which cause congestion in few links (2001-3001, 1001-2001, 1001-2008 etc.). OpenNetMon controller uses forwarding logic same as l2\_learning pox module. This module enables STP (Spanning Tree Protocol) in order to avoid loops. [15] STP provides a means to prevent loops by blocking links in an Ethernet network. With enabled STP it is restricted to use all links hence accumulating the traffic to fewer active links.

As link bandwidth is getting shared with multiple flows it reduces overall transfer rate. Hence total time taken in data transfer is getting increased for congested flows (**Table 2**).

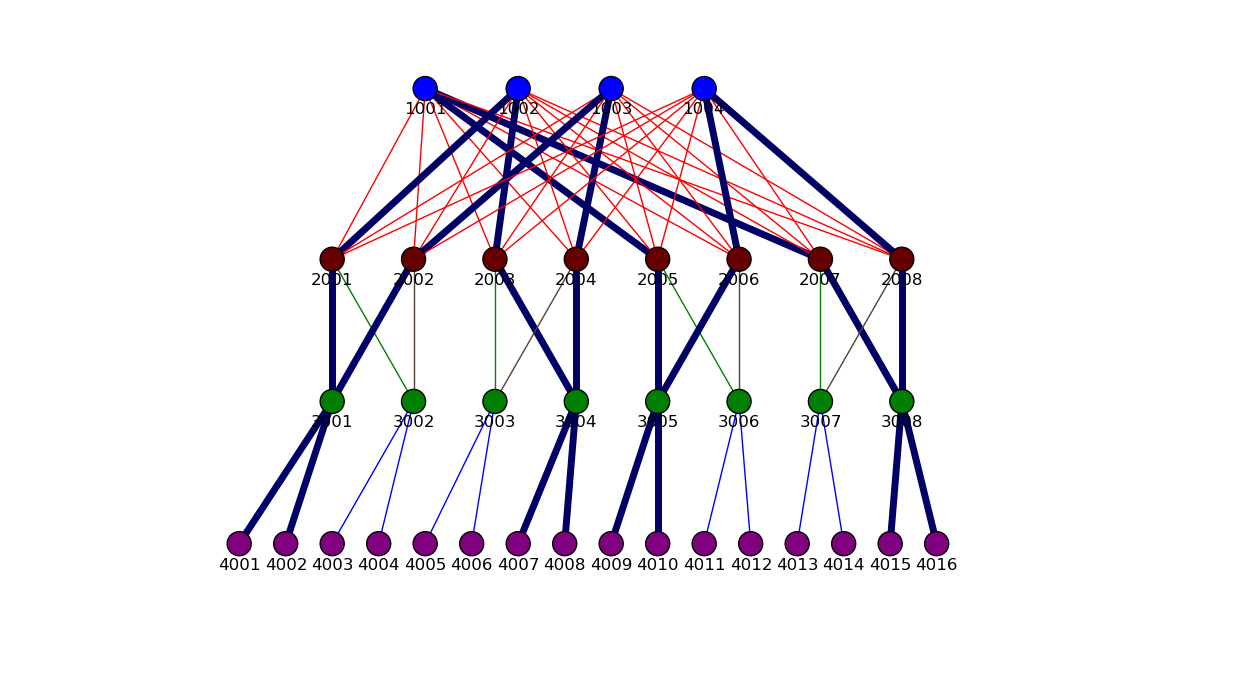


Figure : Routes getting traversed for flows with ECMP forwarding

With ECMP forwarding module if links are getting congested then instead of sharing linkwidth for two flows it picks other possible shortest path to route the packet. In our experiment for all four flows different routes are getting traversed utilizing all core switches (as shown in **Fig 4**). In this way we are avoiding the network traffic hence resulting in high transfer rate (**Table 2**). Total time taken in data transfer has been reduced up to 50% in compare to OpenNetMon controller module.

**With Hadoop MR**

We sorted data of 5GB in a cluster of 16 nodes connected through an SDN-enabled fattree topology (As shown in **Fig 2**) and compared transfer time in shuffle phase with OpenNetMon and ECMP controller module. Data has been sorted using six nodes (master, slave2, slave5, slave9, slave10 and slave16) out of the 16 nodes from the topology. All the six nodes are running map tasks while reduce tasks are being executed on two nodes. There are two reduce tasks which are taking place on reducer nodes. For TeraSort steps please refer Appendix B.

Conclusion

With ECMP controller module transfer time in shuffle phase has been reduced up to large extent. With OpenNetMon controller module it’s almost taking double time in compare to ECMP controller module. During shuffle phase as there are simultaneous data transfers occurring from all mapper nodes to reducer nodes the whole network got congested. With ECMP controller module we install multiple shortest paths by utilizing all core switches to route the packet to destination which reduces the congestion and hence results in less transfer time. But OpenNetMon controller module utilizes single core switch so most of the links remain unutilized and increases traffic on few links resulting more transfer time. We have also tested and validated this experiment with 10GB and 20GB data.

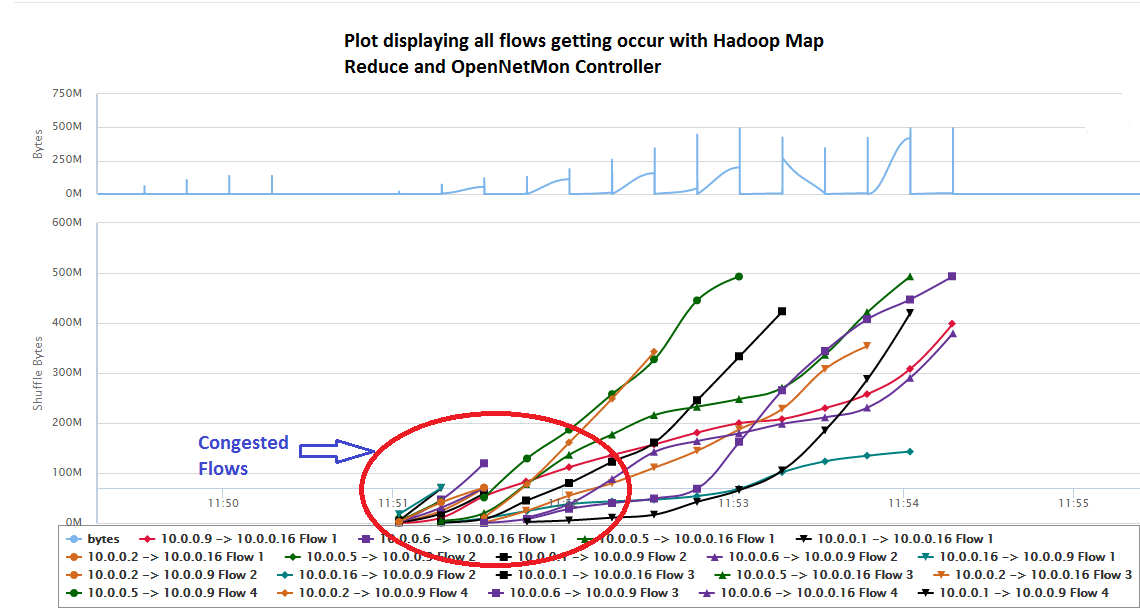


Figure : Congested Flows with OpenNetMon Controller Module

We have plotted transfer flows occurred during Hadoop map reduce shuffle phase. Some of the flows among these are getting congested (as shown in **Fig 6**) hence increasing the transfer time. ECMP controller module distributes these flows across other unused links which reduces the congestion as well as transfer time.

With ECMP implementation these congestions are getting reduced (As shown in **Fig 7**) which further reduces transfer time. Total transfer time has been reduced to half of the time taken with OpenNetMon controller. Also we have plotted graph with shuffle bytes and transfer time as parameters for ECMP as well as OpenNetMon. The plotted graph is steeper with ECMP in compare to OpenNetMon due to reduced transfer time.

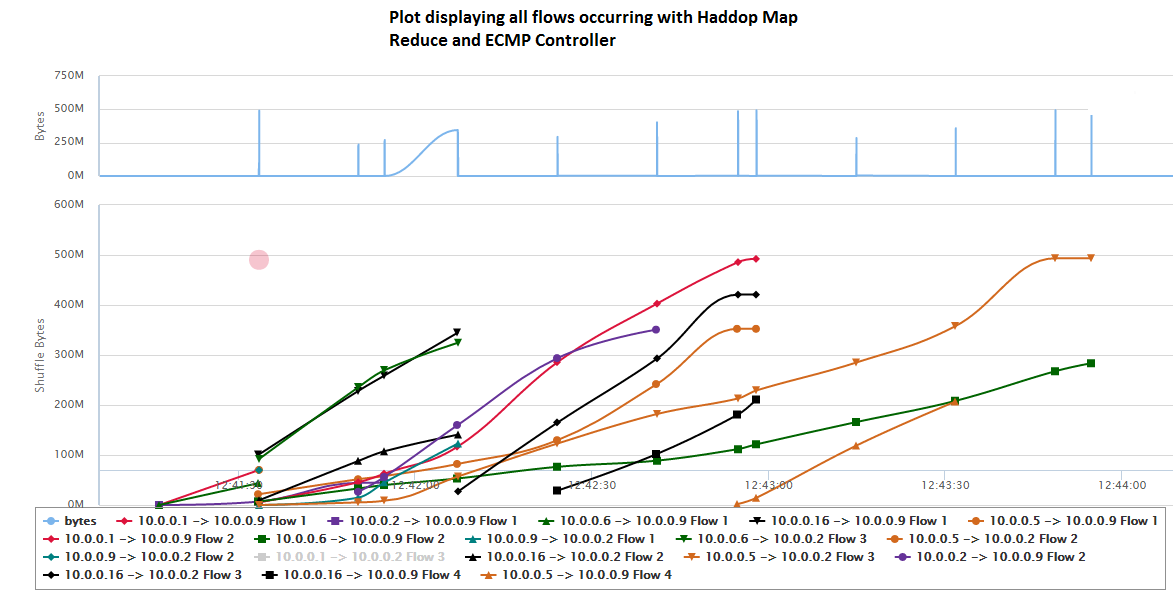
****

Figure : Flows with ECMP Controller module

**APPENDIX A**

Table 3 displays the data collected for 1GB data transfer from source host to destination host with iperf generator. It displays time taken as well as transfer rate for data flow with ECMP forwarding and without ECMP forwarding.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Source Host** | **Destination Host** | **Data Transferred** | **Without ECMP Forwarding** | | **With ECMP Forwarding** | |
| **Time taken(in min)** | **Transfer rate(Mbits/sec)** | **Time taken(in min)** | **Transfer rate(Mbits/sec)** |
| 10.0.0.1 | 10.0.0.8 | 1GB | 2.31 | 62 | 1.46 | 96.8 |
| 10.0.0.2 | 10.0.0.7 | 1GB | 2.56 | 55.8 | 1.49 | 95.5 |
| 10.0.0.9 | 10.0.0.15 | 1GB | 2.97 | 48.1 | 1.52 | 93.2 |
| 10.0.0.10 | 10.0.0.16 | 1GB | 3.04 | 47.1 | 1.57 | 91.7 |

Table : Time taken in 1GB data transfer from source host to destination host

Figure 8 displays the flow chart of ECMP module implementation for better understanding. It is describing briefly the whole process i.e., how packets are getting routed to the destination.

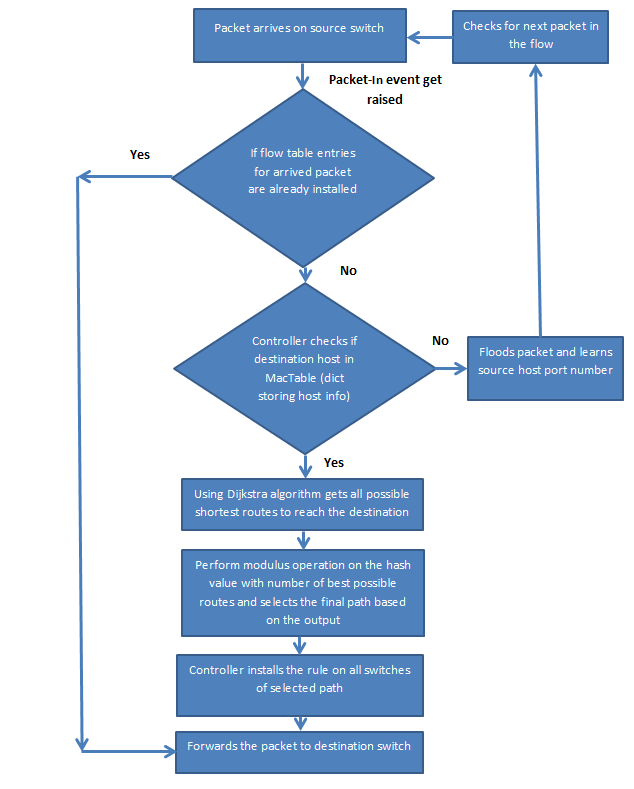
****

Figure : Flow diagram displaying ECMP implementation

APPENDIX B

**TeraSort**

A full TeraSort consists of the following steps:

* Generating the input data via TeraGen.
* Running the TeraSort on the input data.
* Validating the sorted output via TeraValidate.

We are running TeraGen and TeraSort as our experiment only needs these two steps.

**TERAGEN**: Generate the input data

TeraGen generates random input data. The syntax to run TeraGen is as follows:

hadoop jar $HADOOP\_PREFIX/share/hadoop/mapreduce/ hadoop-\*examples\*.jar teragen <number of 100-byte rows> <output dir>

The first parameter specifies the number of rows of input data to generate, each of which having a size of 100 bytes. The data format is as follows: <10 bytes key><10 bytes rowid><78 bytes

filler>

**TERASORT**: Sort the input data

The output from TeraGen is used as an input for TeraSort. The syntax to run TeraSort is as follows:

hadoop jar $HADOOP\_PREFIX/share/hadoop/mapreduce/ hadoop-\*examples\*.jar terasort <input dir> <output dir>

**APPENDIX C**

**Measurement and Analysis**

We have used the ECMP Flow Monitoring module to measure data and time over network. It periodically sends probe-packets to edge switches at a pre-defined rate to obtain the flow statistics. With each query, Flow Monitoring module receives the amount of data sent and time taken for data transfer, and generates csv files that are used to obtain the time and data statistics over network. In this experiment we have measured the time taken to transfer the data over network. When Transfer is done we can get statistics for each flow e.g., maximum byte count per flow, packet count, source port, destination port, time duration from this .csv file. Based on these statistics we have plotted different transfer flows occurring at different time for Hadoop Map Reduce shuffle phase transfer by using per flow statistics (as shown in **Fig 6 & Fig 7**).

**ADDITIONAL RESOURCES**

1. <http://talentica.com/pdf/Hadoop-Map-Reduce-Shuffle-Phase-Measurement.pdf>
2. <https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>
3. <http://code.activestate.com/recipes/119466-dijkstras-algorithm-for-shortest-paths/>
4. <https://www.juniper.net/documentation/en_US/junos13.3/topics/concept/routing-policy-security-ecmp-flow-based-forwarding-understanding.html>
5. <https://tools.ietf.org/html/rfc2992>
6. <http://lib.tkk.fi/Dipl/2011/urn100416.pdf>
7. <http://www.dia.uniroma3.it/~compunet/www/docs/chiesa/ecmp.pdf>
8. <https://www.youtube.com/watch?v=hUfKCbRXFR8>
9. <https://www.youtube.com/watch?v=qx9sJ3O3JM0>
10. <http://cse.unl.edu/~byrav/INFOCOM2011/workshops/papers/p614-xi.pdf>
11. <http://eeweb.poly.edu/chao/docs/public/onpuwmr.pdf>
12. <https://openflow.stanford.edu/display/ONL/POX+Wiki>
13. <https://docs.docker.com/>
14. <https://www.youtube.com/watch?v=aLipr7tTuA4>
15. <https://en.wikipedia.org/wiki/Spanning_Tree_Protocol>