SDN via APIs: Review of Open SDN / OpenFlow

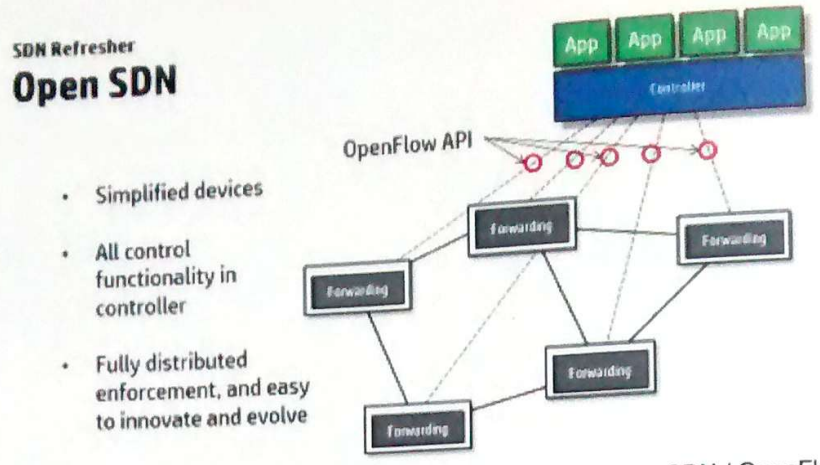


Figure 1-44: SDN via APIs: Review of Open SDN / OpenFlow

Again, recall that Open SDN looked as portrayed in the picture above, with control functionality residing on the controller; with simpler devices; and with OpenFlow being the protocol for communicating with devices below.

NOTES

SDN via APIs: Proprietary Protocol

What you get:

- **Some network programmability**
- **"Proprietary openness"**
- **Little or no device simplification**

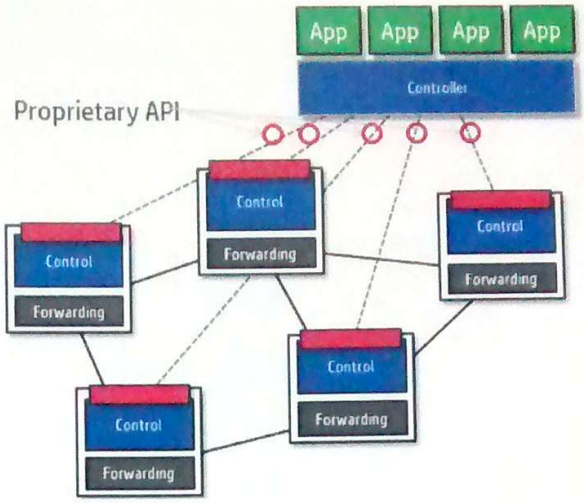


Figure 1-45: SDN via APIs: Proprietary Protocol

Here is a representation of the design of SDN via APIs. In the picture it is clear that:

- Devices are not necessarily simplified, but retain much of the control functionality.
- There is a layer of superior APIs on the devices for communication with the controller.
- The communication between the devices and the controller uses these newer, superior, but proprietary protocols.
- There is indeed a controller, presenting an API above for applications to contribute control functionality

So this is an improvement over today's situation of autonomous, independent devices, because they have APIs which facilitate automatic and programmatic changes to network configuration, as is required in the data center.

NOTES

SDN via APIs: Today's Devices

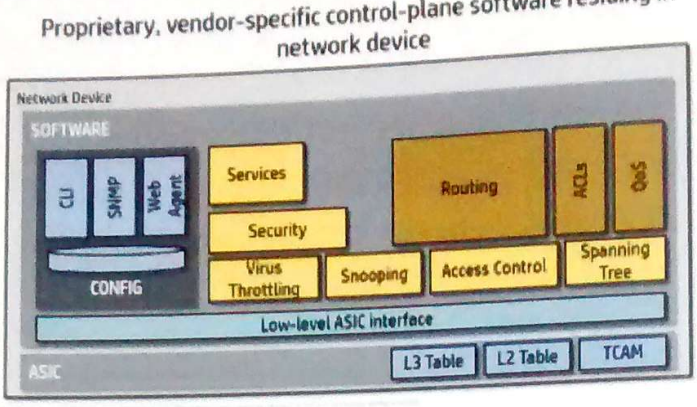


Figure 1-46: SDN via APIs: Today's Devices

Here again is today's networking device, with all of that control software running inside.

NOTES

SDN via APIs: A Better API

SDN VIA APIs: to non-OpenFlow devices

SDN via APIs: Provide an API on top of the switch's existing configuration mechanisms

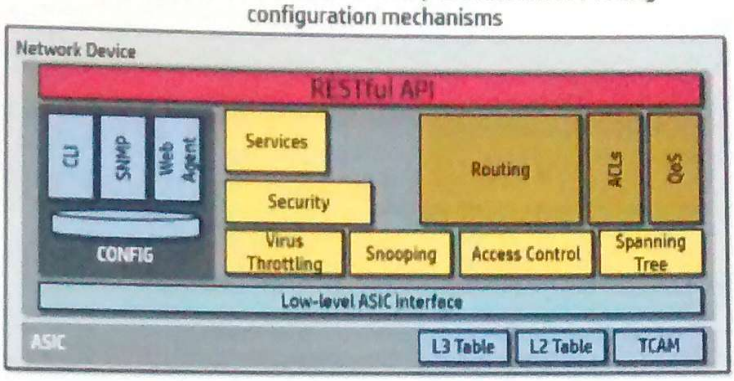


Figure 1-47: SDN via APIs: A Better API

Here we see the same device, but with a better API for affecting the configuration and behavior of the device.

Note that it would be possible, assuming the APIs were designed appropriately, to remove much of that control functionality from the device. In fact, you could have an API which was the equivalent of OpenFlow, although proprietary; having this type of API would allow the device to shed the control functionality and behave just like an Open SDN device.

However today, most device implementations of SDN via APIs retain most control functionality on the device, and so this device simplification goal is not met.

NOTES

SDN via APIs: Summary

SDN Definition #2: **APIs**

SDN via APIs: Response from many entrenched networking vendors.

- Leave most control plane on device
- Same devices, centralized controller
- Application development on vendor's controller
- Proprietary protocol

Proponents: Cisco, Arista, Brocade, Alcatel-Lucent,...

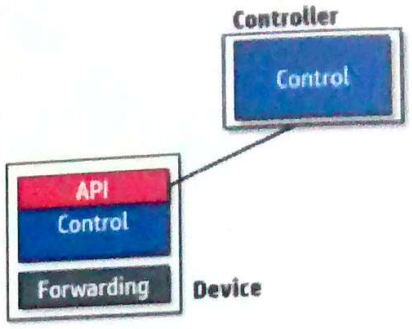


Figure 1-48: SDN via APIs: Summary

This summary describes what one gets with SDN via APIs:

- There is a controller
- It is possible to write applications on the controller, although they will be proprietary for that vendor
- The communication protocol between controller and device is proprietary, which is another way of ensuring vendor lock-in

Some of the networking vendors who have their own SDN via APIs strategy are: Cisco, Arista, Brocade, Alcatel-Lucent, and others.

NOTES

SDN via APIs: Summary (continued)

Programmability

- Can overcome agility issues
- Can allow centralized control

- Doesn't address device cost & simplicity
- Doesn't address openness for innovation
- Doesn't solve datacenter issues (VLANs etc)

Figure 1-49: SDN via APIs: Summary - programmability

In review, SDN via APIs can overcome some agility issues; and it does give centralized control, which is beneficial in and of itself, in addition to providing a platform for developing APIs.

However there are some limitations to the value of SDN via APIs:

- In most cases device cost is not reduced, simplicity is not achieved
- It isn't really open, in the sense that it provides APIs but these are proprietary and specific to the vendor's device
- Data center issues (MAC address table, VLAN, etc.) are not addressed

NOTES

SDN via Overlays: Virtualized Networks

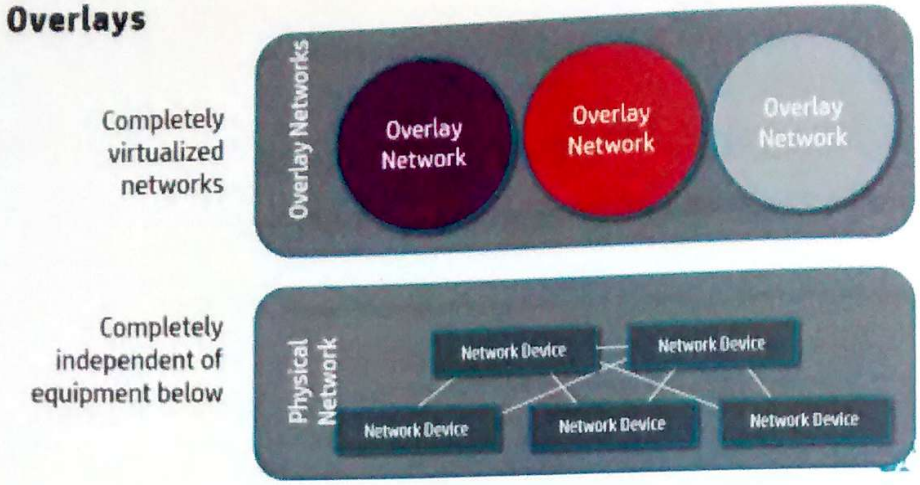


Figure 1-50: SDN via Overlays: Virtualized Networks

SDN via Overlays takes an entirely different approach; in this solution, the physical network itself is ignored entirely, with networking control taking place at a 'virtual network' level.

What this means is that the data center issues we discussed earlier are addressed at a virtual level, without touching the physical network. The physical network can be anything - routed layer 3, large layer 2, whatever you like. The solution will use some existing networking technology - namely tunnels - to create virtual circuits across the top of the existing physical network.

As a result, the network below in the figure is not touched at all - the solution is not concerned with those devices. All it cares about are the virtual networks, shown in the picture as existing completely above the physical network.

NOTES

SDN via Overlays: Doesn't Touch Devices

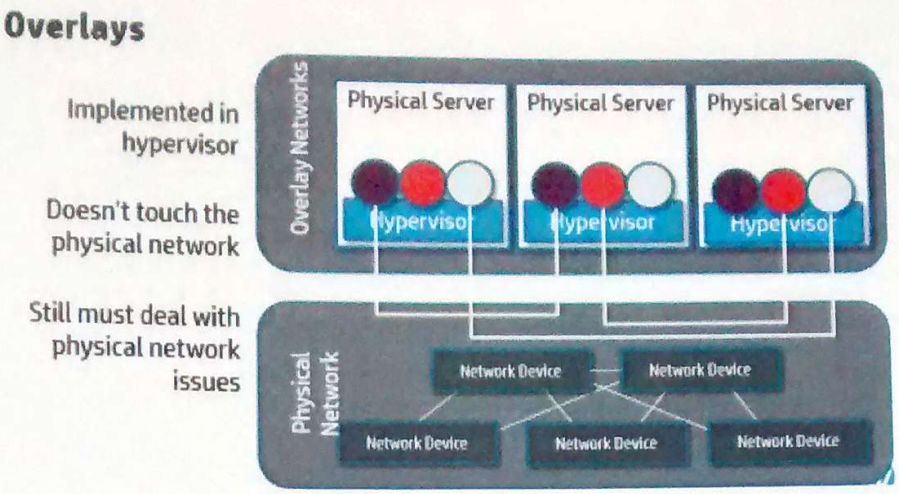


Figure 1-51: SDN via Overlays: Doesn't Touch Devices

The ramifications then are that the SDN via Overlay solution makes use of the hypervisors that exist on each physical server. This hypervisor has an internal virtual switch which routes traffic between nodes.

The controller (not shown in the picture) dynamically configures each hypervisor to forward packets between virtual machine using tunnels. Tunnels (borrowed from network virtualization standards described earlier) are used to meet demands for multi-tenancy, and to reduce the number of MAC addresses in the network.

Thus, as shown in the picture, the purple nodes can talk to each other; the orange and grey nodes as well. Traffic is segregated using tunnels, and MAC addresses are reduced by an order of magnitude (maybe more) because of the fact that host MAC addresses are never seen on the physical network - they are encapsulated into the interior of the tunneled packet.

NOTES

SDN via Overlays: MAC-in-IP Tunnels

Overlay Tunneling Alternatives

- **VXLAN** (Cisco), **NVGRE** (Microsoft), **STT** (Nicira)
- **Uses MAC-in-IP tunneling**

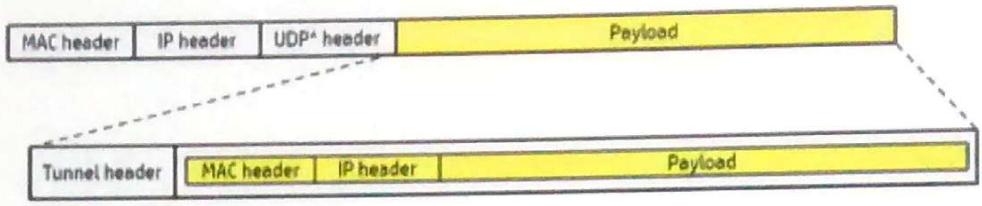


Figure 1-52: SDN via Overlays: MAC-in-IP Tunnels

Tunneling alternatives for implementing SDN via Overlays are typically:

- VXLAN, which was originally pioneered by Cisco and VMware, although recently with their Nicira acquisition, VMware also promotes STT.
- NVGRE, which was originally pioneered by Microsoft and others.
- STT, originally by Nicira, which is now part of VMware.

All of these alternatives feature MAC-in-IP tunneling, which works as shown in the diagram:

- The original packet, including MAC addresses, is encapsulated into a tunneling packet, including the tunnel header.
- The tunneled packet is placed into an IP packet (UDP in the example, obviously this means VXLAN, but it would be GRE or TCP for the other tunnel types.
- The packet is then unicast from one tunnel endpoint to the other, as described in the following pages.

NOTES

SDN via Overlays: Tunneling Operation

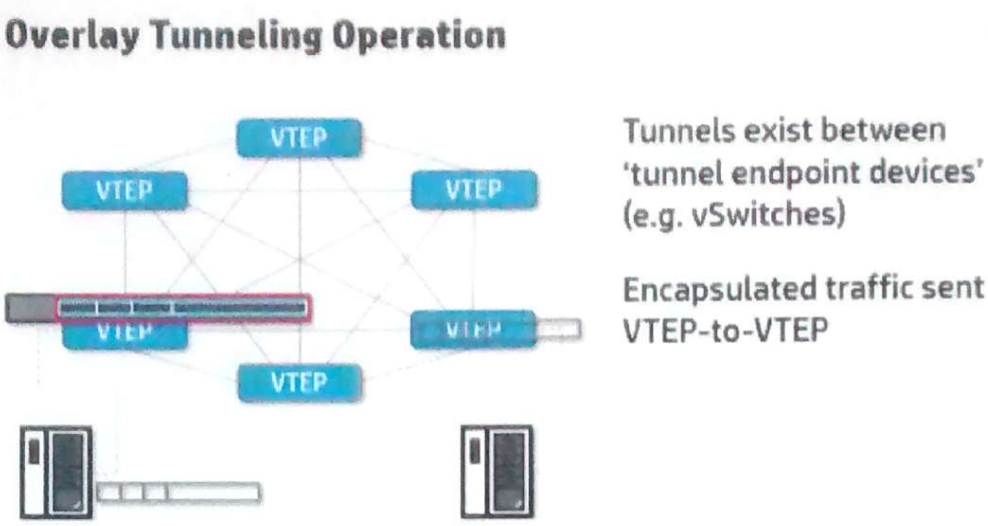


Figure 1-53: SDN via Overlays: Tunneling Operation

This diagram shows the packet originating at the host on the left, getting encapsulated at the VTEP which attaches the host to the network (probably in a hypervisor). The packet is then passed as a unicast, across the network to the destination switch VTEP, where it is decapsulated, and passed to the destination host.

The picture shows a virtual representation of the VTEPs all connected to each other, since they can communicate via unicast packets, irrespective of the underlying network. The virtual networks connecting VTEPs are call the overlay, and the physical network underneath is referred to as the underlay.

NOTES

SDN via Overlays: Summary

SDN Definition #3: **Overlays**

SDN via Overlays: Response mainly from server-based vendors and OpenStack proponents

- Leave devices alone
- Use virtual switches in hypervisors
- Virtual networks on top of physical network
- Open or proprietary protocol

Proponents: VMware, Plumgrid, ...

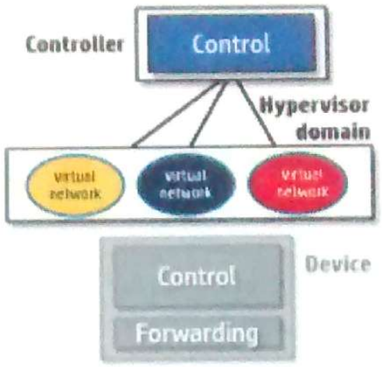


Figure 1-54: SDN via Overlays: Summary

In summary, SDN via Overlays does the following:

- Does not touch the physical network below at all - that underlay network can be left alone, and can be built out of whatever the customer chooses.
- Typically (although not required), the virtual tunnel endpoints will be implemented in virtual switches within hypervisors running on servers.
- There will be a controller, which manages the tunnels, and the routing tables required to direct traffic into the appropriate tunnels

Some vendors who promote SDN via Overlays most enthusiastically are VMware and PlumGrid. Microsoft has implemented SDN via Overlays, and many networking vendors (e.g. Cisco, IBM, NEC, and Juniper) have virtual network solutions of this nature.

NOTES

SDN via Overlays: Summary (continued)

Tunneling (encapsulation)

- Can overcome multi-tenancy issues
- Can overcome MAC address table size issues
- Can overcome VLAN deficiencies

- Doesn't address physical network
- Doesn't simplify or enable innovation

Figure 1-55: SDN via Overlays: Summary

To review the benefits of SDN via Overlays, they:

- Can be used to overcome multi-tenancy issues, since they use tunnels to segregate traffic from different customers.
- Can overcome MAC address table issues, since the only MAC addresses which appear on the physical network are the MACs of the VTEPs - this can reduce the number of MAC by an order of magnitude or more.

Some of the shortcomings of the overlay solution:

- It doesn't address the shortcomings of the physical network - if you have a bad physical network to begin with, it will likely remain bad (unless that badness was related to MAC address table overflow, or VLAN exhaustion.
- The whole point of SDN initially was to change the way networking was done - the clean slate idea - simplifying devices, enabling innovation, etc. This overlay solution achieves none of that.

NOTES

A Closer Look at SDN

A Closer Look at SDN

- SDN **Components** (devices, controllers, applications)
- **OpenFlow** protocol basics (including newer versions)

SDN Use Cases

- Applications of SDN in **real life** (datacenter, WAN, campus)
- **HP** applications of SDN in real life

Figure 1-56: A Closer Look at SDN

At this point it is time to take a closer look at SDN: the components that make up an SDN solution, the protocol (OpenFlow) that enables it (in the pure, Open SDN way), and some use cases that have driven the interest in SDN that we see today.

NOTES

Anatomy of an SDN *Hardware* Device

Anatomy of an SDN Device

Hardware

- L2 & L3 forwarding tables
- TCAMs for matching fields other than MAC and IP address

Operation

- Handle matching flows locally
- Drop or forward non-matching flows to controller and await instructions

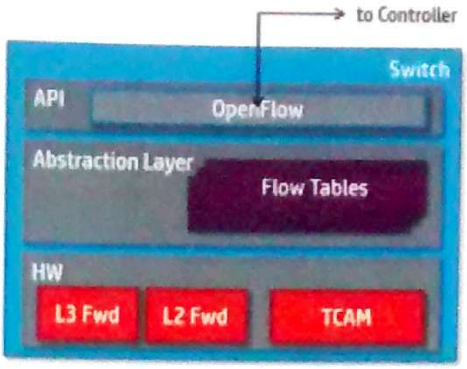


Figure 1-57: Anatomy of an SDN Hardware Device

First we look at SDN devices. In particular we look at both *hardware* and *software* SDN devices.

A hardware SDN device will make use of the existing hardware in the device in order to provide the enforcement of the rules that are programmed by the controller. This is the way that SDN was envisioned in the early days - recall back to the discussion of ForCES, Clean Slate, and Ethane.

The hardware of the device is:

- The ASIC that hosts the various hardware tables that are used to make forwarding and routing decisions.
- A layer 3 forwarding (routing) table that is used for making IP-level forwarding decisions, based on destination IP address.
- A layer 2 forwarding table that is used for making Ethernet-level forwarding decisions, based on the destination MAC (physical) address.
- A TCAM (ternary content addressable memory), which is used for more complicated matches against the incoming packet, matching against many layer 2, 3, and 4 fields, as well as allowing wildcards for fields that are irrelevant for the specific match.

At a level above the hardware is the abstraction represented by the flow table (or flow tables). These tables exist in the software on the device and are translated into entries in the respective hardware tables below.

At the level above those tables is the actual protocol - in this case OpenFlow - which is used to communicate with the controller. This two-way communication is used to exchange status messages, to pass packets to the controller, and to receive outgoing packets and flow modification messages.

Anatomy of an SDN *Software* Device

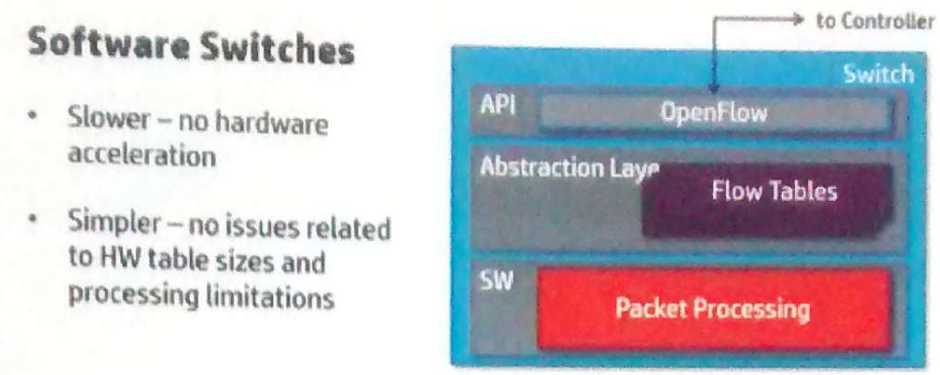


Figure 1-58: Anatomy of an SDN Software Device

The above figure depicts a software SDN device. The upper layers are the same, the only difference is the lower level. Recall that in a hardware device, there was an ASIC with tables for matching incoming packets and taking actions.

In a software switch, this HW has been replaced by logic which does the lookups for matching incoming packets, and takes the appropriate actions.

There are a couple ramifications to the software switch:

- It will be slower, since the packets are handled in software rather than via hardware tables.
- It will be simpler to implement, meaning that there are no translate-to-hardware issues which are prevalent and must be considered with a hardware device.

As an example of the 'simpler' argument, it will typically be true that the latest features of OpenFlow will be implemented first in software switches. The reason for this is because it can be difficult to translate new OpenFlow features into hardware, whereas it is a simple exercise to implement the feature in software using hash maps and such.

Secondly there may be hardware limitations that do not exist in software counterparts. For example, flow table sizes: in hardware, there will be fixed limit to the number of flow entries that can be held in a hardware flow table. And device manufacturers will always be weighing the tradeoffs between putting in more hardware to support larger tables, versus saving costs with smaller tables.

In software, there are of course limitations to the size of memory available, but they are much more variable and can be much larger (and are less expensive) than what is true with hardware.

Consequently software implementations will always be faster, and more feature-complete, then those implemented in hardware. But of course hardware implementations will be faster and more desirable for any environment where throughput is an issue.

SDN Device Hybrid Modes

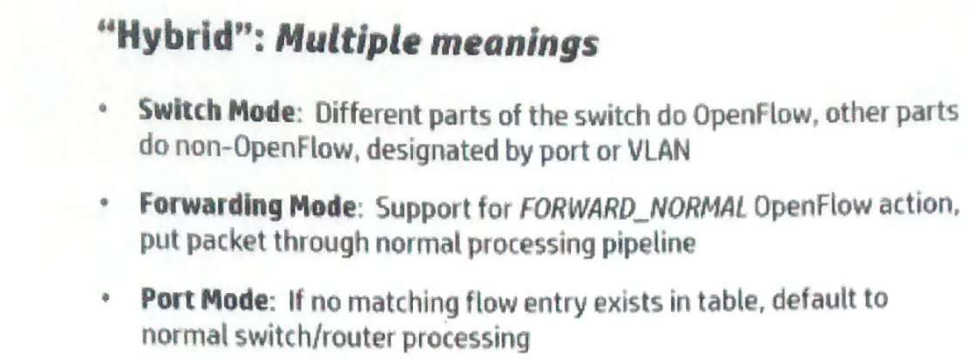


Figure 1-59: SDN Device Hybrid Modes

This discussion of SDN devices would not be complete without considering the topic of 'hybrid' devices. That is, devices which are capable of operating in both OpenFlow, and non-OpenFlow mode, at the same time.

These types of devices are obviously desirable for customers who are interested in gradually moving towards OpenFlow, but want to retain some of their traditional functionality for certain parts of their network.

In general there are three types of 'hybrid' behavior that are promoted these days. The terminology is not fixed at this time, but the general categories are:

- Switch Mode. In this category, the switch itself is segregated into OpenFlow and non-OpenFlow portions. This can be based on criteria such as port, or VLAN. A typical use case would be to have certain subnets using SDN, and others still using traditional networking, as part of a gradual roll-out of the technology.
- Forwarding Mode. In this category, within OpenFlow functionality, switches can support the 'forward normal' optional action, which specifies for a specific rule that the switch should forward the packet using the 'normal' processing pipeline - e.g. as it would if OpenFlow was not involved. This allows an application to tell the switch in certain situations, just forward the packet normally.
- Port Mode. In this category, for specific ports or VLANs, the switch can be instructed that if an incoming packet does not match any of the rules, it should forward the packet 'normally'. Without this rule, the switch will generally have two choices if a packet does not match any rule: it can be dropped, or it can be forwarded to the controller. This hybrid mode feature allows for the third option, to forward the packet normally.

NOTES

SDN Controller APIs

Northbound API:

- Non-standard today (ONF WG just formed)
- C++, Java, Python, REST

Southbound API:

- OpenFlow, proprietary, or both

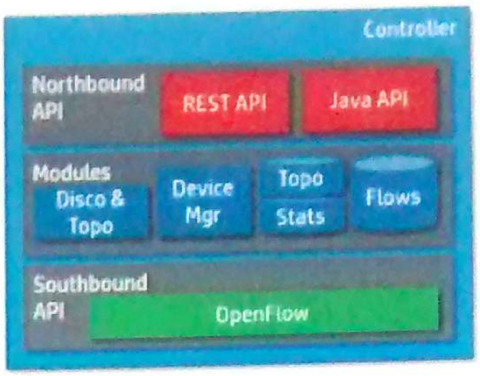


Figure 1-60: SDN Controller APIs

Now we move to a discussion of the controller itself. Controllers can be open source or can be commercial. They contain an upper application-facing 'northbound' API, as well as a lower device-facing 'southbound' API. In between is the basic controller functionality, which we will discuss later.

The Northbound API:

- Provides interfaces for applications
- Isn't standardized today (although the Open Networking Foundation has started a working group).
- Can use various languages such as C++, Java, Python, or can be implemented via REST (with limitations we will discuss later).

The Southbound API:

- For Open SDN is OpenFlow
- For other controllers (e.g. OpenDaylight) it can be other protocols such as CLI, SNMP, or various routing protocols

The next page looks at the Northbound API in more detail.

NOTES

SDN Controller: Northbound API

Northbound API **Events**

- Not available from REST
- Switch & user device events
- Packet events

Northbound API **Functions**

- Add, delete, or modify flows
- Actions to take in response to events received
  - Drop, modify, forward packet
  - Add, delete, modify flows

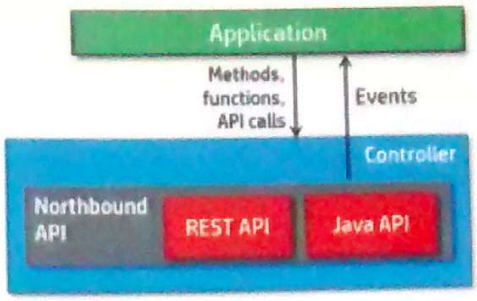


Figure 1-61: SDN Controller: Northbound API

Looking a little closer at the northbound API, we see that there are basically two options:

- Use the native language of the controller, e.g. Java.
- Use the RESTful API provided by the controller, which allows you to use any language you choose. You can even write shell scripts to invoke the REST calls to access controller functionality.

Be aware that there are limitations if you want to use the REST API, you will not receive notifications from the controller, since REST is a one-way mechanism, from the requester (your application) to the controller. The ramifications of this are that **you will not be able to receive packets from the switches and act on them**. This of course is very serious for certain types of application, which depend on being able to listen for unmatched packets, and respond with actions and flow modifications.

Events that can be received:

- Switch events (new switch discovered, port status changes)
- End node events (node discovered, IP address learned/changed, connection point changed)
- Packet events (packet received from switch)

There are of course a number of functions or methods that may be called on the REST or internal (Java) APIs. These include:

- Flow modifications
- Actions (based on receiving packets - clearly this is not available for REST-based applications)

We spend a lot of time in later sections discussing these APIs.

NOTES

SDN Controller Applications

Standard applications

- GUI
- Learning Switch
- Routing

Additional applications

- Load balancer
- Firewall

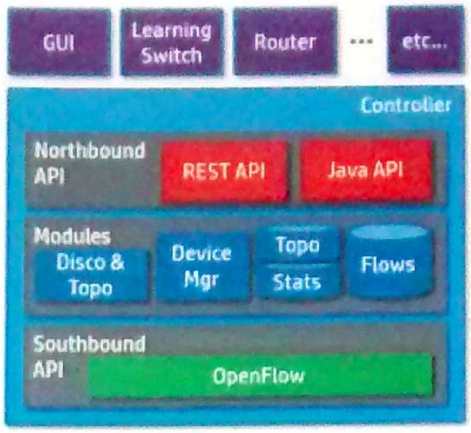


Figure 1-62: SDN Controller Applications

Any controller available today comes with a fairly significant set of applications, which perform a number of helpful functions. Some of the standard functions available are:

- A GUI for looking at the network, topology, end nodes, etc.
- Learning Switch allowing your network of OpenFlow devices to work automatically by dynamically setting flows based on observed traffic.
- Routing - layer 3 forwarding

Some controllers provide functionality for load balancers and firewalls as well.

One of the ramifications of this is that controllers will often perform topology discovery, using techniques such as LLDP, to learn how each switch is connected to the others.

NOTES

SDN Controller Considerations

- No standard Northbound API
- Coordination between applications
- Scalability, High-availability, Performance

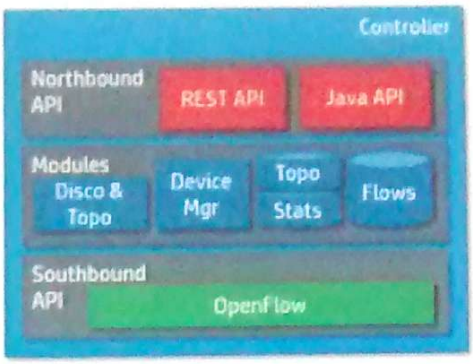


Figure 1-63: SDN Controller Considerations

There are some considerations which must be taken into account when contemplating creating SDN applications. Some of this are discussed below:

- No standard northbound API. This is an issue if you wish to create an application that runs on multiple controllers. There is a working group within ONF that is looking into creating a standard interface, but the standard - as well as implementations of the standard - are a ways off at this moment.
- Coordination between applications is very important to consider. What happens if multiple applications attempt to write conflicting rules on the switch? What happens if multiple applications attempt to receive the same packets that have been forwarded to the controller? What happens if one application handles the response to the switch, and doesn't allow other applications to perform actions as well? What happens if the application takes actions, but subsequent application negate or overwrite those actions? All of this things must be considered when creating an application.
- If you are building a reactive application - one which depends on switches forwarding unmatched packets to the controller (or matched packets with an action of "forward to controller") - if you are building this type of application, then obviously you will need to be aware of scalability and performance issues. What happens if too many packets are getting forwarded, such that the links or the controller gets overloaded? Will the latency be too great in replying to the switch? And what if the controller goes down - if the application relies on having packets forwarded to the controller, this will disrupt the behavior of the application.

All of these are legitimate considerations to bear in mind when building an SDN application.

NOTES

OpenFlow Overview

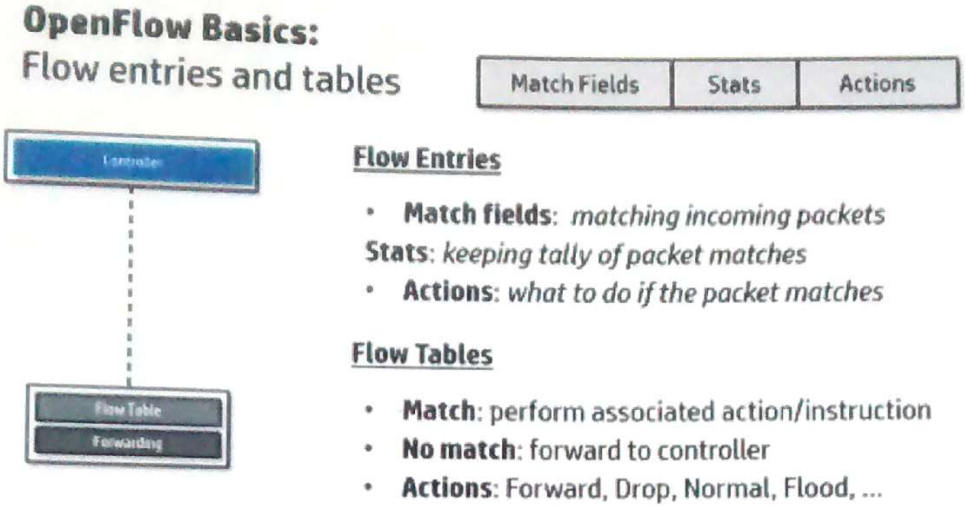


Figure 1-64: OpenFlow Overview

The OpenFlow standard itself is maintained and extended by the "Open Networking Foundation", or ONF. The organization is made up of customers (rather than networking vendors), and consists of a number of working groups such as Forwarding Abstractions, Extensibility, Optical Transport, Northbound Interface, and Wireless & Mobility.

OpenFlow, as mentioned, is the communication protocol that is used to carry messages between the switch and its controller. Communication can go in either direction - initiated by the switch or by the controller.

Some of the basics of OpenFlow are:

- Communication can be in the open or on a secure channel
- On the switch:
  - Information on the device is stored in the form of Flow Entries, which together constitute a Flow Table.
  - Flow entries are made up of three major parts:
    - Match Fields, which are used to match against incoming packets
    - Stats, which keep track of counters for each flow
    - Actions, which are performed if the incoming packet matches the specific flow entry.
  - Flow Tables are made up of Flow Entries; when packets arrive, they are matched against the Flow Entries.
    - If there is a match, the appropriate action is taken (and counters incremented)
    - If there is no match, the packet will be forwarded to the controller (or dropped - drop is the default for later versions of OF, check your switch behavior)
  - Actions tell the switch what to do if there is a match; some examples will be to output the packet to a specific port, to drop the packet, to forward the packet using 'normal' (non-OpenFlow) processing, flood the packet, etc.

NOTES

OpenFlow Match Fields

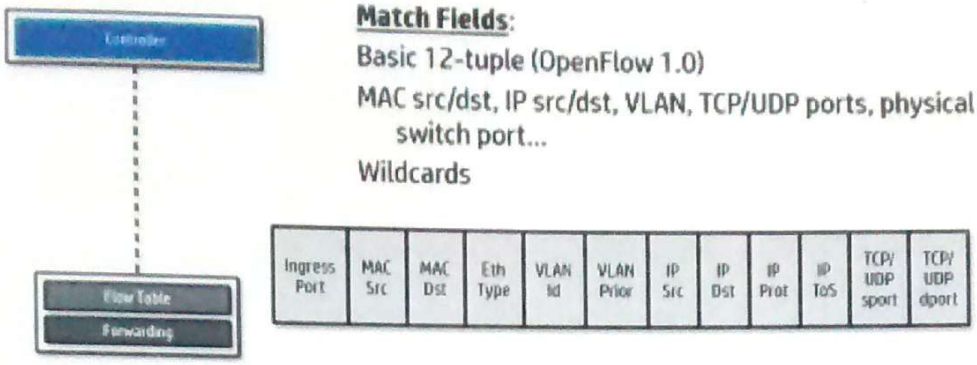


Figure 1-65: OpenFlow Match Fields

Looking more closely at the match fields (for OF 1.0 in this example), it consists of a basic 12-tuple:

- The first field is the ingress port - the port on which the packet arrived
- The following eleven fields all come from the Ethernet header - MAC source and destination, IP source and destination, TCP/UDP source and destination ports, etc.
- Note that in the match, wildcards are allowed, meaning that some or even all of the fields can be marked as "don't care".

Those familiar with TCAM operation recognize that this maps directly to how TCAMs operate, matching incoming packets against a match field defining what parts to the packet to examine to determine if it matches specific entries in the TCAM. TCAMs are generally used in traditional switches for policy-based routing (PBR) and access control lists (ACLs).

The eleven fields from the Ethernet packet are:

- MAC source address
- MAC destination address
- Ethertype
- VLAN identifier
- VLAN priority
- IP source address
- IP destination address
- IP protocol (e.g. TCP, UDP)
- IP type of service (TOS)
- TCP or UDP source port
- TCP or UDP destination port

Subsequent versions of OpenFlow add extra functionality and fields to this list.

OpenFlow Flow Entries

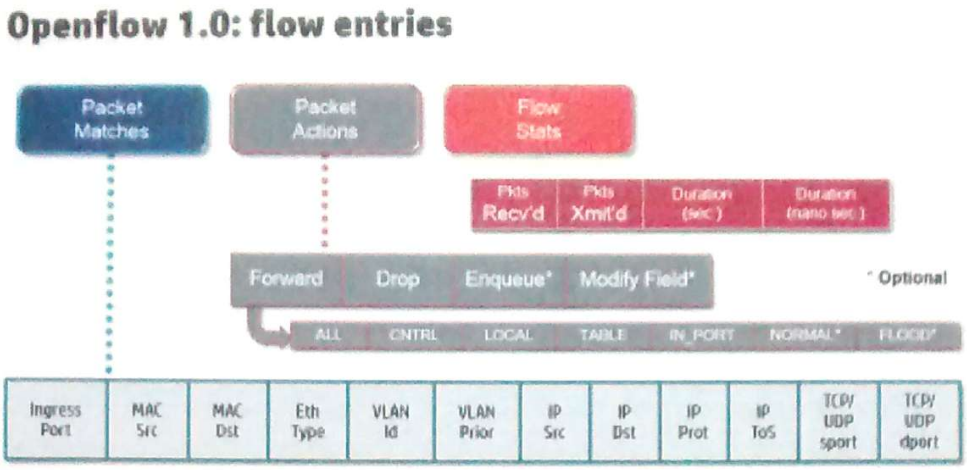


Figure 1-66: OpenFlow Flow Entries

The diagram above shows another view of the match fields, along with potential actions, and the statistics that can be gathered:

- The match fields are precisely what was described in the previous pages.
- The actions can include forward (to various destination keywords), drop, enqueue (for internal prioritization), and modify field
  - Modify field instructs the switch to actually change the packet before forwarding the packet.
  - Mainly the header fields can be modified, e.g. addresses, priority, ports, etc.
- The diagram also shows the statistics that can be gathered, mainly packets received and transmitted, and the length of time the flow has been in existence.

NOTES

OpenFlow Flow Tables

- Prioritized list of *Flow Entries*
- Evaluated in order, execute first match found

Priority	Match Fields	Actions	Stats	Timers
Priority	Match Fields	Actions	Stats	Timers
Priority	Match Fields	Actions	Stats	Timers
	• • •			
Priority	Match Fields	Actions	Stats	Timers

- Each flow has a timeout ('idle' and 'hard')

Figure 1-67: OpenFlow Flow Tables

Looking a bit deeper at the Flow Entries, we can see that there are some other fields that are important for each entry.

Remember that the Flow Table consists of a set of Flow Entries, and that these entries are evaluated in order in order to determine the first match against the received packet. In order to have the entries evaluated in order, each entry must have a priority associated with it. Consequently the diagram shows the priority attached to each flow.

In addition to this, each entry can have times associated with them. There are two types of timers:

- A Hard Timer continues to count down and once it has gotten to zero, the flow is removed, regardless of whether the flow has had matching incoming packets or not.
- An Idle Timer counts down as long as no packet has arrived which matches this particular flow. Once a packet arrives that matches this flow, the timer will be reset and begins to count down again.

In this way, flows may come and go, depending on these timer values which have been set by the application.

NOTES

OpenFlow Flow Entry Types

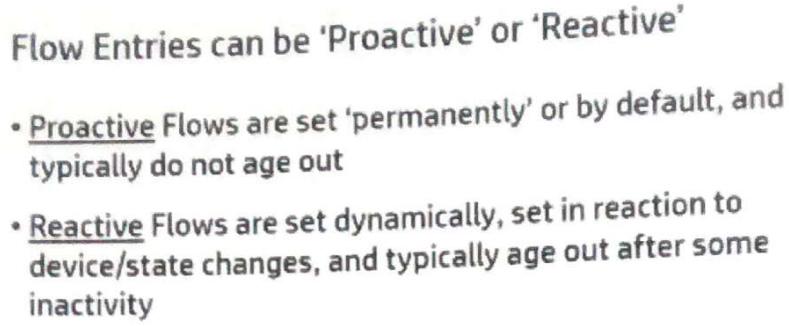


Figure 1-68: OpenFlow Flow Entry Types

Now that we know a little bit about Flow Tables and the Flow Entries that go into these tables, it is time to learn a little about different types of flows.

It is important to note that these distinctions listed above - "proactive" and "reactive" - are general in nature, and refer to the manner in which flows are used. There are not specifically-defined "proactive" and "reactive" flow types which are used within the OpenFlow protocol - these characterizations are used for understanding both flows and the applications that set them.

There are generally two type of flows which will be set on a switch:

- Proactive Flows. These flow entries are typically set initially, when the switch is discovered, and they tend to be either permanent (no timeouts), or else last for a long time. They are set in anticipation of traffic arriving that matches the flow entry.
- Reactive Flows. These flow entries are typically set in response to something that has been observed in the network, most likely as a result of a packet getting forwarded from the switch to the controller. In such a case, the application will want to tell the switch to take action on the received packet, and then most importantly, it will set one or more flow entries on that switch so that the next time a packet of this nature arrives at the switch, the switch can handle it locally, without forwarding the packet to the controller.

NOTES

OpenFlow Ports

Physical ports
(correspond to actual ports on the switch)

Logical ports
(higher-level abstractions, e.g. LAGs, Tunnels, ...)

Reserved ports
(ALL, CONTROLLER, LOCAL, NORMAL, FLOOD, ...)

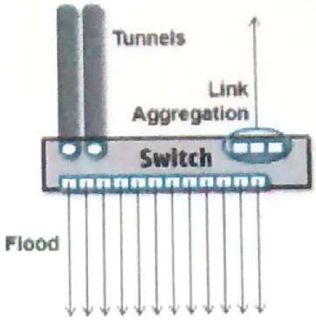


Figure 1-69: OpenFlow Ports

One of the most important actions that can be set for a flow entry is the Output action, which instructs the switch to forward the packet out the appropriate port.

With successive versions of OpenFlow, the definition of a port has been expanded. OpenFlow 1.1 expanded the definition from just referring to a physical port, to include the definition of logical (virtual) ports as well.

This means that an application can now instruct the switch to forward a packet out a specific logical port such as a tunnel or a link aggregation group.

Of course there have always been a certain number of pre-defined special 'ports', which are really reserved values, representing forwarding to all ports, to the controller, etc.

NOTES

OpenFlow Flow Entry Examples (1)

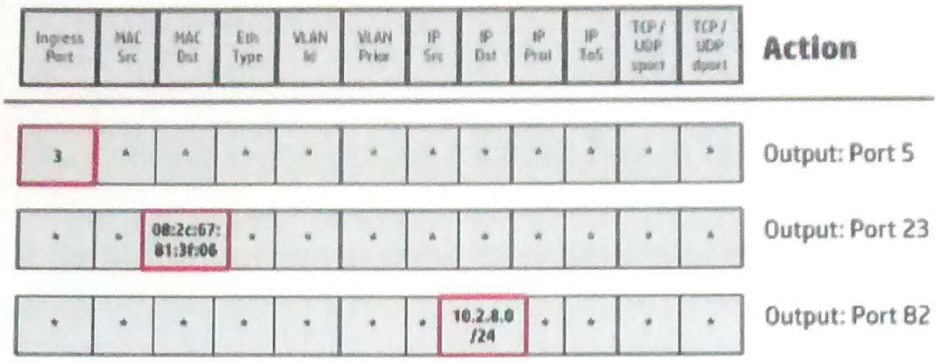


Figure 1-70: OpenFlow Flow Entry Examples (1)

Here are some examples that can be examined to determine the use of these specific flows. Take a moment to look at the flow entries one at a time, and try to figure out for what purpose each might be used.

Here are the answers:

- The first example looks only at the input port (in this case port 3), and instructs the switch to forward each packet coming in on that port, to get forwarded out the specified port (port 5). This flow clearly does not examine any part of the Ethernet packet at all - all it cares about is the input port, and based on that port, it will take an action to output it to a specified port.

  This type of flow could be used for a layer 1 device such as a patch panel. It could be used with an application that turns a device into a mechanism for connecting two wires, such as you would find in a test lab.

  Note also that optical devices are basically layer 1 as well, only dealing with connecting ports, one to another. There is an application we look at later - data center offload - which utilizes an optical switch in this way.

- The second example moves up the networking chain a little bit - up to layer 2, as can be seen from the fact that it attempts to match based on the incoming packet's destination MAC address. Based on that field's value, it instructs the switch to forward the packet out a specific port (in this case, port 23). What type of device forwards packets based on the destination MAC address?

  Of course this is a simple implementation of a bridge or learning switch, which looks at the destination MAC and forwards out the appropriate port.

  So one can see that it would be easy to take a simple device consisting of HW and OpenFlow, and turn that inexpensive device into a learning switch. Perhaps it could even do away with problems like spanning tree in the process.

- The third example moves up a little bit more. It matches the destination IP address, and based on that value (with mask applied), it will forward the

packet out the appropriate port (in this case port 82).

Layer 3 forwarding of this nature is of course what a router would do. The possibility would be to take that simple device with HW and openflow, and turn it into a router using flows such as this, set by an intelligent SDN application running on the controller.

NOTES

OpenFlow Flow Entry Examples (2)

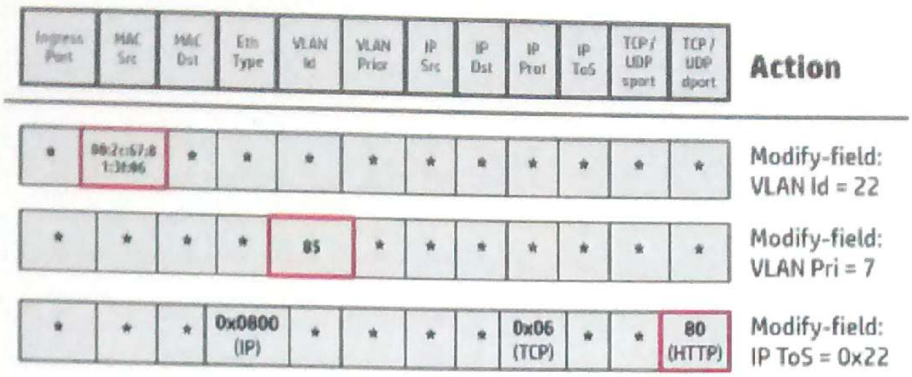


Figure 1-71: OpenFlow Flow Entry Examples (2)

Here are three more examples of flows - your job is to figure out how the flows might be used. These examples are a little more involved than those on the previous page.

- In the first match, the switch is instructed to match packets based on source MAC address; based on the source MAC address, the action is to modify the VLAN for the outgoing packet (which will then presumably be sent out into the network after the VLAN has been set.

  The question then is, what type of application might use flows such as this? Fundamentally, it is segregating users based on their MAC address, and putting them into different VLANs.

  If you are familiar with campus networks, hospitality, enterprises etc., treating devices or users different, you are on the right track. Some type of flow such as this could be used to segregate guests on the network from regular users; it could be used to segregate certain device types into quarantined portions of the network (e.g. put all iPads and tablets into a specific subnet).

- In the second match, the flow is set so that it matches incoming packets based on their VLAN. In this case, the assumption is that devices are already segregated into different VLANs; and the action is to modify the VLAN priority to be set to a high value.

  What type of application would use a flow such as this?

  This is clearly a QoS type of application attempting to make sure that traffic is assigned the correct priority and queuing as it passes through the network. Voice over IP traffic would require low-latency, and thus high priority. The same for video. And something like a backup requires only that the backup eventually finish, without too much concern for latency or delay, hence it would fall into this category as well.

- The last match is checking the incoming packet against three fields: Ethertype (it must be IP), IP Protocol (it must be TCP), and destination port (it must be 80). As a result of a match, the action would be to modify the

ToS priority field to a low value (0x22).

What is an application such as this trying to accomplish?

This type of flow could be used by an application that was trying to perform some traffic shaping on the network - lowering the priority of HTTP (TCP port 80) traffic. Perhaps this rule is configured to be set only at certain times of the day, e.g. working hours. At other times, HTTP traffic flows normally.

These are just a few examples of flows that could be set, and the applications that would likely set flows of each type. With this information you now have the knowledge to begin to think about designing your own flows to suit your own application needs.

NOTES

OpenFlow 1.1 Changes

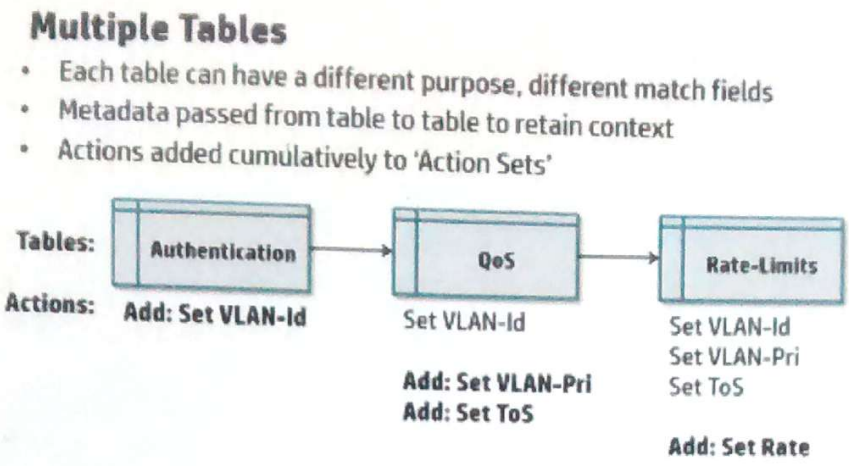


Figure 1-72: OpenFlow 1.1 Changes – multiple tables

We now consider some changes that have been made in different versions of OpenFlow. Recall that the first real version of OpenFlow that was widely implemented was OpenFlow 1.0. In fact, many vendor devices still only support OpenFlow 1.0.

OpenFlow 1.1 added a number of significant features; perhaps the most significant is the definition of multiple tables. Remember that OpenFlow 1.0 had just one Flow Table. OpenFlow 1.1 allows there to be multiple tables, which can be chained together via any logic defined by the application.

One use of this feature would be to have different tables server different purposes, linked in series, as shown in the diagram. In the figure:

- Packets arrive and are passed through the authentication table, which determines if the user (based on MAC address) is allowed into the network. If they are, then the VLAN is set appropriate to the policy that applies to them.
- Packets are then passed through the next table, which determines what QoS to set for this user. The matches could be based on the type of traffic, the incoming VLAN, the existing priority (CoS or ToS), or whatever. Based on this, the priority gets set (or re-set).
- The last table could be used for setting the rate-limits. Now prior to OF 1.3, per-flow meters were not available, however HP switches had these available. So we use them in the example here, where rate-limits are set for the user.

Imagine setting these using one table - you would need a huge number of flows, because you would need a specific entry for every single possible combination. You would end up with devices X VLAN X Protocol X CoS X ..., which would fill up any flow table quite quickly. Using multiple tables greatly reduces the number of flows required.

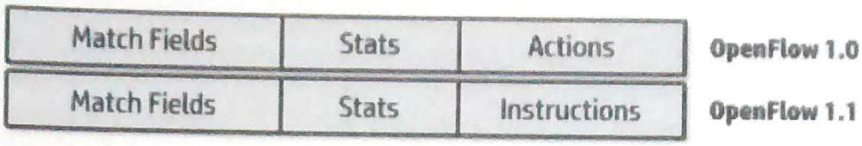
For extra credit, consider most learning switch applications, which end up setting two flows for each source-destination pair. The number of flows required for this type of application will be (numberOfDevices)X(numberOfDevices-1). So for example, if there are six devices, the number of flows will be 6X5=30. If there are 60 devices, the number of flows required will be 60X59=3540.

Using multiple tables, one for source and one for destination, the number of flows for 6 devices would be 12 (rather than 30), and 120 (rather than 3540). You can see that the savings in number of flows grows significantly as the number of devices increases.

If you didn't get all of that last discussion, don't worry, we will discuss this application in subsequent sections.

NOTES

OpenFlow 1.1 Changes (cont'd)



Actions → Instructions

- OF 1.0: Each flow entry is associated with zero or more Actions
- OF 1.1: Each flow entry is associated with a set of Instructions:
  - Changes packet (Apply- or Clear-Action(s))
  - Changes Action Set (Write-Action)
  - Changes pipeline processing (Write-Metadata or Goto-Table)

Figure 1-73: OpenFlow 1.1 Changes

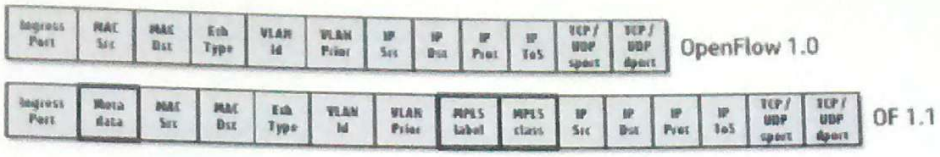
It naturally flows from the understanding of multiple tables in OF 1.1, that the definition of 'actions' would change somewhat, because one of the 'actions' that could be taken based on a flow table match might be to go to another specific table, and evaluate the packet based on that. This doesn't fall into the normal definition of 'action', and so that terminology has been changed to 'instruction'.

Some of the instructions can now be:

- Changes to the packet (apply current actions immediately, or even clear the current set of actions)
- Add some new actions
- Change processing (transfer control to another table, perhaps passing some meta-data to that table)

NOTES

OpenFlow 1.1 Changes (cont'd)



New Match Fields

- **Metadata**: For communication passed between tables
- **MPLS Label**: Matches on *outermost* MPLS tag
- **MPLS Traffic Class**: Matches on *outermost* MPLS tag

Figure 1-74: OpenFlow 1.1 Changes – match fields

Following along with the multiple tables' idea, it is possible to pass data from one table to another - a field has been added to the match which is the Metadata, used for this purpose of passing along hints or info to the subsequent table.

Another addition to the match, coming about in OpenFlow 1.1, are fields related to MPLS. The MPLS label and class can now be matched, so that MPLS may be supported.

Note that the matching is based on the outermost tag - inner tags are not matched.

NOTES

OpenFlow 1.1. Changes (cont'd)

Pushing / Popping Tags

- Push / Pop VLAN tags (QinQ)
- Push / Pop MPLS tags (MPLS)
- Inserted as the *outermost* tag
- Tag-stacking encapsulation by ISPs

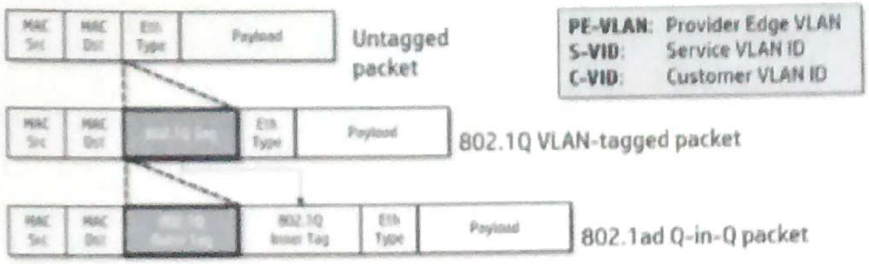


Figure 1-75: OpenFlow 1.1 Changes – push/pop tags

Part of the support for MPLS includes the ability to push and pop tags. This is standard need for MPLS so it is no surprise that this functionality would need to be supported in OF 1.1 if it was going to support MPLS.

In addition to the pushing and popping of MPLS tags, the protocol has been enhanced to push and pop VLAN tags as well. Pushing and popping of these tags is fundamentally similar to what is done for MPLS, so it follows that both of these 'tag stacking' features would be supported.

Note that tags are inserted as the outermost tag, which is the common manner in which tags are pushed onto packets as they pass an edge. The reverse is also true - the outermost tag is what gets removed when it exits the domain for which the tag was applied.

NOTES

OpenFlow 1.1 Changes (cont'd)

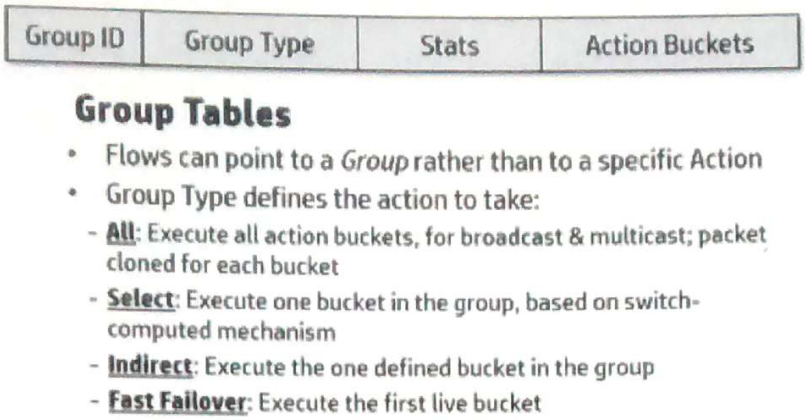


Figure 1-76: OpenFlow 1.1 Changes – group tables

OpenFlow 1.1 also added the concept of Group Tables. These group tables serve a number of functions, depending on the type of group it is. This means that actions associated with specific flows, rather than explicitly saying something like "output=1", can instead make a reference to a group table.

Note the term "action bucket" - this is basically just a set of actions, grouped together for the purposes of being referenced as one in the group types defined below

Group tables can server a number of purposes, depending on the group type:

- ALL: this type of group executes all action buckets, meaning that it can be useful for things like broadcast and multicast. The packet gets cloned as each action is performed.
- INDIRECT: this type of group executes the one defined action bucket, which makes this useful for defining a set of actions once, then having flow entries point to this one group, rather than each flow entry having to define the same set of actions (which is inefficient, prone to error, etc.).
- SELECT: this type of group is useful for allowing the switch to perform some form of load-balancing, of its own choosing, without consulting the controller. In this situation, only one of the action buckets is executed, having been selected by the switch.
- FAST FAILOVER: this type of group is useful for allowing the switch to perform its own form of redundancy. The idea would be that there would be multiple action buckets, and the switch will use the currently-selected action bucket, as long as it is 'live'. If any of the links in the action bucket goes down, then the switch will choose a different action bucket - thus providing an immediate failover mechanism.

So these group tables greatly improve the ability of the switch to operate in an efficient and reliable method, without requiring consultation from the controller.

Looking back at OpenFlow 1.1, clearly a number of cool and important features were added - multiple tables especially. However - and this is important - this is a point where the theory of OpenFlow diverged a bit from the practice, in terms of what was implement-able. Consider that with OF 1.0, there was a direct translation from a Flow Table to a TCAM.

However with OF 1.1, the idea of having *multiple* tables became an issue - how would the network device vendors implement that feature? Non-trivial for sure. This is a case where a software switch is much easier to implement than a hardware version.

Consequently, while all vendors jumped on the OF 1.0 bandwagon, very few (if any) implemented OF 1.1. Those that did, typically implemented features in software, for prototype purposes.

NOTES

Openflow 1.2 Changes

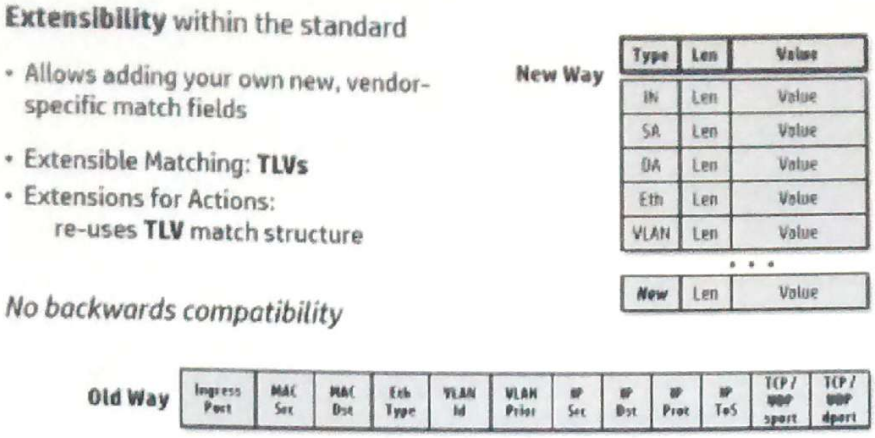


Figure 1-77: OpenFlow 1.2 Changes

OpenFlow 1.2 added more functionality into the mix, the two main features being:

- Extensibility: the ability to extend OpenFlow with the attributes of a vendors' choosing, both for matching and modifying.
- IPv6 support: the ability to match on IP addresses, type, code, flow labels, etc.

Extensibility changed the way that matching worked in OpenFlow. With OpenFlow 1.0 and 1.1, we used a fixed 12-tuple or 15-tuple to define wildcards and match field values. This changes with OF 1.2 to a variable structure using TLV (type-length-value) fields. This allows for vendors to add their own match fields, particular to their devices, without violating the OF standard, which was impossible with OF 1.0 or 1.1.

The diagram shows an example of the old way of doing things, and for comparison a list of TLVs for specifying the non-wildcarded fields the new way.

An important thing to note is that the same TLV structure is used for actions, where fields are to be modified.

NOTES

OpenFlow 1.2 Changes (cont'd)

Adds **IPv6** support

- Match on IPv6 source/destination address
- Match on IPv6 type, code, neighbor discovery
- Match on IPv6 flow label

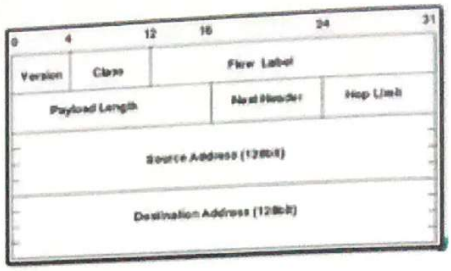


Figure 1-78: OpenFlow 1.2 Changes; adding IPv6

Not much to say here except that IPv6 is now supported in OF. Matches are possible on source/dest IPv6 address, IPv6 type, code and neighbor discovery fields.

It is also possible to match on the IPv6 flow label.

NOTES

can add fields to openflow standards.

OpenFlow 1.3 Changes

Openflow 1.3: PBB – spb(m)

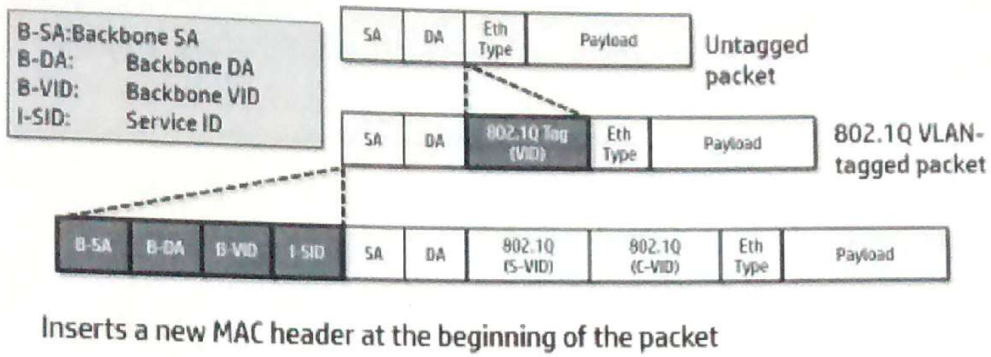


Figure 1-79: OpenFlow 1.3 Changes

For OpenFlow 1.3, the major enhancement that we mention here is support for Provider Backbone Bridging (PBB), which uses MAC-in-MAC tunneling.

PBB is needed by service providers who have packets which traverse provider edges and must get encapsulated and tagged as they pass through the provider network. One method of doing this is using MAC-in-MAC encapsulation, as shown in the diagram.

The image with MAC-in-MAC is that a brand new MAC header is prepended onto the beginning of the packet. As can be seen in the packet, the prepended header consists of more than just the source and destination MAC - it also includes the new VID for this frame, and the Service ID. Going into the details of PBB is beyond the scope of this class, but suffice to say that this was an important addition to the standard.

NOTES

Environments for SDN Applications

SDN in Action

Environments for SDN

• **Datacenter**	Virtualization, multi-tenancy, recovery from failures, traffic engineering and load-balancing
• **WAN/Backbone**	Resiliency, reliability, determinism, traffic engineering and load-balancing
• **Campus**	Network access control, guest access, BYOD, hospitality networks
• **Security**	Firewalls, intrusion detection and prevention, blacklists, enforced quarantine

Figure 1-80: Environments for SDN Applications

As it turns out, there are many environments that are applicable for SDN.

- For data center environments, SDN can help with virtualization, multi-tenancy, recovery from link failures and system failures, and traffic engineering / load balancing.
- For WAN or backbone environments, SDN can help with resiliency, reliability, determinism and predictability, as well as traffic engineering and load balancing.
- In enterprise and campus environments, SDN can help achieve simplified network access control, guest access, BYOD (bring your own device), and hospitality.
- In the security domain, SDN can help in providing firewalls, in funneling traffic to IDS/IPS (intrusion detection systems, intrusion prevention systems), in creating blacklist filters, and in enforcing quarantine for devices.

NOTES

SDN Application Environments: Data Centers

SDN in Action

Datacenters and SDN

The bulk of the research being done, and of the investment being made around SDN technology, is happening in the datacenter space.

Why so much interest around SDN in the datacenter?

- Specific agility needs
- Specific resiliency needs
- Specific traffic engineering needs
- Specific multi-tenancy needs

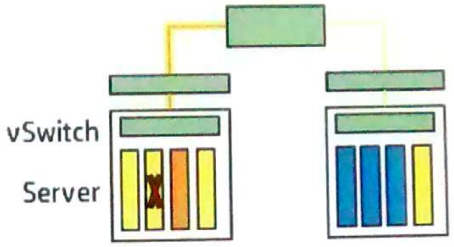


Figure 1-81: SDN Application Environments: Data Centers

Looking at data centers, SDN can help with many situations:

- When a server administrator wants to move a VM from one physical server to another, SDN's flexibility and automation allows the VM's networks to be provisioned automatically on the destination device.
- When a server goes down and must be re-started on a different physical server, SDN can again create the necessary network resources in the new location. And when the downed server comes back online, SDN can reverse those changes.
- When there is a need for multi-tenancy - even on a massive scale - SDN can use overlays to provide segregation of tenant traffic across the same physical links.
- When there is a need to maximize the utilization of various links in the network, SDN simplifies the task of reacting to traffic loads and re-routing certain flows on-demand.

NOTES

SDN Application Environments: WANs

SDN in Action

WAN Environments and SDN

Case Study: Google has already implemented SDN for their WANs

- COTS silicon
- OpenFlow 1.0, 1.3
- Failover and traffic shaping
- Superior resiliency, predictability
- So successful they are moving datacenter networking to SDN

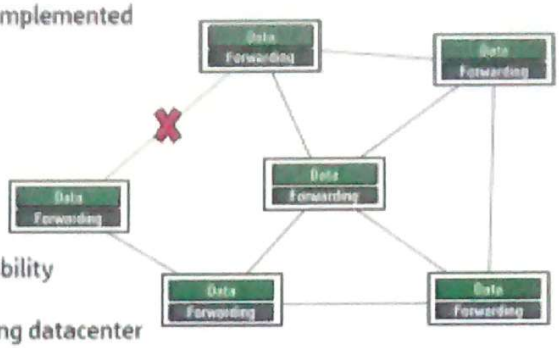


Figure 1-82: SDN Application Environments: WANs

WAN environments are sensitive to the cost of links between sites, and so they want to maximize their use of the bandwidth they own. Consequently issues related to traffic engineering are important.

In addition, recovery from faulty links is important. Recover must be immediate, and it must be predictable, which is not the case with current technology.

The best example of the applicability of SDN to WAN environments has been Google, who implemented SDN on the WAN links connecting their data centers. They did with:

- Using common off the shelf (COTS) networking chips.
- Using OpenFlow (with enhancements)

What the found was that they realized:

- Superior resiliency (recovery from failure)
- Superior predictability (it recovers the same way from the failures - not dependent on timing as are today's solutions)

The results of this 'experiment' is that they are working on moving their data centers to SDN.

NOTES

Google WAN Implementation

SDN in Action

Google WAN without openflow

- Automonomous competition for paths
- Only one wins, others retry
- Repeat until everybody has a path

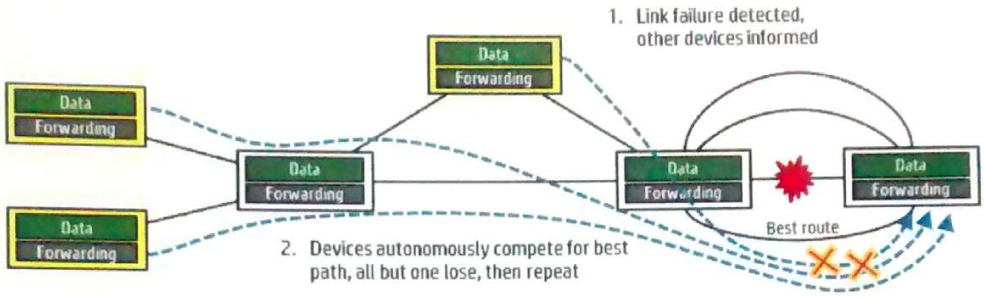


Figure 1-83: Google WAN Implementation

This is an example that Google has shown, indicating why they prefer SDN solutions for the WAN:

- In the figure, the initial link between the devices on the right fails (shown by the red explosion)
- There are three possible ways to route that traffic to overcome the failed link. One of these links is clearly the best choice.
- In today's systems, all three other switches are going to attempt to claim that best choice route. Only one will succeed. Which one succeeds with depend in part on timing and luck.
- The ones that failed will have to try again, this time fighting over the two top links. Again, one will succeed and the other will fail.

It is this randomness and unpredictability that Google hoped to resolve with SDN.

NOTES

Google WAN Implementation (cont'd)

SDN in Action

Google WAN with openflow

- Optimal path computation
- Repeatable path computation

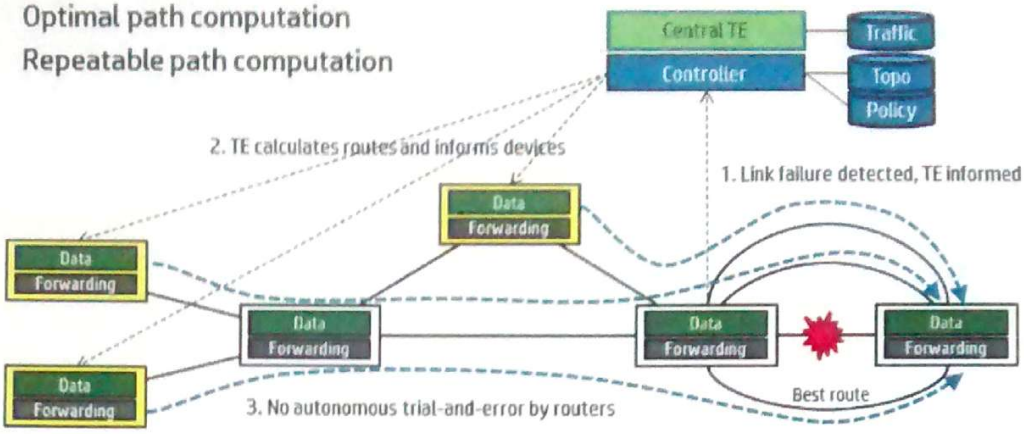


Figure 1-84: Google WAN Implementation (cont'd)

With the SDN solution from Google, the central controller is able to make optimal decisions with network-wide views.

- When the link goes down, the controller is notified, and examines the path requirements of the three other switches.
- The controller takes traffic, topology, and policy into consideration when determining which switches get which route to the destination.
- The controller sets flows in the switches in the network to achieve the requirements based on these calculations.

NOTES

SDN App Environments: Routed Networks

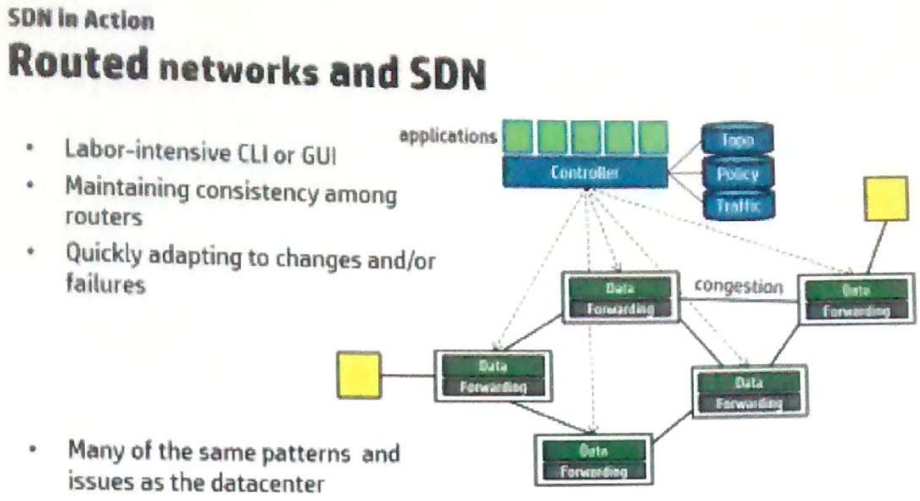


Figure 1-85: SDN App Environments: Routed Networks

In general routed environments, many of the same issues exist as were true with WANs, described earlier.

Here is a brief look at what SDN offers routed environments:

- Departure from the days of being forced to use the labor-intensive *and error prone* CLI commands.
- Consistent configuration is guaranteed because it is being driven by a central policy server.
- Consistency of configurations - the bane of many network administrators' existence - is guaranteed by the central controller.

In addition to these situations, the network is able to react to changes due to failures or other disruptions in the network.

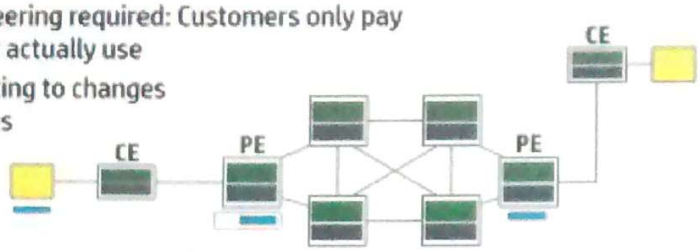
NOTES

SDN App Environments: Carrier / Provider

SDN in Action

Carrier Environments and SDN

- Many boundaries requiring encapsulation
- Traffic engineering required: Customers only pay for what they actually use
- Quickly adapting to changes and/or failures



- Multiple customers, domains, layers, geographies
- Monetization: Squeezing costs, NFV

Figure 1-86: SDN App Environments: Carrier / Provider

We have mentioned carriers before - their need to deal with traffic that traverses boundaries, their need to reduce costs, their need to provide differentiated levels of service based on how much a customer has paid for higher performance, and their need to do all of this in a fast and efficient manner.

Some of the ways SDN addresses these:

- Encapsulation when crossing boundaries is now possible (MAC-in-MAC, Q-in-Q, MPLS) with OpenFlow 1.3.
- SDN features a network-wide view and the potential to incorporate traffic data into making routing decisions.
- We've discussed in WANs how SDN can provide quick and deterministic reaction to failure.
- SDN allows the customer to make the most efficient use of the network through agility and the ability to route traffic based on a broad range of inputs.

NOTES

SDN App Environments: Load Balancing

SDN In Action

Load Balancing and SDN

Load-balancing well-suited for SDN
- Flow-based forwarding decisions
- SDN's agility and automation

Challenges for SDN
- Stateful needs
- Deep packet inspection needs

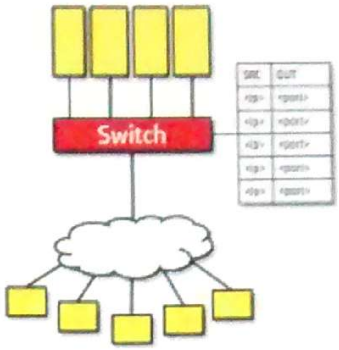


Figure 1-87: SDN App Environments: Load Balancing

Load Balancers today are expensive hardware appliances which are costly to purchase and replace. When an upgrade is needed, quite often it requires a new piece of hardware - updates happen on hardware timelines, rather than software.

In comparison, a switch can provide basic levels of load balancing, basing the distribution on simple values such as source IP address. Doing this saves costs and allows new functionality to be added by updating the application on the controller, not needing to update the device in most cases.

Of course there are challenges as well:

- If you need a stateful firewall, this solution will not achieve that, without forwarding too much traffic to the controller. OF rules are not stateful by nature.
- If you need deeper packet inspection (DPI) than is possible with the version of OpenFlow you are using, you will not be able to use this solution. The only alternative is forwarding the traffic to the controller, which will introduce undue delay.

So if one wants to have a load balancer implemented in a switch, some compromises must be made.

NOTES

SDN App Environments: Firewalls

SDN in Action

Firewalls and SDN

Firewalls well-suited to SDN
- Block/allow IP addresses
- Block/allow TCP/UDP ports

Challenges for SDN
- Complex/stateful firewall rules
- Deep packet inspection needs

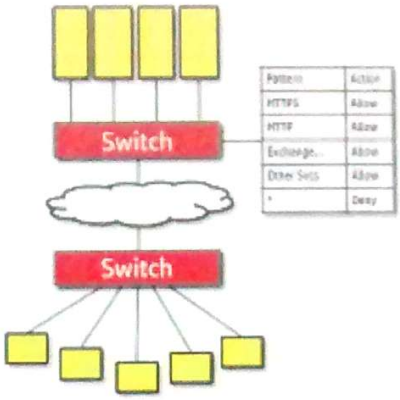


Figure 1-88: SDN App Environments: Firewalls

Implementing a firewall is similar to a load-balancer, in that it is simple in the more basic deployments scenarios. Firewalls which block traffic based on basic parts of the Ethernet header - such as are found in the OF 1.0 twelve-tuple - can easily be implemented.

However the challenges are similar to those present with load-balancers - statefulness is not supported today with OpenFlow. In a similar way, DPI is not possible, without special extensions (perhaps using OF 1.2).

NOTES

SDN App Environments: Campus / Enterprise

SDN in Action

Campus Environments and SDN

- Access control solutions today are expensive, error-prone, complicated, and cumbersome
- SDN simplifies the solution
  - MAC authentication via OpenFlow
  - Redirection to support BYOD
- Challenges for SDN
  - Flow table sizes
  - Co-existence with 802.1X

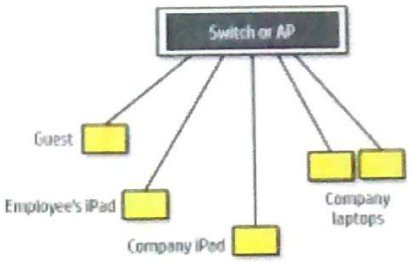


Figure 1-89: SDN App Environments: Campus / Enterprise

If you recall, the first SDN implementation - Ethane - was a network access control application, allowing devices access to the network based on policy, etc. It is not a far stretch to consider using this type of functionality for a full-fledged campus access application.

Using appropriate rules, performing redirections to get user devices registered, allowing guests and putting them into their own VLAN - these are all easily done with OpenFlow. Support for users' own devices can also be done by allowing them to register these devices through the redirection mentioned earlier.

Some of the more advanced functionality related to securing user devices - certificates and 802.1X - need to be achieved in conjunction with these features.

NOTES

Summary

In this module we have introduced software defined networking by:

- Looking at the history of networking, and at the precursors to SDN
- Examining the reasons why SDN became so important
- Defining the different types of SDN that exist today
- Described how SDN actually works, including a detailed look at OpenFlow

In the next sections we will look at:

- The basics of building an SDN application
- The method and techniques for creating SDN application designs
- The general APIs used in SDN applications today, for all controllers
- The specific APIs used in the HP SDN Controller

Those final three modules will be filled with labs in which you get an opportunity to experience the process of designing and implementing your own SDN application.