MINT 709 CAPS Project

Student Name: Carmen Eissa
Supervised by: Juned Noonari
August 2014

MPLS-TE with SDN/OpenFlow Approach

Problems Statement:
Traffic Engineering and dynamic bandwidth allocation issue, with optimized routing etc.

Solution:
Use SDN based solution.

Analysis:
Implication on MPLS, LDP, RSVP etc? Do we still need LDP? Can flow table be manipulated to mimic MPLS like capabilities and create FECs? Can we eliminated use of MPLS completely? what will happen to existing network and mpls? Or should it be better to use MPLS at controller along with IGP?

1.0 Background Work:

Section 1.a - MPLS Functionality Background Study

MPLS Networks:

The MPLS protocol was formed on the basis of combining the best parts of layer 2 forwarding/switching with the best parts of layer 3 IP routing to form a technology that shares the extremely fast-packet forwarding that ATM invented with the very flexible and complex path signaling techniques adopted from the IP world.[**]

MPLS is used for optimizing traffic forwarding through a network, it is based on the concept of label switching: an independent and unique "label" is added to each data packet and this label is used to switch and route the packet through the network. The label is simple, essentially a shorthand version of the packet's header information, so network equipment can be optimized around processing the label and forwarding traffic. [2]

The multiprotocol label switching (MPLS) allows data packets to transfer faster through the telecommunications network from one node to another. The advantage of using this technology is that it does not stick to just one process but works on multiple switching and cabling systems as well as different data transfer protocol. This is a good way to allow data packets to move from IP routers in a more efficient and faster way. Along with VPN system, MPLS works together to form a network. [3]

It is Key to realize the differences in the way MPLS and IP routing forward data across a network. Traditional IP packet forwarding uses the IP destination address in the packet's header to make an independent forwarding decision at each router in the network. These hop-by-hop decisions are based on network layer routing protocols, such as Open Shortest Path First (OSPF). These routing protocols are designed to find the shortest path through the network, and do not consider other factors, such as latency or traffic congestion. On the other hand, MPLS creates a connection-based model overlaid onto the traditionally connectionless framework of IP routed networks. This connection-oriented architecture opened the door to a wealth of new possibilities for managing traffic on an IP network. MPLS builds on IP, combining the intelligence of routing, which is fundamental to the operation of the Internet and IP networks, with the high performance of switching.

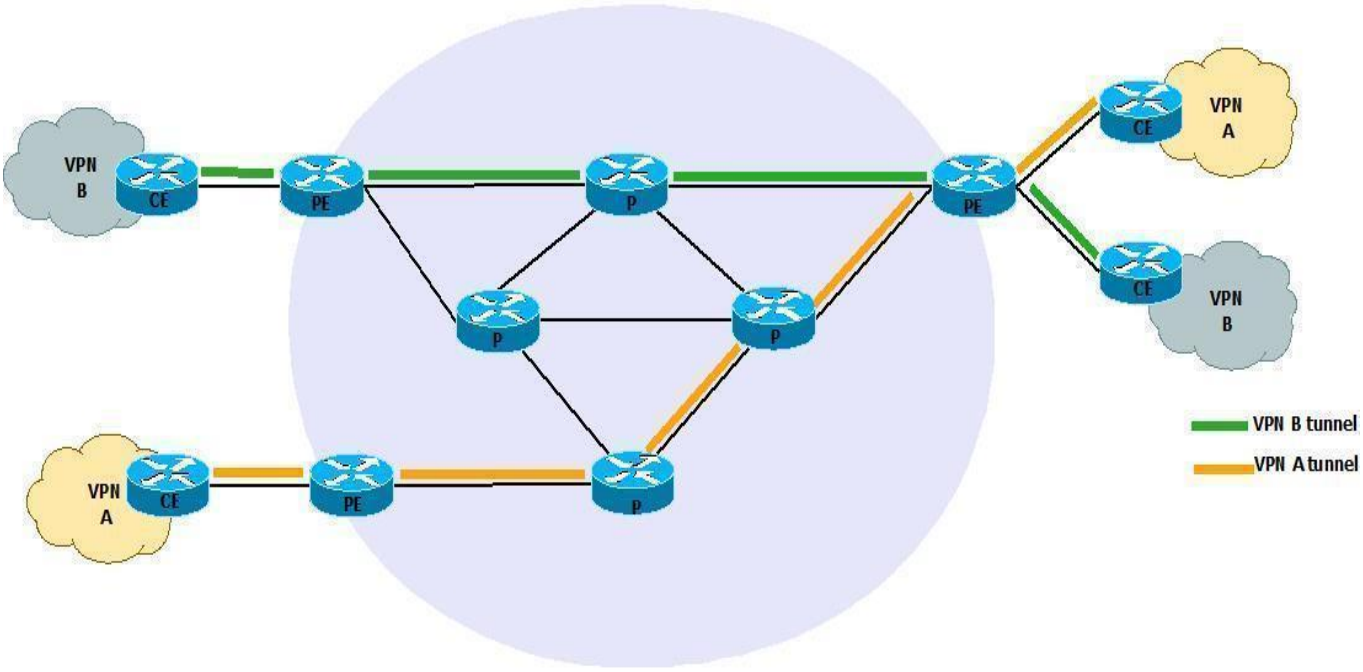


Figure 1.a.1

The internal part of the network, which is usually a WAN, the Provider routers (P in the above diagram) are label switch routers (LSR) which means that they only make forwarding decisions for forwarding MPLS frames. At the edge of the network, the Provider Edge routers (PE in the above diagram) are label edge routers (LER), they Push/Pop a 32-bit header to the data packet, this header includes a label or virtual circuit identifier, as well as forwarding frames in appropriate direction.

Paths through the MPLS network are established before data flow by using a routing protocol that allows the LER to discover the topology of the MPLS network, a tunnel that is established through the network is called a Label Switched Path (LSP). Frames that have the same destination and same QoS (QoS defines priority/class of service) form a forwarding equivalence class (FEC) within an LSP. Edge routers establish FECs for new traffic and inform other LERs and LSRs about this new FEC using the label distribution protocol (LDP); Then, LSRs agree on the labels they will use on each link for this FEC.

Typical scenario:
- A packet arrives at an LER, this ingress LER pushes a header into the data packet with a label identifying its FEC.
- This MPLS frame is then forwarded through the MPLS network by the LSRs that changes the label for each next link.
- At the other edge, the egress LER pops off the header and forwards the data packet. [4]

The original idea behind MPLS was for fast and efficient switching of any layer-3 protocol, but in practice, MPLS is used almost exclusively for IP.

Traffic Engineering:

Traffic engineering is a method of optimizing the performance of a telecommunications network by dynamically analyzing, predicting and regulating the behavior of data transmitted over that network. Traffic engineering is also known as teletraffic engineering and traffic management. The techniques of traffic engineering can be applied to networks of all kinds, including the PSTN, LANs , WANs, cellular telephone networks, proprietary business and the Internet.

The theory of traffic engineering was originally conceived by A.K. Erlang, a Danish mathematician who developed methods of signal traffic measurement in the early 1900s. Traffic engineering makes use of a statistical concept known as the law of large numbers (LLN), which states that as an experiment is repeated, the observed frequency of a specific outcome approaches the theoretical frequency of that outcome over an entire population. In telecommunications terms, the LLN says that *the overall behavior of a large network can be predicted with reasonable certainty even if the behavior of any single packet cannot be predicted*. When the level of network traffic nears, reaches or exceeds the design maximum, the network is said to be congested.

Traffic Engineering is needed in the internet mainly since current IGPs always use the shortest paths to forward traffic. Using shortest paths conserves network resources, but it may also cause the following problems:

- ➔ The shortest paths from different sources overlap at some links, causing congestion on those links.
- ➔ The traffic from a source to a destination exceeds the capacity of the shortest path, while a longer path between these two routers is under-utilized. [5,6]

MPLS TE Overview:

In a traditional IP forwarding paradigm, packets are forwarded on a per-hop basis where a route lookup is performed on each router from source to destination. The destination-based forwarding paradigm leads to suboptimal use of available bandwidth between a pair of routers in the service provider network. Predominantly, the suboptimal paths are under-utilized in IP networks. To avoid packet drops due to inefficient use of available bandwidth and to provide better performance, TE is employed to steer some of the traffic destined to follow the optimal path to a suboptimal path to enable better bandwidth management and utilization between a pair of routers. TE, hence, relieves temporary congestion in the core of the network on the primary or

optimal cost links. TE maps flow between two routers appropriately to enable efficient use of already available bandwidth in the core of the network. The key to implementing a scalable and efficient TE methodology in the core of the network is to gather information on the traffic patterns as they traverse the core of the network so that bandwidth guarantees can be established. Hence, TE tunnels, Tunnel1 and Tunnel2, can be configured on the edge router PE that can map to separate paths (say PATH1, PATH2), enabling efficient bandwidth utilization. TE tunnels are, thus, data flows between a specific source and destination that might have properties or attributes associated with them.

The attributes associated with a tunnel, in addition to the ingress (headend) and egress (tailend) points of the network, can include the bandwidth requirements and the QoS for data that will be forwarded utilizing this tunnel. Traffic is forwarded along the path defined as the TE tunnel by using MPLS label switching. Hence, TE tunnels are assigned specific label switched paths (LSPs) in the network from source to destination, which are usually PE routers. MPLS LSPs have a one-to-one mapping with TE tunnels, and TE tunnels are not bound to a specific path through the SP network to a destination PE router. Unless configured explicitly, TE tunnels can reroute packets via any path through the network associated with an MPLS LSP. This path might be defined by the IGP used in the core; MPLS TE also lends itself to a resilient design in which a secondary path can be used when the primary path fails between two routers in a network.[7]

VPLS:

Virtual private LAN service (VPLS) is a technology that makes it possible to connect local area networks (LANs) over the IP Networks, so that they appear to subscribers like a single Ethernet LAN. A VPLS uses multiprotocol label switching (MPLS) to create the appearance of a virtual private network (VPN) at each subscriber location. A VPLS moves each subscriber's Ethernet packets seamlessly to other locations by tunneling them through the provider network, independent of traffic from other Network users. Fault-tolerance ensures that each packet arrives intact at its intended destination. A VPLS is easy to use because subscribers do not have to connect directly to the Network. Instead, they connect as if to an Ethernet network.

A VPLS can provide point-to-point (only 2 end point VPLS, however Pseudowires is traditionally used for point to point connectivity) and multipoint services, as well as any-to-any capability. It is possible to build a VPLS over a wide geographic area, and the technology allows for subscribers to change locations easily. The service is also scalable. A VPLS can serve anywhere from a few subscribers up to hundreds of thousands. [1]

Creating the MPLS Overlay

In the context of SDN, MPLS is an addition to the packet header—an encapsulation that allows the operator of an IP network to create overlays or logical tunnels on the IP network (the underlay), as shown in Figure 1.a.2.

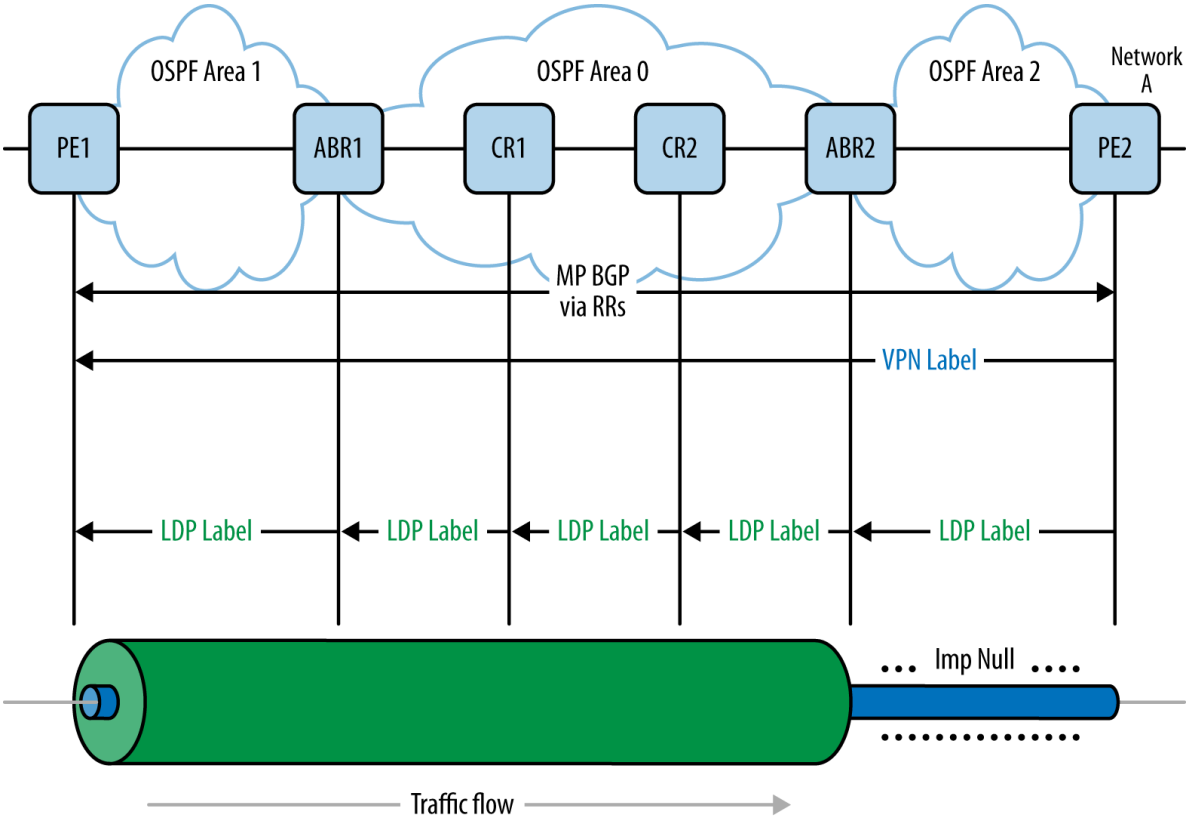


Figure 1.a.2. An MPLS VPN (VRF label distribution via route reflection) over an OSPF multiarea underlay

The label itself is 24 bits, which means there are 1,048,575 labels (the labels 0 through 15 are reserved), as shown in Figure 1.a.3.

Figure 1.a.3. MPLS label

Labels can be stacked in a LIFO (last in, first out) order. The stacking of labels allows for the creation of multiple services or tunnels across a network. These were precursors to today's network overlays.

- A single label can enable an expedited lookup in the label table versus the IP forwarding table.
- Two labels create an abstraction that enables isolation, like that of the VPN where the external label expedited forwarding to an element with multiple virtual instances (VRFs) whose discriminator is the inner label, as shown in Figure 1.a.4.

- Three or four labels create abstractions that enable the same forwarding through an intervening tunnel (unprotected or protected), like VPNs constructed over traffic engineering tunnels (with or without fast reroute protection).

Like the IGP, many books have been written about the operation of MPLS, so we will not attempt to explain it all, but again, a general description will help with our SDN discussion going forward.

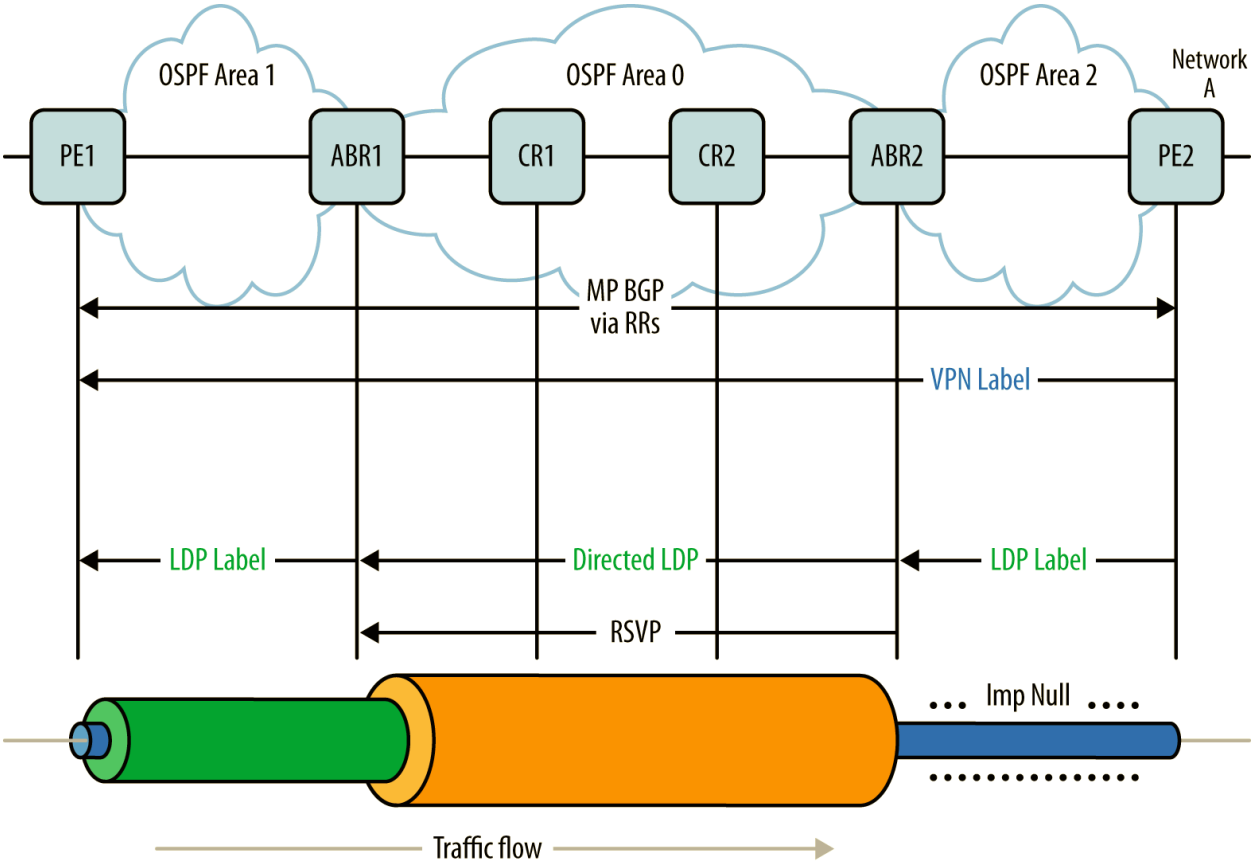


Figure 1.a.4. An MPLS VPN (VRF label distribution via route reflection) over an MPLS-TE core (all over an OSPF underlay) [Source: SDN: Software Defined Networks by Thomas D. Nadeau and Ken Gray]

The main aspects of MPLS operation involve label allocation, address binding, and label distribution—all of which are controlled by configuration:

- The label distribution protocols can be LDP, RSVP (and BGP for the labeled unicast address family). These control protocols have neighbor/session forming behaviors and information exchange.
- Label allocation is normally dynamic, but label scale can be controlled somewhat in some vendor implementations particularly in the context of VPNs by per-VRF allocation or per-prefix/per-platform allocation. The assignment of these labels can be ordered (but this is not a requirement).

- Label distribution can be downstream on-demand (e.g., RSVP for traffic engineering) or downstream unsolicited which is the default behavior of LDP.

Like the IGP, certain aspects of MPLS control plane behavior can be controlled by global and local configuration with the same limitations listed previously. This includes the ability to filter label advertisements, control label retention policy, control label range and the use and distribution of reserved labels. The network element can perform label actions that include push, pop, swap, multiple push, and swap-and-push (in addition to forward). Historically, not all network elements were capable of performing all of these actions, nor were they capable of adequately supporting deeper label stacks.

When MPLS is deployed, the forwarding behavior of the data plane changes from longest destination prefix match to a match of the topmost label on the label stack.
However, the forwarding path will still follow the acyclic graph computed for the destination prefix. While this leads to a more expeditious lookup, it adds complexity by maintaining additional tables and references between the IP forwarding table and the label table. MPLS also adds to the overall complexity of the distributed IP control paradigm.

The specific application of MPLS traffic-engineered tunnels allows the operator to control the path of tunnels and thus exploit areas of the network not used for ordinary destination prefix-based forwarding. These MPLS tunnels are loaded based on the next hop address of a class of prefixes, called a Forwarding Equivalence Class (FEC). A FEC can also be a set of policies that specifically identify specific flows or quality of service characteristics of the flows such as those used by policy-based routing.

Like the IP IGP, MPLS has been enhanced over time, particularly in the area of multipath load balancing through innovations like the creation of sub-LSPs and entropy labels.

Section 1.b - SDN Controllers in Relationship to MPLS

SDN Networks:

When the Internet (the network of networks) was invented, during the second world war, the concept was to have a static sturdy reliable network, with no central point that can fail and cause the whole network to go down, however the trade off was inflexibility. This concept became no longer sufficient for the modern communication world. which led to the concept of SDN, or Software Defined Networks, which in a nutshell means to be able to program most if not all network operations using ordinary programming languages.

Since SDN means the physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices; it can cause a great shift in the network engineering world from distributed protocols to centralized APIs, where networking will be part of computing and not so separate from it.

In abstraction, for the control plane to accomplish its task, it must:

- Figure out what network looks like (topology)
- Figure out how to accomplish a goal on a given topology
- Tell the switches what to do (configure forwarding state)

Figure 1.b.0 illustrates the logical structure of an SDN system. A central controller performs all complex functions, including routing, naming, policy declaration, and security checks. This plane constitutes the *SDN Control Plane*, and consists of one or more SDN servers.[11]

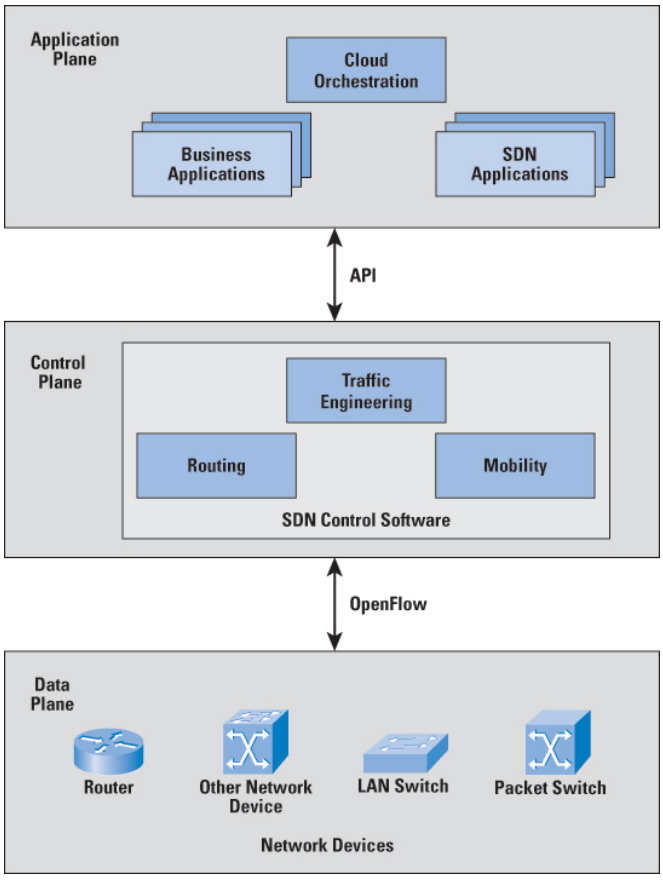


Figure 1.b.0 [Source: William Stalling on Cisco website]

For SDN, there are two control plane abstractions:

1. Global network view: provides information about current network, which is implemented with “Network Operating System”

2. Forwarding model: provides a standard way of defining forwarding state by using OpenFlow specification of <header,action> flow entries

Most open source SDN controllers revolve around the OpenFlow protocol, few of the commercial products use the protocol exclusively, most use it in conjunction with other protocols. Besides the use of OpenFlow and proprietary protocols, there are SDN controllers that leverage IP/MPLS network functionality to create MPLS VPNs as a layer 3-over-layer 3 tenant separation model for data center or MPLS LSPs for overlays in the WAN.

OpenFlow Protocol:

The origins of OpenFlow can be traced back to 2006, when Martin Casado, a PhD student at Stanford University, California, developed something called Ethane.Intended as a way of centrally managing global policy, it used a "flow-based network and controller with a focus on network security", according to OpenFlowNetworks.com, a site dedicated to tracking the emerging technology, along with SDN. That idea eventually led to what became known as OpenFlow

According to ONF (Open Network Foundation) OpenFlow is a standardized protocol for remotely interacting with the forwarding behaviors of switches from multiple vendors (cross-vendors switch forwarding); which provides a way to control the switches behavior throughout a network dynamically and programmatically. Upon this low-level primitive, researchers can build networks with new high-level properties. For example, OpenFlow enables more secure default-off networks, wireless networks with smooth handoffs, scalable data center networks, host mobility, more energy-efficient networks and new wide-area networks – to name a few. [8]

In regards to the OpenFlow architecture, the control plane in all switches and routers in the network is moved to a separate controller/server; this controller communicates with the network switched over a secure channel using the OpenFlow protocol. The controller's software dynamically programs the switches, modifying the flow specifications, which controls the routes of packets through the network.

It is key to realize that the OpenFlow protocol is a set of protocols and an API, in other words, the controller does nothing without an application program, or more, giving instructions on which flows go on which elements. Figure 1.b.1 shows the OpenFlow logical architecture where some of the control plane applications will function on the controller, emulating the behavior of traditional control plane applications.

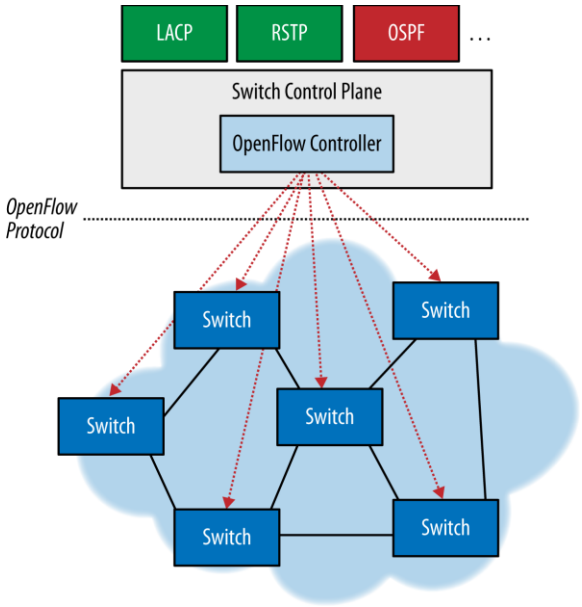


Figure 1.b.1 [Source: SDN: Software Defined Networks by Thomas D. Nadeau and Ken Gray]
Currently, OpenFlow protocols are divided in two parts:

1- A wire protocol to:

- ➔ Establish a control session.
- ➔ Define a message structure for exchanging flow modifications (flow-mods).
- ➔ Collect statistics, and defines the fundamental structure of a switch (ports and tables).

Since the flow entries are no longer stored in a permanent storage in the network component, this makes the OpenFlow protocol interestingly attractive to the network community, as it introduces the concept of substituting the ephemeral state of for the rigid and unstandardized definitions of various vendors' protocol configuration. Also, in an OpenFlow flow entry, the entire packet header (specially the layer 2 and layer 3 fields) are available for match and modify actions. These have evolved over the different releases of OpenFlow. Figure 1.b.2 illustrates the complexity of implementing the L2+L3+ACL forwarding functionality. The combination of primitives supported from table to table leads to a very large combination of possibilities to support.

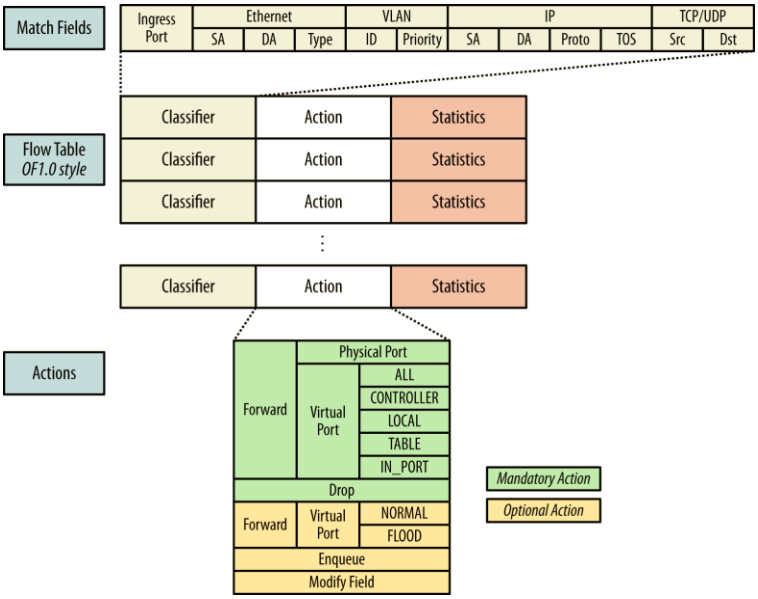


Figure 1.b.2 [Source: SDN: Software Defined Networks by Thomas D. Nadeau and Ken Gray]

OpenFlow has an 11-tuple match space; Comparing this to the distributed MPLS model, it is a strikingly different in width of operator control.

2- A configuration and management protocol, of-config based on NETCONF protocol to:

- ➔ allocate physical switch ports to a particular controller.
- ➔ define high availability (active/standby) and behaviors on controller connection failure.

Although OpenFlow can configure the basic operation of OpenFlow command/control it still can't boot or maintain an element. The following figure 1.b.3 shows the OpenFlow controller components:

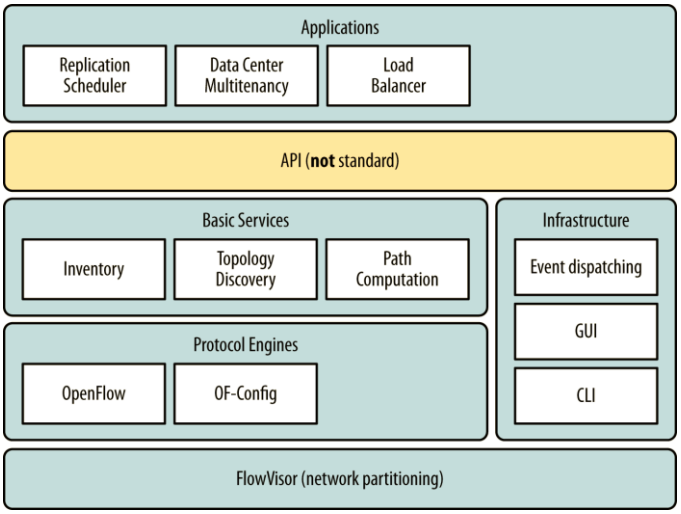


Figure 1.b.3 [Source: SDN: Software Defined Networks by Thomas D. Nadeau and Ken Gray]

OpenFlow protocol is still developing and it didn't reach it's finalized architecture yet. For example, while OpenFlow provides a standardized southbound (controller to element agent) protocol for instantiating flows, there is no standard for either the northbound (application facing) API or the east/west state distribution protocol that allows both application portability and controller vendor interoperability. This is why there were some proposals to implement "Hybrid Networks" using dual function switches that dedicate different port interfaces to either OpenFlow Packets or no OpenFlow packets, where the OpenFlow services are segregated from the native (traditional) switch services. One of these proposals is the SIN (Ships In the Night) as shown in the Figure 1.b.4 below:

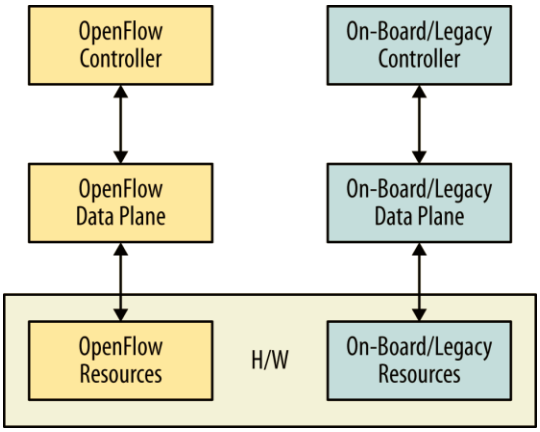


Figure 1.b.4 [Source: SDN: Software Defined Networks by Thomas D. Nadeau and Ken Gray]

Even though there are still questions about the level of abstraction implemented by OpenFlow and whether its API represents a complete SDN API, there is ongoing efforts around hybrid operation that may make it easier to integrate its capability for matching/qualifying traffic in traditional/distributed networks or at the boundaries between OpenFlow domains and native domains.

SDN Controllers:

Figure 1.b.5 depicts a logical view of the SDN architecture. Network intelligence is (logically) centralized in software-based SDN controllers, which maintain a global view of the network. As a result, the network appears to the applications and policy engines as a single, logical switch. With SDN, enterprises and carriers gain vendor-independent control over the entire network from a single logical point, which greatly simplifies the network design and operation. SDN also greatly simplifies the network devices themselves, since they no longer need to understand and process thousands of protocol standards but merely accept instructions from the SDN controllers [9]

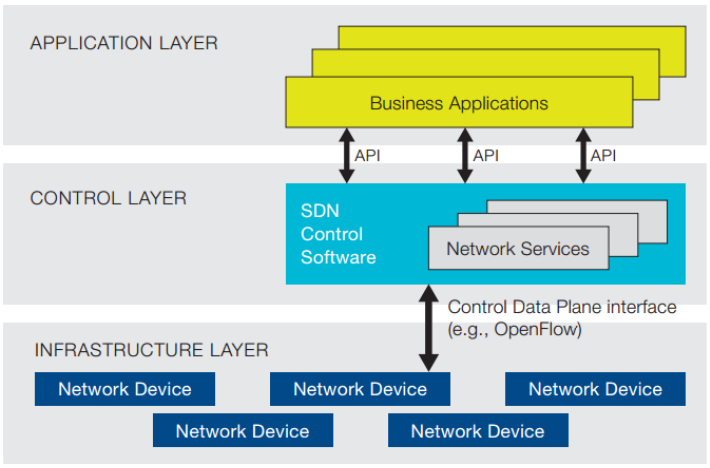


Figure 1.b.5 Software-Defined Network Architecture [Source: ONF SDN White Paper]

The most important concepts of SDN are:

- ➔ Programmability.
- ➔ The separation of the control and data planes.
- ➔ The management of volatile network state in a centralized control model, regardless of the degree of centralization.

The above concepts are ideally introduced via the SDN framework, which is eventually incorporated in an SDN controller.

Ideally, an SDN controller provides services that can realize a distributed control plane, as well as the concepts of temporary state management and centralization. In reality, any given instance of a controller will provide a slice or subset of this functionality, as well as its own take on these concepts.

The general description of an SDN controller is a software system or collection of systems that together provides:

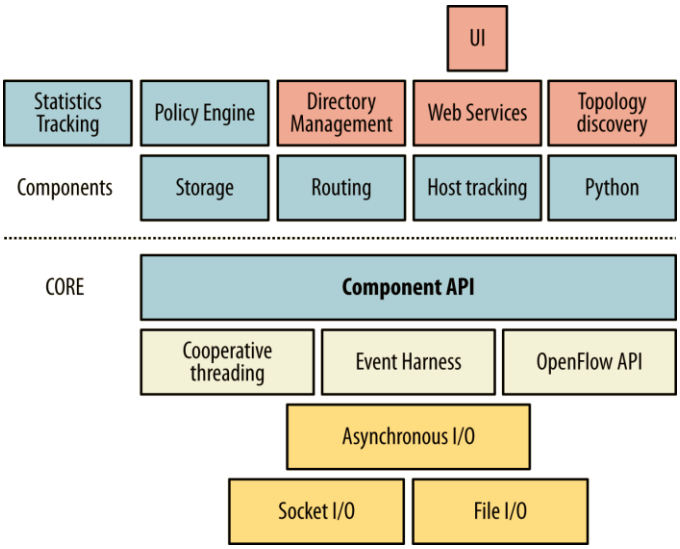
- ➔ Management of network state, and in some cases, the management and distribution of this state, may involve a database.
- ➔ A high-level data model that captures the relationships between managed resources, policies and services provided by the controller (usually using the YANG modeling language)
- ➔ A REST API is provided that reveals the controller services to an application to facilitate the controller-to-application interaction. This interface is ideally derived from the data model that describes the services and features of the controller.
- ➔ A secure TCP control session between controller and the associated agents in the network elements

➔ A standards-based protocol for the provisioning of application-driven network state on network elements

➔ A device, topology, and service discovery mechanism; a path computation system; and potentially other network-centric or resource-centric information services

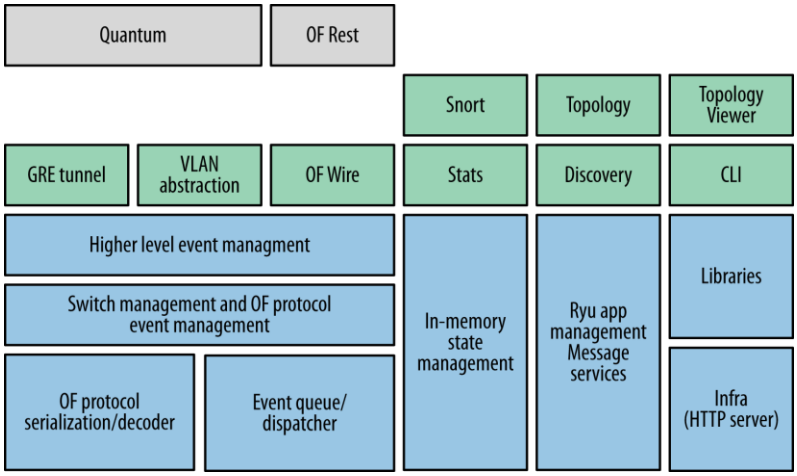
Currently, many SDN controllers are being used and tested; The topmost open source controllers in terms of their usage are : POX, Ryu, Trema, FloodLight, and OpenDaylight. [13]

- **POX**: a python-based SDN controller, inherited from the NOX controller, is used to explore SDN debugging, network virtualization, controller design, and programming models.



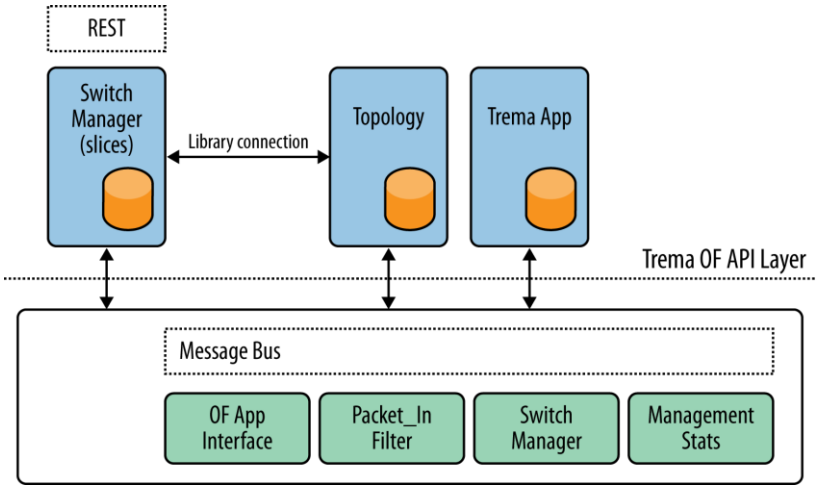
[Source: SDN: Software Defined Networks by Thomas D. Nadeau and Ken Gray]

- **Ryu**: a component-based SDN controller (supported by NTT Labs). Ryu has a set of predefined components that can be modified, extended, and composed for creating a customized controller application. Any programming language can be used to develop a new component.



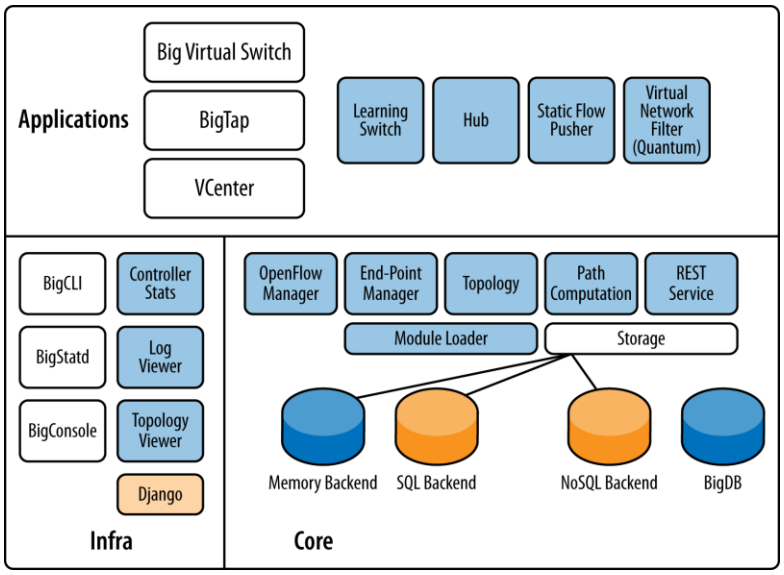
[Source: SDN: Software Defined Networks by Thomas D. Nadeau and Ken Gray]

- **Trema**: supported by NEC labs and its most important design goals are easy-to-write-code and performance. The scripting language "Ruby" is used to increase productivity. The compiler language "C" is used to increase performance.



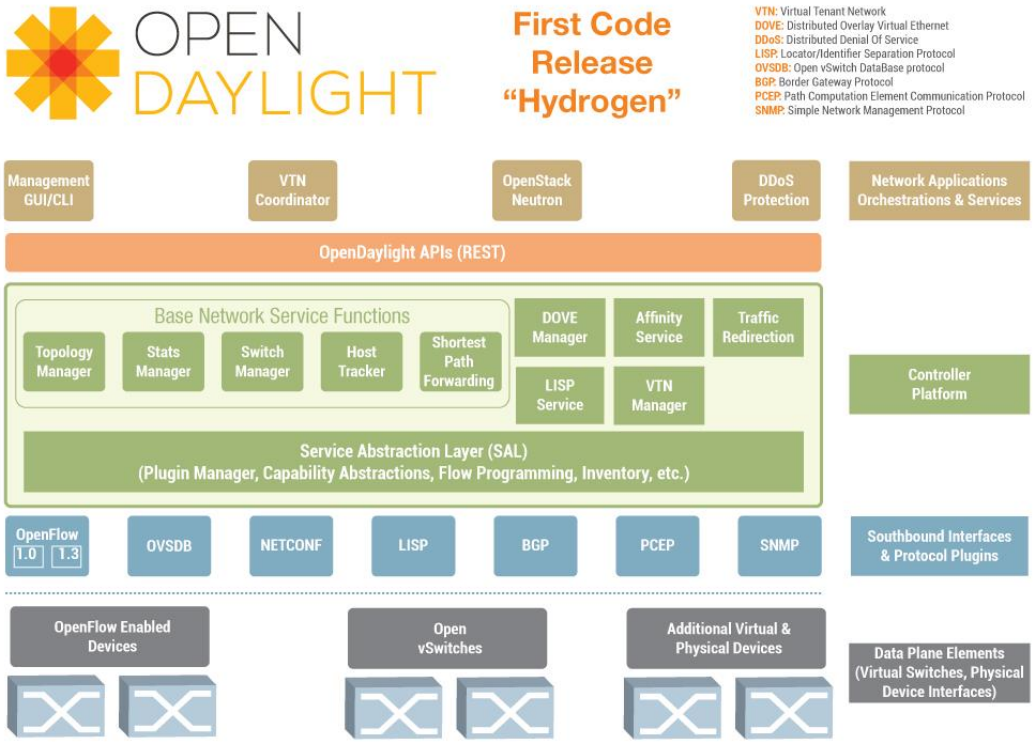
[Source: SDN: Software Defined Networks by Thomas D. Nadeau and Ken Gray]

- **FloodLight**: consists of a set of modules, where each module provides a service to the other modules and to the control logic application through either a simple Java API or a REST API. The controller can run on the top of Linux, Mac and Windows OS.



[Source: SDN: Software Defined Networks by Thomas D. Nadeau and Ken Gray]

- **OpenDaylight**: a project under linux distribution. The goal of the project is to create robust code that covers most of the major components of the SDN architecture, to gain acceptance among the vendors and users, and to have a growing community that contributes to the code and uses the code for commercial products.



[Source: http://www.opendaylight.org/]

The decision to choose which SDN controller depends on the goals and requirements of the project, the following is a short comparison between these five SDN controllers:

	POX	Ryu	Trema	FloodLight	OpenDaylight
Interfaces [SB=SouthBound]	SB (OpenFlow)	SB (OpenFlow)	SB (OpenFlow)	SB (OpenFlow)	SB (OpenFlow)
Virtualization	Mininet & Open vSwitch	Mininet & Open vSwitch	Built-in Emulation Virtual Tool	Mininet & Open vSwitch	Mininet & Open vSwitch
GUI	Yes	Yes	No	Web UI (using REST)	Yes
REST API	No	Yes	No	Yes	Yes
Productivity	Medium	Medium	High	Medium	Medium
Open Source	Yes	Yes	Yes	Yes	Yes
Documentation	Poor	Medium	Medium	Good	Medium
Language Support	Python	Python Specific + Message Passing Reference	C/Ruby	Java + any REST supporting language	Java

Modularity	Medium	Medium	Medium	High	High
Platform Support	Linux, Windows, and Mac OS	Mostly Linux	Only Linux	Linux, Windows, and Mac OS	Linux
OpenFlow Support	OF v1.0	OF v1.0, v2.0, v3.0 & Nicira Extensions	OF v1.0	OF v1.0	OF v1.0
TLS Support	Yes	Yes	Yes	Yes	Yes

2.0 Case Study

Section 2.a - MPLS challenges in comparison to SDN challenges

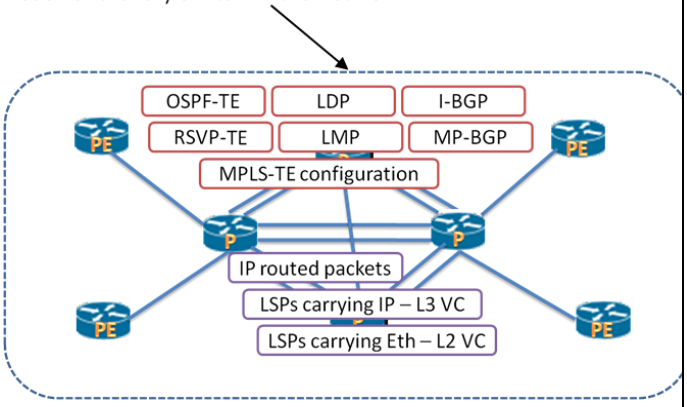
MPLS networks have evolved over the last 10-15 years to become critically important for ISPs. They provide two key services: traffic engineering in IP networks and L2 or L3 enterprise VPNs. However as carriers deploy MPLS networks, they find that

(a) Even though the MPLS data plane was meant to be simple, vendors end up supporting MPLS as an additional feature on complex, energy hogging, expensive core routers; and

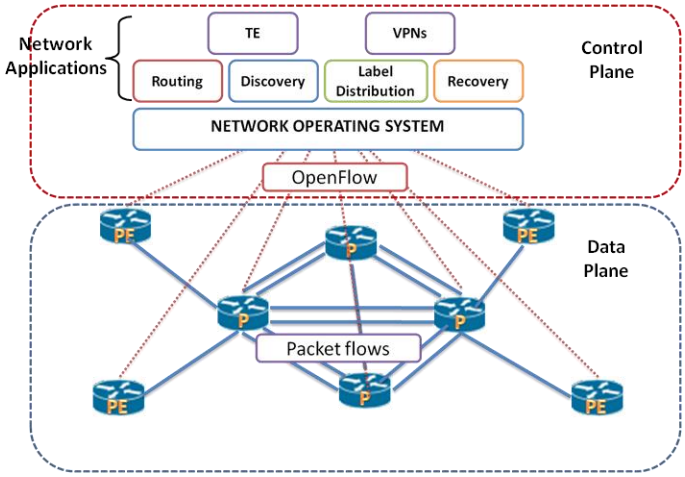
(b) The IP/MPLS control plane has become exceedingly complex with a wide variety of protocols tightly intertwined with the associated data-plane mechanisms.

So, MPLS doesn't come cheap or simple

MPLS with SDN/Openflow



Today's IP/MPLS Networks



SDN based MPLS

[10]

In the year of 2011, two PhD students at Stanford University* were able to build and test a network to showcase the possibilities SDN/OpenFlow holds in the MPLS world, over the period of just two months (this is how innovation was made simply possible by SDN/OpenFlow).

We will use their model to show how SDN/OpenFlow can emulate the various features on an MPLS Control protocols, that will entirely eliminate the use of LDP, RSVP, ...etc.

In any MPLS network there are simple data plane mechanisms of Pushing on, swapping and popping off the MPLS labels in a label-switched path. In addition, there are a number of control plane protocols that control the operation of these networks and provide the services they enables.

Any changes to these services, however small, or the creation of a new service necessarily require and involve the change to these these protocols or the creation of an entirely new one. Both of which are lengthy and time consuming processes. And yet the data plane mechanisms remain simple the same: PUSH - SWAP - POP operations.

With OpenFlow, we take a different approach, which allows us to go beyond what MPLS provides today. We can provide all the services that MPLS networks provide; but more importantly, OpenFlow made it possible to make changes to existing services or create new ones by changing the networking applications that run on the network operating system. Which means that, new capabilities are no longer tied to extensions to protocols that need to be implemented on each and every router in the network. Plus, OpenFlow doesn't need to change either, for all it

gives is control over the simple, PUSH - SWAP - POP data plane operations which remain the same.

To demonstrate the capability to replicate what MPLS provides today, they built a system in Mininet environment that emulated a wide area network with multiple instances of Open vSwitch and OpenFlow enabled software switch, which was modified to include the standard MPLS data plane. see Figure 2.0

OpenFlow was used as the only control plane protocol and then layered a traffic engineering service on top of that system.

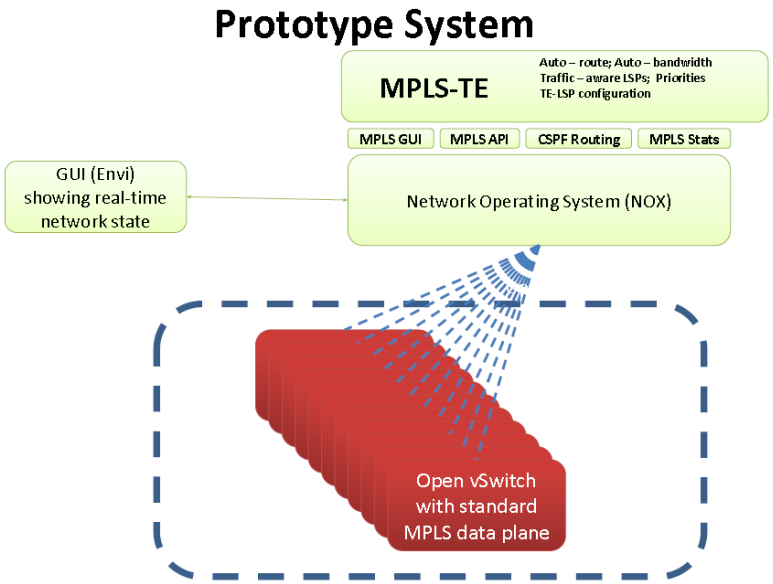


Figure 2.0

Note: traffic engineering is to steer traffic over routes that aren't necessarily the shortest path in the network amongst other things; aiming to avoid congestion and more efficiently utilize network resources.

The MPLS solution would involve creating tunnels, routing traffic through them, and using several tunnel features that help in maintaining and manipulating these tunnels. Such as:

- ➔ Auto-Route
- ➔ Auto-Bandwidth
- ➔ Priorities
- ➔ Load-Share
- ➔ DiffServ aware Traffic Engineering DS-TE

These features were demonstrated without the use of **any** MPLS control plane protocols.

Figure 2.1 shows flows for three different traffic types, represented by three colours, originating in San Francisco and distant to New York, Kansas, and Houston.

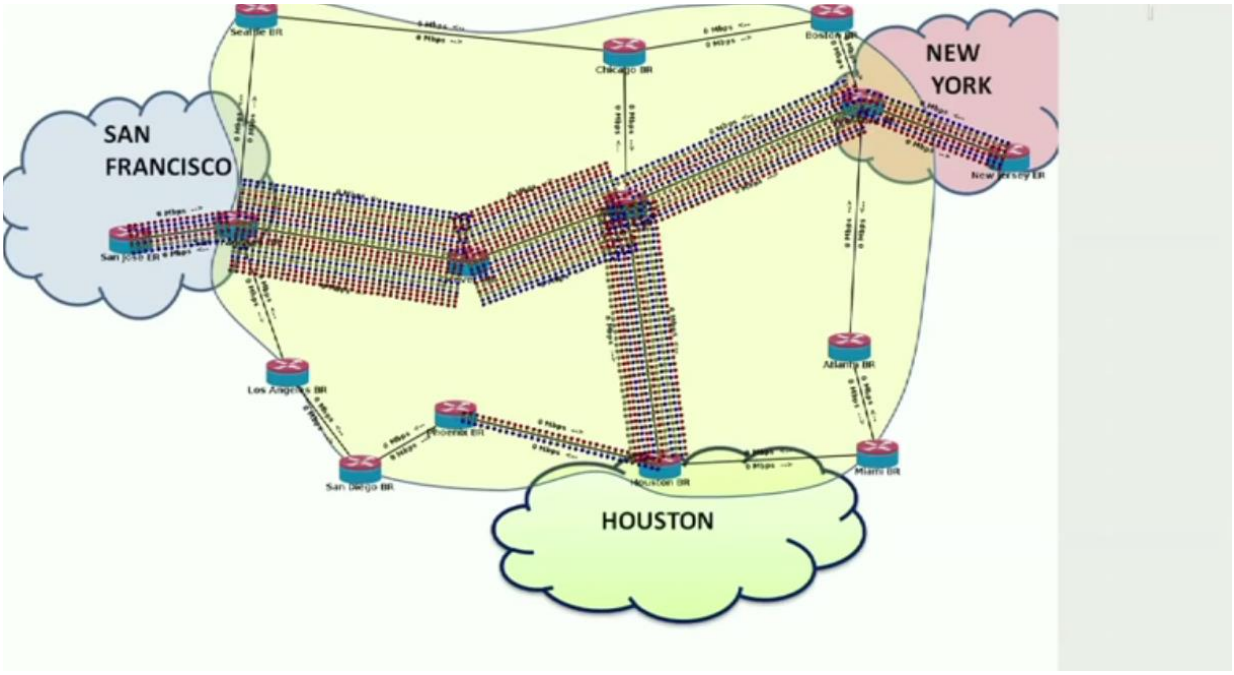


Figure 2.1

At first the traffic was routed just as in a regular IP network, that will take the shortest path in the network based on the destination IP address. It is obvious that this can potentially congest the links between San Francisco, Denver and Kansas.

To start the traffic engineering process, as shown in Figure 2.2, a tunnel was created between San Francisco and New York (green), and a tunnel between San Francisco and Houston (blue) with the following characteristics:

Route: SFO-DEN-KAN-NYC	**Route: SFO-DEN-KAN-HOU**
ResBw: 123 Mbps	ResBw: 700 Mbps
Priority: 0	Priority: 0
Usage: 14 Mbps	Usage: 19 Mbps
AutoBw: OFF	AutoBw: OFF
Traffic: VOIP|VIDEO	Traffic: ALL

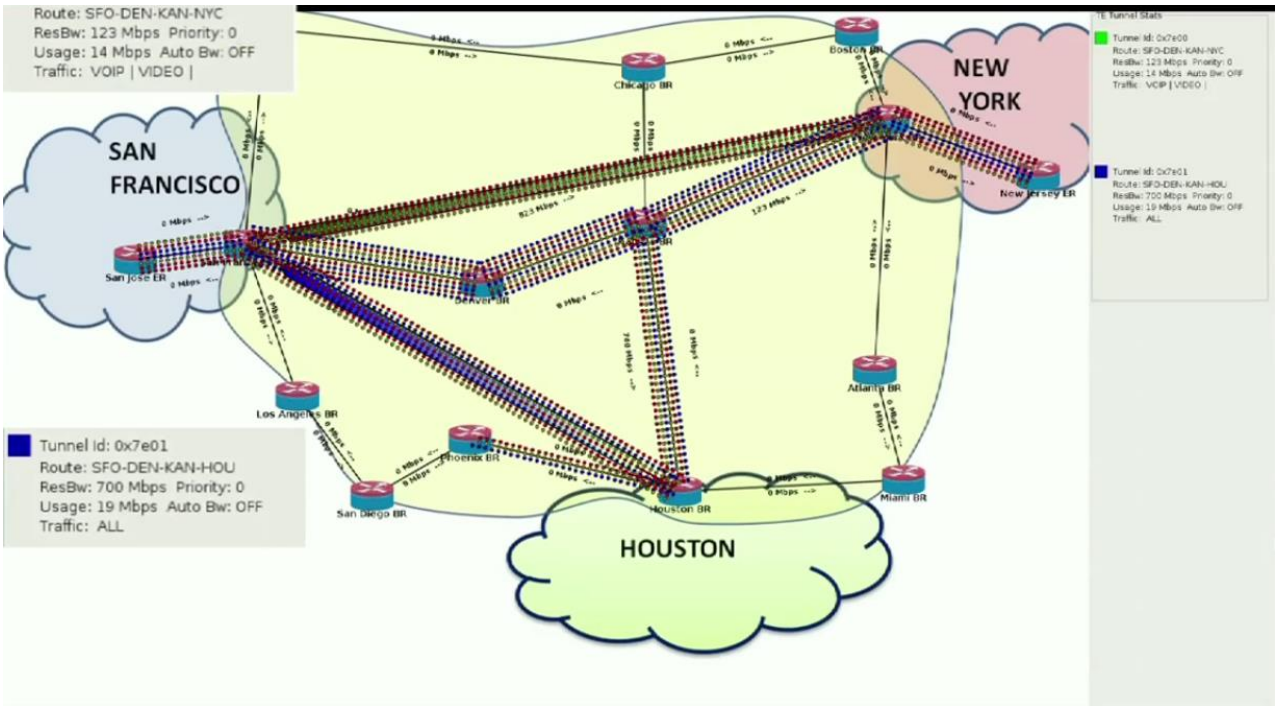


Figure 2.2

We can see that the two tunnels created are still routed over the San_Francisco-Denver-Kansas links, but the accumulative bandwidth that was reserved over those links is 823 Mbps, leaving only 77 Mbps of unreserved bandwidth on those links. If we try to create another tunnel between San Francisco and Kansas with bandwidth reservation greater than 77 Mbps (Figure 2.3), the traffic engineering algorithm forces the tunnel and the traffic it carries to be routed over the under-utilized link between San_Francisco-Seattle-Chicago

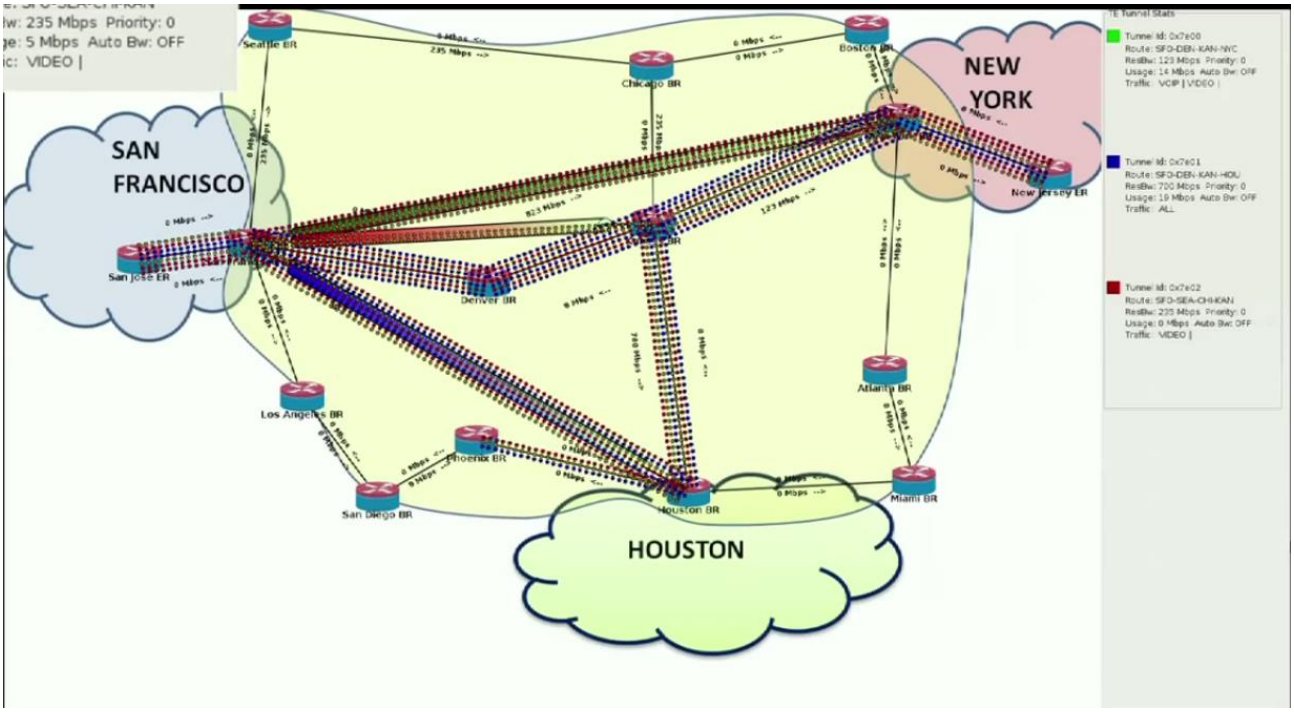


Figure 2.3

We can also create tunnels of specific traffic type. For example:

Route: SFO-SEA-CHI-KAN	**Route: SFO-DEN-KAN-HOU**
ResBw: 235 Mbps	ResBw: 700 Mbps
Priority: 0	Priority: 0
Usage: 5 Mbps	Usage: 19 Mbps
AutoBw: OFF	AutoBw: OFF
Traffic:VIDEO	Traffic: ALL

The San_Francisco-Kansas tunnel accepts only video traffic, while the San_Francisco-Houston tunnel accepts all types of traffic. We can also load balance traffic flows over a tunnel and IP links. For Example: traffic flows between San Francisco and Kansas are routed over a regular IP links between San Francisco-Denver-Kansas. Also, as shown in Figure 2.4, over the orange tunnel which actually takes the Seattle-Chicago route.

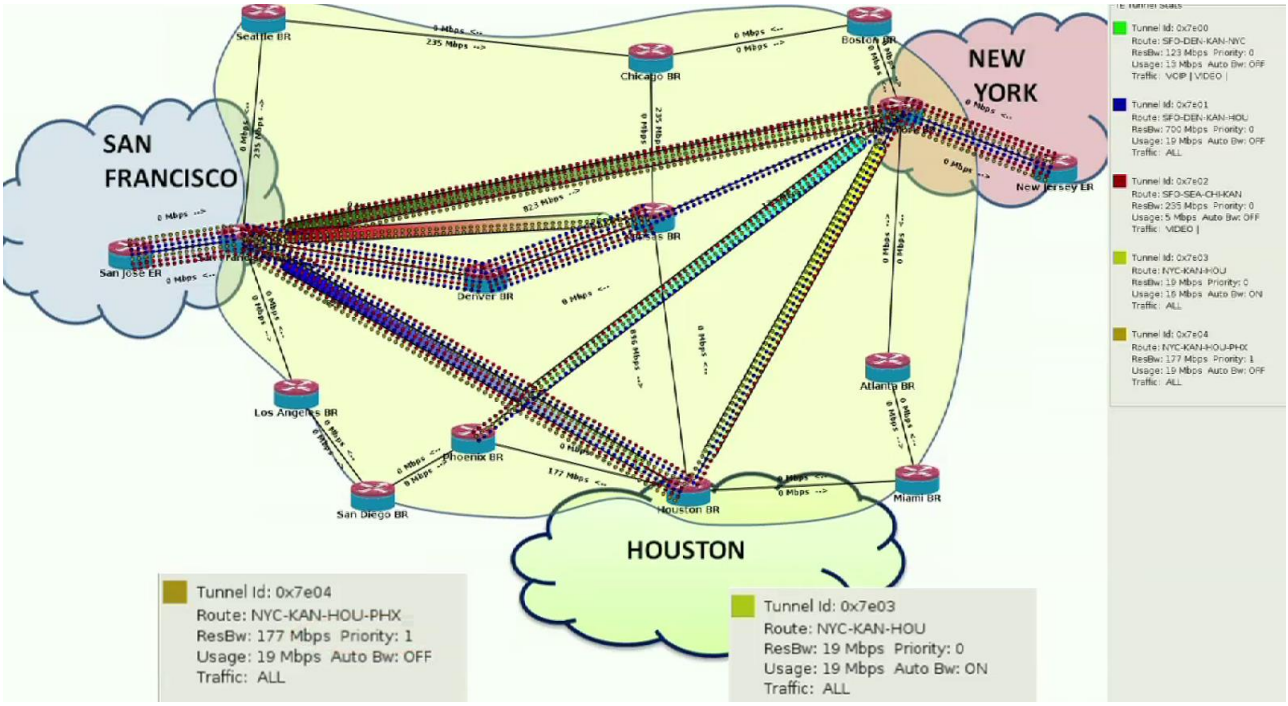


Figure 2.4

Route: NYC-KAN-HOU-PHX	**Route: NYC-KAN-HOU**
ResBw: 177 Mbps	ResBw: **10 Mbps → 19 Mbps**
Priority: 1	Priority: 0
Usage: 19 Mbps	Usage: 19 Mbps
AutoBw: OFF	AutoBw: ON
Traffic: ALL	Traffic: ALL

We can assign other properties to tunnels, for example:

★ The yellow tunnel between New York and Houston, this tunnel has AutoBw feature turned ON, which makes its bandwidth reservation track the usage of the tunnel (the ResBw of 10 Mbps automatically changes to 19 Mbps).

★ The orange tunnel between New York and Phoenix has a priority of 1, which makes it of a lower priority than all the other tunnels of priority 0. the tunnels priorities comes into play when interacting with the AutoBw feature. Note that both tunnels are originating from New York are routed over the Kansas_Houston link, nearly maxing out the bandwidth reservation possible on that link; Now, since AutoBw is ON on the yellow tunnel, any increase in traffic through that tunnel will be tracked by a corresponding

increase in its ResBw (if usage increased to 39Mbps, ResBw increases to 39 Mbps as well). But such an increase will not be possible o the maxed out Kansas-Houston link without bumping off the lower priority New York-Phoenix tunnel, see Figure 2.5

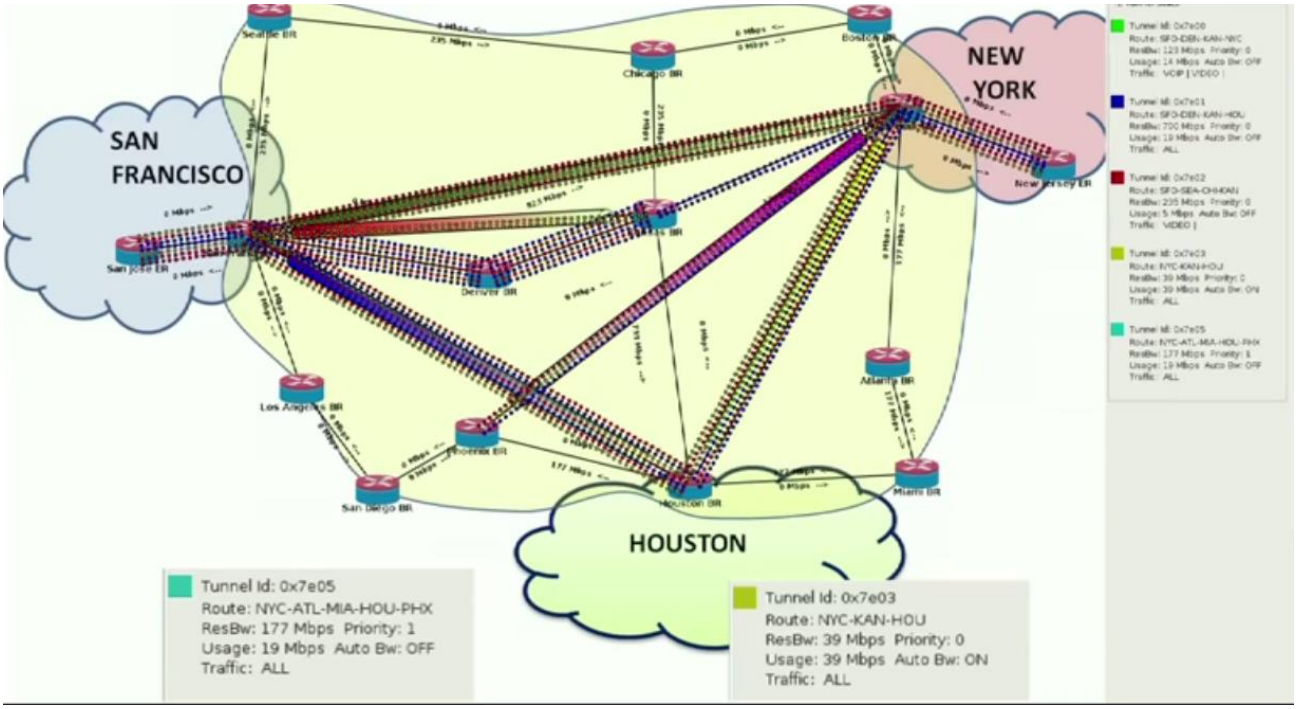


Figure 2.5

Route: NYC-ATL-MIA-HOU-PHX	**Route: NYC-KAN-HOU**
ResBw: 177 Mbps	ResBw: **39 Mbps**
Priority: 1	Priority: 0
Usage: 19 Mbps	Usage: 39 Mbps
AutoBw: OFF	AutoBw: ON
Traffic: ALL	Traffic: ALL

Here we see the blue lower priority New-York-Phoenix tunnel forced to reroute over the Atlanta-Miami links.

To summarize, it took about 4000 Lines-of-Code, two grad students, two months of work, to fully replace all MPLS control plane protocols with only OpenFlow protocol API. innovation steering wheel is now in the hands of all of us who can program..

Section 2.b - What Type of Future Work is Happening in the Industry

SDN is described by many as "The perfect storm" to the networking world. Some networkers think that SDN is an extraordinary technology that's going to change the world of networking. Others see SDN as yet another in a long string of quirky networking ideas that never gained acceptance.

When SDN business model was first proposed, the idea of cross vendor network elements communication, prepared many big vendors to resist the idea. No surprise in that, since now clients can purchase one main expensive network equipment, and integrate it with many less expensive ones. Now, SDN has already reached the researching and trialing stage, and some major companies headspear the industry by already deploying (NTT, Japan). [15]

The companies in the networking world (Juniper, Cisco, Fujitsu, Alcatel, HP, Siemens, ..etc.) currently are facing the choice to either join in and jump into the SDN train to get involved with the development of SDN/OpenFlow, or get left behind. When we started looking into this SDN/OpenFlow project a year ago, Cisco was one of the opposers to invest in this new emerging technology; Few month later, Cisco is knee-deep in the virtualization and SDN adaptation.

On August 2014, David Ramel wrote an article with the title "Software-defined networking gaining traction", in that article he reviews the latest SDN industry progress reports:

"Two recent surveys show similar growth expectations for the technology, with about half of respondents reporting they'll soon have it in production.

Infonetics Research's SDN Strategies: North American Enterprise Survey reported that 45 percent of respondents – who now use SDN or expect to evaluate it – anticipate having SDN in production in their data centers by end of next year, jumping to 87 percent by end of 2016. Meanwhile, Juniper Networks Inc. announced its own Software-Defined Networking Progress Report that found some 53 percent of respondents plan to adopt SDN, with 74 percent of those saying that will happen within the next year. Juniper, however, said its survey also revealed "two distinct camps," with about 47 percent of respondents saying they had no plans whatsoever to adopt SDN."

So SDN and its related technologies, such as OpenFlow, Network Fabric Virtualization (NFV) and open networking in general, have matured enough that the questions are starting to move from, "what is SDN?" to "how do we use SDN in our business?"

One of the key initiatives that Dan Pitt, Executive Director of the Open Networking Foundation (ONF) talked about for the coming months was:

- ➔ Work on setting the standards for OpenFlow version 1.3 conformance. As more and more networking vendors build hardware with OpenFlow 1.3 capabilities, the ONF will work to make sure that these devices meet the standard. This is key, as some of the groundbreaking work going on in adding OpenFlow capabilities to unique hardware, including chip-based OpenFlow.

- ➔ Work on enabling carrier-grade SDN capabilities and continued efforts to have SDN and NFV work well together.

- ➔ Finding more ways that organizations can understand the abilities of SDN and how they can leverage it to meet their complex infrastructure needs. Key to this effort will be the introduction of use cases for SDN, which the ONF plans to unveil at upcoming conferences and summits.

3.0 Lab Work

Section 3 - Mininet

Mininet is a *network emulator*. It runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code. A Mininet host behaves just like a real machine; you can ssh into it and run arbitrary programs (including anything that is installed on the underlying Linux system.) The programs you run can send packets through what

seems like a real Ethernet interface, with a given link speed and delay. Packets get processed by what looks like a real Ethernet switch, router, or middlebox, with a given amount of queueing. When two programs, like an iperf client and server, communicate through Mininet, the measured performance should match that of two (slower) native machines. [14]

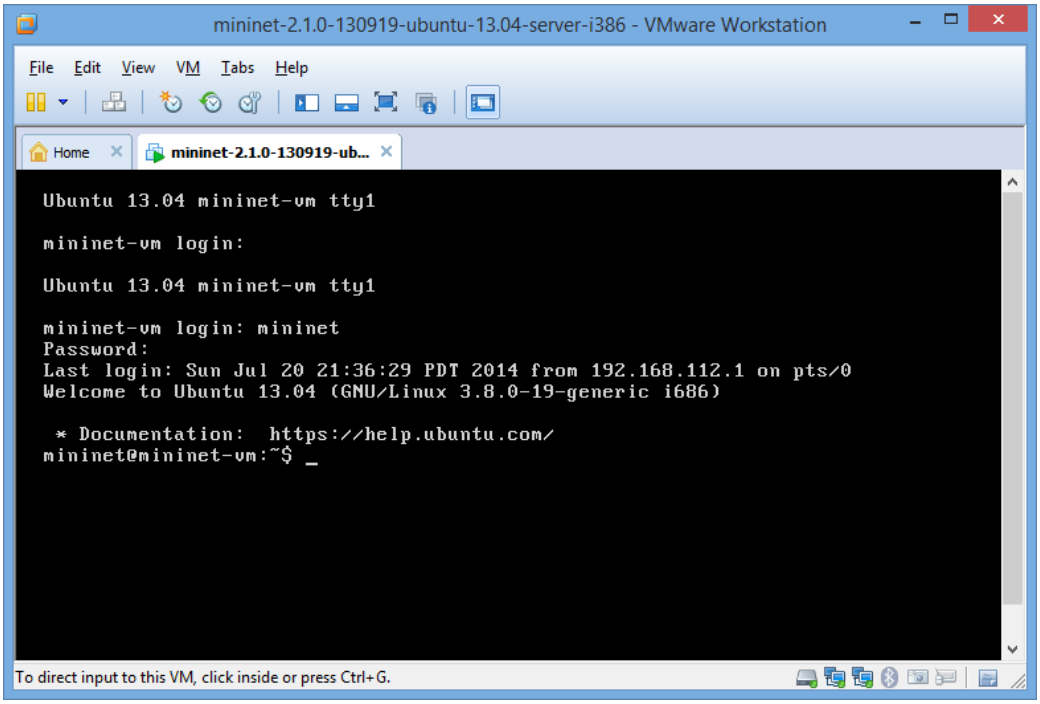
One reason Mininet is widely used for experimentation is that it allows you to create custom topologies, many of which have been demonstrated as being quite complex and realistic, such as larger, Internet-like topologies that can be used for BGP research. Mininet also allows for the full customization of packet forwarding.

Open vSwitch is an open source OpenFlow capable virtual switch that is typically used with hypervisors to interconnect virtual machines with a host and virtual machines between different hosts across networks.It is also used on some dedicated switching hardware. It can be a critical piece in an SDN solution.

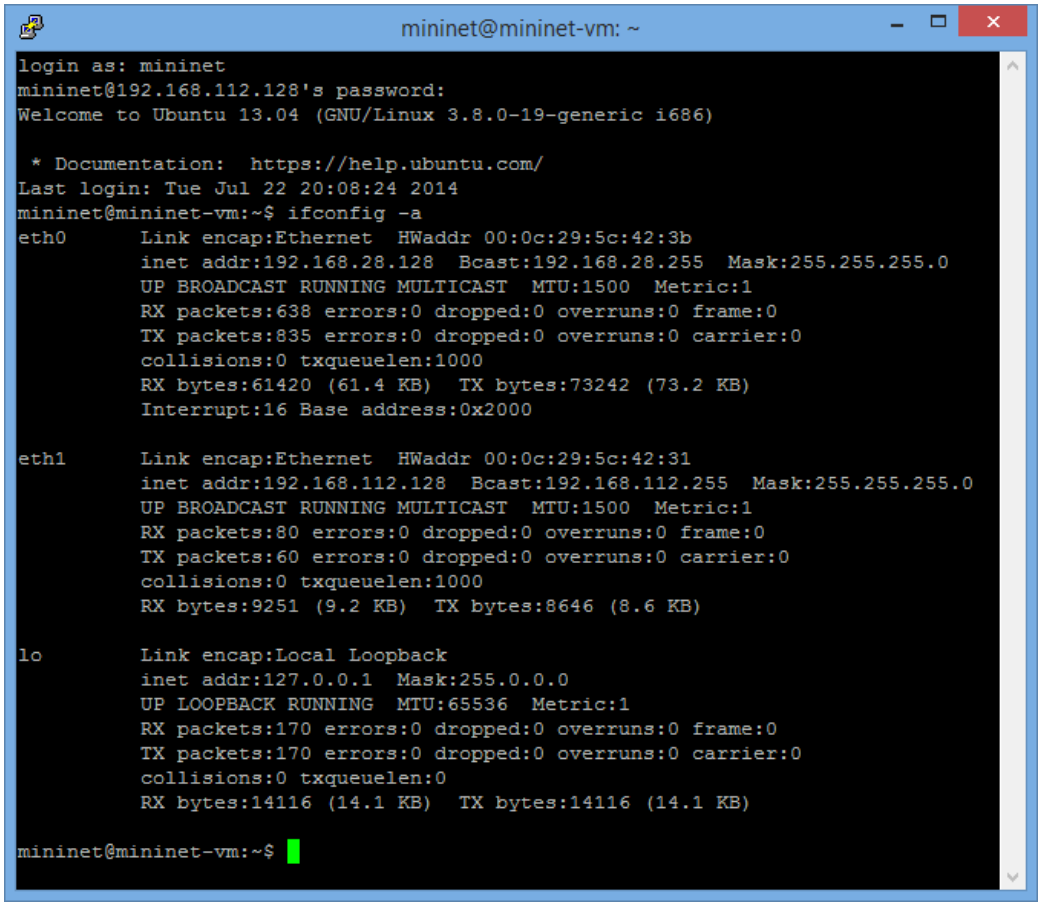
Steps:

1. Download the Mininet VM image. mininet-2.1.0-130919-ubuntu-13.04-server-i386
2. Download and install a virtualization system. VMware Workstation for Windows.
3. Sign up for the mininet-discuss mailing list. This is the source for Mininet support and discussion with the Mininet community.
4. The VM has two network interfaces. One should be a NAT interface that it can use to access the Internet, and the other should be a host-only interface to enable it to communicate with the host machine.
5. Install X server (Xming) to enable X11 forwarding to track packets via Wireshark & SSH terminal (PuTTY)

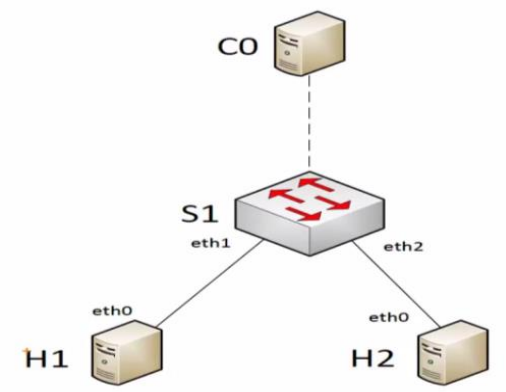
6. Run the Mininet System, username: mininet password: mininet

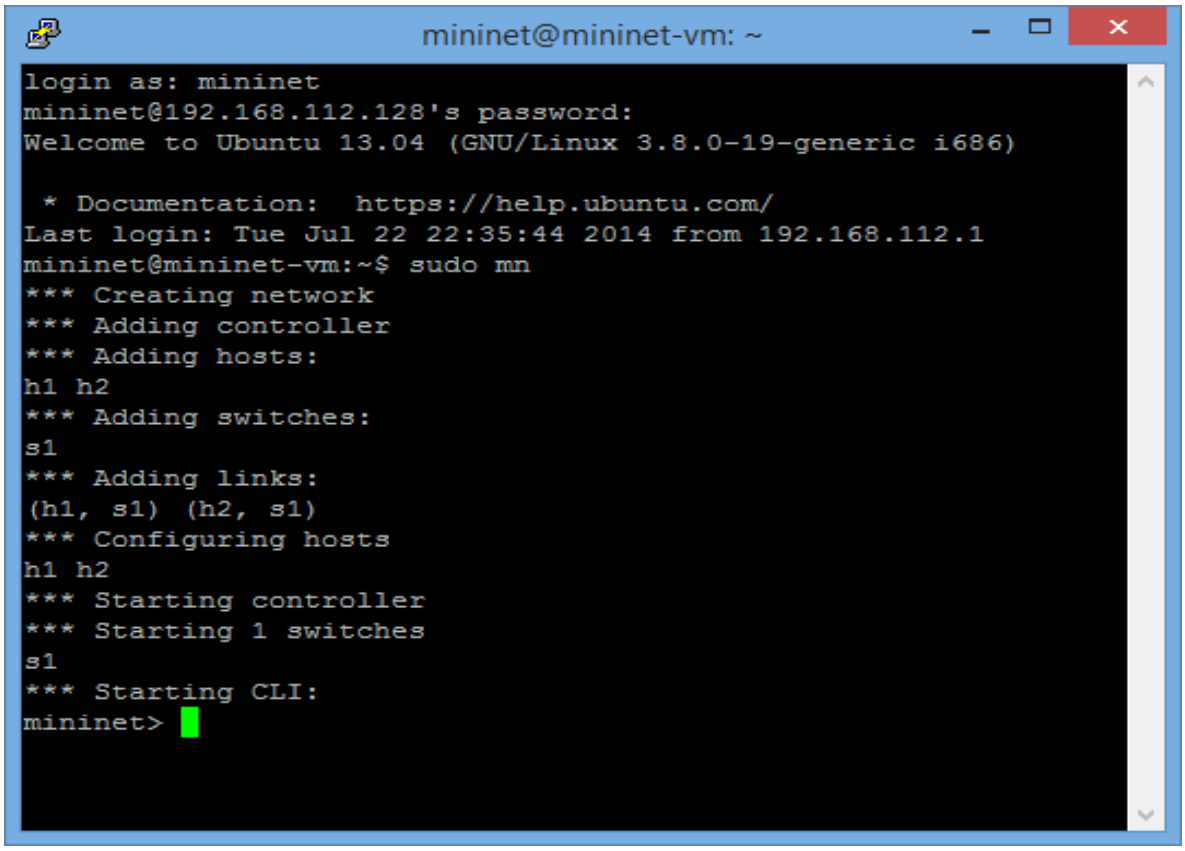


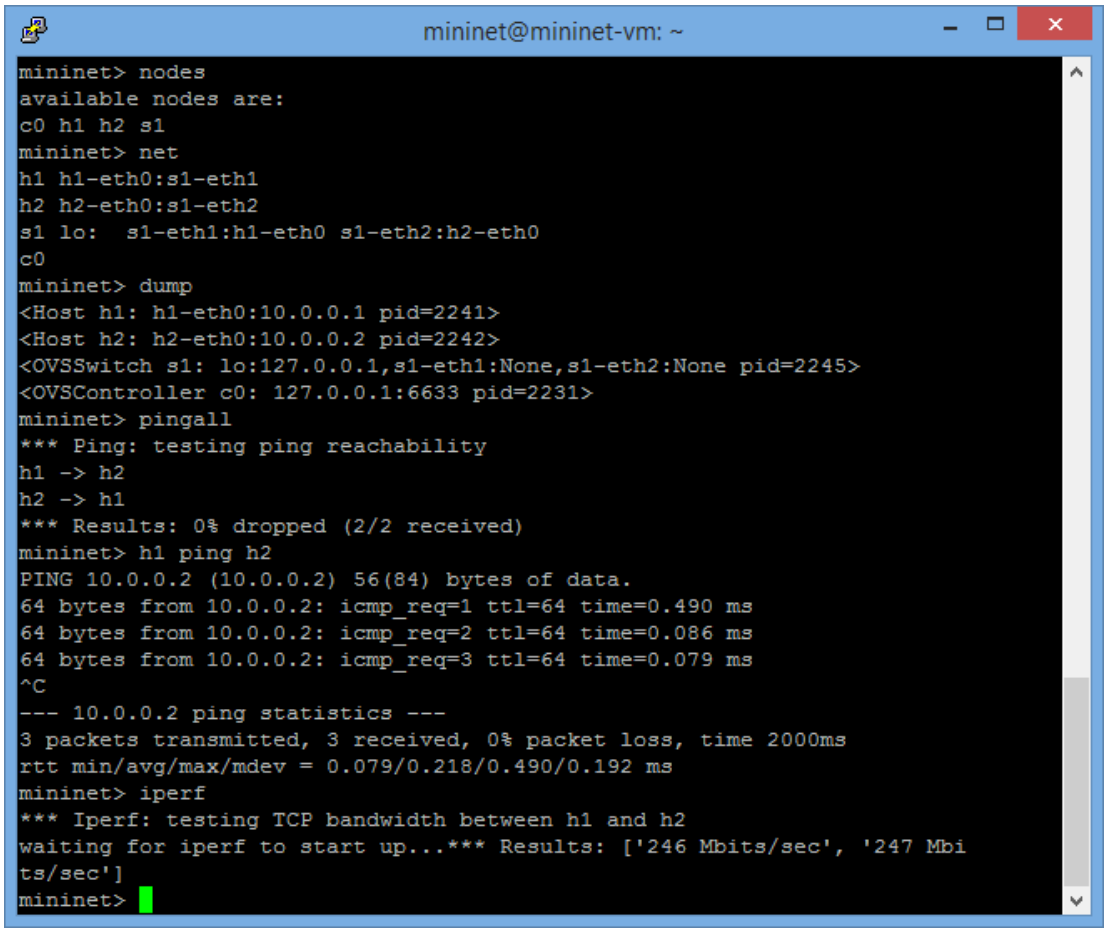
7. SSH into the host-only interface at its associated IP address



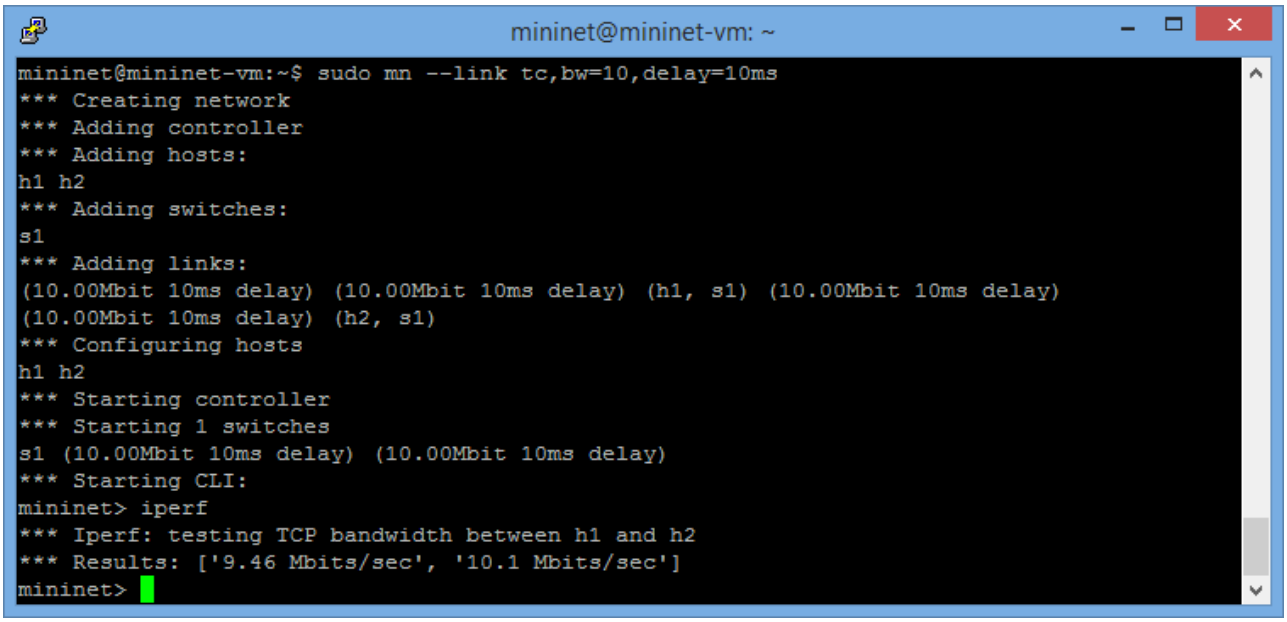
8. Run sudo mn
   This will create a simple SDN network with one OpenFlow reference controller, two hosts(h1, h2), and one routing switch (S1)





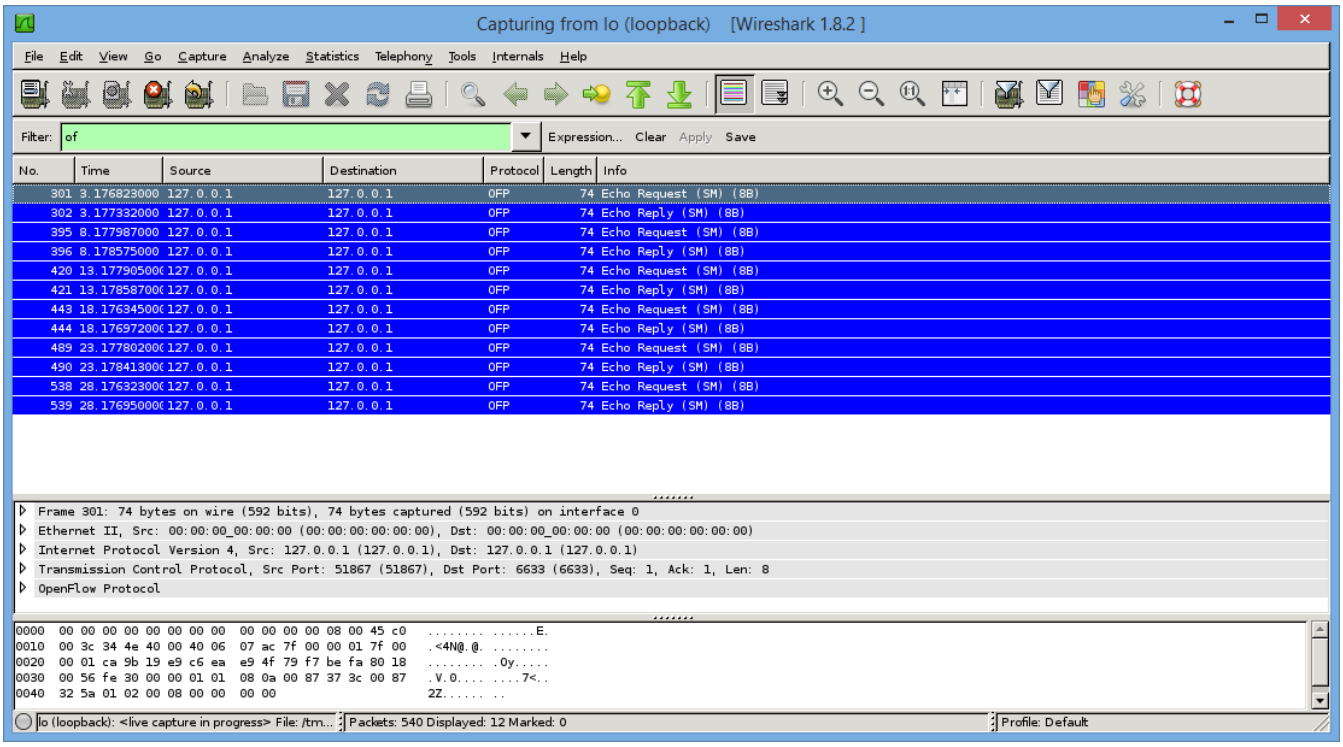


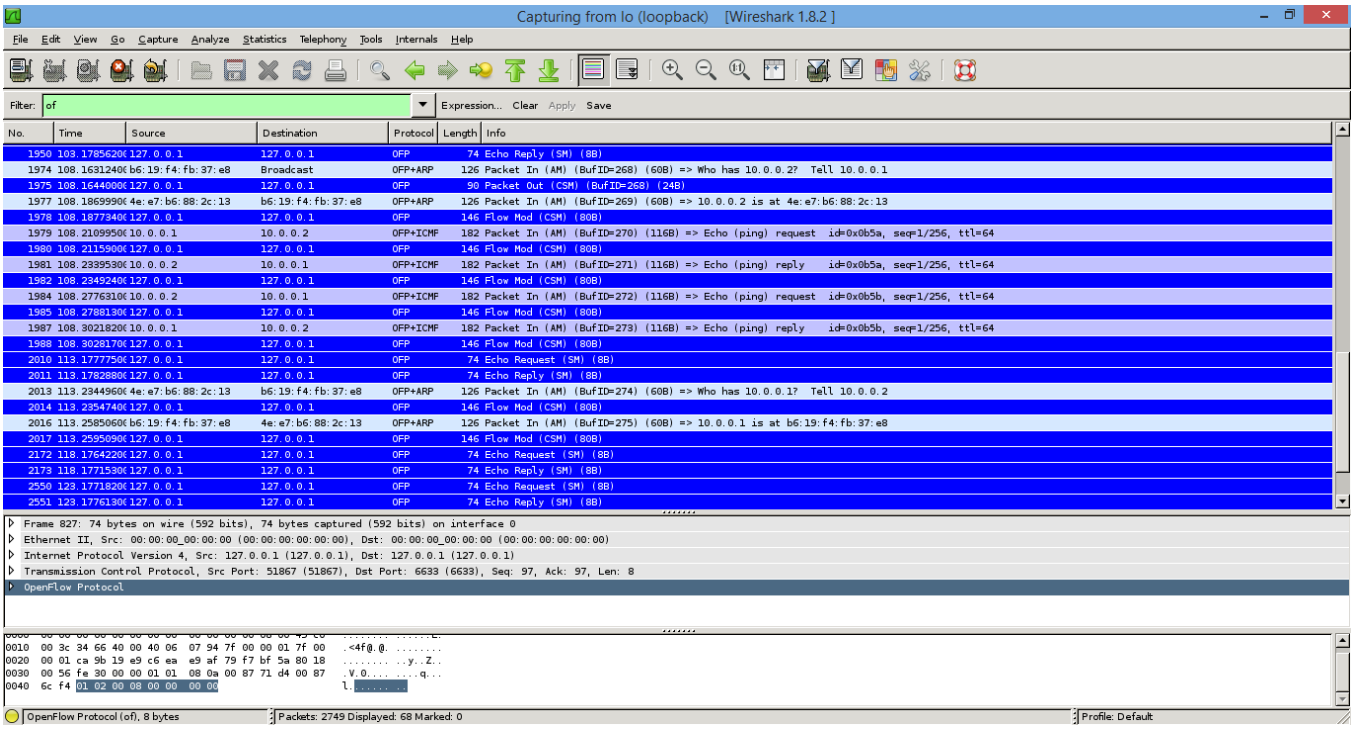
We can control the link bandwidth and delay as follows:



Run [pingall]

Start a second SSH session, and then run [sudo wireshark &]





We can see some OpenFlow packages, as shown in the screenshot above

Now we try to run an SDN controller other than the default controller (C0)

1. Using Floodlight controller (Java based controller)

To install JDK and Ant→

```
$ sudo apt-get install build-essential default-jdk ant python-dev eclipse
```

To install prereqs →

```
$ time sudo apt-get install build-essential default-jdk ant python-dev
```

To Download Floodlight from Github and build:

```
$ git clone git://github.com/floodlight/floodlight.git
$ cd floodlight
$ git checkout fl-last-passed-build
$ sudo apt-get  install ant
```

Now we can directly run the floodlight.jar file produced by ant

```
$ java -jar target/floodlight.jar
```

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

2. Using POX (Python based)

to make sure any other controller instances is closed run:

```
$ sudo killall controller
```

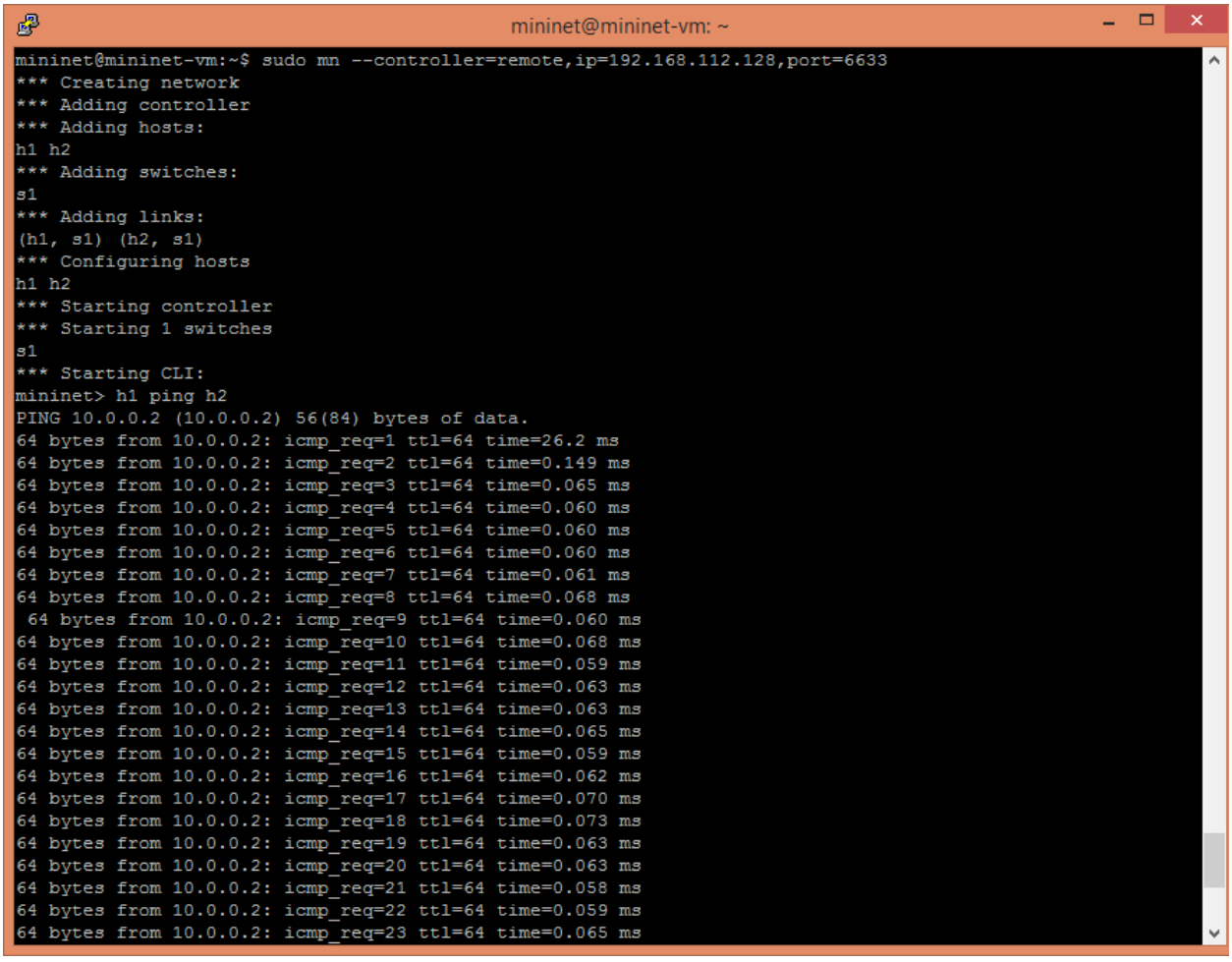
Since, the mininet vm I am working on was downloaded from ww.minnet.org, it already includes POX SDN controller, so I made it run on a second virtual machine "mininet-vm2"

```
mininet@mininet-vm2:~$ cd pox
mininet@mininet-vm2:~/pox$ python ./pox.py forwarding.l2_learning
POX 0.1.0 (betta) / Copyright 2011-2013 James McCauley, et al.
INFO:core:POX 0.1.0 (betta) is up.
```

This will start POX with the basic layer-2 forwarding switch with the controller, then SSH to the first virtual machine "mininet-vm" and start mininet:

```
mininet@mininet-vm:~$ sudo mn --controller=remote,ip=192.168.112.128,port=6633
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
```

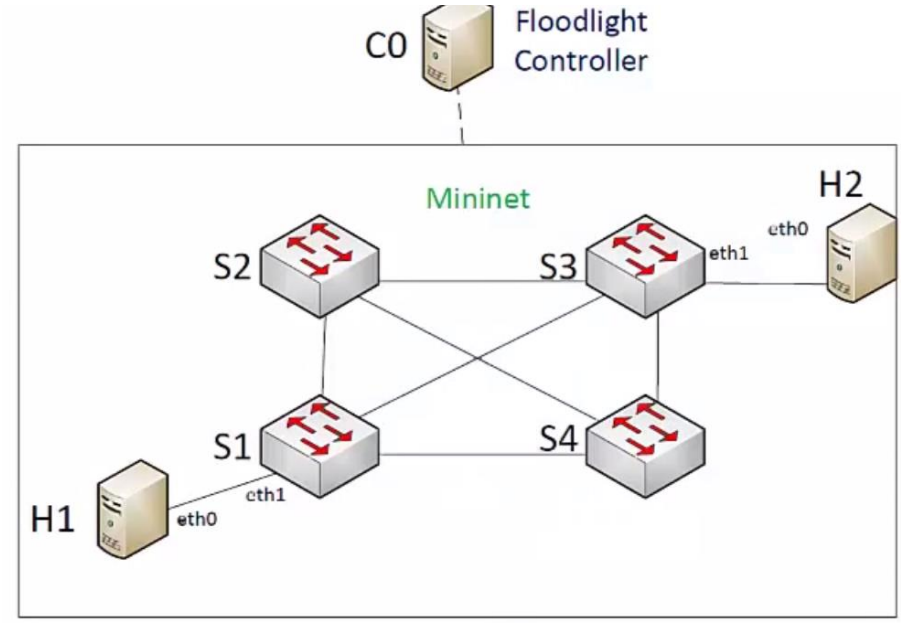
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet>



Host h1, is able to ping host h2, which indicate that the remote controller is working, when we kill the POX controller session, (^c) we still see ping packets delivered, this is because the flow is cached on an OpenFlow vSwitch .

Custom Network Topology:

We now create a more complex network as shown below:



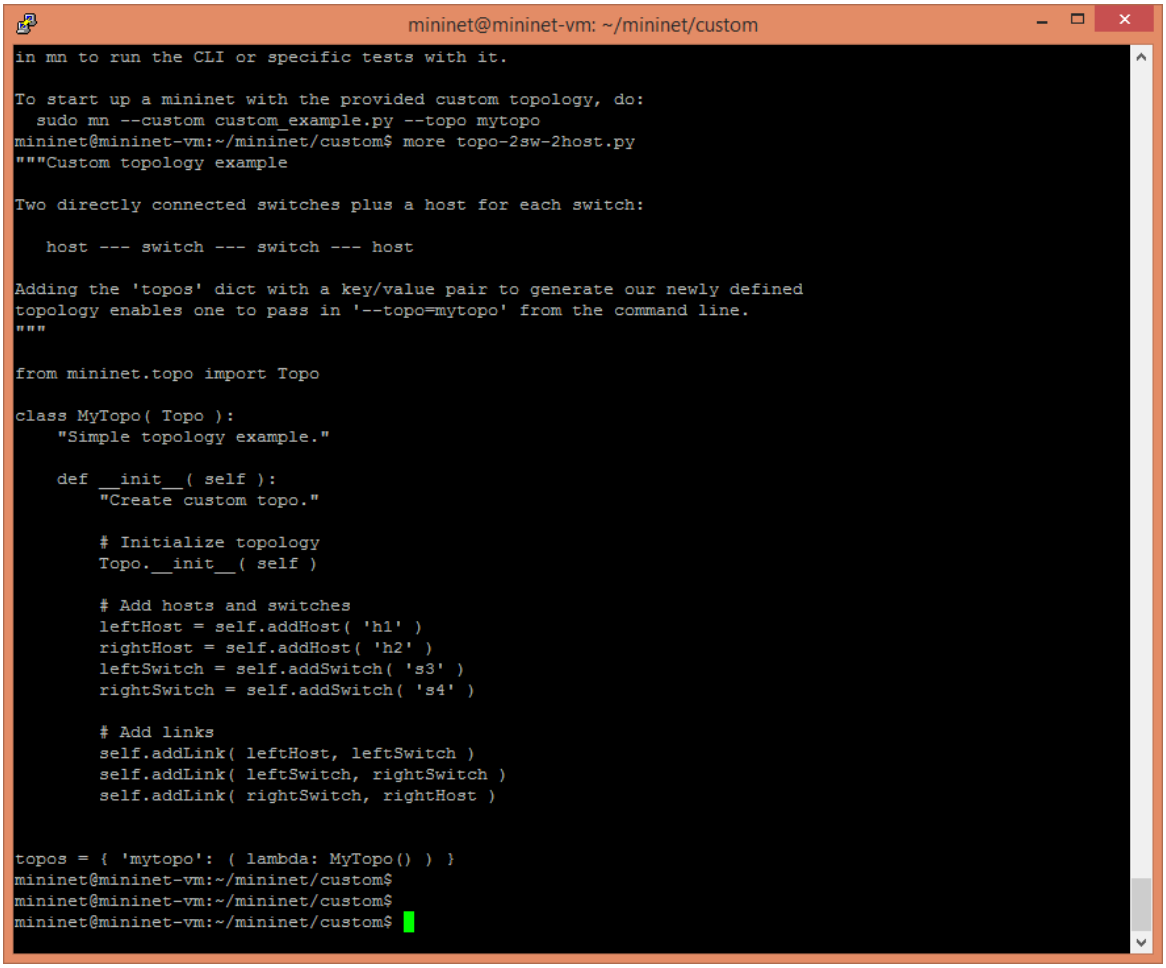
Here, we have four switches connected in a full mesh , and one controller, which can be Floodlight, or POX.

Note: just because we have an SDN enabled network, doesn't mean that suddenly we have no concerns about loops. This why it's very important to implement this network with the controller, which will detect and prevent packets to travel in a loop.

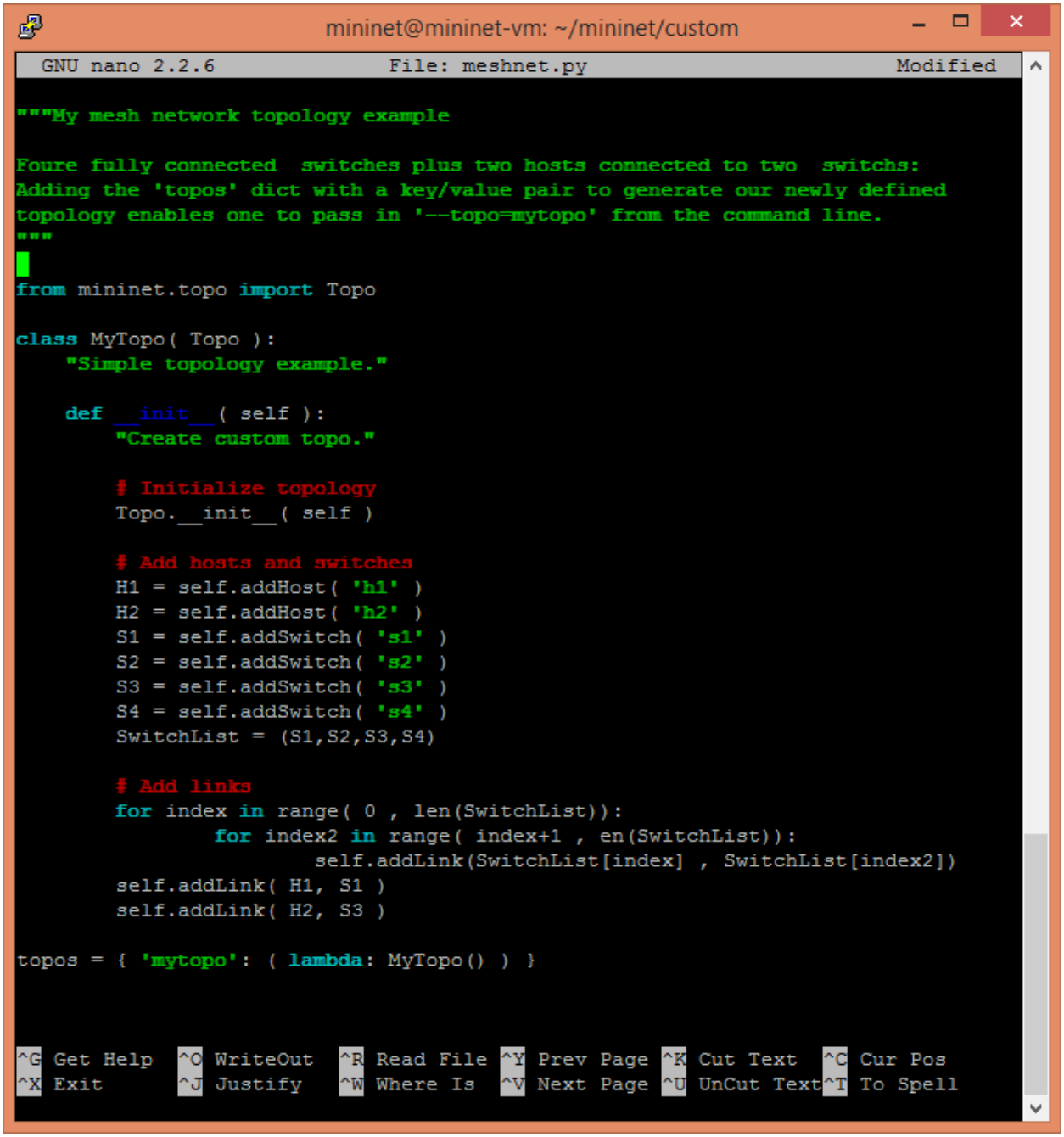
To create the above network, we will be editing a python file located at root/mininet/custom/

```
mininet@mininet-vm:~$ cd mininet/custom
mininet@mininet-vm:~/mininet/custom$ more readme
readme: No such file or directory
mininet@mininet-vm:~/mininet/custom$ ls
mesh.py.save  README  topo-2sw-2host.py
mininet@mininet-vm:~/mininet/custom$ more README
This directory should hold configuration files for custom mininets.

See custom_example.py, which loads the default minimal topology.  The advantage o
f 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
mininet@mininet-vm:~/mininet/custom$
```

There is an example file already in that directory called topo-2sw-2host.py:



I copied the content of the example and modified to create my mesh topology as shown below:



Now to run this topology, first we get our SDN controller running on a remote location (another vm) and then we run the topology by →

```
$ sudo mn --custom meshnet.py --topo mytopo controller=remote,ip=192.168.112.128
```

```
mininet@mininet-vm:~$ cd mininet/custom
mininet@mininet-vm:~/mininet/custom$ nano meshnet.py
mininet@mininet-vm:~/mininet/custom$ sudo mn --custom meshnet.py --topo mytopo -
-controller=remote
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(h1, s1) (h2, s3) (s1, s2) (s1, s3) (s1, s4) (s2, s3) (s2, s4) (s3, s4)
*** Configuring hosts
h1 h2
*** Starting controller
*** Starting 4 switches
s1 s2 s3 s4
*** Starting CLI:
mininet>
```

mininet> dump

```
*** Starting CLI:
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=2672>
<Host h2: h2-eth0:10.0.0.2 pid=2673>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None,s1-eth4:None p
id=2676>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None pid=2681>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None,s3-eth3:None,s3-eth4:None p
id=2686>
<OVSSwitch s4: lo:127.0.0.1,s4-eth1:None,s4-eth2:None,s4-eth3:None pid=2691>
<RemoteController c0: 127.0.0.1:6633 pid=2665>
mininet>
```

mininet> pingall

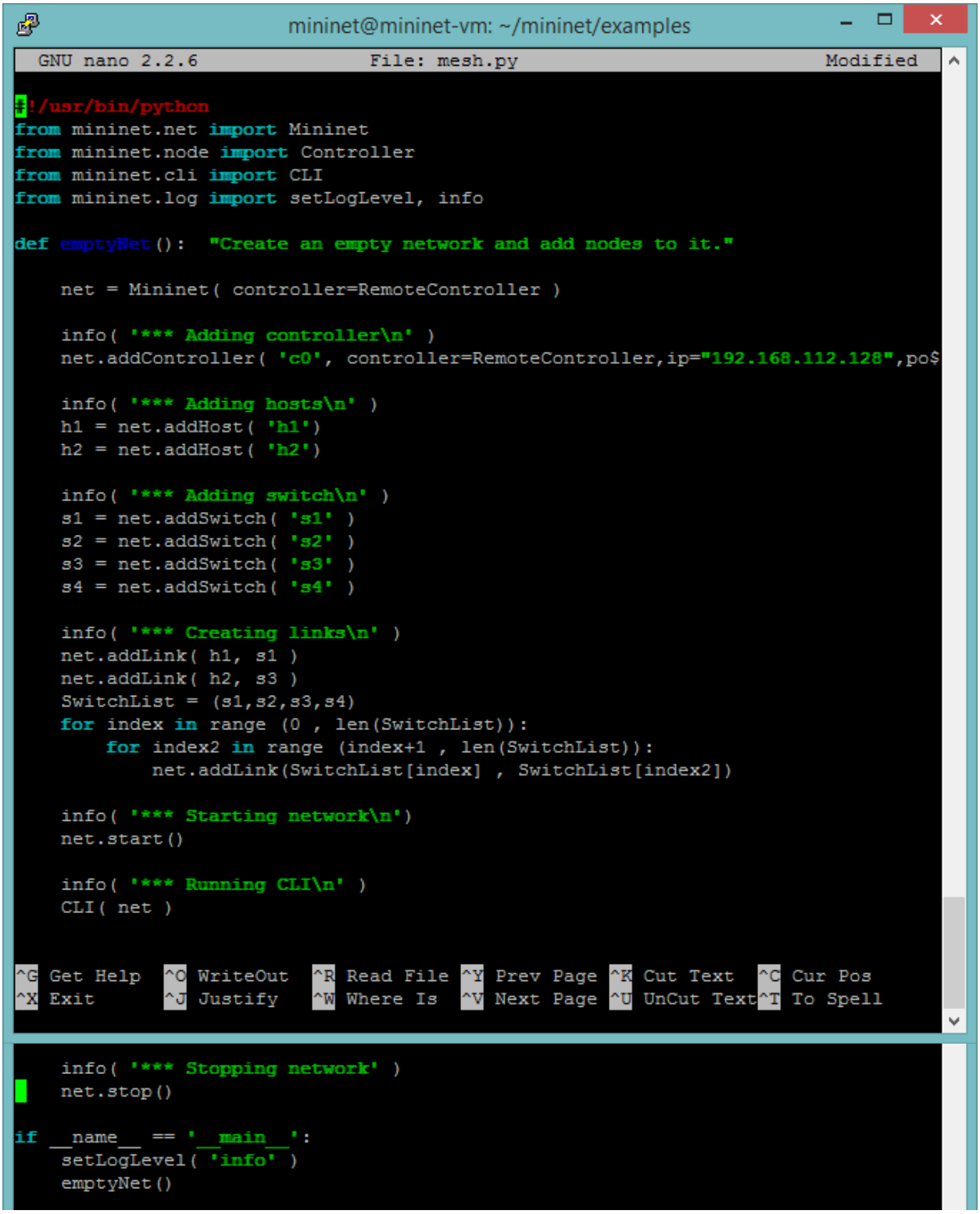
```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet>
```

mininet> net

```
mininet> net
h1 h1-eth0:s1-eth4
h2 h2-eth0:s3-eth4
s1 lo:  s1-eth1:s2-eth1 s1-eth2:s3-eth1 s1-eth3:s4-eth1 s1-eth4:h1-eth0
s2 lo:  s2-eth1:s1-eth1 s2-eth2:s3-eth2 s2-eth3:s4-eth2
s3 lo:  s3-eth1:s1-eth2 s3-eth2:s2-eth2 s3-eth3:s4-eth3 s3-eth4:h2-eth0
s4 lo:  s4-eth1:s1-eth3 s4-eth2:s2-eth3 s4-eth3:s3-eth3
c0
```

The network is connected, as planned, all nodes were created..

Here is another program file that can give more capabilities in setting the network elements:



Note in the program above, that we used the python class RemoteController instead of the python class Controller.

In general, here is how a Mininet Workflow can look like:

Creating a Network

You can create a network with a single command. For example,

```
sudo mn --switch ovsk --controller ref --topo tree,depth=2,fanout=8 --test pingall
```

starts a network with a tree topology of depth 2 and fanout 8 (i.e. 64 hosts connected to 9 switches), using Open vSwitch switches under the control of the OpenFlow/Stanford reference controller, and runs the **pingall** test to check connectivity between every pair of nodes. (This takes about 30 seconds on my laptop.)

Interacting with a Network

Mininet's CLI allows you to control, and manage your entire virtual network from a single console. For example, the CLI command

```
mininet> h2 ping h3
```

tells host **h2** to ping host **h2**'s IP address. *Any available Linux command or program can be run on any virtual host*. You can easily start a web server on one host and make an HTTP request from another:

```
mininet> h2 python -m SimpleHTTPServer 80 >& /tmp/http.log &
mininet> h3 wget -O - h2
```

Customizing a Network

Mininet's API allows you to create custom networks with a few lines of Python. For example, the following script

```
from mininet.net import Mininet
from mininet.topolib import TreeTopo
tree4 = TreeTopo(depth=2,fanout=2)
net = Mininet(topo=tree4)
net.start()
h1, h4 = net.hosts[0], net.hosts[3]
print h1.cmd('ping -c1 %s' % h4.IP())
net.stop()
```

creates a small network (4 hosts, 3 switches), and pings one host from another. The Mininet distribution includes several text-based and graphical applications which we hope will be instructive and inspire you to create cool and useful apps for your own network designs.

Sharing a Network

Mininet is distributed as a virtual machine (VM) image with all dependencies pre-installed, runnable on common virtual machine monitors such as VMware, Xen and VirtualBox. This provides a convenient container for distribution; once a prototype has been developed, the VM image may be distributed to others to run, examine and modify. A complete, compressed Mininet VM is about 1GB. (Mininet can also be installed natively - **apt-get install mininet** on Ubuntu.) It is Open Source after all.

Running on Hardware

Once a design works on Mininet, it can be deployed on hardware for real-world use, testing and measurement. To successfully port to hardware on the first try, every Mininet-emulated component must act in the same way as its corresponding physical one. The virtual topology should match the physical one; virtual Ethernet pairs must be replaced by link-level Ethernet connectivity. Hosts emulated as processes should be replaced by hosts with their own OS image. In addition, each emulated OpenFlow switch should be replaced by a physical one configured to point to the controller. However, the controller does not need to change. When Mininet is running, the controller "sees" a physical network of switches, made possible by an interface with well-defined state semantics.

There is much more to be explored and tested in the mininet virtualization world, the following are links to some mininet projects that can be tested in the future:

- MPLS-TE Demo http://archive.openflow.org/wk/index.php/MPLS-TE_Demo#Mininet_and_OVS
- VXLAN overlay networks with Open vSwitch

References:

- * SDN: Software Defined Networks by Thomas D. Nadeau and Ken Gray
- [1] http://searchnetworking.techtarget.com/definition/virtual-private-LAN-service
- [2] http://www.ixiacom.com/pdfs/library/white_papers/mpls.pdf
- [3] http://mplslearner.wordpress.com/2013/05/03/understanding-mpls/

- [4] http://en.wikipedia.org/wiki/Multiprotocol_Label_Switching
- [5]http://searchtelecom.techtarget.com/definition/traffic-engineering
- [6] http://homes.cs.washington.edu/~arvind/cs425/doc/traffic-mpls.pdf
- [7]http://www.ciscopress.com/articles/article.asp?p=426640&seqNum=2
- [8] http://archive.openflow.org/wk/index.php/OpenFlow_Tutorial
- [9] ONF SDN White Paper https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf
- [10] http://archive.openflow.org/wk/index.php/MPLS_with_OpenFlow/SDN
- [11] http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_16-1/161_sdn.html
- [12] http://sit.sit.fraunhofer.de/mne/publications/download/SDNControllers.pdf
- [13] Feature-based Comparison and Selection of Software Defined Networking (SDN) Controllers, by Rahamatullah Khondoker, Adel Zaalouk, Ronald Marx, Kpatcha Bayarou
- [14] http://mininet.org/
- [15]www.lightreading.com/carrier-sdn/sdn-architectures/open-sdn-driving-the-operator-market/a/d-id/708018