# Enforcing Link Utilization with Traffic Engineering on SDN

Erik de Britto e Silva, Gustavo Pantuza, Frederico Sampaio, Bruno P. Santos,
Luiz F. M. Vieira, Marcos A. M. Vieira, Daniel Macedo
Universidade Federal de Minas Gerais
{erik, pantuza, fredmbs, bruno.ps, lfvieira, mmvieira, damacedo}@dcc.ufmg.br

## ABSTRACT

One of the main problems with the increasing Internet traffic is low throughput caused by bottlenecks, creating critical periods of packet loss. A straightforward solution to this problem is increasing the link capacity, which leads to over provisioning. Unfortunately, it causes poor link utilization during ordinary usage. In some situations we can add more links, adding robustness by traffic balancing.

We propose a traffic engineering solution using Software Defined Networks. Our goal is to enforce link utilization dividing the traffic over multiple links through a simple, easy to implement, low cost, efficient and scalable solution. The solution can be deployed on data centers or at the edge of an Autonomous System.

Single link bottleneck is a problem that can be solved with over provisioning but wasting resources. We propose an alternative solution and performed our experiments on a commercial OpenFlow Switch. The solution met our goal of enforcing link utilization through traffic division. Using three links, we achieved in the worst case 37% more throughput, in the best case almost 57% more compared with a single link

## Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications; C.2.4 [**Distributed Systems**]: Network operating systems—*SDN*

## Keywords

SDN, Traffic Engineering, Network Management, OpenFlow

## 1. INTRODUCTION

Internet global traffic is growing exponentially and there is a forecast of growing from its 51 Exabytes in 2013 to almost its triple until 2018 reaching 132 Exabytes [5] (1 Exabyte = $10^{18}$ bytes). A problem to be solved is how to meet this growing demand in the next years. The usual solution is upgrading the physical link but it isn't scalable and have a higher cost. The physical links can reach its technological speed limit so more new advanced technologies will be needed. A simple option to this problem is to employ Traffic Engineering that is explored in this work.

Traffic capacity of the largest data and information providers on the Internet, as content providers and service providers in the cloud, is dimensioned to attend more than the peak of traffic in its data centers, focusing on the critical periods. Even employing multiple ways of optimizing the traffic delivery, such data centers are over provisioned [22]. Small enterprises can obtain scalable solutions such as the Amazon AWS service [9]. But until here the growing traffic problem is solved only buying more resources, which isn't scalable. An alternative solution to this problem is enforcing link utilization via Traffic Engineering with Software Defined Network (SDN).

If we concentrate only on the largest volumes of outgoing traffic to the Internet of a site or Autonomous System, we will find two situations: first an institution or an enterprise with much outgoing traffic to the Internet from its inside users and second a services or data provider with a large volume of outgoing traffic. It is a two way situation, simply put: the clients sending outgoing traffic to the providers and the providers sending traffic to the clients. Just increasing the outgoing speed of the physical links is increasingly more costly, can be slower delayed for days, weeks or months, it cannot be the best solution depending on the situation. It will cause poor link utilization on normal usage maintaining constant the higher cost just for critical periods.

Surely one link serving all the outgoing and incoming traffic is less robust for whatever institution or enterprise. Beyond the limit of traffic congestion due this only link, it can suffer limits from cost or capacity of the other site of the link out of our control. If we add one or more links we can employ a traffic engineering solution adaptive to the variations of the traffic volume through the activation and distribution of the traffic over all the available outgoing links. It will not only diminish the congestion of the outgoing traffic compared to a single link, but it will improve its incoming flow too.

Using the SDN architecture paradigm through the OpenFlow protocol [16] we can innovate with original solutions based on different abstractions of the network and services.

It allows link control through switches or routers, the elements that perform the forwarding of the network data.

The main contribution of the present work is the definition and the implementation of a Traffic Engineering solution determining network paths or routes flowing trough links on a physical SDN network, so it will enforce link utilization dividing the traffic to all available links. Secondary contributions are to present a simple, easy to implement, low cost, efficient and scalable solution. It can be employed to balance the outgoing traffic of a network to the Internet or inside a data center.

In the next sections we the main aspects of Traffic Engineering and SDN. Then we propose, implement and assess a solution of traffic engineering over SDN to divide the outgoing traffic over multiples links.

## 1.1 Traffic Engineering

Traffic engineering involves the adaptation of traffic to the network conditions, with the goals of attaining a good performance to the user and an efficient utilization of the network resources [4]. The outgoing traffic flow on an Autonomous System (AS) to the Internet, as it happens on an enterprise or even on an Internet Services Provider, is extremely elastic [15], suffering enormous variations depending on certain periods of a day or certain conditions. A solution of traffic engineering employed on this periods can smooth or avoid congestion, resulting in more utilization of all physical links, providing a better economic return of the underlying communication structure and increasing the robustness of it.

The SDN approach is not only just a new approach, it brings new nonexistent aspects, such as complexity reduction and dynamic management in real time.

There are two main needs for a new architectural paradigm on traffic engineering solutions. In the first place there is a need of an architecture capable to differentiating the different traffic of different applications in order to provide an adequate and specific service for each type of traffic, in a very short period of time, in order of some miliseconds. In the second place, to meet the fast increase use of cloud computing, a network management must enhance the utilization of resources to augment the systems performance [3]. SDN is an emerging network architecture that can meet these needs [1].

## 1.2 Software Defined Networks - SDN and OpenFlow

Software Defined Networks - SDN is a network architecture paradigm that separates the Control Plane, now hosted on the SDN Controller, from the Data Plane (or Forwarding Plane) on the SDN Switch, as shown on Figure 1. This functional separation brings enormous benefits turning routing systems more manageable, less complex and more adaptable to the needs of new services. The intrinsic delay of installing a new flow on the SDN switch is our main trouble.

The SDN controller is responsible for the selection of the paths, so all the policies information (each policy is a different Traffic Engineering Method) are on the controller. It

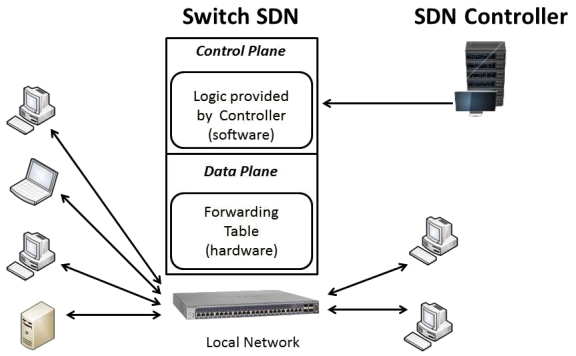


Figure 1: Software Defined Networks - SDN

is possible to implement the function of traffic management centralized or partially centralized on the SDN controller [13].

The OpenFlow protocol [16] is based on the SDN paradigm and has a standardized interface for the communication of the controller with the forwarding element.

# 2. MOTIVATION

In the present work we show some advantages on the deployment of Traffic Engineering over SDN.

The solution of forwarding a huge volume of traffic by dividing it over multiples flows is a simple and efficient method of Traffic Engineering among all its available methods [14]. The Traffic Engineering methods are easy to implement, have light computational overhead and results in good link utilization regarding flow division.

The SDN architecture brings innovation and provides new solutions to network problems with many advantages [20]. Below we enumerate some important advantages:

1. The network isn't proprietary - vendor independent
2. Network control is easy programmable - more agility
3. Replaces dedicated monolithic hardware devices (hardware appliances) by software based devices (software appliances)
4. Easier implementation of network virtualization
5. Cheaper commodity devices
6. Reduction of operational and maintenance costs

The main advantage of SDN is the decoupling of the Data Plane from the Control. This functional separation brings enormous benefits turning routing systems more manageable, less complex and more adaptable to the needs of new services.

We can use the Pox environment [19] to make our SDN Controller with Traffic Engineering. It is best suited for research and development of prototypes. It is an OpenFlow [16] platform with many useful characteristics:

1. Python based and event oriented - high level language
2. Available within the Mininet Simulator [7]
3. Easily installation and configuration in Linux environments
4. Can control real physical network devices as OpenFlow switches

We can insert Traffic Engineering methods in a SDN controller providing an simple, flexible and efficient solution that meets the problem of dealing with large volumes of traffic.

# 3. RELATED WORK

In this work we have the main goal of dividing a volume of traffic among some links using Traffic Engineering methods, many existing solutions focus on traffic load balancing dividing all the requests of one same service over multiple replica servers. Objectively we are concentrated on the link state, in fact the nearest present real load already delivered - the amount of bytes delivered on each link. We can use our solution on networks with just a few number of links as with larger networks with a large number of outgoing links. We can employ this kind of traffic load balancing or traffic division on the outgoing links of a data center parallel to a load balancing of replica servers, it adds robustness, efficiency, and agility.

In the next paragraphs we describe the related work divided into groups: Traffic Engineering on a network partially SDN , Traffic Engineering on a non SDN Network, Traffic Engineering on a SDN network, hybrid load balancing on a SDN network, wildcard balancing and a web traffic balancing on a non structured SDN network with OpenFlow.

*Traffic Engineering on a network partially SDN*: SDN can be incrementally introduced into existing networks to get significant improvements in network utilization as well as to reduce packet losses and delays, as shown by the authors on [1]. Our work have a similar goal of enforcing link utilization reducing packet losses and delays but differently we show a real implementation in a physical environment completely SDN with no simulation.

*Traffic Engineering on a non SDN Network - DIFFSERV/MPLS*: Traffic Engineering is evolving since its started dealing with telephony traffic [3], it is an old area if compared with the new paradigm of SDN. In computer networks the division of flows is a problem similar to maintaining QoS (Quality of Service). We can understand that the QoS depends on the maintenance of a guaranteed flow. Such flow will or will not be distributed or forwarded through different links, routes or paths.

There is a work applying Traffic Engineering in a DIFFSERV/MPLS network similar to the present work, dealing with QoS on a IP test bed [2]. It shows that the traffic of MPLS packets carrying its LSP (Link Switch Path) marked as most important provides prioritization and directs such packets toward the switches creating tunnels. Its drawback is that it establishes static routes so it isn't scalable. In our work, if one route (physical link) turns inactive, the flow division is dynamically adjusted among the still active routes (physical links) so it can scale increasing the flow in each route and decrease the flow on each route when a new route is added. On MPLS the availability of backups routes must be provided even when using Traffic Engineering to maintain QoS (in this case the Traffic Engineering is not able to adjust to network failures using new available routes or isolating the inactive route). Using SDN we add scalability and efficiency to deal with QoS, each packet passes trough a flow without the processing that all packets must pass when on a MPLS network. With SDN it will be faster on the same hardware or devices.

When on DIFFSERV there is a band reservation for each class of service one by one, to make the QoS service priority. It adds the cost of maintaining one record with the available amount of network band to each priority service, all the time in all the network routers. The LSP with Traffic Engineering to maintain the band reservation are called DIFFSERV-TE LSP. It don't scale well and isn't robust as the SDN solution of our present work. In addition, if one class of traffic decrease or is removed, there will be no liberation of the reserved class to the other classes of service in real time. In the DIFFSERV case the IGP protocol advertises the band allocation for each class of traffic and the BC (Bandwidth Constraints), in a more complex and more slower way compared to SDN. If DIFFSERV is using the MAM (Maximum Allocation Model) there is a waste of band, all the classes of traffic are isolated and the band reservation for each one, being guaranteed, cannot be redivided neither shared. However, DIFFSERV using MAN isn't easy to understand and manage. If DIFFSERV is using the RDM (Russian Doll Allocation Model), the traffic classes share the available bandwidth on the best effort model, in other words, it is necessary a preemption to ensure for each traffic class its minimum band guaranty. Without preemption a traffic class can use all the available bandwidth in detriment of the other classes. As its main advantages comparatively, SDN is dynamic, virtually in real time and without the limitations described above of MPLS or DIFFSERV.

*Traffic Engineering on a SDN Network - B4: Google's Global WAN*: One of the successful, earliest and largest application of Traffic Engineering with SDN is B4, the Google's Global WAN. It spreads around the globe connecting its data centers [13]. Although our present work is of a smallest scale of values, complexity and in smallest magnitudes, our goal is to achieve many of the same advantages of B4 on a real implementation of Traffic Engineering on SDN too. Some common characteristics of this work and B4 are:

- Need to control and monitor the available bandwidth
- Maximization of link utilization,
- Motivation to use Traffic Engineering to maintain flows dynamically divided

- Need to a simpler and more efficient network by the use of advanced protocols advanced management features and advanced monitoring
- Dynamic band reallocation in case of failure
- Use of commercial switches
- Decoupling the Control Plane from the Data Plane.

There are some differences, in B4 some loss of packets are tolerable and even some larger delays due to its elastic traffic properties, in the present work we focus only on the actual load of delivered bytes on each link not occurring loss of packets and greater delays as our test bed is a LAN. The intrinsic delay of installing a new Flow on the switch is our main trouble. We note that B4 is not an open solution as our present work.

*Hybrid Load Balancing on a SDN network - DUET*: The Duet solution [10] uses the SDN paradigm. It manages a data center providing cloud services doing load balancing of the server's load. It uses a hybrid set of ordinary non SDN switches and a set of software switches to guarantee the routes redistribution in case of failure. The ordinary switches with an available API, are programmable as physical multiplexers by means of its ECMP (Equal Cost Multi-Path Routing) tables and tunneling. There is a DUET controller monitoring the load and the topology in all switches, it calculates routes on the network and distributes the ECMP programing and tunneling routes over all the physical and software switches. This is the way that the load balancing is made. Thus for all set of client addresses of the services, there is a table duplicated among the physical switches, where changes of load and of topology are monitored and recalculated, forcing the controller to reprogram the tables on all physical and software switches. The software switches receives virtual addresses of the servers (IP and port) and redirects them to a set of physical and software switches. DUET distributes the load among the virtual IPs over multiple software switches mapping some virtual IPs among them. Next, each software switch maps its virtual address and its target clients, to a set of IPs belonging to physical switches of the internal network.

The use o DUET is for a set of switches inside a data center but it too motivates our work with some of its results: high capacity and small latency values, high availability and the attribution of routes dynamically adapting to patterns variations of traffic and network failures. When comparing DUET with solutions completely based on software switches for data centers it provides 10 times more capacity than a software load balancing solution, with the latency reduced 10 times too. With the cost of a fraction of software balancing and quickly adapting to the network dynamics, including its failures, as some of our goals.

*Wildcard Balancing - OpenFlow-based Server Load Balancing Gone Wild*: The wildcard balancing [21] is a solution that installs flows proactively instead of processing the flow on each new incoming packet and after then installing it on an OpenFlow switch. With wildcards all prefixes of the clients of a data center are installed to its replicas servers. This way the load balancing is made. Just a few packets are directed to the controller resulting in a small impact on the network throughput. The services requests from the clients are forwarded directly to the replica server already set. There is a algorithm to this and other to change the rules to new weights of balancing. In our present work we can configure a map of destinations and distribute them over multiple switches to perform the traffic division and it will be the reverse necessity of a data center - to balance its outgoing traffic. A data center have many internal physical links and proportionally few external links (fat tree for example). In our present work we don't use this previous installation of flows but it can be interesting as a future work to avoid bursts of new flows and overload of the OpenFlow controller. In SDN there is a intrinsic latency on a new flow installation so in our work each time we install a flow, say from a client to a server, we install the reverse flow from the server to the client simultaneously to mitigate the latency.

*Plug-n-Serve: Load-balancing web traffic using OpenFlow*: The traffic load balancing done inside a LAN with OpenFlow switches on a real environment, called Plug-n-Serve [11] have some advantages that are goals of our work. It is designed enabling the servers to connect or disconnect on any available switch. In our work there is a focus only on the outgoing traffic to the edge of the network. There are some differences: it considers the network congestion by monitoring the latency and forwards the packets to the routes with the smallest response times (least RTT). There is a monitor of server state and load that are out of our scope, hereafter we can add the RTT - Round Trip Time to access the load of different links to replica servers. It is parameter to assess the delay of each link and we can see it as the quality of the communication to a destination server, we can send more flow to the link with the smallest delay.

In summary, our work implements a real Traffic Engineering solution on an open SDN environment, dividing the flows among links unlike any of the described related works.

# 4. ARCHITECTURE: TRAFFIC ENGINEERING ON SDN

In the next subsections we present the Logical and Physical Design of our architecture that does enforce link utilization with a new approach using Traffic Engineering on SDN.

## 4.1 Logical Design

In this work we implement an OpenFlow 1.0 Controller (our SDN Controller) coded on Python language (Version 2.7) running on the POX environment [19].

We show on Figure 2 an usage example of our solution. It can be seen as a way of performing traffic division, on a network such as an Autonomous System, using an OpenFlow switch controlling on the available links. The switch forwards flows directed to edge router(s) outgoing physical links.

The controller is event-driven, the initial event occurs when the OpenFlow switch connects to the OpenFlow controller. All events take place when a new packet arrives on a switch's port. If there isn't a rule already defined and active on the switch for this packet, the switch generates a Packet_In

Table 1: Servers, Controller and Client hardware configurations

| | NIC MODEL | NIC SPEED Mbits/s | RAM GB | CPU Intel(R) | OS |
|---|---|---|---|---|---|
| ***SRV100*** | Qualcomm L2 Fast Eth. | 100 | 1 | Pentium(R) Dual CPU E2200 @ 2.20GHz | Fedora 20 - 32 bits |
| ***SRV101*** | Qualcomm L2 Fast Eth. | 100 | 2 | Core(TM) 2 Duo CPU E7200 @ 2.53GHz | Fedora 20 - 64 bits |
| ***SRV102*** | Realtek RTL8111 Gigabit Eth. | 1000 | 3.6 | Core(TM) i3 CPU 530 @ 2.93GHz | CentOS 6.5 - 64 bits |
| ***CONTROLLER*** | Marvell 88E8036 Fast Eth. | 100 | 2 | Core(TM) 2 T500 CPU @ 1.66 GHz | Ubuntu 12.04 – 64 bits |
| ***CLIENT*** | Realtek RTL8169 Gigabit Eth. | 1000 | 8 | Core(TM) i5-2310 CPU @ 2.90 GHz | CentOS 6.4 - 64 bits |

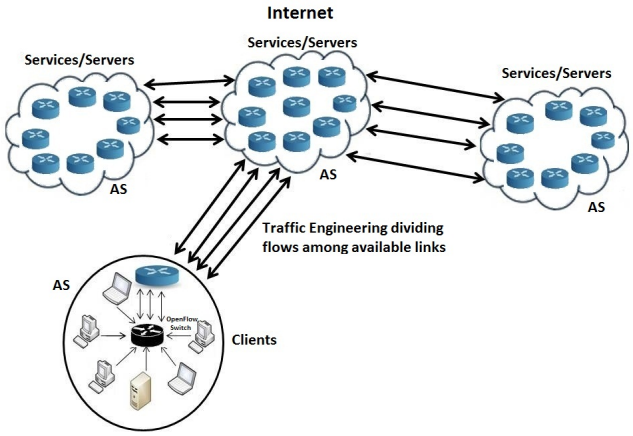


Figure 2: Usage Example

Event to the controller. Then the controller installs or not a rule activating an OpenFlow *FLOW* on the switch. The *FLOW* probably will allow the switch to forward a set of similar packets to the same output port or host destination. After some timers expiration, the OpenFlow Hardware Timeout and Software Timeout parameters, the existing *FLOWS* are discarded from the switch. The load is assessed by the Flow Statistics Request sent by the Controller to the Switch, the Switch responds with a Flow Statistics record to the Controller. With this record the Controller computes the link that will receive the new flow and then installs a new rule on the Switch Flow Table, as show on Figure 3.

Regarding the use of Traffic Engineering for traffic division, a possible technique is Load Balancing, where we consider the flows as the load to be distributed over multiple network links. Such flow distribution follows a policy defined according one or more criteria, below we show the two policies more commonly used:

- Round robin: division by equal number of flows sequentially assigned to each link
- Load based: division by flows according the volume already trafficked on each link or the load until the present time. The flow goes to the link with the minimum amount of bytes already flowed.

The most versatile policy is the Load based, it can divide the traffic with justice, balanced over the links according the current load or even do the division with weights. For example: weighted by the speed where more flows are forwarded over the fastest links. The policies are implemented on parameters to balance the load on the diverse conditions that can be met, e.g.: events, timetables, clients, robustness, security, destination, services, and machines; we can use one or a combination of them to divide flows using Traffic Engineering.

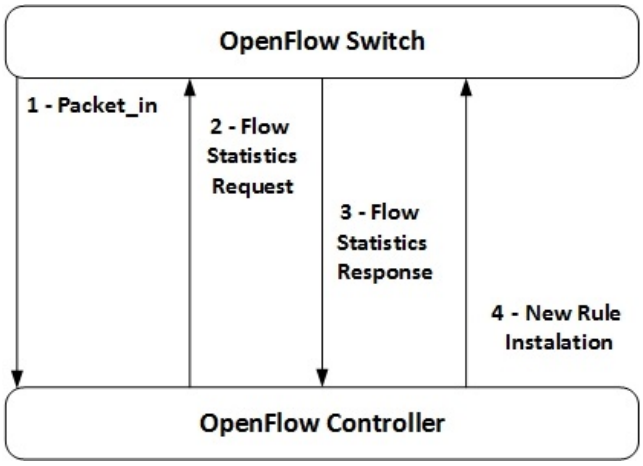


Figure 3: Data and Command Flow Between Switch and Controller - Scenario where a Packet_In is a new flow

### Policies Design

We defined and implemented three policies from Traffic Engineering methods on our SDN controller code:

1. Round robin
2. Load
3. No

### Round Robin policy

The first policy is the most fair and distributes evenly the number of http “GET” load, the Round robin policy shares the number of requests equally among all available servers. The requests are distributed to be serviced sequentially among all servers on a ring data structure so a new request goes to the next server with the least accumulated numbers of requests. This policy even if there is a huge capacity difference among the servers sends to each one the same number of requests. In this case we have a statically configured closed control system using just the event of a new requisition to be assigned to the next server, a increasing counter of each

server requests is the feedback variable. It is a light computing method as we can see on Algorithm 1, where the main computation are made at lines 6 and 7. Respectively, they are a mathematic integer division and a modulus arithmetic operation after an integer addition. At line 5 we receive and handle an event from the switch caused by a new packet that is a valid REQUISITION and at line 8 the controller installs the flow on the chosen link (each server is connected to a different link).

**Algorithm 1:** Round robin

```
1 SERVERS ← [SRV100, SRV101, SRV102]; // Server list
2 SERV-LEN ← 3;
3 INDEX ← 0;
4 SERVER ← 0;
5 while NEW-REQUEST do
6 |   SERVER ← SERVERS[INDEX / SERV-LEN];
7 |   INDEX ← (INDEX + 1) mod SERV-LEN;
8 |   ControllerSendsSwitch(NEW-REQUEST,SERVER);
9 end
```

**Load policy**

The Load policy is based on the OpenFlow switch statistics provided to our OpenFlow controller. It is the total accumulated number of bytes already trafficked between each server individually and the client, the Load Policy sends the actual incoming request to the server with the smallest accumulated number of bytes recorded among all servers. The load algorithm is shown bellow on Algorithm 2. The code of this policy sends a flow statistics request from the controller to the switch (line 5) and continues processing even if the request response isn't timely sent by the switch. It can be the old statistics already stored or really new values. It demands a computing time to sum all flows from the data structures returned by the switch - Update function on line 7, on line 8 it computes the link with the smallest Load or smallest bytes delivered trough MinLD function, and on line 9 it installs the flow on the chosen link. In this case we have a closed control system with the amount of delivered bytes as the dynamic feedback variable.

**Algorithm 2:** Load

```
1  SRV100L ← 0; // Load on SERVER100
2  SRV101L ← 0; // Load on SERVER101
3  SRV102L ← 0; // Load on SERVER102
4  SERVER ← 0; // Server ID of least load
5  while NEW-REQUEST do
6  |   ControllerRequests(FLOW-STATISTICS,TABLE);
7  |   // even if doesn't return an event with a new
   |      table
8  |   Update(TABLE,SRV100L,SRV101L,SRV102L);
9  |   MinLd(SERVER,SRV100L,SRV101L,SRV102L);
10 |   ControllerSendsSwitch(NEW-REQUEST,SERVER);
11 end
```

**No policy**

The No policy is the option we use in the case of just a single server, the first server, so we can assess its performance without a policy and compare with our two implemented policies. In this case we don't have an active policy at all.

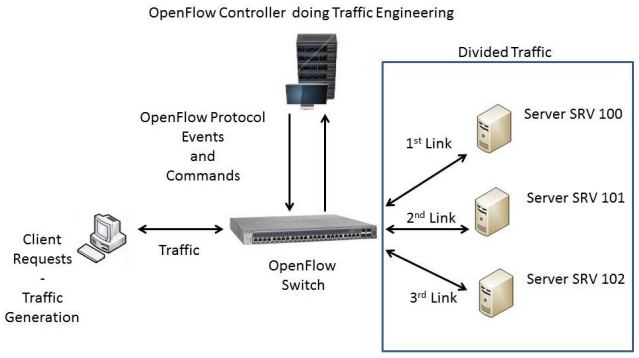


Figure 4: Physical Design

## 4.2 Physical Design

Our testbed has six components: an OpenFlow Linux controller, an OpenFlow switch, a Linux client and three Linux servers. It has the simplest topology, each machine network adapter port is connected to a different switch's port as on Figure 4.

The OpenFlow controller is a Linux machine running Ubuntu Server 12.04 - 32 bits with one ethernet port connected to the OpenFlow switch controller port. On this connection we have a secure channel between the OpenFlow switch and the Openflow controller. This secure channel is on a distinct switch's port number previously assigned on the switch configuration.

The client is a Linux Machine running CentOS - 64 bits. It is generates the requests to the Server(s) and is connected to an ordinary port of the OpenFlow switch at 1 Gigabit/s. The load is generated on it by a shell script starting sequential wget [18] instances, each one requests the same 100 Mbytes file.

The OpenFlow switch is a Hewlett Packard HP commercial switch HP 2920-24G with OpenFlow firmware K 15.5 installed. It has 24 ports at 1 Gigabit/s capacity, configured on HP OpenFlow instance aggregate mode. This OpenFlow aggregate instance connects all the switch's ports on the same network except the OpenFlow Controller port that is isolated from them all.

We used three heterogeneous Linux machines as the http servers. They don't have the same hardware and software. On Table 1, we can see CentOS, Fedora and Ubuntu Servers, 32-bits and 64-bits motherboard's architectures, each one with different CPU, network adapters, memory chips and hard disks.

# 5. EXPERIMENTS

We choose http service using port 8080 with TCP connections to assess the throughput in bytes per second of our experiments. All methods improvements are compared with the throughput gains against one link (one server - No Policy) shown at the end of this paper on Table 5.

As we see in the next sections, we first assess the network latency with httperf [17]. Secondly we assess the network bandwidth with iperf [8]. Finally we use rounds of http GET tests recording the different download rates of each Traffic Engineering method at the client machine.

## 5.1 Network Latency - HTTPERF

One of the main components of the experiment is to get the magnitude and the approximated value of the time needed to establish one connection on the switch flow table by the controller of our experiments. With this we have some parameters of the behavior of physical test bed network.

We got two measurements with httperf tool, one with Load policy active on the controller and the other with the No policy. Httperf performed 2000 connections starting with 100 requisitions increasing 100 for each second, we consider the http server performance to be under 100 miliseconds per call as they are shown on the table 2.

Table 2: Network Latency Measurements

Httperf Latency with LOAD Policy

| | |
|---|---|
| Request Rate | 94.1 req/s (10.6 ms/req) |
| Reply Rate | Average 93.2 ms – Standard Deviation 8.3 |
| | Minimum 81.2 ms – Maximum 99.6 ms |

Httperf Latency with NO Policy

| | |
|---|---|
| Request Rate | 94.8 req/s (10.5 ms/req) |
| Reply Rate | Average 93.8 ms – Standard Deviation 7.4 |
| | Minimum 83.4 ms – Maximum 99.2 ms |

## 5.2 Network bandwidth - IPERF

We got three measurements with iperf tool, each directly from the client to each link/server individually. The duration of each iperf measurement was 10 minutes (200 seconds), as shown on table 3, we can consider the average bandwidth to be near 12 Mbit/s.

Table 3: Network Bandwidth Measurements

| | SRV100 | SRV101 | SRV102 |
|---|---|---|---|
| Bandwidth (Mbits/s) | 12,2 | 12,1 | 12,1 |

## 5.3 Traffic Division by Policies

There was a period before the experiments that many rounds of undocumented tests were made to know the limits of the assembled test bed. We only show the experiments accomplished with no TCP connection resets, connections errors or stalled connections.

On the experiment the client will issue http service requests to the same virtual httpd server address. These requests are considered the service load for the experiment. Each request is a packet sent to the switch, if it doesn't belong to a *FLOW* already activated it must be handled by the controller. The controller, depending on the active policy algorithm, will compute to which server it will forward this packet when activating a new *FLOW*. The servers are running Apache Server 2.4 [12] servicing to each request a file with 100 Mbytes size. The requests are done by a client using the http GET primitive.

On each request for a new *FLOW* - a new http GET request - the controller sends to switch a request to collect the *FLOW STATISTICS* recorded and updated by the switch. At this time the past recorded statistics can be regarded as the newest depending on the statistics arriving time to the controller. After the controller requests the actual switch flow statistics it cannot stop and wait for the event providing the real time statistics from the switch, thus sometimes the flow statistics are lagging and have old values. We have no guarantee if the flow statistics are old or new. Depending on the active policy, we use or not the switch flow statistics as we see below. In the case of one server halt, the controller isolate it and continues dividing the traffic between the available servers.

The HP Openflow switch of our test bed has many limitations when on OpenFlow mode, one interesting is that the flow statistics are refreshed at the switch in 20 seconds intervals. That is the reason we have a round with 21 seconds delay to start a new http request.

At this part of the experiments we made rounds of loops at the client issuing wget instances under the policies No, Load and Round Robin. Each wget requested the download of the same 100 Mbytes file, at the end we recorded the throughput of each wget result.

For each policy we executed the following test rounds as depicted in the table 4. The "0" second delay is obtained when we don't put a "sleep x" on the shell script loop that starts the wget sequentially. The delay between one connection and the start of the next is done with a "sleep x" seconds inside the loop.

Table 4: Test rounds performed on each policy

| Round Number | Number of Sequential Connections | Delay to start a new connection | Size of the file |
|---|---|---|---|
| 1 | 5 | 0 second | 100 Mbytes |
| 2 | 10 | 1 seconds | 100 Mbytes |
| 3 | 10 | 5 seconds | 100 Mbytes |
| 4 | 10 | 10 seconds | 100 Mbytes |
| 5 | 10 | 21 seconds | 100 Mbytes |

### *5.3.1* No Policy

The experiment with No policy as stated before is to record the traffic performance of a single server, our SRV100 machine.

As all these rounds are executed on the same sever there is a smaller value of the Standard Deviation and some outliers can be explained when the Apache Server installs file buffers and minor network disturbances from common standard protocols like DNS and ARP running on an ordinary Linux server.

**No Policy Evaluation** The small test round of 5 connections reached the largest rate with mean of 374.8 Kbytes/s, the 10 connections with 21 seconds delay to start a new connection reached the mean of 262.5 Kbytes/s. Limitations from the server, from the client and mainly from the switch could arise due to many simultaneous connections.

### 5.3.2 Load Policy

The Load policy assigns the next starting request of the client to the server with the smallest load of bytes. This load of bytes is the volume of already trafficked bytes as reported by the OpenFlow switch to the controller. As stated before it can be a old value that is refreshed at 20 seconds interval (HP OpenFlow 1.3 Administrator Guide)[6]. Although it was supposed to be the more efficient method of traffic division it is inaccurate as it can use an old value due to switch limitations.

**Load Policy Evaluation** The small test round of just 5 connections reached the largest rate with the mean of 368.6 Kbytes/s just a small difference compared with the No policy and it can be justified that for smallest traffic there is an overhead of the Load policy that doesn't exist with the No policy. In the rounds among 1 and 10 seconds delay it reaches almost 50% of more throughput as expected when we divide a one way traffic among three links. At the 21 seconds delay round it reaches 342.3 Kbytes/s possibly adding the overhead of the load policy with some errors on the volume already delivered as reported by the switch statistics.

**Time Granularity Drawback** The main drawback intrinsic our experiment environment is the accumulated delay of the following steps:

1. Event generation inside the switch
2. The communication of the event to the controller by TCP/IP ethernet connection at 100 Mbits/s
3. The event catching by the Pox Python application in a generic multipurpose Linux OS commercial PC
4. The event processing and ending with a controller protocol command to the switch
5. The communication of the controller command to the switch again by TCP/IP
6. The switch communication catching and execution system, unknown to us

By the continuous use of our experiment's switch we assume that it is still a all software implementation of an OpenFlow Open Vswitch (the standard OpenFlow Switch) over a ordinary TCP/IP commercial switch hardware. So the delay of our test bed is unexpected higher for a LAN, reaching its minimum greater than 80 ms, as we can see on our measurements on Table 2 of this paper. We are assured of this assumption as the measured bandwidth as shown on Table 3 falls to a peak less that 13 Mbit/s on an advertised 1 Gbit/s commercial switch.

The owner's manual shows that there are speed limitation when using the switch OpenFlow mode so we could not increase the switch internal speed limit greater that 2000 packets per second. Additionally we could not turn off software processing leaving only hardware processing on, when the switch is on OpenFlow mode - this is in fact the main cause of our assumptions about the poor performance on OpenFlow mode.

When there is traffic of 10 Mbit/s, 1,250,000 Bytes are delivered per second, so on each delay at the switch flow statistics we can got a error of this amount times the pace of the arrival of new connections. In the Load Policy with a 20 seconds statistics internal refresh rate we can have a error of 25,000,000 Bytes that can result in a wrong load computation among links or servers.

### 5.3.3 Round robin Policy

The round robin policy assigns sequentially the next connection to the next server distributing the load evenly among all the available servers. In the case of a errors or inaccuracies of the Load policy as stated before, we can expect it to have the most throughput of all the policies on our present work.

**Round robin Policy Evaluation** It reached the largest rate of all the policies on the 5 connections round since it computes the assignment based on roughly the number of present connections it is a faster, lighter and accurate operation reaching the largest mean of 515.6 Kbytes/s. It is in fact a division of small loads to almost free servers, so the overhead is lighter as the load on each server. On the rounds of 1 second and 5 seconds delay, it loses for the Load policy throughput possibly because the overhead of the load compensates for a more equal load distribution among the servers overcharging none of them. At the rounds of 10 and 21 seconds delay it surpasses the Load policy about 10% higher possibly due some errors in the statistics maintaining its old values and more overhead added due the statistics computation and delay at the controller on the Load policy. This overhead doesn't exists with the Round robin policy.

Definitively the accumulated delay as shown in the preceding item **Time Granularity Drawback** has minor computation delay in some cases, just as it happens to the Round robin policy. This policy uses a few integer manipulations internally, just counting the connections already open on each link, not depending on events on the switch beyond the Packet_In (when a new unknown flow arrives = when a new service request arrives).

## 6. CONCLUSION

In this work, we showed one simple, easy and still efficient implementation of Traffic Engineering on a SDN enforcing link utilization. We reached the desired traffic division and link utilization, using only the volume of bytes already delivered. We implemented and validated the whole system in a SDN real physical environment, using the OpenFlow standard protocol. It is a promising solution, specially with the evolution and dissemination of SDN switches.

Table 5:
Policy Results Comparison
The highest average rates in KBytes/s are marked bold

| POLICY | ROUND 1<br>5 conns<br>0 s delay | ROUND 2<br>10 conns<br>1 s delay | ROUND 3<br>10 conns<br>5 s delay | ROUND 4<br>10 conns<br>10 s delay | ROUND 5<br>10 conns<br>21 s delay |
|---|---|---|---|---|---|
| No | 374.8 | 175.4 | 184.4 | 198.8 | 262.5 |
| Load | 368.6 | ***256.6*** | ***286.0*** | 280.4 | 342.3 |
| Round Robin | ***515.6*** | 241.7 | 268.8 | ***312.0*** | ***369.1*** |
| Higher than No | **37.57%** | **46.29%** | **55.10%** | **56.94%** | **40.61%** |

The Policy Comparison is shown below on Table 5 and on

Figure 5. We succeeded on our objective doing a division of traffic among some links, enforcing link utilization towards two methods of Traffic Engineering on a SDN OpenFlow test bed. Although we expected the Load policy to function more smoothly, it was not possible and some differences arose compared with the throughput of the Round robin policy, as we evaluated on the Experiments Section. The Round robin and Load throughput differs about 10% except in one case, we are sure that a better implementation of OpenFlow at the switch will provide a larger difference favoring Load policy when the switch statistics are provided more accurately through a more faster switch response.

As expected, the Load and the Round robin policies had more throughput than the No policy. We note that the No policy is processed by the same controller code that executes all the policies. So it adds an overhead to all new requests depending on the active policy computation.

The link utilization is enforced all the time by the selection of a destination link when a new connection request (or a new flow) arrives at the controller.

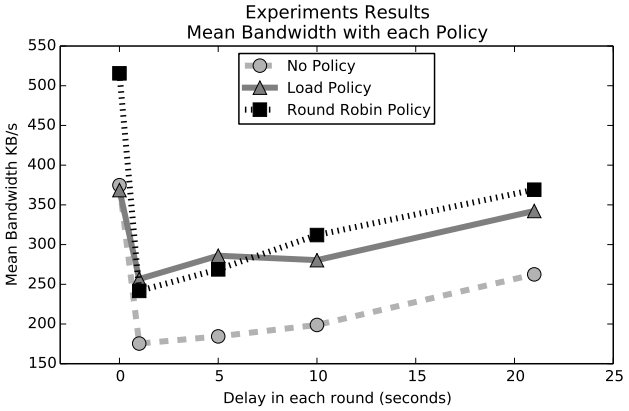


Figure 5: Policy Comparison

# 7. FUTURE WORK

The controller implemented was as simple as possible still reaching good results. We can use environments faster than Pox and add more functionality just in case of needing better response times.

The BGP protocol possibly can be configured to leave some edge routers aside from propagate its routes to the same destiny. Thus we can integrate our solution with a BGP environment, performing the traffic division among those routers faster than BGP alone. We believe that this would be the fastest and cheapest solution for implementing it in Autonomous Systems.

It would be useful to add timetables for each protocol division and priority, and to add users information to directs their individual paths over prioritized links.

In a data center environment, we can investigate new solutions that tolerate a small delay in traffic and improve its throughput, using different hardware.

For small offices or home offices a cheap wireless router can be implemented with the option of dividing the traffic among at least two different Internet links, a DSL or cable modem link and an additional 4G LTE usb cellular modem link. It can be developed over an OpenFlow open source available firmware, for existing commercial wireless routers.

# References


[1] S. Agarwal, M. Kodialam, and T. Lakshman. Traffic engineering in software defined networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2211–2219, April 2013.

[2] I. F. Akyildiz, T. Anjali, L. Chen, J. C. De Oliveira, C. Scoglio, A. Sciuto, J. A. Smith, and G. Uhl. Invited a new traffic engineering manager for diffserv/mpls networks: Design and implementation on an ip qos testbed. *Comput. Commun.*, 26(4):388–403, Mar. 2003.

[3] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou. A roadmap for traffic engineering in sdn-openflow networks. *Computer Networks*, 71(0):1 – 30, 2014.

[4] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and I. Widjaja. RFC 3272: Overview and Principles of Internet Traffic Engineering. Technical report, IETF, 2002.

[5] CISCO. White paper - cisco visual networking index: Global mobile data traffic forecast update, 2013–2018. Technical report, CISCO, June 2014.

[6] H. P. Company. Hp openflow 1.3 administrator guide wired switches k/ka/kb/wb 1 5.15, 2015.

[7] R. de Oliveira, A. Shinoda, C. Schweitzer, and L. Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*, pages 1–6, June 2014.

[8] J. D. et al. Iperf project. 2008.

[9] F. Ferraris, D. Franceschelli, M. Gioiosa, D. Lucia, D. Ardagna, E. Di Nitto, and T. Sharif. Evaluating the auto scaling performance of flexiscale and amazon ec2 clouds. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 423–429, Sept 2012.

[10] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 27–38, New York, NY, USA, 2014. ACM.

[11] N. Handigol, S. Seetharaman, M. Flajslik, and R. Johari. Plug-n-serve: Load-balancing web traffic using openflow. *Demo at ACM SIGCOMM*, 2009.

[12] J. Jagielski. Apache httpd v2.4: Hello cloud: Buy you a drink?, Nov 2011.

[13] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, Aug. 2013.

[14] G. Leduc, H. Abrahamsson, S. Balon, S. Bessler, M. D'Arienzo, O. Delcourt, J. Domingo-Pascual, S. Cerav-Erbas, I. Gojmerac, X. Masip, A. Pescapè, B. Quoitin, S. Romano, E. Salvadori, F. Skivée, H. Tran, S. Uhlig, and H. Ümit. An open source traffic engineering toolbox. *Computer Communications*, 29(5):593 – 610, 2006. Networks of Excellence.

[15] C. L. Lim and A. Tang. Traffic engineering with elastic traffic. In *Global Communications Conference (GLOBECOM), 2013 IEEE*, pages 3095–3101, Dec 2013.

[16] 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.*, 38(2):69–74, Mar. 2008.

[17] D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[18] H. Niksic et al. Gnu wget 1.16.1, Nov 2014.

[19] L. Rodrigues Prete, C. Schweitzer, A. Shinoda, and R. Santos de Oliveira. Simulation in an sdn network scenario using the pox controller. In *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*, pages 1–6, June 2014.

[20] A. Valdivieso Caraguay, L. Barona Lopez, and L. Garcia Villalba. Evolution and challenges of software defined networking. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, pages 1–7, Nov 2013.

[21] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[22] D. Xu, B. Fan, and X. Liu. Efficient server provisioning and offloading policies for internet datacenters with dynamic load-demand. *IEEE Transactions on Computers*, 99(PrePrints):1, 2014.