

**BSc in Telecommunications Engineering**

**TEL3214**

**Computer Communication Networks**

**Lecture 12**  
**Software Defined Networks**

Eng Diarmuid O'Briain, CEng, CISSP



Department of Electrical and Computer Engineering,  
College of Engineering, Design, Art and Technology,  
Makerere University

Copyright © 2017 Diarmuid Ó Briain

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

## Table of Contents

<b>1. Introduction.....</b>	<b>5</b>
1.1 Why the need for change.....	6
<b>2. The Data Centre problem.....</b>	<b>8</b>
2.1 SDN in the Data Centre.....	9
<b>3. SDN Architecture.....</b>	<b>10</b>
<b>4. SDN operation.....</b>	<b>11</b>
4.1 Flow Tables.....	13
4.2 Group Tables.....	13
4.3 Meter Tables.....	13
<b>5. SDN Controllers.....</b>	<b>15</b>
5.1 SDN Applications.....	16
5.2 Link Discovery Module.....	17
5.3 Topology Manager.....	17
5.4 Virtual Routing Engine (VRE).....	18
<b>6. Mininet.....</b>	<b>19</b>
6.1 Getting the SDN Virtual Machine.....	19
6.2 Build a Mininet test network.....	22
6.3 Configuring hosts.....	29
6.4 Configuring links.....	29
<b>7. OpenFlow traffic review.....</b>	<b>31</b>
7.1 Webserver test.....	38
<b>8. POX Controller.....</b>	<b>41</b>
8.1 Running POX.....	41
8.2 Testing POX.....	42
<b>9. Project Floodlight.....</b>	<b>43</b>
9.1 Running Floodlight.....	43
9.2 Testing Floodlight.....	43
<b>10. OpenDaylight.....</b>	<b>44</b>
10.1 Running ODL karaf.....	44
10.2 Installing openDaylight User eXperience (DULX) features.....	44
10.3 Testing the ODL installation.....	45
10.4 DLUX User interface.....	46
<b>11. Custom Topologies.....</b>	<b>47</b>
11.1 Create a custom topology.....	50
<b>12. Custom script to ODL remote controller.....</b>	<b>55</b>
12.1 Run OpenDaylight.....	55
12.2 OpenDaylight User Experience (DLUX).....	56
12.3 Start Mininet network.....	56
<b>13. North Bound Interface (NBI).....</b>	<b>60</b>
13.1 Frenetic.....	60
<b>14. Networks Function Virtualisation (NFV).....</b>	<b>62</b>
14.2 Open Platform NFV.....	66
14.3 Ongoing research.....	67
14.4 Software Defined WAN (SD-WAN).....	68
<b>15. The future of Broadband.....</b>	<b>70</b>
<b>16. SDN Lab.....</b>	<b>71</b>
<b>17. List of Abbreviations.....</b>	<b>72</b>

## Illustration Index

Illustration 1: Elastic Compute to Elastic Network.....	6
Illustration 2: Data Centre racks.....	8
Illustration 3: Traditional Data Centre.....	8
Illustration 4: SDN in the Data Centre.....	9
Illustration 5: SDN Architecture.....	10
Illustration 6: SDN Architecture.....	10
Illustration 7: SDN Operation.....	11
Illustration 8: OpenFlow switch tables.....	13
Illustration 9: SDN Routing islands.....	16
Illustration 10: SDN Routing service.....	17
Illustration 11: Appliance to import.....	19
Illustration 12: Appliance settings.....	20
Illustration 13: VirtualBox Dashboard.....	20
Illustration 14: Bridged adapter.....	21
Illustration 15: Basic test network.....	23
Illustration 16: Testing Mininet network.....	26
Illustration 17: xterm window over SSH.....	27
Illustration 18: Wireshark OpenFlow traffic.....	31
Illustration 19: SDN Action.....	31
Illustration 20: SDN Flow-MOD.....	33
Illustration 21: SDN Flow-MOD #2.....	34
Illustration 22: karaf.....	44
Illustration 23: Dlux login.....	46
Illustration 24: DLUX Topology.....	46
Illustration 25: Mininet custom topology example.....	50
Illustration 26: Test network with ODL.....	55
Illustration 27: Dlux topology dashboard.....	59
Illustration 28: Pyretic.....	60
Illustration 29: Kenetic.....	61
Illustration 30: Network Function Virtualisation.....	62
Illustration 31: NFV Ecosystem.....	63
Illustration 32: vCPE.....	64
Illustration 33: Software Defined WAN (SD-WAN).....	68
Illustration 34: SDN Lab.....	71

## 1. Introduction

Over the last ten years or so the landscape in computing has changed dramatically with the Cloud, large-scale data centres and virtualisation. Over the last few years networks have increased in speed and there has been a convergence on Ethernet as the standard for all links, to the point that the difference between Local Area Network (LAN), Metropolitan Area Network (MAN) and Wide Area Network (WAN) has diminished dramatically. What has not changed in that time however is the core switching and routing functions which are generally delivered on a hardware based stand-alone device that is self sufficient in terms of the data it switches or routes and the control necessary to make that happen. In a bid to outdo each other to maintain advantage in the market companies like Cisco, Juniper and HP have loaded their devices with features that over time have resulted in network devices that rely on aged protocols like Border Gateway Protocol (BGP) to communicate and networks have levels of header encapsulation that eat into the Maximum Transfer Unit (MTU) size of the packets. This layering of abstractions on top of other abstractions is not conducive to Network Management, where traffic patterns are decided within each layer independently.

It is not uncommon for a packet to arrive at an Internet service provider (ISP) network with a Virtual LAN (VLAN) tag, the ISP adding another VLAN tag before passing the packet to an upstream ISP who adds an Multi-Protocol Label Switching (MPLS) header as it is switched across their IP network.

While the underlying networks have converged towards the all Ethernet / all Internet Protocol (IP) model, in some form the number of services have increased rapidly. In the past ISPs provided Internet Access in the form of Broadband and possibly layered a voice service either as a circuit switched out of band telephone line or as a Voice over Internet Protocol (VoIP) service with some packet priority mechanism to give Quality of Service (QoS). In more recent years this service set is increasingly being supplemented with TeleVision over IP (TVoIP) that more often than not requires a separate Set Top Box (STB) for its provision.

Network resilience is an important characteristic to ISP network designers, yet duplication of service paths may give the appearance of redundancy where there is in fact none. For example a tier three ISP getting a service from two independent ISPs, one a tier two provider and the second, an incumbent tier one provider, all to ensure path resilience for a customer. The tier one ISP provides an MPLS circuit at the customer end and drops that off at a data centre at a major city, the tier two ISP provides a Network Termination Unit (NTU) at the customer end and drops the traffic off at another data centre in a major city. However there is generally a limit on the number of actual fibre paths between major cities or even the circuit supplied by the tier two ISP many in fact have a portion passing through the tier one network. From the ISP providing the service to the customer there is the appearance of separate paths however a strategic failure of a fibre bundle could in fact expose this.

### 1.1 Why the need for change



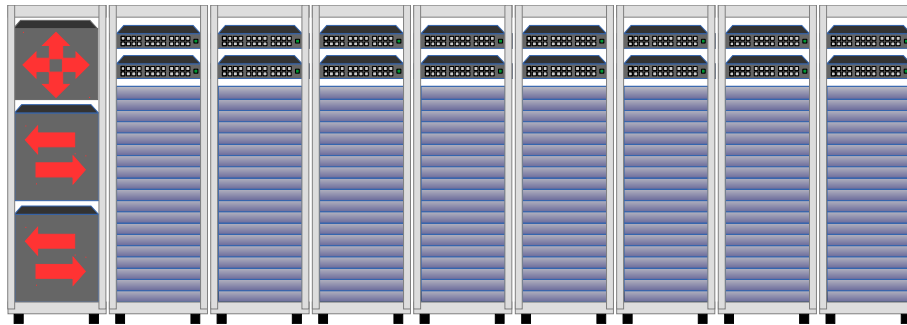
*Illustration 1: Elastic Compute to Elastic Network*

In the not too distant past Network Engineers were taught to design networks according to the 80/20 rule where 80% of traffic stays within the local network and should be switched, whereas 20% left the local network to the Internet. This has completely changed with 80% leaving the local network, mainly to access cloud based services and only 20% remains within the local network. This has transformed today's traffic patterns.

As virtualisation and led to the cloud and AWS launching the concept of Elastic Compute the network has not adjusted until now. SDN promises Elastic Network to match Elastic Compute and support the move to cloud based services, the rise in big data and the Internet of Things (IoT).

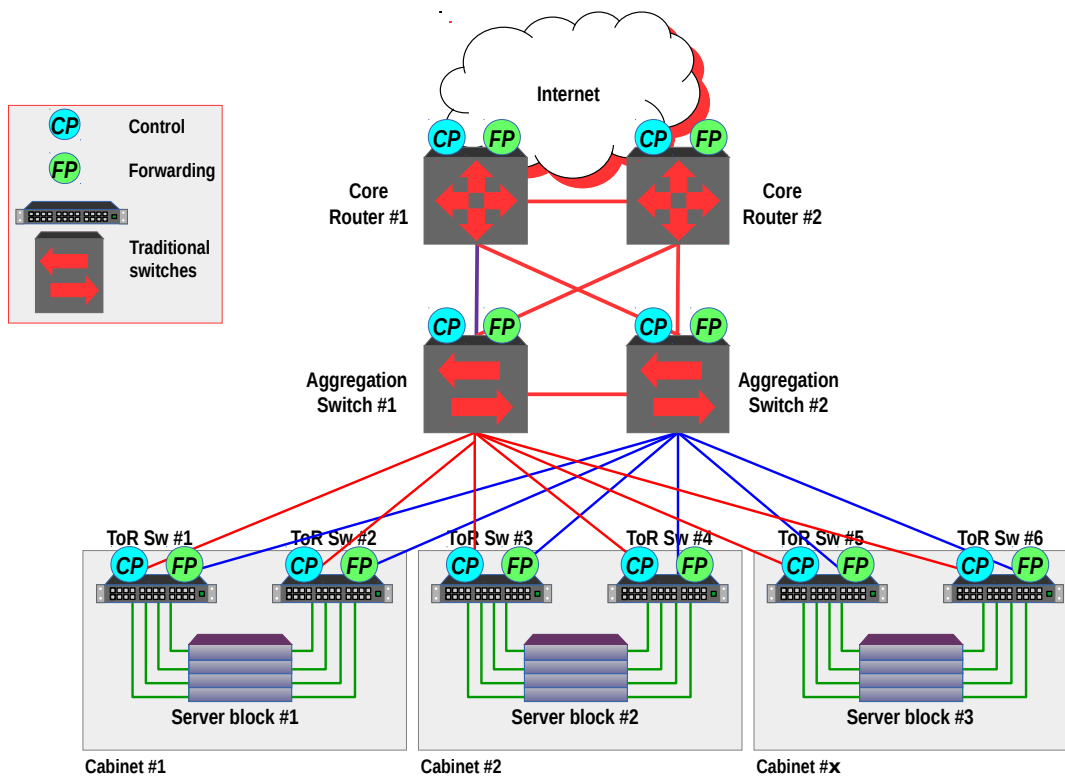
Current network models discourages change, Companies are finding they are unable to scale and are dependent on a small number of vendors.

## 2. The Data Centre problem



*Illustration 2: Data Centre racks*

Traditional networking has proven a problem in Data Centres for some time. Consider Illustration 2 where racks of servers have two Top of Rack (ToR) switches that are linked to each server in the rack and to each of the aggregation switches that are in turn linked to core routers. As can be see in Illustration 3 each switch and router in this scenario has responsibility in the Forwarding Plane (FP) moving frames from port to port as well as in Control Plane (CP) identifying the paths which frames should be forwarded over.



*Illustration 3: Traditional Data Centre*

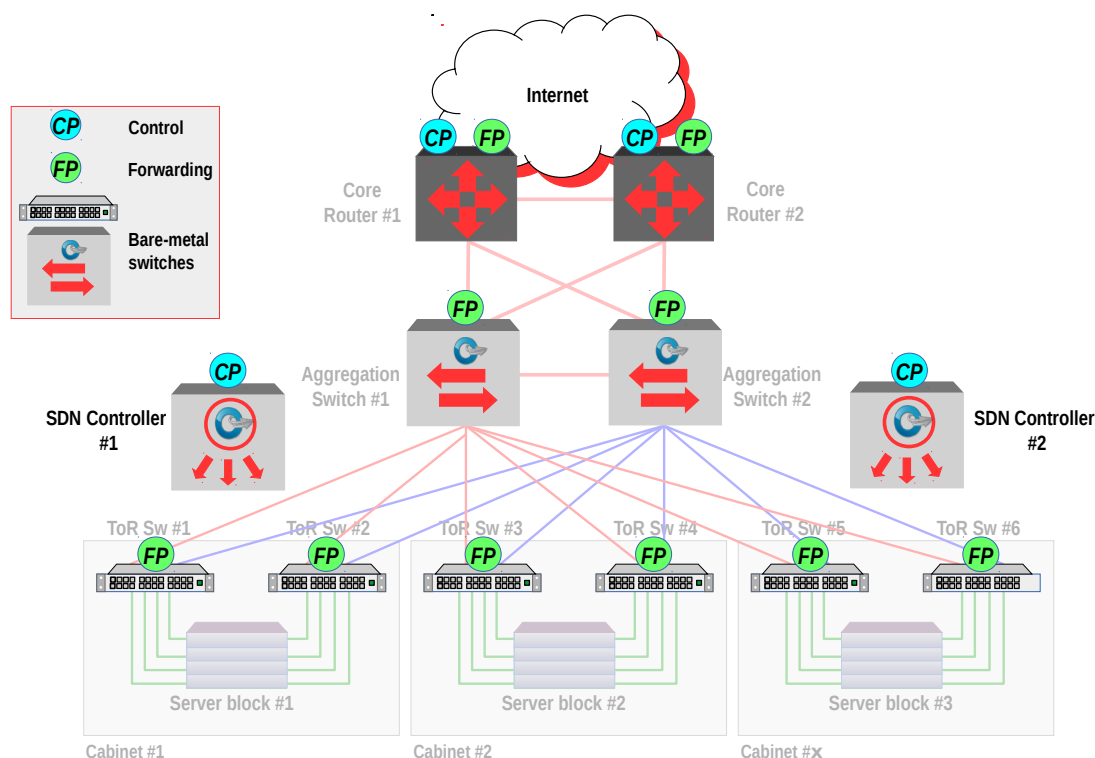


Why ?

Would the Data Centre be better off with some massive single switch into which all servers are connected ?

No practical, no certainly not, well SDN solves this question.

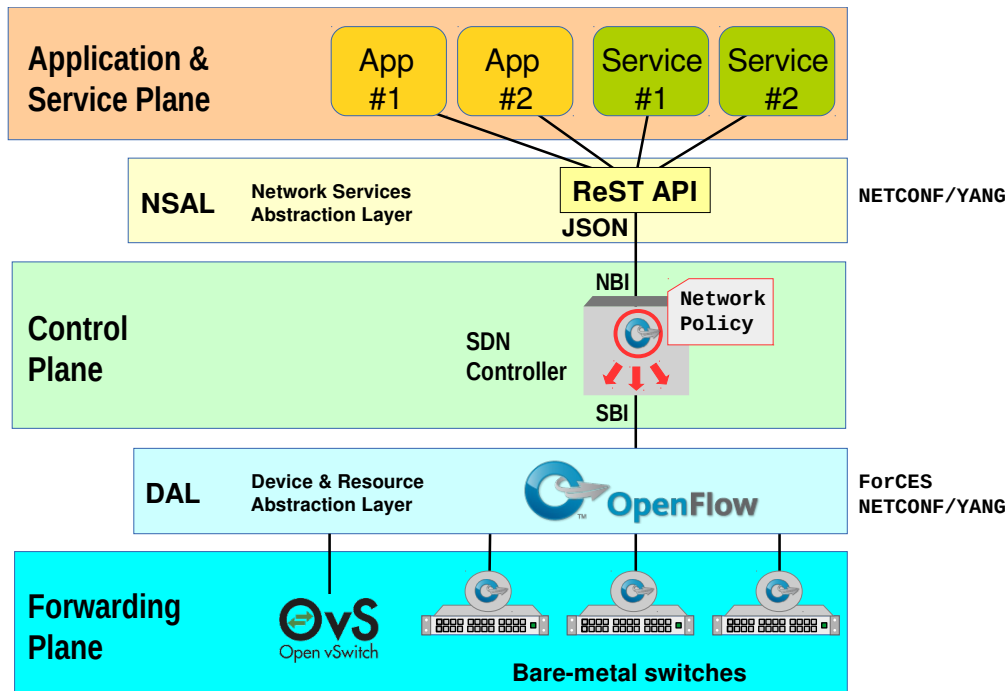
## 2.1 SDN in the Data Centre



*Illustration 4: SDN in the Data Centre*

Illustration 4 Shows the how SDN changes the data centre. While each rack continues to have two ToR switches they are bare-metal switches with responsibility for the FP only. A communications channel called OpenFlow to the two SDN Controllers provide a link to the CP. The SDN Controllers make switching decisions based on a network policy and forward decisions to the switches. In this way the Data Centre has the one big switch in terms of control but the hardware is distributed.

### 3. SDN Architecture



*Illustration 5: SDN Architecture*

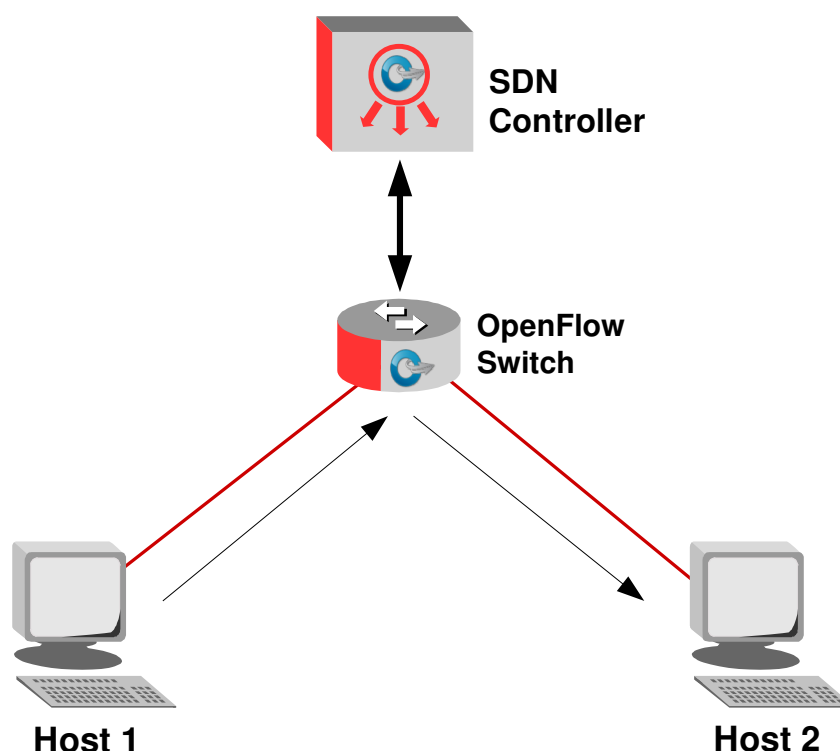
Communication between the SDN Controller and the bare-metal switches is an essential component of SDN. This is achieved over the Device and Resource Abstraction Layer (DAL) on the South Bound Interface (SBI) of the SDN Controller. OpenFlow is a simple protocol that the SDN Controller uses over a secure channel (Transport Layer Security (TLS) on TCP port 6633 to modify the flow table in a supporting switch.

Further work on the OpenFlow protocol and an initial specification in 2008 for a virtual Switch daemon (vswitchd), produced for the GNU/Linux kernel led to the Open virtual Switch (OvS). OVS offers a soft switch solution that operates over OpenFlow and can be used in virtualised situations where a physical switch is unnecessary. The overall SDN architecture is demonstrated in Illustration 5 with the Forwarding and Control planes, linked by OpenFlow offering services from the Application plane via a RESTful Application Program Interface (API).

OpenFlow has also evolved, coming under the management of the Open Networking Foundation (ONF) founded in 2011 for the promotion and adoption of SDN through open standards development. OpenFlow has evolved to version 1.5.1.

## 4. SDN operation

In order to understand how SDN switching works consider a traditional switch. A frame arrives at a switch port, the switch inspects the frame header and determines if it has a record for the destination MAC address. If it doesn't, then the frame is forwarded on all ports and the source port MAC is recorded upon which the port the frame was received. If it does then the frame is only forwarded to the known port associated with the destination MAC. All these decisions are made in the individual switch.



*Illustration 7: SDN Operation*

In an SDN Network when as is shown in detail in the mininet example labs below, when a frame arrives at an OpenFlow Switch, it performs a table entry check and if it finds that it has a *table-miss*, which means there is no flow entry associated with this frame, it will send an OpenFlow *Packet In (OFPT\_PACKET\_IN)* message to the SDN Controller with a unique Buffer Identifier for a decision. The SDN Controller responds to the OpenFlow Switch using the same Buffer Identifier with the decision to *Output to switch port* on all ports.

The response from the second host arrives at the OpenFlow Switch and is given a new Buffer Identifier, again there is a table-miss so the OpenFlow Switch sends an OpenFlow *OFPT\_PACKET\_IN* message to SDN Controller. The SDN Controller now sends an OpenFlow *Flow MOD* to the OpenFlow Switch to add an entry to the Flow Table for this now known traffic. Subsequent similar packets are then forwarded automatically by the OpenFlow Switch until the idle time-out of 60 seconds has been exceeded and then the process must be repeated.

The next packet in from the original host triggers another *OFPT\_PACKET\_IN*. This time the SDN Controller knows the port that the second host is connect on, so it sends an OpenFlow *Flow MOD* to the OpenFlow Switch to add an entry to the Flow Table for the traffic. Subsequent similar packets are then forwarded automatically by the OpenFlow Switch until the idle time-out of 60 seconds has been exceeded and then the process must be repeated.

In the example the frame that arrived was unmatched by the OpenFlow Switch. It is typical for the SDN Controller to *pre-load* the OpenFlow Switch with flows. It is also not simply the MAC fields in the frame header nor the IP Addresses in the packet header. Flows can be based on a multitude of values within the overall frame and its sub packet and even transport session headers:

- The port the frame arrived on
- The source Ethernet port
- The destination Ethernet port
- The source IPv4 or IPv6 address
- The destination IPv4 or IPv6 address
- IPv6 Flow Label
- IPv6 Extension Header pseudo-field
- ICMPv6 type or code
- Target IP address, source or target link layer address in IPv6 Neighbour Discovery (ND)
- VLAN Identifier (VLAN-ID)
- VLAN Priority Code Point (PCP)
- Differentiated Services (DiffServ) Code Point (DSCP)
- IP Header Explicit Congestion Notification (ECN)
- IPv4 or IPv6 Protocol number
- TCP Source, Destination port or flags
- UDP Source or Destination port
- Stream Control Transmission Protocol (SCTP) Source or Destination port
- ICMP Type or Code
- ARP Opcode
- MAC Addresses in ARP payload
- IP Addresses in ARP payload
- The LABEL, Traffic Class (TC) or Bottom of Stack (BoS) in first MPLS Shim header
- User Customer Address (UCA) field in the first Provider Backbone Bridge (PBB) instance tag

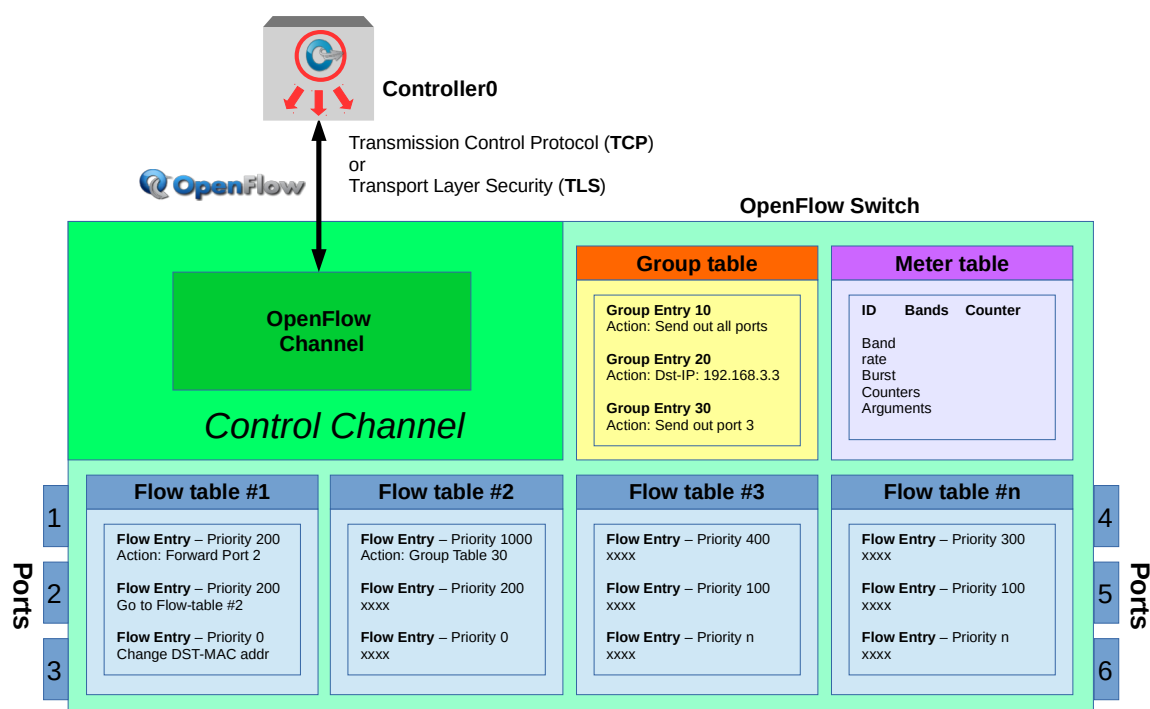


Illustration 8: OpenFlow switch tables

## 4.1 Flow Tables

In the forwarding instructions the controller specifies to the OpenFlow switch the group of parameters used to define individual flows and what action to carry out on frames that match the flow. The OpenFlow switch has as can be seen in the diagram multiple *Flow Tables*. In the initial OpenFlow version this was limited to a single table however the Application Specific Integrated Circuit (ASIC) hardware in switches were capable of much more so later versions of the OpenFlow protocol allowed for multiple tables which improves performance and scalability.

## 4.2 Group Tables

OpenFlow protocol also added *Group Tables* consisting of group entries. A flow entry can be pointed to a group, which enables OpenFlow to have additional methods of forwarding:

- *SELECT*: for load sharing and redundancy
- *ALL*: for multicast or broadcast forwarding
- *INDIRECT*: which allows for multiple flow entries to point to a common group ID
- *FAST FAILOVER*: which enables the switch to change forwarding without requiring communication with the SDN Controller in the event of a port failure

## 4.3 Meter Tables

The Meter Tables consists of *per-flow meters* used by OpenFlow to implement QoS. Each *per-flow meter* measures the rate of frames assigned to it and controls the rate of those frames. Each meter consists of one or more meter bands which specify the rate at which the band applies and how frames are processed. Each meter band is identified by its rate and contains:

- *Band type*: defines how packet are processed
- *rate*: defines the lowest rate at which the band can apply
- *burst*: defines the granularity of the meter band
- *counters*: updated when packets are processed by a meter band
- *type specific arguments*
  - *drop*: discard the packet. Can be used as a rate limiter band
  - *dscp remark*: increase the drop precedence of the DSCP field in the IP header. Can act a simple DiffServ policer

## 5. SDN Controllers

Now that a standard SBI existed the evolution of controllers as well as work on a NBI became important. Network Operating System (NOX) a C++ based first generation controller was developed by Nicira Networks and donated to the research community. A Python version of the NOX Controller called POX was developed for rapid development and prototyping.

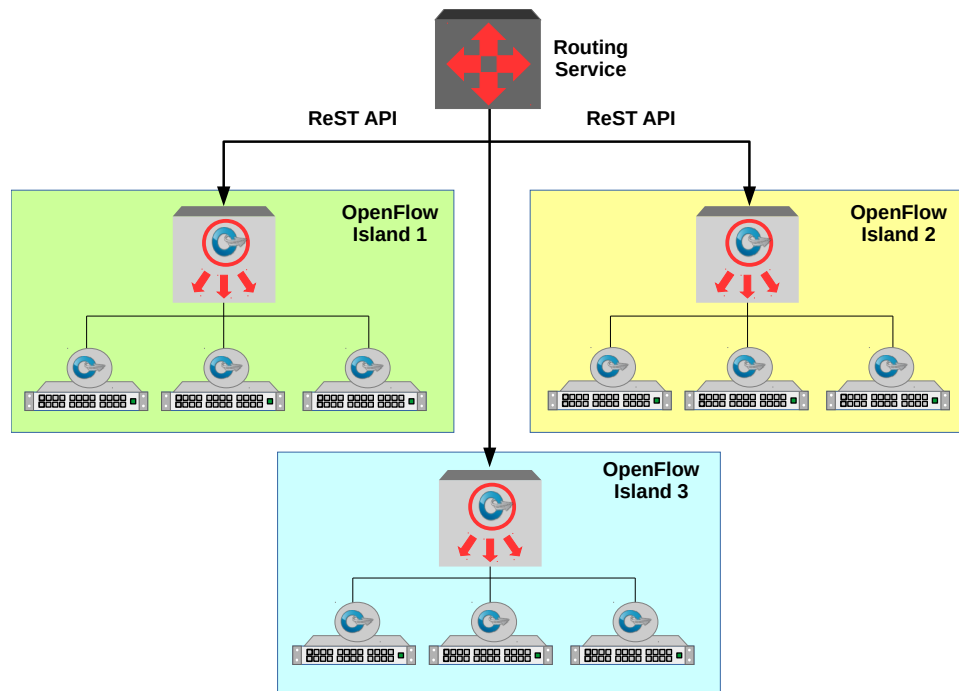
Another Python based SDN Controller is 'RYU' (Japanese: flow), available under the Apache 2.0 license has OpenStack integration and supports OpenFlow 1.0 – 1.4 plus Nicira extensions. Ryu has a Web Server Gateway Interface (WSGI) and by using this function, it is possible to create a REST API (called RESTful API), which is a useful NBI link with other systems or browsers in an application tier.

A commercial grade Java SDN Controller developed by Big Switch Networks evolved from a Java based research SDN Controller called Beacon as Project Floodlight. This project code is also Apache 2 licensed. It, like RYU, has a RESTful API.

The other big SDN Controller is a Linux Foundation collaborative project called OpenDaylight (ODL), developed in Java. The latest version of the platform designated Helium is a follow on from the first release of ODL called Hydrogen. This project was designed to take advantage of existing Linux Foundation projects, like integration with OpenStack as well as developments with high availability, clustering and security. ODL OpenFlow plugin supports OpenFlow versions 1.0 and 1.3. Like RYU and Project Floodlight an application tier is made possible through a RESTful API as well as an Authentication, Authorisation and Accounting (AAA) AuthN filter.

### 5.1 SDN Applications

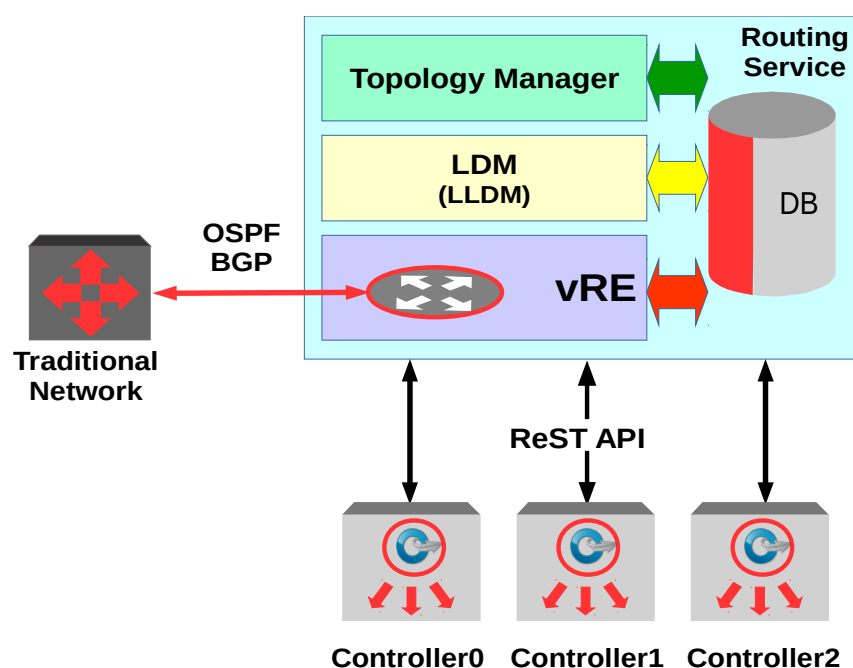
#### 5.1.1 SDN Routing Service



*Illustration 9: SDN Routing islands*

As we have seen the SDN Controller manages switches on the CDPI using OpenFlow protocol. On the NBI the SDN Controller interfaces using REST API with application services. In traditional networks networks are linked by routers. In an SDN Network groups of switches are managed by a controller and this is called an *OpenFlow Island*. In an SDN Network the *Flows* that the SDN Controller send to the individual switches are controlled by SDN Applications. One typical example is the routing service.





*Illustration 10: SDN Routing service*

Routing in this case is an SDN Application and consists of:

- Link Discovery Module (LDM)
- Topology Manager
- Virtual Routing Engine (VRE)

## 5.2 Link Discovery Module

The LDM discovers and maintains the status of all physical links on the network. When OpenFlow Switches discover other switches via Link Layer Discovery Protocol (LLDP) this information is passed to the Link Discovery Module (LDM). Additionally when unknown traffic is discovered by an OpenFlow Switch as described above the SDN Controller also passes this to the LDM. In this way the LDM derives the information to build a picture of the overall network topology as a Neighbour Database.

## 5.3 Topology Manager

The Topology Manager builds the topology from the Neighbour Database. It generates the logical OpenFlow Islands and determines the shortest path between OpenFlow nodes. From this the Topology Manager can build the individual topology databases for the controllers which contain the shortest paths plus alternate paths to each OpenFlow node or hosts connected to them.

## 5.4 Virtual Routing Engine (VRE)

The function of the VRE is to allow SDN networks interoperate with traditional networks. It builds a virtual networking topology to represent the SDN network to the traditional networks using traditional routing protocols like Open Shortest Path First (OSPF) and Border Gateway Protocol (BGP).

While it is essential for SDN to have an application like the Routing Service to handling routing within the SDN and to interact with traditional networks, the SDN architecture lends itself readily to newer SDN Applications that can interact with the Controller over the REST API and thereby influence the OpenFlow Switches in new and imaginative ways not possible in today's traditional networks.

## 6. Mininet

*Mininet* is a project that creates a virtual network on a computer, a *network emulator*. On it it is possible to develop a network of hosts, switches, routers and links based on a single GNU/Linux kernel. Mininet uses Linux Containers (LXC) lightweight virtualisation to allow for experimentation with SDNs and SDN Controllers. For example a SDN Controller can be given a network of devices to work with and because they are based on the GNU/Linux kernel behave exactly as a stand-alone GNU/Linux device.

To allow for experimentation establish a Mininet VM image to work with. Install Oracle VirtualBox as a hypervisor first (<https://www.virtualbox.org>).

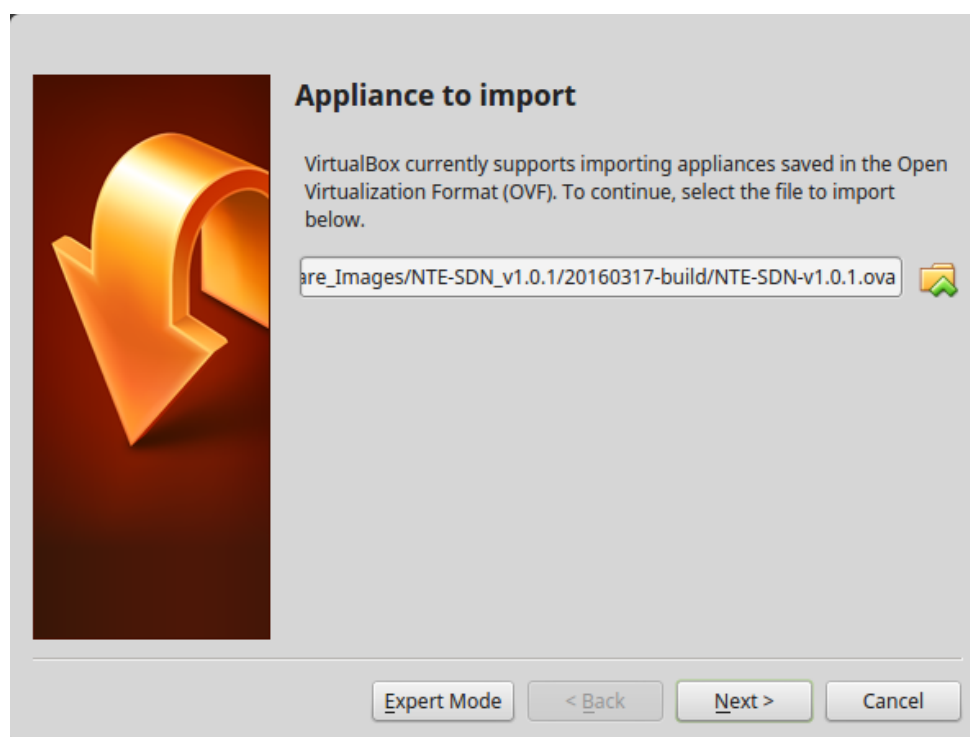
### 6.1 Getting the SDN Virtual Machine

Download the course NTE-SDN VM .ova from the course page.

Extract the *NTE-SDN-v1.0.1.ova* file to the computer.

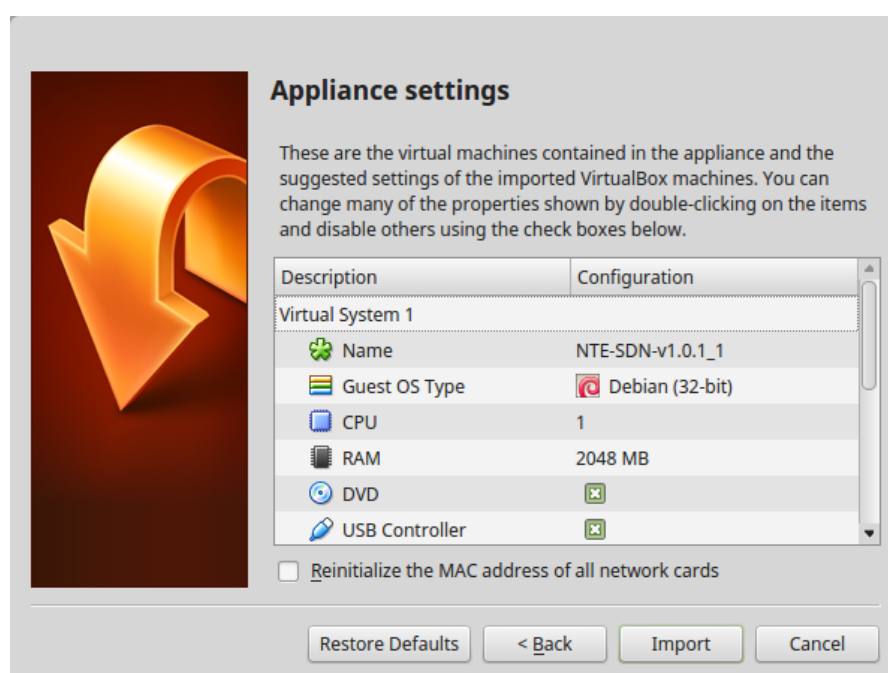
If you wish to understand how this VM was built the refer to the Appendix 02.

**File > Import Appliance**



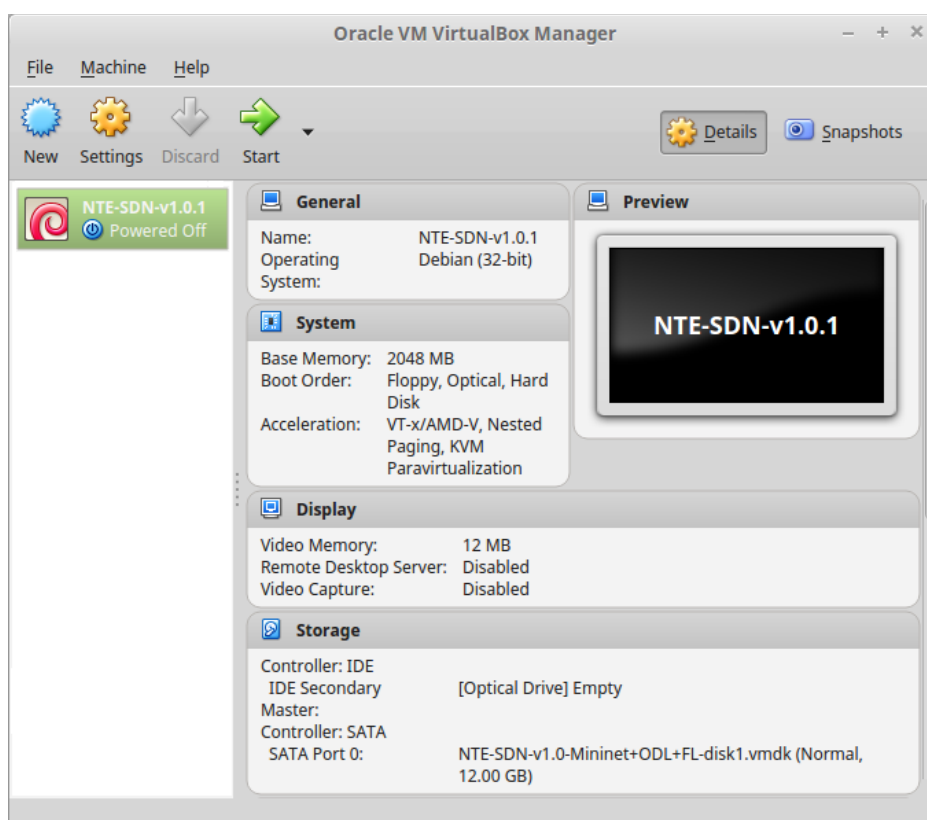
*Illustration 11: Appliance to import*

Select the ***NTE-SDN-v1.0.1.ova*** and click ***Next >***.



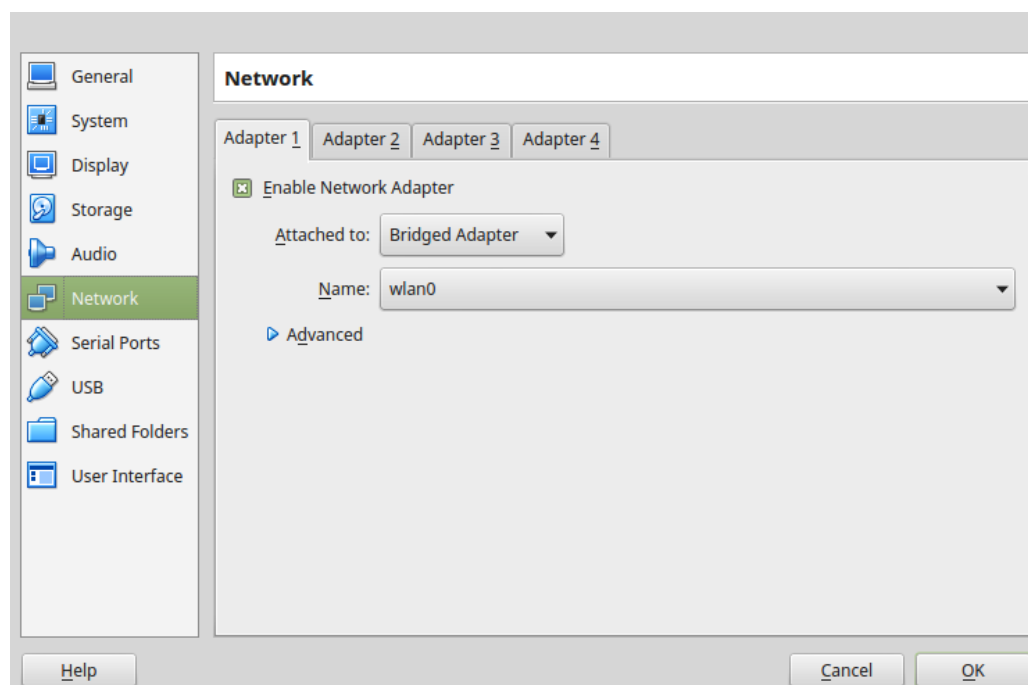
*Illustration 12: Appliance settings*

Check the tickbox for “**Reinitialise the MAC address of all network cards**”  
Select **Import**.



*Illustration 13: VirtualBox Dashboard*

The new SDN VM is created. Before starting it configure the Network interface as a **Bridged Adapter**. To do this Right click on the newly created VM, select Settings and Network, make the change from *Attached to: NAT* to *Attached to: Bridged Adapter*. Select **OK**.



*Illustration 14: Bridged adapter*

Get the IP address of the mininet VM.

```
sdn@SDN-i386:~$ ip addr list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
    default qlen 1000
    link/ether 08:00:27:e6:cb:e2 brd ff:ff:ff:ff:ff:ff
    inet 192.168.22.83/24 brd 192.168.25.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fee6:cbe2/64 scope link
        valid_lft forever preferred_lft forever
```

From the host computer SSH to the VM guest.

```
user@host:~$ ssh -X sdn@192.168.22.83
sdn@192.168.22.83's password: sdn
sdn@SDN-i386:~$
```

## 6.2 Build a Mininet test network

Look at the startup options for Mininet.

```
user@host:~$ ssh -X sdn@192.168.22.83
```

```
sdn@SDN-i386:~$ sudo mn --help
```

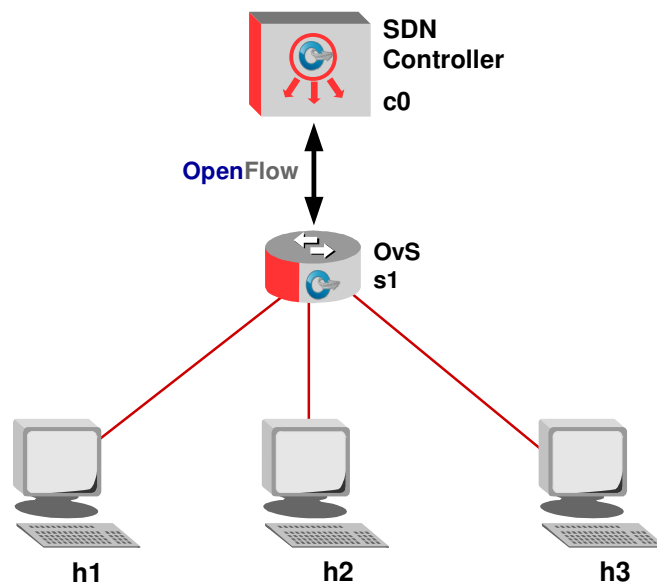
```
Usage: mn [options]
```

```
(type mn -h for details)
```

The mn utility creates Mininet network from the command line. It can create parametrized topologies, invoke the Mininet CLI, and run tests.

Options:

```
-h, --help                show this help message and exit
--switch=SWITCH           default|ivs|lxbr|ovs|ovsbr|ovsk|user[,param=value...]
                           ovs=OVSSwitch default=OVSSwitch ovsk=OVSSwitch
                           lxbr=LinuxBridge user=UserSwitch ivs=IVSSwitch
                           ovsbr=OVSBridge
--host=HOST               cfs|proc|rt[,param=value...]
                           rt=CPULimitedHost{'sched': 'rt'} proc=Host
                           cfs=CPULimitedHost{'sched': 'cfs'}
--controller=CONTROLLER  default|none|nox|ovsc|ref|remote|ryu[,param=value...]
                           ovsc=OVSController none=NullController
                           remote=RemoteController default=DefaultController
                           nox=NOX ryu=Ryu ref=Controller
--link=LINK               default|ovs|tc[,param=value...] default=Link
                           ovs=OVSLink tc=TCLink
--topo=TOPO               linear|minimal|reversed|single|torus|tree[,param=value
                           ...] linear=LinearTopo
                           reversed=SingleSwitchReversedTopo tree=TreeTopo
                           single=SingleSwitchTopo torus=TorusTopo
                           minimal=MinimalTopo
-c, --clean               clean and exit
--custom=CUSTOM           read custom classes or params from .py file(s)
--test=TEST               cli|build|pingall|pingpair|iperf|all|iperfudp|none
-x, --xterms              spawn xterms for each node
-i IPBASE, --ipbase=IPBASE
                           base IP address for hosts
--mac                     automatically set host MACs
--arp                     set all-pairs ARP entries
-v VERBOSITY, --verbosity=VERBOSITY
                           info|warning|critical|error|debug|output
--innamespace             sw and ctrl in namespace?
--listenport=LISTENPORT  base port for passive switch listening
--nolistenport            don't use passive listening port
--pre=PRE                 CLI script to run before tests
--post=POST               CLI script to run after tests
--pin                     pin hosts to CPU cores (requires --host cfs or -host rt)
--nat                     [option=val...] adds a NAT to the topology that
                           connects Mininet hosts to the physical network.
                           Warning: This may route any traffic on the machine
                           that uses Mininet's IP subnet into the Mininet
                           network. If you need to change Mininet's IP subnet,
                           see the --ipbase option.
--version                 prints the version and exits
--cluster=server1,server2...
                           run on multiple servers (experimental!)
--placement=block|random
                           node placement for --cluster (experimental!)
```



*Illustration 15: Basic test network*

Establish a basic network with an SDN Controller (*c0*) an Open vSwitch (OVS) and three hosts.

Options:

- Switch
  - *ivs* - Indigo Virtual Switch
  - *lxbr* - Linux Bridge
  - *ovs* - Open vSwitch
  - *ovsbr* - Open vSwitch in standalone/bridge mode
  - *ovsk* - OpenFlow 1.3 switch
  - *ovsl* - Open vSwitch legacy kernel-space switch using *ovs-openflowd*
- Controller
  - *nox* - Nicira Networks OpenFlow controller
  - *ovsc* - Open vSwitch controller
  - *ref* - OpenFlow reference controller
  - *remote* - Controller running outside of mininet (i.e. OpenDaylight for example)
  - *ryu* - RYU Network Operating System

- topo
  - linear - Linear topology of k switches, with n hosts per switch
  - minimal - Single switch and two hosts
  - reversed - Single switch connected to k hosts, with reversed ports, the lowest-numbered host is connected to the highest-numbered port
  - single - Single switch connected to k hosts
  - torus - 2-D Torus mesh interconnect topology
  - tree - a tree network with a given depth and fanout
- mac - automatically set host MACs

Create a basic network to start with from the *mn* mininet launch command.

```
sdn@SDN-i386:~$ sudo mn --topo tree,depth=1,fanout=3 --switch ovsk
--controller ref --mac
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(s1, h1) (s1, h2) (s1, h3)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 1 switches
s1
*** Starting CLI:
```

Review the network elements and the links between them.

```
mininet> hosts
*** Unknown command: hosts
mininet> nodes
available nodes are:
c0 h1 h2 h3 s1

mininet> links
s1-eth1<->h1-eth0 (OK OK)
s1-eth2<->h2-eth0 (OK OK)
s1-eth3<->h3-eth0 (OK OK)

mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=7960>
<Host h2: h2-eth0:10.0.0.2 pid=7963>
<Host h3: h3-eth0:10.0.0.3 pid=7965>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=7970>
<Controller c0: 127.0.0.1:6633 pid=7952>
```



To run commands on the hosts, use the hostname followed by the command. For example look at the IP address on *h1* and route table on *h2*.

```
mininet> h2 ip addr show | grep eth0
159: h2-eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group default qlen 1000
    inet 10.0.0.2/8 brd 10.255.255.255 scope global h2-eth0
```

```
mininet> h2 ip route
10.0.0.0/8 dev h2-eth0 proto kernel scope link src 10.0.0.2
```

Test connectivity from one host to another.

Test options in the cli are:

- build
- pingall - Ping between all hosts
- pingallfull - Ping between all hosts, including times
- pingpair - Ping between first two hosts
- iperf <host1> <host2> - Run TCP iperf between two hosts
- iperfudp <bw i.e. 100M> <host1> <host2> - UDP iperf

```
mininet> h1 ping -c1 h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=4.66 ms

--- 10.0.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 4.664/4.664/4.664/0.000 ms
```

Look at network links between elements.

```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s1-eth3
s1 lo: s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:h3-eth0
c0
```

The Mininet command *pingall* is useful for checking connectivity between hosts.

```
mininet> pingall
*** Ping: testing ping reachability
h1 → h2 h3
h2 → h1 h3
h3 → h1 h2
*** Results: 0% dropped (6/6 received)
```

*iperf* can be ran from the Mininet prompt to test bandwidth between links. Here is an example between *h1* and *h2*.

```
Mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['18.0 Gbits/sec', '18.0 Gbits/sec']
```

Running commands on hosts can be done directly from the Mininet command shell as seen above or individual *xterms* can be ran for hosts. In the example see the IPv4 addresses for each host.

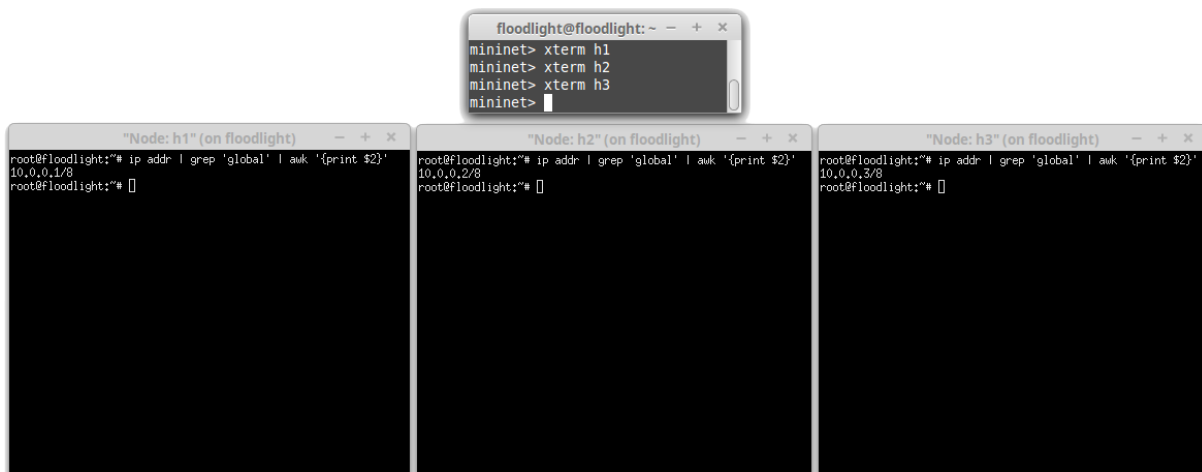


Illustration 16: Testing Mininet network

### 6.2.1 X11 error running xterm

If the connection is over SSH with a `-X` switch for X11 forwarding, then the following error may be displayed.

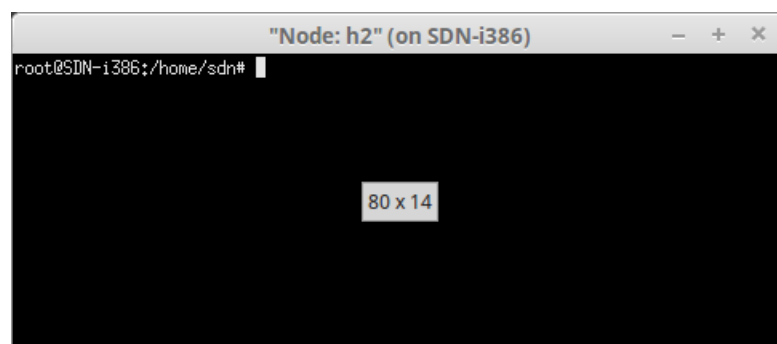
```
mininet> xterm h2
mininet> X11 connection rejected because of wrong authentication.
```

The error is caused because the `-X` connection is made as the user `sdn` but the Mininet has been ran as root using `sudo`. To rectify use the X authority file utility `xauth` to determine the X11 magic cookie for the user `ssh` and then set the same magic cookie for the root user to resolve the issue.

```
sdn@SDN-i386:~$ xauth list $DISPLAY
SDN-i386/unix:10  MIT-MAGIC-COOKIE-1 1603e2caa81a3a6f50245bb17cf9e546

sdn@SDN-i386:~$ sudo -s
root@SDN-i386:/home/sdn# xauth add SDN-i386/unix:10  MIT-MAGIC-COOKIE-1
1603e2caa81a3a6f50245bb17cf9e546
root@SDN-i386:/home/sdn# exit

sdn@SDN-i386:~$ sudo mn --topo tree,depth=1,fanout=3 --switch ovsk
--controller ref --mac
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(s1, h1) (s1, h2) (s1, h3)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 1 switches
s1
*** Starting CLI:
mininet> xterm h2
```



*Illustration 17: xterm window over SSH*

### 6.2.2 Exiting mininet

To exit Mininet use the *exit* command.

```
mininet> exit
*** Stopping 1 controllers
c0
*** Stopping 1 switches
s1 ...
*** Stopping 3 links

*** Stopping 3 hosts
h1 h2 h3
*** Done
completed in 3.536 seconds
```

It is a good idea to follow this up with *sudo mn -clean* or *-c* to tidy up before running another network.

```
sdn@SDN-i386:~$ sudo mn --clean
*** Removing excess controllers/ofprotocols/ofdatapaths/pings/noxes
killall controller ofprotocol ofdatapath ping nox_core lt-nox_core ovs-
openflowd ovs-controller udpbwtest mnexec ivs 2> /dev/null
killall -9 controller ofprotocol ofdatapath ping nox_core lt-nox_core ovs-
openflowd ovs-controller udpbwtest mnexec ivs 2> /dev/null
pkill -9 -f "sudo mnexec"
*** Removing junk from /tmp
rm -f /tmp/vconn* /tmp/vlogs* /tmp/*.out /tmp/*.log
*** Removing old X11 tunnels
*** Removing excess kernel datapaths
ps ax | egrep -o 'dp[0-9]+' | sed 's/dp/nl:/'
*** Removing OVS datapaths
ovs-vsctl --timeout=1 list-br
ovs-vsctl --timeout=1 list-br
*** Removing all links of the pattern foo-ethX
ip link show | egrep -o '([-_.[:alnum:]]+-eth[[:digit:]]+)'
ip link show
*** Killing stale mininet node processes
pkill -9 -f mininet:
*** Shutting down stale tunnels
pkill -9 -f Tunnel=Ethernet
pkill -9 -f .ssh/mn
rm -f ~/.ssh/mn/*
*** Cleanup complete.
```

## 6.3 Configuring hosts

Recreate the same network but this time specifying the bandwidth of links and adding delay. The options:

- host
  - cfs - Completely Fair Scheduler (CFS)
  - proc
  - rt, - Real Time (RT)
    - cpu=<positive fraction, or -1> - CPU bandwidth limit

Note: RT\_GROUP\_SCHED must be enabled in the kernel to change the *rt,cpu*.

```
sdn@SDN-i386:~$ sudo mn --topo tree,depth=1,fanout=3 --switch ovsk
--controller ref --mac --host rt,cpu=0.25
```

## 6.4 Configuring links

Recreate the same network but this time for traffic control (tc) specify the bandwidth of links and adding delay. The options:

- link tc
  - bw=<value> - Value in Mb/s
  - delay=<value> - Time unit expressed as '5ms', '50us' or '1s'
  - max\_queue\_size=<x> - Queue size in packets
  - loss=<0 - 100> - Percentage loss
  - use\_htb=<True | False> - Hierarchical Token Bucket (HTB) rate limiter

```
sdn@SDN-i386:~$ sudo mn --topo tree,depth=1,fanout=3 --switch ovsk
--controller ref --mac --link tc,bw=100,delay=20ms
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(100.00Mbit 20ms delay) (100.00Mbit 20ms delay) (s1, h1) (100.00Mbit 20ms
delay) (100.00Mbit 20ms delay) (s1, h2) (100.00Mbit 20ms delay)
(100.00Mbit 20ms delay) (s1, h3)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 1 switches
s1 (100.00Mbit 20ms delay) (100.00Mbit 20ms delay) (100.00Mbit 20ms delay)
*** Starting CLI:
```

Repeat the *iperf* test between *h1* and *h2*. Note the difference in results from the earlier test. \*\*\* Results: ['18.0 Gbits/sec', '18.0 Gbits/sec'].

```
mininet> iperf h1 h2
```

```
*** Iperf: testing TCP bandwidth between h1 and h2
```

```
*** Results: ['85.7 Mbits/sec', '99.3 Mbits/sec']
```

## 7. OpenFlow traffic review

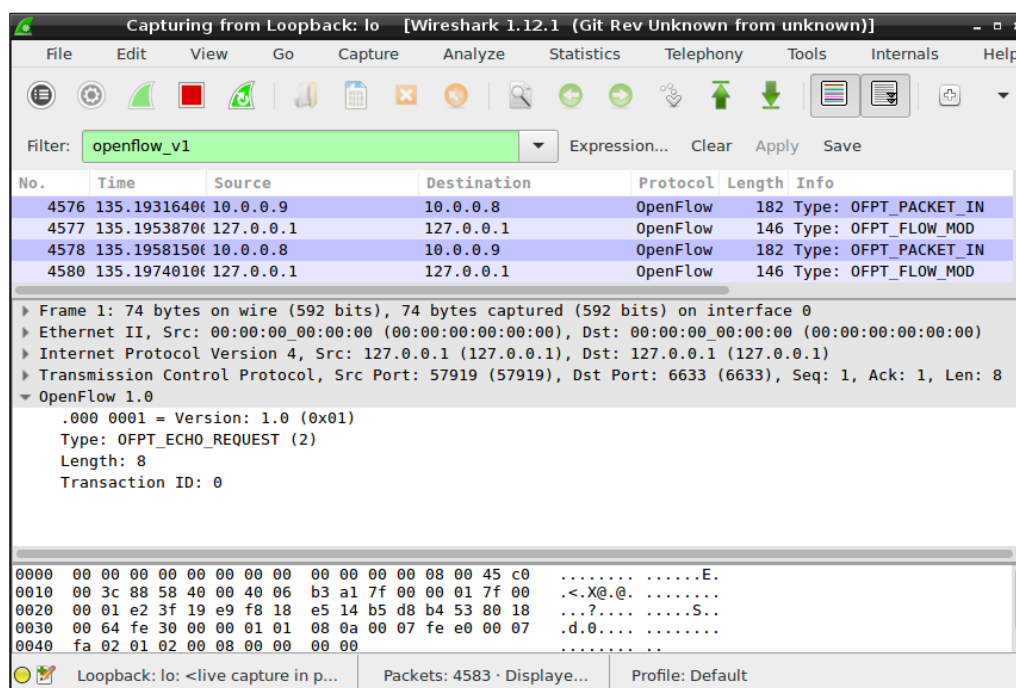


Illustration 18: Wireshark OpenFlow traffic

The OpenFlow traffic between the Controller *c0* and the OvS *s1*. As the controller and switch share the same VM guest the control channel is via the loopback interface, so monitoring the loopback *lo0* interface will give access to this messaging. In this case the messaging is using OpenFlow v1.0 so using the filter *openflow\_v1* will show the communications between the devices.

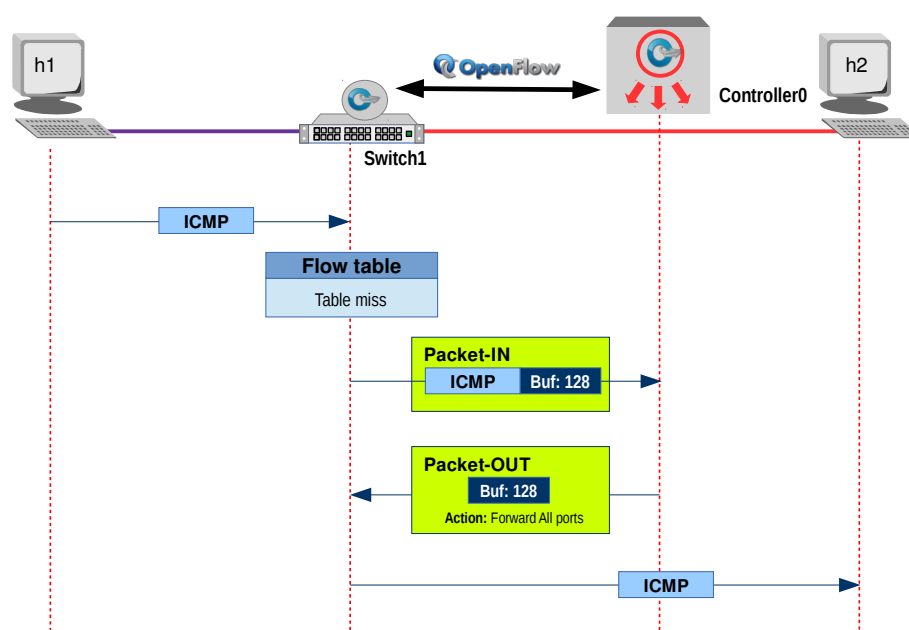


Illustration 19: SDN Action

As demonstrated in Illustration 19 an Internet Control Message Protocol (ICMP) arrives at *s1*. It does a table entry check and finds that it has a *table-miss* no flow entry. The OvS *s1* sends an OpenFlow *Packet In* (*OFPT\_PACKET\_IN*) to the controller *c0* with a unique Buffer IDentifier *128* for a decision.

```

Frame 1: OFPT_PACKET_IN
Ethernet II, Src: 00:00:00_00:00:00, Dst: 00:00:00_00:00:00
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 60348, Dst Port: 6633, Seq: 9,
Ack: 9, Len: 60
OpenFlow
  version: 1
  type: OFPT_PACKET_IN (10)
  length: 60
  xid: 0
  buffer_id: 128
  total_len: 42
  in_port: 1
  reason: OFPR_NO_MATCH (0)
  Ethernet packet

```

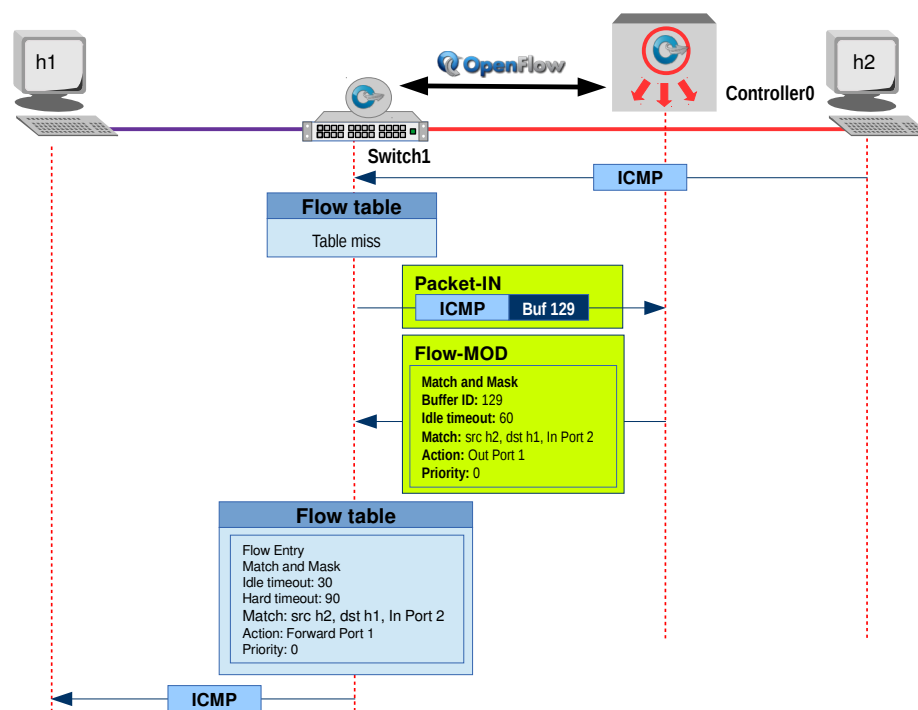
The controller *c0* responds to the OvS *s1* using the Buffer IDentifier *128* with the decision to *Output to switch port* on all ports.

```

Frame 2: OFPT_PACKET_OUT
Ethernet II, Src: 00:00:00_00:00:00, Dst: 00:00:00_00:00:00
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 6633, Dst Port: 60348, Seq: 9,
Ack: 69, Len: 24
OpenFlow
  version: 1
  type: OFPT_PACKET_OUT (13)
  length: 24
  xid: 0
  buffer_id: 128
  in_port: 1
  actions_len: 8
  of_action list
    of_action_output
      type: OFPAT_OUTPUT (0)
      len: 8
      port: 65531 (All ports)
      max_len: 0

```





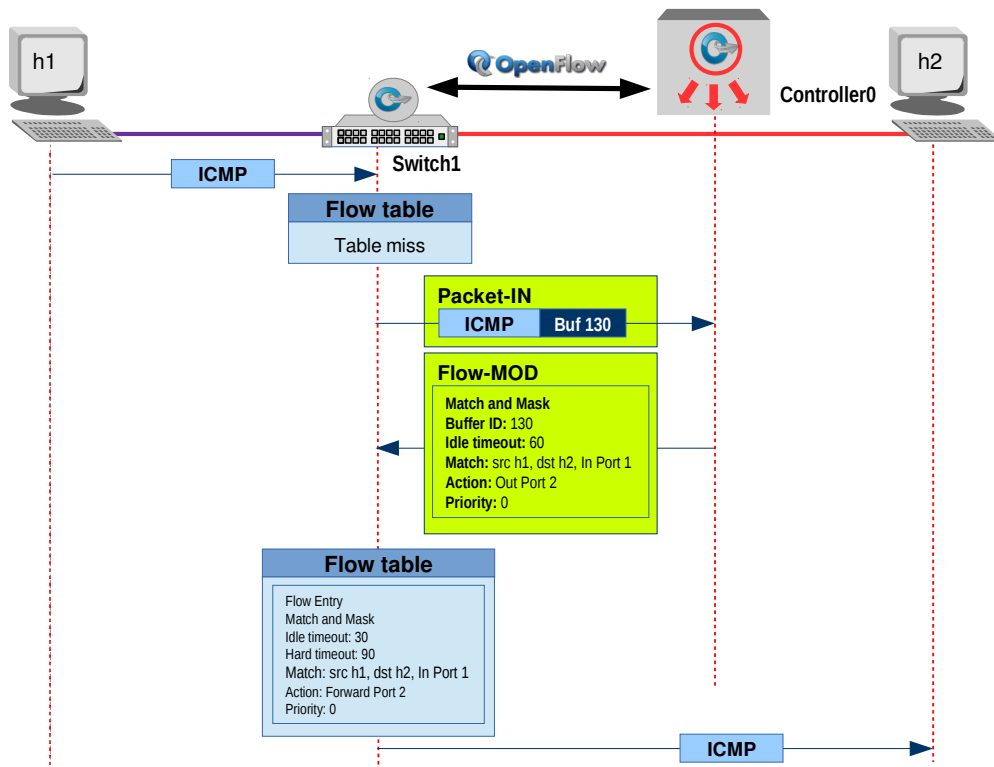
*Illustration 20: SDN Flow-MOD*

The response from the host *h2* can be seen in Illustration 20, it arrives at the OvS *s1* and is given a Buffer Identifier *129*, again there is a table-miss so the OvS *s1* sends an OpenFlow *OFPT\_PACKET\_IN* message to the controller *c0*.

Frame 3: **OFPT\_PACKET\_IN**

```

Ethernet II, Src: 00:00:00_00:00:00, Dst: 00:00:00_00:00:00
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 60348, Dst Port: 6633, Seq: 69,
Ack: 33, Len: 60
OpenFlow
  version: 1
  type: OFPT_PACKET_IN (10)
  length: 60
  xid: 0
  buffer_id: 129
  total_len: 42
  in_port: 2
  reason: OFPR_NO_MATCH (0)
  Ethernet packet
  
```



*Illustration 21: SDN Flow-MOD #2*

In this case as shown in Illustration 21, the controller *c0* sends an OpenFlow *Flow MOD* to the OvS *s1* to add an entry to the Flow Table for traffic from 10.0.0.2 → 10.0.0.1 on Ethernet port 2 → port 1. Subsequent similar packets are then forwarded automatically by the OvS until the idle time-out of 60 seconds has been exceeded and then the process must be repeated.

Frame 4: **OPENFLOW FLOW MODIFICATION**

Ethernet II, Src: 00:00:00\_00:00:00, Dst: 00:00:00\_00:00:00

Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

Transmission Control Protocol, Src Port: 6633, Dst Port: 60348, Seq: 33,

Ack: 129, Len: 80

## OpenFlow

version: 1

type: OFPT\_FLOW\_MOD (14)

length: 80

xid: 0

## of\_match

wildcards: 0x0000000000000000

in\_port: 2

eth\_src: 00:00:00\_00:00:02

eth\_dst: 00:00:00\_00:00:01

vlan\_vid: 65535

vlan\_pcp: 0

eth\_type: 2054

ip\_dscp: 0

ip\_proto: 2

ipv4\_src: 10.0.0.2

ipv4\_dst: 10.0.0.1

tcp\_src: 0

tcp\_dst: 0

cookie: 0

\_command: 0

idle\_timeout: 60

hard\_timeout: 0

priority: 0

buffer\_id: 129

out\_port: 0

flags: Unknown (0x00000000)

## of\_action list

## of\_action\_output

type: **OFPAT\_OUTPUT** (0)

len: 8

**port: 1**

max\_len: 0

The next packet in from *h1* triggers another *OFPT\_PACKET\_IN*.

Frame 5: **OFPT\_PACKET\_IN**

Ethernet II, Src: 00:00:00\_00:00:00, Dst: 00:00:00\_00:00:00

Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

Transmission Control Protocol, Src Port: 60348, Dst Port: 6633, Seq: 129,  
Ack: 113, Len: 116

OpenFlow

version: 1

type: OFPT\_PACKET\_IN (10)

length: 116

xid: 0

buffer\_id: 130

total\_len: 98

in\_port: 1

reason: **OFPR\_NO\_MATCH** (0)

Ethernet packet

This time, Controller *c0* knows the port that *h2* is on so it sends an OpenFlow *Flow MOD* to the OvS *s1* to add an entry to the Flow Table for traffic from 10.0.0.1 → 10.0.0.2 on Ethernet port 1 → port 2. Subsequent similar packets are then forwarded automatically by the OvS until the idle time-out of 60 seconds has been exceeded and then the process must be repeated.

```
Frame 6: OPENFLOW FLOW MODIFICATION
Ethernet II, Src: 00:00:00_00:00:00, Dst: 00:00:00_00:00:00
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 6633, Dst Port: 60348, Seq: 113,
Ack: 245, Len: 80
OpenFlow
  version: 1
  type: OFPT_FLOW_MOD (14)
  length: 80
  xid: 0
  of_match
    wildcards: 0x0000000000000000
    in_port: 1
    eth_src: 00:00:00_00:00:01
    eth_dst: 00:00:00_00:00:02
    vlan_vid: 65535
    vlan_pcp: 0
    eth_type: 2048
    ip_dscp: 0
    ip_proto: 1
    ipv4_src: 10.0.0.1
    ipv4_dst: 10.0.0.2
    tcp_src: 8
    tcp_dst: 0
  cookie: 0
  _command: 0
  idle_timeout: 60
  hard_timeout: 0
  priority: 0, ip=[controller IP]
  buffer_id: 130
  out_port: 0
  flags: Unknown (0x00000000)
  of_action list
    of_action_output
      type: OFPAT_OUTPUT (0)
      len: 8
      port: 2
      max_len: 0
```

*Ping* is not the only command that can run on a host. Mininet hosts can run any command or application that is available to the underlying Linux system and its file system. It is possible to enter any bash command, including job control (&, jobs, kill, etc..)

Next, run a simple HTTP server on *h1*, making a request from *h3*, then shut down the web server.

## 7.1 Webserver test

*lynx* text based web client on the mininet VM.

Check the IP addresses of the *h1* and *h3* hosts. Confirm connectivity between them.

```
mininet> h1 ip addr | grep "inet.*eth0"
    inet 10.0.0.1/8 brd 10.255.255.255 scope global h1-eth0
```

```
mininet> h3 ip addr | grep "inet.*eth0"
    inet 10.0.0.3/8 brd 10.255.255.255 scope global h3-eth0
```

```
mininet> h1 ping -c1 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=4.46 ms
```

*xterm* *h1* and *h3* so you have access to individual shells for each host.

```
mininet> xterm h1
mininet> xterm h3
```

Run a webserver on *h1* *xterm*.

```
root@SDN-i386:~# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
```

Use *lynx* on the *h3* *xterm* to access the webserver.

```
root@SDN-i386:~# lynx 10.0.0.1
```

Directory listing for / (p1 of 2)

Directory listing for /

---

```
* .bash_history
* .bash_logout
* .bashrc
* .cache/
* .gitconfig
* .mininet_history
* .profile
* .rnd
* .ssh/
* .wireshark/
* .Xauthority
* install-mininet-vm.sh
* loxigen/
* mininet/
* oflops/
* oftest/
* openflow/
```

-- press space for next page --

Arrow keys: Up and Down to move. Right to follow a link; Left to go back.

H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list

On the webserver xterm on *h1* the following message pops up.

```
10.0.0.3 - - [18/Mar/2015 01:11:34] "GET / HTTP/1.0" 200 -
```

An alternative is to run the server in the mininet console as a background process and the use lynx to view it.

```
mininet> h1 python -m SimpleHTTPServer 80 &  
Serving HTTP on 0.0.0.0 port 80 ...
```

```
mininet> h2 lynx h1
```

Directory listing for / (p1 of 2)

Directory listing for /

---

```
* .bash_history  
* .bash_logout  
* .bashrc  
* .cache/  
* .gitconfig  
* .mininet_history  
* .profile  
* .rnd  
* .ssh/  
* .wireshark/  
* .Xauthority  
* install-mininet-vm.sh  
* loxigen/  
* mininet/  
* oflops/  
* oftest/  
* openflow/
```

-- press space for next page --

Arrow keys: Up and Down to move. Right to follow a link; Left to go back.

H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list

Kill the webserver on host *h1*.

```
mininet> h1 ps -ef | grep SimpleHTTPServer  
root      2624  1406  0 01:17 pts/4    00:00:00 python -m SimpleHTTPServer 80  
root      2669  1406  0 01:19 pts/4    00:00:00 grep SimpleHTTPServer
```

```
mininet> h1 kill 2624
```

So what happened ?

From → to	Ver	Len	Type	BufID	Reason/Action
S1 → C0	OF 1.0	158	OFPT_PACKET_IN	342	Reason: OFPR_NO_MATCH 10.0.0.3 → 10.0.0.1 TCP SYN 39109 → 80 In pt: 3
C0 → S1	OF 1.0	90	OFPT_PACKET_OUT	342	Action: OFPAT_OUTPUT In pt: 3 Out pt: 65531
S1 → C0	OF 1.0	158	OFPT_PACKET_IN	343	Reason: OFPR_NO_MATCH 10.0.0.1 → 10.0.0.3 TCP SYN, ACK 80 → 39109 In pt: 1
C0 → S1	OF 1.0	146	OFPT_FLOW_MOD	343	Action: OFPR_FLOW_MOD 10.0.0.1 → 10.0.0.3 TCP SRC: 80 TCP DST: 39109 In pt: 1 Out pt:3
S1 → C0	OF 1.0	150	OFPT_PACKET_IN	344	Reason: OFPR_NO_MATCH 10.0.0.3 → 10.0.0.1 TCP ACK 39109 → 80 In pt: 3
C0 → S1	OF 1.0	146	OFPT_FLOW_MOD	344	Action: OFPR_FLOW_MOD 10.0.0.3 → 10.0.0.1 TCP SRC: 39109 TCP DST: 80 In pt: 3 Out pt:1

*h3* tried to send an Ethernet frame containing a TCP SYN message to port 80 on the *h1* webserver.

The OvS *s1* does not have a flow for this in its flow table so it encapsulated the message in an OpenFlow *OFPT\_PACKET\_IN* message with Buffer ID 342 and a Reason code of *OFPR\_NO\_MATCH*.

Controller *C0* responded with an *Output to switch port (OFPAT\_OUTPUT)* message telling the OvS *s1* to send on all its ports.

When the responding SYN, ACK is received the OvS *s1* has no match for the return path either so it encapsulates in an OpenFlow *OFPT\_PACKET\_IN* message with Buffer ID 343 and a Reason code of *OFPR\_NO\_MATCH*.

This time the Ethernet port for the destination is known as a result of the earlier message so the Controller *c0* instructs the OvS *s1* with a OpenFlow Flow Modification message to map HTTP traffic for 10.0.0.1 → 10.0.0.3 in on port 1 to be forwarded to port 3.

The next response from *h3* will again be forwarded as an *OFPT\_PACKET\_IN* message with Buffer ID 344 and a Reason code of *OFPR\_NO\_MATCH* to Controller *c0*.

As the Ethernet port for *h1* is now known an OpenFlow Flow Modification message to map HTTP traffic for 10.0.0.3 → 10.0.0.1 in on port 3 to be forwarded to port 1 is sent to the OvS *s1* from the Controller *c0*.

All similar traffic to/from *h1* to *h3* will now be handled by the OvS *s1* with need for recourse to the Controller *c0* until the idle timeout of 60 seconds has passed.



## 8. POX Controller

POX is an SDN networking platform written in Python. It is an OpenFlow controller, but can also function as an OpenFlow switch, and can be useful for writing networking software in general.

### 8.1 Running POX

```
sdn@SDN-i386:~$ ~/pox/pox.py forwarding.l2_learning
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:core:POX 0.2.0 (carp) is up.
```

An alternative which gives 'pretty' output is:

```
sdn@SDN-i386:~$ ~/pox/pox.py forwarding.l2_pairs \
info.packet_dump samples.pretty_log log.level --DEBUG
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:forwarding.l2_pairs:Pair-Learning switch running.
INFO:info.packet_dump:Packet dumper running
[core                               ] POX 0.2.0 (carp) going up...
[core                               ] Running on CPython (2.7.9/Aug 13 2016 16:41:35)
[core                               ] Platform is Linux-3.16.0-4-586-i686-with-debian-8.7
[core                               ] POX 0.2.0 (carp) is up.
[openflow.of_01                     ] Listening on 0.0.0.0:6633
[openflow.of_01                     ] [00-00-00-00-00-03 1] connected
[openflow.of_01                     ] [00-00-00-00-00-02 2] connected
[openflow.of_01                     ] [00-00-00-00-00-01 3] connected
[dump:00-00-00-00-00-03 ] [ethernet][ipv4][udp][172 bytes]
[dump:00-00-00-00-00-02 ] [ethernet][ipv4][udp][172 bytes]
[dump:00-00-00-00-00-01 ] [ethernet][ipv4][udp][172 bytes]
```

## 8.2 Testing POX

Run a mininet topology where Floodlight is the SDN controller.

```
sdn@SDN-i386:~$ sudo mn --topo tree,depth=2,fanout=3 --switch ovsk
--controller=remote,ip=127.0.0.1,port=6633 --mac
*** Creating network
*** Adding controller
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s2, h1) (s2, h2) (s2, h3) (s3, h4) (s3, h5) (s3,
h6) (s4, h7) (s4, h8) (s4, h9)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:

mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9
h2 -> h1 h3 h4 h5 h6 h7 h8 h9
h3 -> h1 h2 h4 h5 h6 h7 h8 h9
h4 -> h1 h2 h3 h5 h6 h7 h8 h9
h5 -> h1 h2 h3 h4 h6 h7 h8 h9
h6 -> h1 h2 h3 h4 h5 h7 h8 h9
h7 -> h1 h2 h3 h4 h5 h6 h8 h9
h8 -> h1 h2 h3 h4 h5 h6 h7 h9
h9 -> h1 h2 h3 h4 h5 h6 h7 h8
*** Results: 0% dropped (72/72 received)
mininet>
```



## 9. Project Floodlight

The Project Floodlight Open SDN Controller operates with OpenFlow switches. It is an enterprise class, Apache licensed, Java based SDN Controller and has its origins with Big Switch Networks.

### 9.1 Running Floodlight

Run the *floodlight.jar* file produced by *ant*.

```
sdn@SDN-i386:~$ cd ~/floodlight
sdn@SDN-i386:/floodlight$ java -jar ./target/floodlight.jar
```

Floodlight will start running and print debug output to your console.

### 9.2 Testing Floodlight

Run a Mininet topology where Floodlight is the SDN controller.

```
sdn@SDN-i386:~$ sudo mn --topo tree,depth=2,fanout=3 --switch ovsk
--controller=remote,ip=127.0.0.1,port=6653 --mac
[sudo] password for sdn:
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s2, h1) (s2, h2) (s2, h3) (s3, h4) (s3, h5) (s3, h6) (s4,
h7) (s4, h8) (s4, h9)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:

mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9
h2 -> h1 h3 h4 h5 h6 h7 h8 h9
h3 -> h1 h2 h4 h5 h6 h7 h8 h9
h4 -> h1 h2 h3 h5 h6 h7 h8 h9
h5 -> h1 h2 h3 h4 h6 h7 h8 h9
h6 -> h1 h2 h3 h4 h5 h7 h8 h9
h7 -> h1 h2 h3 h4 h5 h6 h8 h9
h8 -> h1 h2 h3 h4 h5 h6 h7 h9
h9 -> h1 h2 h3 h4 h5 h6 h7 h8
*** Results: 0% dropped (72/72 received)
```



## 10. OpenDaylight

Project OpenDaylight is a Linux Foundation Collaborative Project. The software combines SDN components including a fully pluggable controller, interfaces, protocol plug-ins and applications to create a framework for SDN and Network Functions Virtualisation (NFV) solutions. The current release is Beryllium.

### 10.1 Running ODL karaf

ODL uses Apache Karaf which is a small Open Services Gateway initiative (OSGi) based runtime which provides a lightweight container onto which various components and applications can be deployed. Karaf provides an ecosystem for ODL.

To run karaf:

```
sdn@SDN-i386:~$ ./odl/bin/karaf
```



*Illustration 22: karaf*

### 10.2 Installing openDaylight User eXperience (DULX) features

On the *karaf* shell install DLUX.

- odl-restconf – Representational STate (REST) like protocol that provides a programmatic interface over Hyper Text Transfer Protocol (HTTP) for accessing data on port 8080 for HTTP requests.
- odl-l2switch-all – Layer2 switch functionality.
- odl-mdsal-apidocs - Model Driven Service Abstraction Layer (MD-SAL) Application Programmable Interface (API) Documentation. They can be accessed at: <http://<IP addr>:8181/apidoc/explorer/index.html>.
- odl-dlux-all - Graphical user interface for OpenDaylight based on the AngularJS framework.

```
opendaylight-user@root> feature:install odl-restconf
opendaylight-user@root> feature:install odl-l2switch-all
opendaylight-user@root> feature:install odl-mdsal-apidocs
opendaylight-user@root> feature:install odl-dlux-all
```

Features can be installed in one command like this also.

```
opendaylight-user@root> feature:install odl-restconf odl-l2switch-all odl-
mdsal-apidocs odl-dlux-all
```

## 10.3 Testing the ODL installation

Run a Mininet topology where the ODL is the SDN controller.

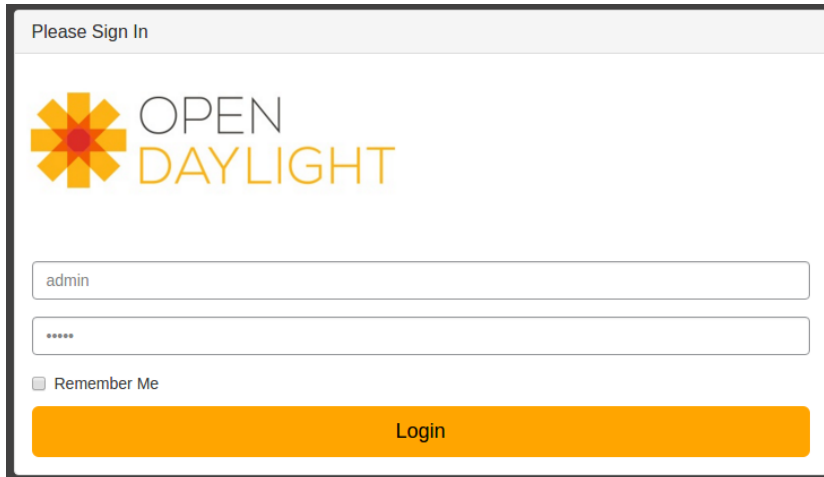
```
sdn@SDN-i386:~$ sudo mn --topo tree,depth=2,fanout=3 --switch ovsk
--controller=remote,ip=127.0.0.1,port=6633 --mac
[sudo] password for sdn:
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s2, h1) (s2, h2) (s2, h3) (s3, h4) (s3, h5) (s3, h6) (s4,
h7) (s4, h8) (s4, h9)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:

mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9
h2 -> h1 h3 h4 h5 h6 h7 h8 h9
h3 -> h1 h2 h4 h5 h6 h7 h8 h9
h4 -> h1 h2 h3 h5 h6 h7 h8 h9
h5 -> h1 h2 h3 h4 h6 h7 h8 h9
h6 -> h1 h2 h3 h4 h5 h7 h8 h9
h7 -> h1 h2 h3 h4 h5 h6 h8 h9
h8 -> h1 h2 h3 h4 h5 h6 h7 h9
h9 -> h1 h2 h3 h4 h5 h6 h7 h8
*** Results: 0% dropped (72/72 received)
```

## 10.4 DLUX User interface

Login to the Dlux interface with a Chrome browser (recommended). The default username is: **admin** and the default password is: **admin**.

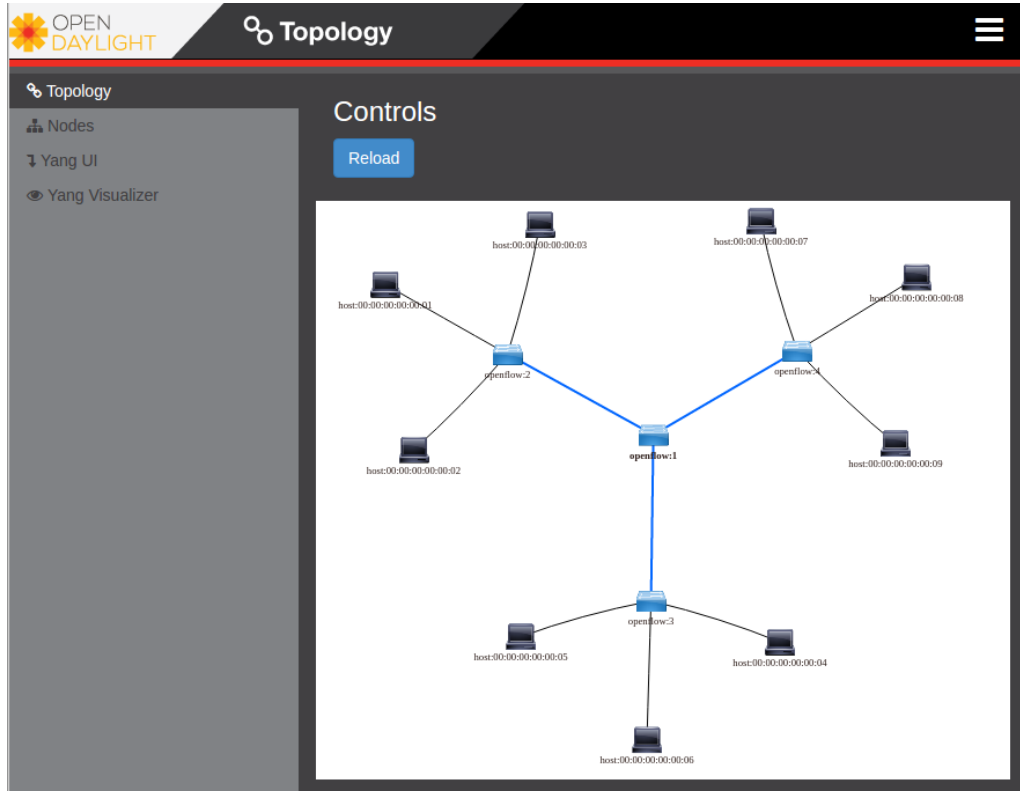
**`http://<IP address>:8181/index.html`**



The image shows the DLUX login page. At the top, it says "Please Sign In". Below that is the "OPEN DAYLIGHT" logo, which consists of a stylized orange flower icon and the text "OPEN DAYLIGHT". There are two input fields: the first is for the username, containing the text "admin", and the second is for the password, containing six dots. Below the password field is a checkbox labeled "Remember Me". At the bottom is a large orange button labeled "Login".

*Illustration 23: Dlux login*

Within the Dlux user interface the Mininet network should be visible under the *Topology* tab.



*Illustration 24: DLUX Topology*

## 11. Custom Topologies

The topologies created thus far have been defined by the *mn* command options and these are limited. It will become necessary to create more customised topologies and this can be achieved using Python scripts. Mininet has example scripts in:

```
~/mininet/examples
```

and custom scripts can be created in:

```
~/mininet/custom
```

Note: For the purpose of NTE-SDN VM however custom exercise scripts are in:

```
~/TEL-3214-exercises
```

```
sdn@SDN-i386:~$ cd ~/mininet/custom
sdn@SDN-i386:~/mininet/custom$ ls
README  topo-2sw-2host.py
```

```
sdn@SDN-i386:~/mininet/custom$ cat README
```

This directory should hold configuration files for custom mininets.

See `custom_example.py`, which loads the default minimal topology. The advantage of defining a mininet in a separate file is that you then use the `--custom` option in `mn` to run the CLI or specific tests with it.

To start up a mininet with the provided custom topology, do:

```
sudo mn --custom custom_example.py --topo mytopo
```

An example is given for a two switch solution with a host in each. The example also incorporates different parameters for each link.

```
sdn@SDN-i386:~$ cat ~/TEL-3214-exercises/topo-2sw-2host.py
```

```
"""Custom topology example
```

```
Two directly connected switches plus a host for each switch:
```

```
host --- switch --- switch --- host
```

Adding the 'topos' dict with a key/value pair to generate our newly defined topology enables one to pass in '--topo=mytopo' from the command line.

```
"""
```

```
from mininet.topo import Topo
```

```
class MyTopo( Topo ):
```

```
    "Simple topology example."
```

```
    def __init__( self ):
```

```
        "Create custom topo."
```

```
        # Initialize topology
```

```
        Topo.__init__( self )
```

```
        # Add hosts and switches
```

```
        leftHost = self.addHost( 'h1' )
```

```
        rightHost = self.addHost( 'h2' )
```

```
        leftSwitch = self.addSwitch( 's3' )
```

```
        rightSwitch = self.addSwitch( 's4' )
```

```
        # Add links
```

```
        self.addLink( leftHost, leftSwitch, bw=50, delay='3ms', loss=10 )
```

```
        self.addLink( leftSwitch, rightSwitch, bw=100, delay='1ms' )
```

```
        self.addLink( rightSwitch, rightHost, bw=50, delay='3ms', loss=10 )
```

```
topos = { 'mytopo': ( lambda: MyTopo() ) }
```



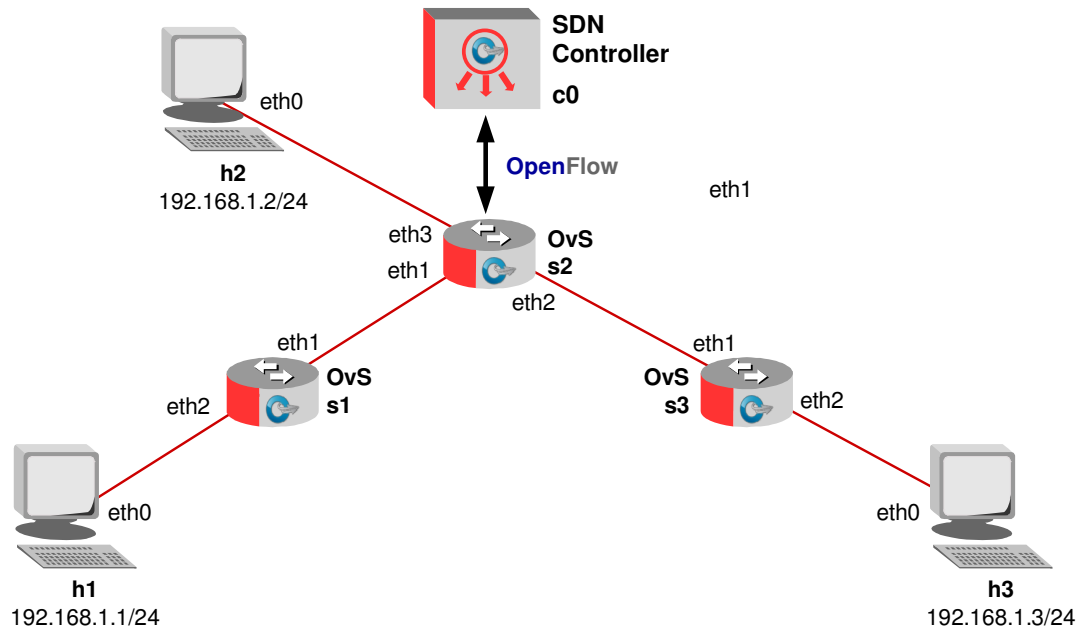
Run this example and confirm it is working.

```
$ sudo mn --custom ~/TEL-3214-exercises/topo-2sw-2host.py --topo mytopo --link=tc
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s3 s4
*** Adding links:
(30.00Mbit 3ms delay 10% loss) (30.00Mbit 3ms delay 10% loss) (h1, s3)
(100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (s3, s4) (50.00Mbit 3ms delay
10% loss) (50.00Mbit 3ms delay 10% loss) (s4, h2)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 2 switches
s3 s4 ... (30.00Mbit 3ms delay 10% loss) (100.00Mbit 1ms delay) (100.00Mbit
1ms delay) (50.00Mbit 3ms delay 10% loss)
*** Starting CLI:
```

To demonstrate the link constraint run the iperfudp between hosts. Note the speed is limited below the speed of the slowest link.

```
mininet> iperfudp 100M h1 h2
*** Iperf: testing UDP bandwidth between h1 and h2
*** Results: ['100M', '26.2 Mbits/sec', '26.2 Mbits/sec']
```

## 11.1 Create a custom topology



*Illustration 25: Mininet custom topology example*

Taking this diagram as an example to build. Use the existing file as a template to build the required custom topology.

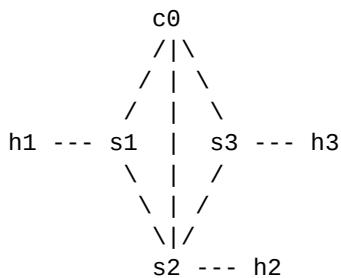
-----Custom-OvS.py-----

```
#!/usr/bin/python
```

```
"""
```

```
Custom topology example
```

```
Three directly connected switches plus a host attached to each switch with a  
controller (c0):
```



```
"""
```

```
from mininet.net import Mininet
from mininet.node import Controller
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink
from mininet.topo import Topo

def customNet():

    "Create a customNet and add devices to it."

    net = Mininet( controller=Controller, link=TCLink )

    # Add controller
    info( 'Adding controller\n' )
    net.addController( 'c0' )

    # Add hosts
    info( 'Adding hosts\n' )
    h1 = net.addHost( 'h1' )
    h2 = net.addHost( 'h2' )
    h3 = net.addHost( 'h3' )

    # Add switches
    info( 'Adding switches\n' )
    s1 = net.addSwitch( 's1' )
    s2 = net.addSwitch( 's2' )
    s3 = net.addSwitch( 's3' )

    # Add links
    info( 'Adding switch links\n' )
    net.addLink( s1, s2, bw=1000, delay='1ms' )
    net.addLink( s2, s3, bw=1000, delay='1ms' )

    info( 'Adding host links\n' )
    net.addLink( h1, s1, bw=50, delay='3ms' )
    net.addLink( h2, s2, bw=50, delay='2ms' )
    net.addLink( h3, s3, bw=50, delay='2ms', loss=15 )

    info( '*** Starting network\n' )
    net.start()

    info( '*** Running CLI\n' )
    CLI( net )

    info( '*** Stopping network' )
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    customNet()
```

-----

Now run the new custom topology.

```
sdn@SDN-i386:~$ sudo ~/TEL-3214-exercises/Custom-OvS.py
Adding controller
Adding hosts
Adding switches
Adding switch links
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay)
(1000.00Mbit 1ms delay) Adding host links
(50.00Mbit 3ms delay) (50.00Mbit 3ms delay) (50.00Mbit 2ms delay) (50.00Mbit
2ms delay) (50.00Mbit 2ms delay 15% loss) (50.00Mbit 2ms delay 15% loss)
*** Starting network
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 (1000.00Mbit 1ms delay) (50.00Mbit 3ms delay) s2 (1000.00Mbit 1ms delay)
(1000.00Mbit 1ms delay) (50.00Mbit 2ms delay) s3 (1000.00Mbit 1ms delay)
(50.00Mbit 2ms delay 15% loss) ...(1000.00Mbit 1ms delay) (50.00Mbit 3ms
delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (50.00Mbit 2ms delay)
(1000.00Mbit 1ms delay) (50.00Mbit 2ms delay 15% loss)
*** Running CLI
*** Starting CLI:
mininet>
```

Reviewing the new network with the *dump*, *net*, *pingall*, *iperf* and *dpctl dump-flows* commands.

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=10517>
<Host h2: h2-eth0:10.0.0.2 pid=10519>
<Host h3: h3-eth0:10.0.0.3 pid=10521>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=10526>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None pid=10529>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None pid=10532>
<Controller c0: 127.0.0.1:6653 pid=10510>

mininet> net
h1 h1-eth0:s1-eth2
h2 h2-eth0:s2-eth3
h3 h3-eth0:s3-eth2
s1 lo: s1-eth1:s2-eth1 s1-eth2:h1-eth0
s2 lo: s2-eth1:s1-eth1 s2-eth2:s3-eth1 s2-eth3:h2-eth0
s3 lo: s3-eth1:s2-eth2 s3-eth2:h3-eth0
c0

mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
```

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> X h2
*** Results: 16% dropped (5/6 received)

mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['43.3 Mbits/sec', '47.7 Mbits/sec']

mininet> iperf h1 h3
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['179 Kbits/sec', '193 Kbits/sec']

mininet> iperf h2 h3
*** Iperf: testing TCP bandwidth between h2 and h3
*** Results: ['318 Kbits/sec', '320 Kbits/sec']

mininet> dpctl dump-flows
*** s1
-----
NXST_FLOW reply (xid=0x4):
*** s2
-----
NXST_FLOW reply (xid=0x4):
*** s3
-----
NXST_FLOW reply (xid=0x4):

mininet> h1 ping 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=2.81 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=2.82 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.595 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.595/2.078/2.826/1.048 ms

mininet> dpctl dump-flows
*** s1
-----
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=3.854s, table=0, n_packets=3, n_bytes=294,
idle_timeout=60, idle_age=1, priority=65535, icmp, in_port=1,
vlan_tci=0x0000, dl_src=b2:a6:82:06:f5:0b, dl_dst=da:f0:ca:b8:ca:79,
nw_src=10.0.0.2, nw_dst=10.0.0.1, nw_tos=0, icmp_type=0, icmp_code=0
actions=output:2 cookie=0x0, duration=2.856s, table=0, n_packets=2,
n_bytes=196, idle_timeout=60, idle_age=1, priority=65535, icmp,
in_port=2, vlan_tci=0x0000, dl_src=da:f0:ca:b8:ca:79,
dl_dst=b2:a6:82:06:f5:0b, nw_src=10.0.0.1, nw_dst=10.0.0.2, nw_tos=0,
icmp_type=8, icmp_code=0 actions=output:1
```

```
*** s2 -----
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=2.859s, table=0, n_packets=2, n_bytes=196,
idle_timeout=60, idle_age=1, priority=65535,icmp,in_port=1,
vlan_tci=0x0000,d1_src=da:f0:ca:b8:ca:79,d1_dst=b2:a6:82:06:f5:0b,
nw_src=10.0.0.1,nw_dst=10.0.0.2,nw_tos=0,icmp_type=8,icmp_code=0
actions=output:3 cookie=0x0, duration=3.86s, table=0, n_packets=3,
n_bytes=294, idle_timeout=60, idle_age=1, priority=65535,icmp,
in_port=3,vlan_tci=0x0000,d1_src=b2:a6:82:06:f5:0b,
d1_dst=da:f0:ca:b8:ca:79,nw_src=10.0.0.2,nw_dst=10.0.0.1,
nw_tos=0,icmp_type=0,icmp_code=0 actions=output:1
*** s3 -----
NXST_FLOW reply (xid=0x4):
```

-----



## 12.2 OpenDaylight User Experience (DLUX)

Access the ODL Server from a Chrome browser as shown earlier.

## 12.3 Start Mininet network

Start a Mininet network on the Mininet computer. In this case point to the remote OpenDaylight remote controller on port 6633, the standard OpenFlow port. The following python script is a variant of the previous script except the controller now points to the remote Open Daylight controller.

```
-----Custom-RemoteODL.py-----

#!/usr/bin/python

"""
Custom topology example

Three directly connected switches plus a host attached to each switch
with a remote ODL SDN Controller (c0):

      c0
      /|\
ODL   /|\   192.168.25.111
...../|\.....
Mininet /|\ 192.168.25.83
      /|\
h1 --- s1 | s3 --- h3
      \|\
      \|\
      \|\
      s2 --- h2

"""
from mininet.net import Mininet
from mininet.node import Controller, RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel, info

# OpenDayLight controller
ODL_CONTROLLER_IP='192.168.25.111'
ODL_CONTROLLER_PORT=6633

# Define remote OpenDaylight Controller

print 'OpenDaylight IP Addr:', ODL_CONTROLLER_IP
print 'OpenDaylight Port:', ODL_CONTROLLER_PORT

def customNet():

    "Create a customNet and add devices to it."

    net = Mininet( topo=None, build=False )
```



```
# Add controller
info( 'Adding controller\n' )
net.addController( 'c0',
                  controller=RemoteController,
                  ip=ODL_CONTROLLER_IP,
                  port=ODL_CONTROLLER_PORT
                )

# Add hosts
info( 'Adding hosts\n' )
h1 = net.addHost( 'h1' )
h2 = net.addHost( 'h2' )
h3 = net.addHost( 'h3' )

# Add switches
info( 'Adding switches\n' )
s1 = net.addSwitch( 's1' )
s2 = net.addSwitch( 's2' )
s3 = net.addSwitch( 's3' )

# Add links
info( 'Adding switch links\n' )
net.addLink( s1, s2 )
net.addLink( s2, s3 )

info( 'Adding host links\n' )
net.addLink( h1, s1 )
net.addLink( h2, s2 )
net.addLink( h3, s3 )

info( '*** Starting network ***\n' )
net.start()

info( '*** Running CLI ***\n' )
CLI( net )

info( '*** Stopping network ***' )
net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    customNet()
```

-----

Run the script.

```
sdn@SDN-i386:~$ sudo ~/TEL-3214-exercises/Custom-RemoteODL.py
OpenDaylight IP Addr: 192.168.25.111
OpenDaylight Port: 6633
Adding controller
Adding hosts
Adding switches
Adding switch links
Adding host links
*** Starting network ***
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3
*** Running CLI ***
*** Starting CLI:
```

Review the topology.

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=10598>
<Host h2: h2-eth0:10.0.0.2 pid=10601>
<Host h3: h3-eth0:10.0.0.3 pid=10603>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=10608>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None
pid=10611>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None pid=10614>
<RemoteController c0: 192.168.25.111:6633 pid=10591>

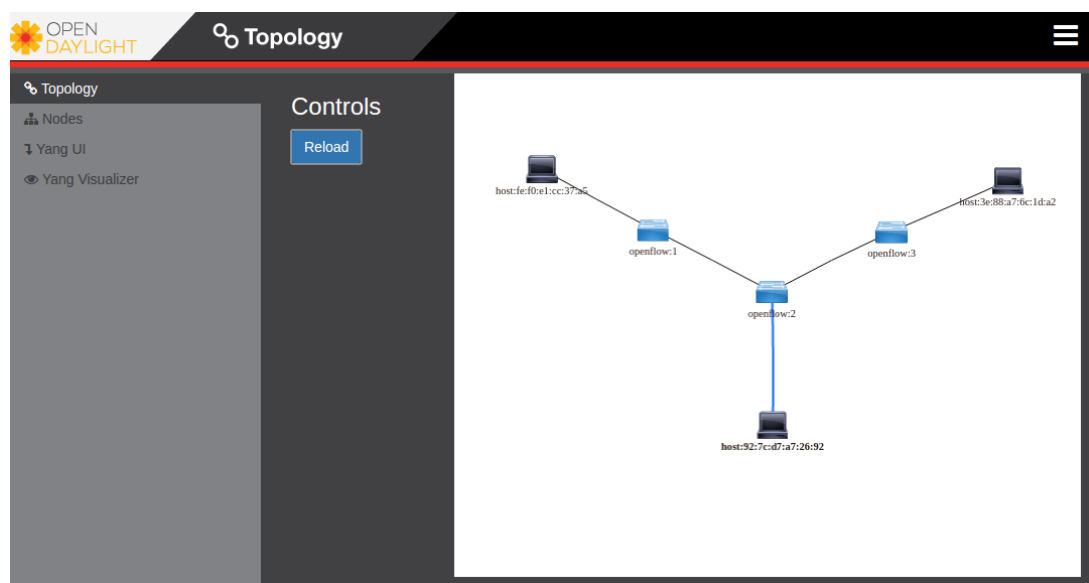
mininet> net
h1 h1-eth0:s1-eth2
h2 h2-eth0:s2-eth3
h3 h3-eth0:s3-eth2
s1 lo: s1-eth1:s2-eth1 s1-eth2:h1-eth0
s2 lo: s2-eth1:s1-eth1 s2-eth2:s3-eth1 s2-eth3:h2-eth0
s3 lo: s3-eth1:s2-eth2 s3-eth2:h3-eth0
c0
```

Test the topology.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
```

```
mininet> iperf h1 h3
*** Iperf: testing TCP bandwidth between h1 and h3
Waiting for iperf to start up...*** Results: ['227 Mbits/sec', '234 Mbits/sec']
```

Now look at the network in DLUX Topology dashboard.



*Illustration 27: Dlux topology dashboard*

## 13. North Bound Interface (NBI)

As SDN evolves it has become apparent that new NBI mechanisms are required to meet the diverse Applications that will adjust the SDN Controller 's Network Policy over the Network Services Abstraction Layer (NSAL).

### 13.1 Frenetic



The Frenetic Project raises the level of abstraction for programming SDNs through the development of simple, reusable, high level abstractions and efficient runtime systems that automatically generate and install corresponding low-level rules on SDN switches.

- High-level abstraction
  - Control.
- Modular constructs
  - Compositional reasoning
  - Sequential (>>)
  - Parallel (|).
- Portability
  - Operate on many devices.
- Rigorous semantic foundations
  - Mechanical program analysis tools.

#### 13.1.1 Pyretic

```
from pyretic.lib.corelib import *  
  
def main():  
    return flood()
```

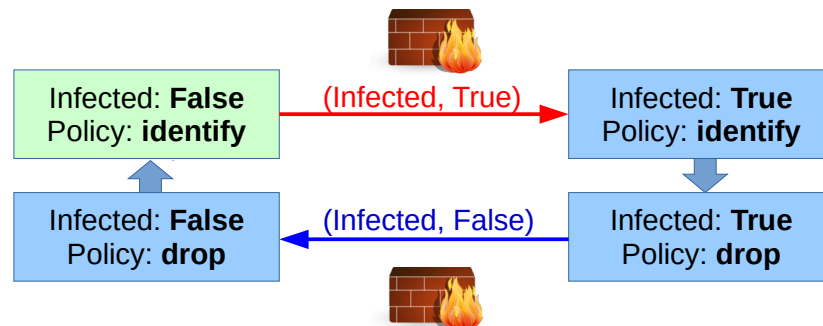
*Illustration 28: Pyretic*

Pyretic is a Frenetic Project implementation embedded in Python. Pyretic however does not answer the question of how an event-reaction logic is embedded in software or how changes are verified as having completed correctly.

### 13.1.2 Kenetic

```
from pyretic.kinetic.fsm_policy import *
from pyretic.kinetic.smv.model_checker import *

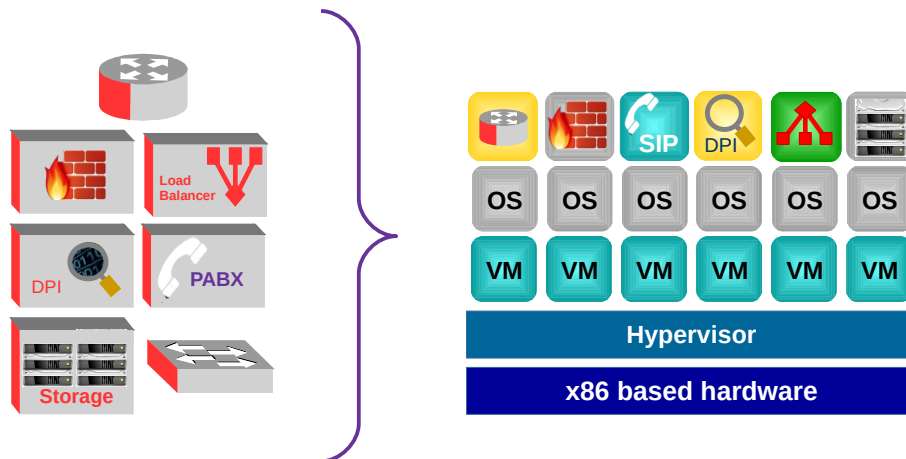
def policy(self):
    self.case(is_true(V('infected')), C(drop))
    self.default(C(identity))
```



*Illustration 29: Kenetic*

Kenetic, a Pyretic module defines network policies as a Finite State machine (FSM). Transitions between states are triggered by different types of dynamic events in the network, like intrusion detection, authentication of hosts, data usage cap reached, etc. For each of these events an operator can enforce different policies or a chain of policies either sequentially or in parallel.

## 14. Networks Function Virtualisation (NFV)



*Illustration 30: Network Function Virtualisation*

At the SDN & OpenFlow World Congress in Darmstadt, Germany in October 2012 a group of Tier 1 service providers launched an initiative called NFV. These operators could see that Virtualisation and Cloud computing could evolve the way services are delivered on networks by consolidation and virtualisation of network equipment on industry standard high volume servers as can be seen in the NFV concept. Functions could also be migrated to centralised virtualised infrastructure while also offering the facility to push virtualisation of functions right out to the end user premises.

While SDN and NFV are complimentary to each other they are not as yet inter-dependent and can therefore be operated either together, or independently.

Obviously moving functions that were heretofore based on specialist hardware presents a number of challenges, such as;

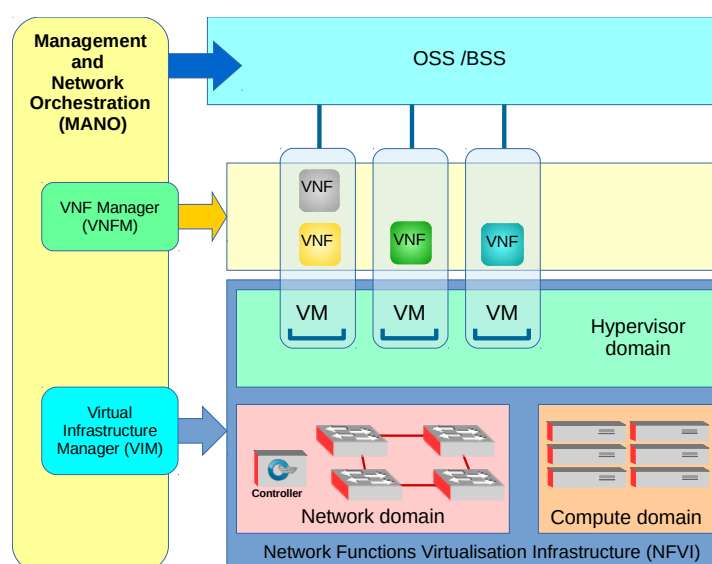
- the portability to a virtualised system and interoperability with existing infrastructure.
- the performance trade-off between standards based hardware and that of specialised, function specific hardware.
- the interaction of the Management and Network Orchestration (MANO) of the distributed functions with the network. Using the benefits of automation to achieve the transformational aspects of NFV.
- the integration of functions into the overall NFV ecosystem and its coexistence with legacy systems.
- the new challenges in terms of security and stability have evolved as a result of cloud computing and virtualisation.

These challenges and newer security challenges will evolve from this new networking system.

The benefits of NFV however make the case for migration so compelling that without doubt it will form the core of services to be offered by service providers well into the

future. Hardware-based appliances have a specific life, which is getting shorter and shorter with the rapid pace of development, and they need regular replacement. This complicates maintenance procedures and customer support with no financial benefit to the service provider.

NFV will transform the design of the network to implement these functions in software, many of these will process centrally thereby allowing for their operation to be migrated and backed up as needed. This will reduce equipment costs and reduce power consumption due to power management features in standard servers and storage, while eliminating the need for specific hardware. Services can be scaled up and down in a similar fashion to that provided by cloud services today. IT MANO mechanisms familiar today in cloud services will facilitate the automatic installation and scaling of capacity by building Virtual Machines (VM) or Containers to meet demand. In this way traffic patterns and service demand can be met in an automated and managed fashion. As a result the service provider can increase the speed to market of both existing NFVs but also decrease the time it takes to innovate new services and deliver them on the virtualised infrastructure.



*Illustration 31: NFV Ecosystem*

Illustration 31 shows the overall NFV ecosystem. The underlying infrastructure collectively is called the Network Functions Virtualisation Infrastructure (NFVI) and it consists of three domains, Network, Compute and Hypervisor/Virtualisation. The Network Domain consists of islands of switches with SDN Controllers or a traditional routed and switched network. The Compute Domain consists of the computing hardware and storage necessary to support the upper layers. The final domain in the NFVI is the Hypervisor/Virtualisation Domain which contains the virtualisation hypervisors and VMs. This can be built using existing hypervisors like Xen, VMWare or using Container technology like Docker. These NFVI domains are managed by a Virtual Infrastructure Manager (VIM).

A Virtual Network Function Manager (VNFM) controls the building of individual Virtual Network Functions (VNF) on the VMs. MANO performs the overall management of the VIM, VNFM and Operations Support Systems (OSS) / Business Support System (BSS) and allows the service provider to quickly deploy and scale VNF services as well as provide and scale resources for VNFs. This system reduces administrator workloads and removes the need for manual administration type tasks. It also offers APIs and other tooling extensions to integrate with existing environments.

#### 14.1.1 Providing NFV to the customer

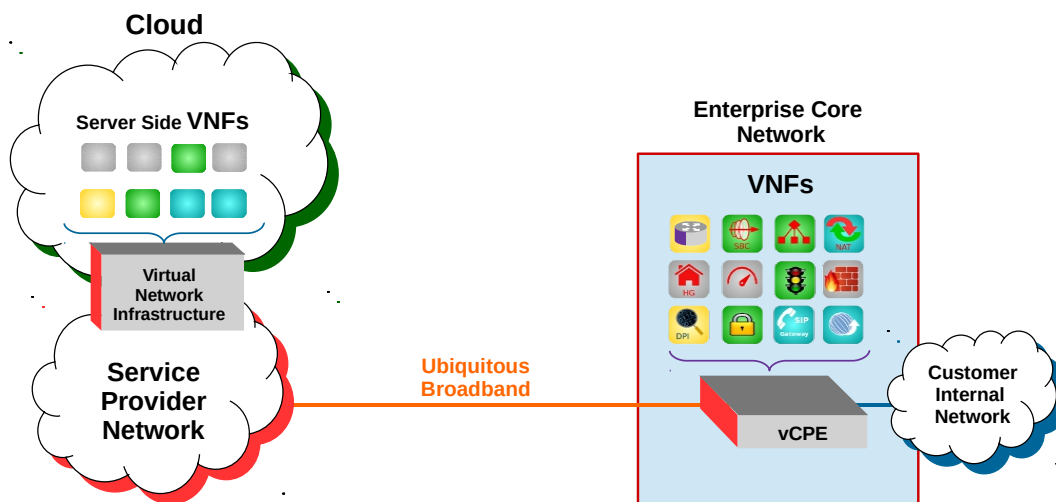


Illustration 32: vCPE

Illustration 32 demonstrates the benefits that ubiquitous high speed broadband gives to the service provider. It provides the ability to supply a vCPE to the customer upon which VNFs can be offered.



Current services that can be converted into NFV style services are:

- Router.
- Session Border Controller (SBC).
- Load Balancer.
- Network Address Translation (NAT).
- Home Gateway (HG).
- Application Acceleration.
- Traffic Management.
- Firewall.
- Deep Packet Inspection (DPI).
- Bulk Encryption.
- Content Caching.
- Session Initiation Protocol Gateway (SIP-GW).

This however is just the beginning, these services already exist on traditional deployment mechanisms. The fact that virtualisation will now be available in the vCPE at the customer premises means that a service provider can deploy new services not envisaged as yet and deploy services on a trial basis, all without equipment changes.

#### **14.1.2 NFV Standards**

After the initial white paper from the Darmstadt-Germany Call for Action in 2012 it was decided to form an Industry Specification Group (ISG) under the European Telecommunications Standards Institute (ETSI). Phase 1 of this group was to "drive convergence on network operator requirements for NFV to include applicable standards, where they already exist, into industry services and products to simultaneously develop new technical requirements with the goal of stimulating innovation and fostering an open ecosystem of vendors" (ETSI, 2012). They issued a progress White Paper in October 2013 and a final paper in October 2014 which drew attention to the second release of ETSI NFV ISG documents that were subsequently published in January 2015. December 2014 was considered to be the end of phase 1 and phase 2 was launched. This saw some reorganisation of the ISG NFV working groups, to focus less on requirements and more on adoption.

The key areas addressed include:

- Stability, Interoperability, Reliability, Availability, Maintainability.
- Intensified collaboration with other bodies.
- Testing and validation to encourage interoperability and solidify implementations.
- Definition of interfaces.
- Establishment of a vibrant NFV ecosystem.
- Performance and assurance considerations.
- Security.

## 14.2 Open Platform NFV

The Linux Foundation established a Collaborative Project called 'Open Platform NFV (OPNFV)' in October 2014. The project intent is to provide a Free and Open-Source Software (FOSS) platform for the deployment of NFV solutions that leverage investments from a community of developers and solution providers.

The initial focus of the OPNFV will be the NFVI and VIM. In reality this means the OPNFV will focus on building interfaces between existing FOSS projects like those listed below.

Creating these interfaces between what are essentially existing elements to create a functional reference platform will be a major win for the technology and certainly contribute to the goals of phase 2 of the ETSI NFV ISG.

- Virtual Infrastructure Management: OpenStack, Apache CloudStack, ...
- Network Controller and Virtualization Infrastructure: OpenDaylight, ...
- Virtualisation and hypervisors: KVM, Xen, libvirt, LXC, ...
- Virtual forwarder: OvS, Linux bridge, ...
- Forwarding-plane interfaces and acceleration: forwarding plane Development Kit (DPDK), Open Data Plane (ODP), ...
- Operating System: GNU/Linux, ...

### 14.3 Ongoing research

SDN is at an early stage of development. The Open Networking Research Center (ONRC) at UC Berkeley and Stanford University has been created to help realise the potential of SDN. The IETF has a Software-Defined Networking Research Group (SDNRG) with the stated goal of identifying the approaches that can be defined, deployed and used in the near term as well identifying future research challenges.

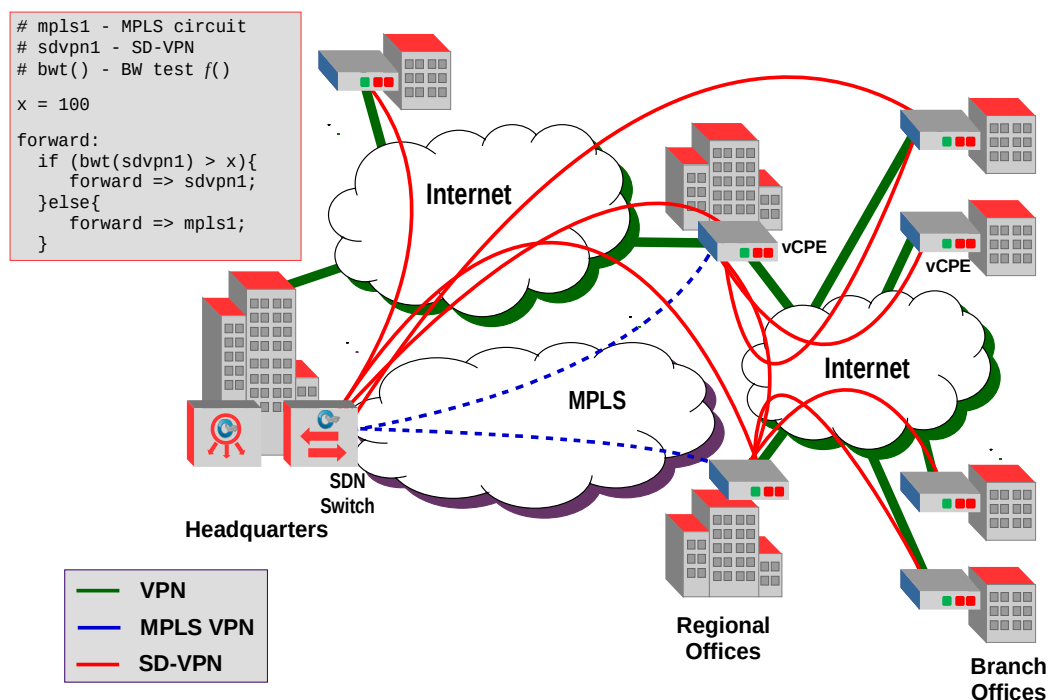
The IETF have also a Network Function Virtualisation Research Group (NFVRG) to focus on research problems associated with NFV-related topics and the research community to address them.

The Linux Foundation believe that with the projects they have in place already, they are in a perfect position to bring these together as a new project Open Platform NFV (OPNFV) to accelerate NFV.

Dr. James Kempf of Ericsson believes that NFV and SDN have traversed the peak of inflated expectation and are starting down the trough of despair. However he has considered the OPNFV initiative of the Linux Foundation which he sees as a complimentary effort to their existing OpenDaylight and OpenStack projects. He believes that there is a lot of work yet to be achieved before reaching the slope of enlightenment and considers that SDN is confined to the data centre for some time to come.

## 14.4 Software Defined WAN (SD-WAN)

In traditional enterprise, Local Area Network (LAN) segments are interconnected across the Wide Area Network (WAN) via Multiprotocol Label Switching (MPLS) circuits operated by Internet Service Providers (ISP). While in many enterprises, small offices maybe connected over the Internet via secure Virtual Private Networks (VPN). At this stage it is common for people to use Voice over Internet Protocol (VoIP) for personal and even business voice calls over the Internet and for the most part it works. Therefore why shouldn't it be possible to manage enterprise traffic over the Internet? Quality of Service (QoS) cannot be guaranteed on the Internet so enterprises continue to employ MPLS circuits for the QoS and the Service Level Agreements (SLA) they receive from the providers. Such circuits and the enterprise routers required at each site makes the cost of such circuits quite expensive.



*Illustration 33: Software Defined WAN (SD-WAN)*

SD-WAN as demonstrated in Illustration 33 provides a solution to the problem. With SD-WAN, MPLS circuits are maintained between the critical sites, say HQ and regional sites. Each site has a virtualised Customer Premises Equipment (vCPE) instead of an enterprise router. For major sites both Internet VPN and MPLS circuits are maintained while smaller sites maintain multiple Internet circuits from more connected sites.

A Software Defined Network (SDN) Controller monitors each circuit, both MPLS and Internet. Taking the example that a Regional Office requires 100 Mb/s of bandwidth at a certain latency with the Headquarters office. The SDN Controller monitors these thresholds and should the Internet circuit meet the requirements it will forward traffic over that circuit instead of the more expensive MPLS option. Should the Internet circuit fall below the threshold then the SDN Controller can redirect the traffic over the MPLS circuit to maintain the expected QoS level. In this way MPLS is only employed when the Internet circuit cannot meet the required SLA. Similar re-routing can occur for network outages as the SDN Controller has an overall view of the WAN circuits, it can detect failures and redirect accordingly.

## 15. The future of Broadband

It is predicted that the future of broadband (Weldon, 2015) will be a new Global-Local paradigm that will supply an elastic network that will give the appearance of infinite bandwidth to the end-user. This paradigm will be achieved increasingly by Global Service Providers (GSP) using Local Service Provider (LSP) infrastructure. Seamless provision will be possible with the GSP providing vCPEs to the customer and LSPs allowing the GSP access to a slice of their Application & Service Plane.

The changes that the elastic network bring about will also impact the cloud. Elastic compute and elastic network will link and develop together. The current centralised cloud cannot continue in its current form and cloud content will need to be brought closer to the customer. Services where latency and bandwidth are critical to the service will naturally be the first to benefit from this. The need for this change can already be seen as Netflix install Content Delivery Networks (CDN) in regions, typically through local Internet eXchange Points (IXP) in Europe. Another pointer to a new model is Netflix support for the Open-IX Association (OIX) to establish European-style IXPs in the United States of America (US).

A new Global-Local paradigm will evolve that creates a service chain with critical functions moving to a new edge cloud and less critical functions at the central site. Virtual eXtendable LAN (VXLAN) and SD-WANs will link the elements of the service chain mapped together via SDN network policies.

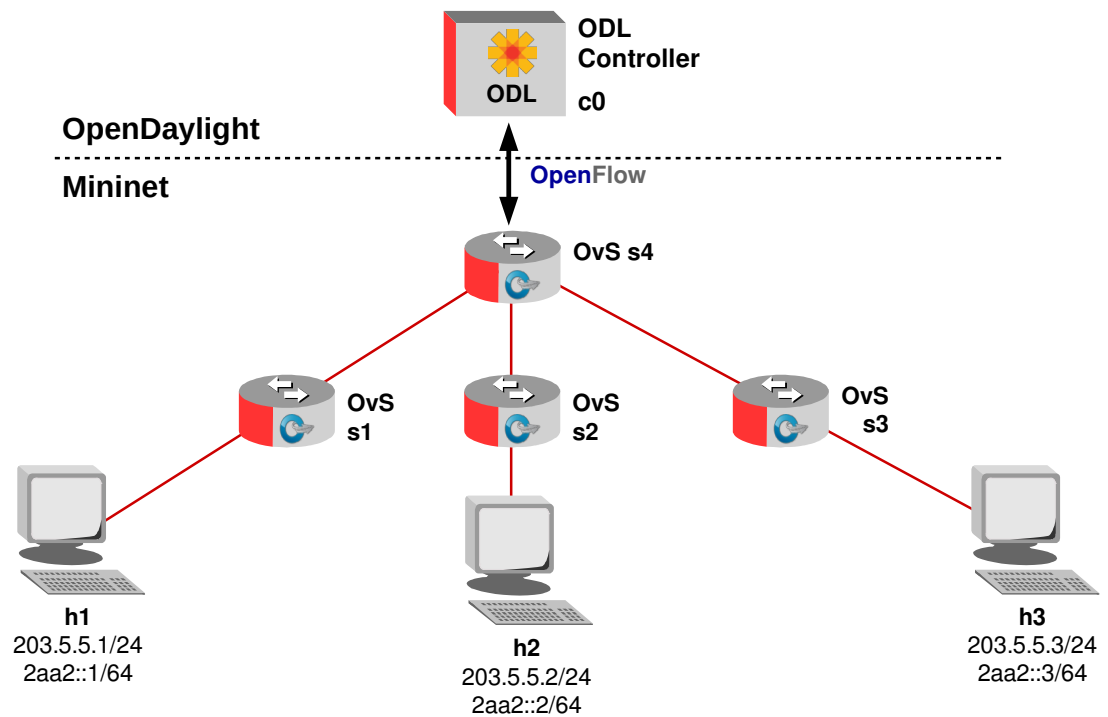
If the edge cloud resides at the LSP it will be interesting to see how the principle of Net Neutrality can be maintained. There are many questions regarding the future evolution of the Global-Local cloud paradigm.

The following is list of possible models that will evolve:

- LSP simply act as an Infrastructure as a Service (IaaS) provider for the GSP.
- LSP provide a hosted Platform as a Service (PaaS) or Software as a Service (SaaS) for the GSP.
- GSPs provide infrastructure locally as is the current case where Netflix provides CDNs at local IXPs. This model is very expensive for the GSP and will probably not scale well in the future.

It is very likely that a mix of these options will exist in the future, the selection of a particular option driven by local circumstances.

## 16. SDN Lab



*Illustration 34: SDN Lab*

- Using two SDN VMs build the network given in Illustration 34 and test connectivity between each device.
- Show a screen capture of the Dlux topology.
- Change out the SDN Controller for POX and Project Floodlight respectfully, demonstrate functionality.

## 17. List of Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
BGP	Border Gateway Protocol
BoS	Bottom of Stack
BSS	Business Support System
CDN	Content Distribution Network
CDPI	Control - Data Plane Interface
COTS	Commercial-off-the-Shelf
CPU	Central Processing Unit
DiffServ	Differentiated Services
DPDK	Dataplane Development Kit
DPI	Deep Packet Inspection
DSCP	DiffServ Code Point
ECN	Explicit Congestion Notification
ETSI	European Telecommunications Standards Institute
FCAPS	Fault, Configuration, Accounting, Performance, and Security Management
FOSS	Free and Open Source Software
HG	Home Gateway
HV	Hypervisor
ICMP	Internet Control Message Protocol
ID	Identifier
IDS	Intrusion Detection System
I/O	Input/Output
IPSec/SSL	IP Security/ Secure Sockets Layer
IPS	Intrusion Prevention System
IPv6	Internet Protocol version 6
ISG	Industry Specification Group
ISG	Industry Specification Group. An ETSI sub-organisation
ISSU	In Service Software Upgrade
IT	Information Technology
LAN	Local Area Network



LB	Load Balancer
LDM	Link Discovery Module
LLDP	Link Layer Discovery Protocol
MAC	Medium Access Control
MANO	Management and Network Orchestration
M2M	Machine-to-Machine communications
MPLS	Multiprotocol Label Switching
NAT	Network Address Translation
ND	Neighbour Discovery
NF	Network Function
NFVI	Network Functions Virtualisation Infrastructure
NFV	Network Functions Virtualisation
NIC	Network Interface Controller
NSD	Network Service Descriptors
ODP	Open Dataplane
ONF	Open Networking Foundation
OpenFlow	Specifications developed by the Open Networking Foundation
OPNFV	Open Platform NFV
OSPF	Open Shortest Path First
OSS	Operations Support System
PBB	Provider Backbone Bridge
PCP	Priority Code Point
QoS	Quality of Service
SBC	Session Border Controller
SCTP	Stream Control Transmission Protocol
SDN	Software Defined Network
SIP-GW	SIP Gateway
SIP	Session Initiation Protocol
SLA	Service Level Agreement
TCP	Transmission Control Protocol
TC	Traffic Class
UCA	User Customer Address
UDP	User Datagram Protocol
vCPE	Virtual Customer Premises Equipment
VLAN	Virtual Local Area Network

VLD	Virtual Link Descriptors
VM	Virtual Machine
VMWare	Proprietary Hypervisor
VNA	Virtualised Network Appliance
VNFD	VNF Descriptors
VNFFGD	VNF Forwarding Graph Descriptors
VNF	Virtual Network Function
VRE	Virtual Routing Engine
WAN	Wide Area Network
Xen	Proprietary Hypervisor