**18ECE320T - Software Defined Networks**

**UNIT 3**

**The OpenFlow Switch**

Figure 1 depicts the basic functions of an OpenFlow V.1.0 switch and its relationship to a controller. As would be expected in a packet switch, we see that the core function is to take packets that arrive on one port (path X on port 2 in the figure) and forward it through another port (port N in the figure), making any necessary packet modifications along the way. The table adjacent to packet matching is a flow table. The wide, gray, double arrow in Figure starts in the decision logic, shows a match with a particular entry in that table, and directs the now-matched packet to an action box on the right. This action box has three fundamental options for the disposition of this arriving packet:

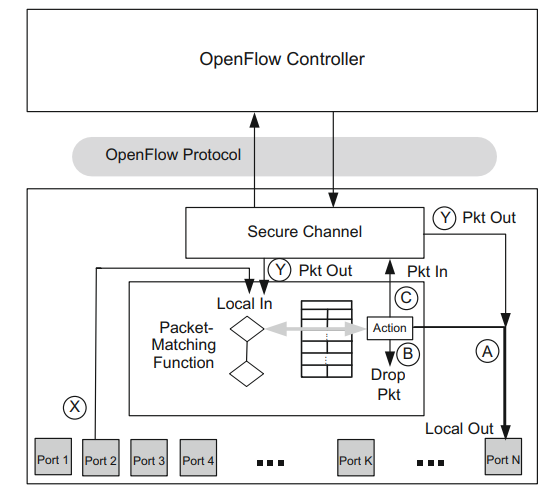
A. Forward the packet out a local port, possibly modifying certain header fields first.

B. Drop the packet.

C. Pass the packet to the controller.

In the case of path C, the packet is passed to the controller over the secure channel shown in the figure. If the controller has either a control message or a data packet to give to the switch, the controller uses this same secure channel in the reverse direction. When the controller has a data packet to forward out through the switch, it uses the OpenFlow PACKET\_OUT message.

**Figure 1: Open Flow V 1.0 Switch**



We see in Figure that such a data packet coming from the controller may take two different paths through the OpenFlow logic, both denoted Y . In the rightmost case, the controller directly specifies the output port and the packet is passed to that port N in the example. In the leftmost path Y case, the controller indicates that it wants to defer the forwarding decision to the packet-matching logic.

An OpenFlow-only switch: is one that forwards packets only according to the OpenFlow logic described above.

An OpenFlow hybrid: is a switch that can also switch packets in its legacy mode as an Ethernet switch or IP router.

**The OpenFlow Controller**

**Why control plan and data plane are separated?**

Modern Internet switches make millions of decisions per second about whether or not to forward an incoming packet, to what set of output ports it should be forwarded, and what header fields in the packet may need to be modified, added, or removed. This is a very complex task and can be carried out at line rates on multigigabit media. Not all functions on the switching datapath can be carried out at line rates, so there has long been the notion of splitting the pure data plane from the control plane.

The data plane matches headers, modifies packets, and forwards them based on a set of forwarding tables and associated logic, and it does this very, very fast. The rate of decisions being made as packets stream into a switch through a 100 Gbps interface is astoundingly high. The control plane runs routing and switching protocols and other logic to determine what the forwarding tables and logic in the data plane should be. This process is very complex and cannot be done at line rates as the packets are being processed, and it is for this reason we have seen the control plane separated from the data plane, even in legacy network switches.

The OpenFlow control plane differs from the legacy control plane in 3 key ways.

* **First,** it can program different data plane elements with a common, standard language, OpenFlow.
* **Second,** it exists on a separate hardware device than the forwarding plane, unlike traditional switches, where the control plane and data plane are instantiated in the same physical box. This separation is made possible because the controller can program the data plane elements remotely over the Internet.
* **Third,** the controller can program multiple data plane elements from a single control plane instance.

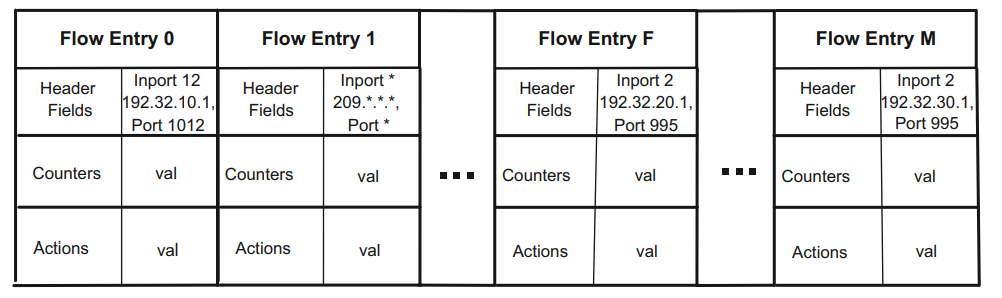
**The OpenFlow Protocol**

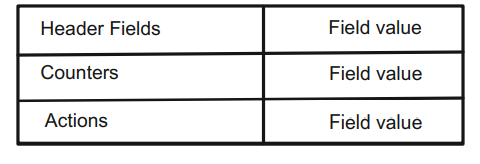
The OpenFlow specification defines the concept of an OpenFlow port. An OpenFlow V.1.0 port corresponds to a physical port. For many years, sophisticated switches have supported multiple queues per physical port. These queues are generally served by scheduling algorithms that allow the provisioning of different quality of service (QoS) levels for different types of packets. OpenFlow embraces this concept and permits a flow to be mapped to an already defined queue at an output port. Thus, the output of a packet on port N may include specifying onto which queue on port N the packet should be placed.

**Flow Table**

The flow table lies at the core of the definition of an OpenFlow switch. We depict a generic flow table in Figure 2. A flow table consists of flow entries, a flow entry consists of header fields, counters, and actions associated with that entry. The header fields are used as match criteria to determine whether an incoming packet matches this entry. If a match exists, the packet belongs to this flow. The counters are used to track statistics relative to this flow, such as how many packets have been forwarded or dropped for this flow. The actions fields prescribe what the switch should do with a packet matching this entry.

**Figure 2. Open Flow V1.0 Flow Table**





**Packet Matching**

When a packet arrives at the OpenFlow switch from an input port (or, in some cases, from the controller), it is matched against the flow table to determine whether there is a matching flow entry. The following match fields associated with the incoming packet may be used for matching against flow entries:

• Switch input port • VLAN ID • VLAN priority • Ethernet source address • Ethernet destination address • Ethernet frame type • IP source address • IP destination address • IP protocol • IP Type of Service (ToS) bits • TCP/UDP source port • TCP/UDP destination port.

These 12 match fields are collectively referred to as the basic 12-tuple of match fields. The flow entry’s match fields may be wildcarded using a bit mask, meaning that any value that matches on the unmasked bits in the incoming packet’s match fields will be a match.

Flow entries are processed in order, and once a match is found, no further match attempts are made against that flow table. For this reason, it is possible for there to be multiple matching flow entries for a packet to be present in a flow table. Only the first flow entry to match is meaningful; the others will not be found, because packet matching stops upon the first match. If the end of the flow table is reached without finding a match, this is called a **table miss.** In the event of a table miss in V.1.0, the packet is forwarded to the controller. If a matching flow entry is found, the actions associated with that flow entry determine how the packet is handled. The most basic action prescribed by an OpenFlow switch entry is how to forward this packet. A good abstraction hides the details of the thing being abstracted while still permitting sufficiently fine-grained control to accomplish the needed tasks.

The V.1.0 specification is silent about which of these 12 match fields are required versus those that are optional. The ONF has clarified this confusion by defining three different types of conformance in its V.1.0 conformance-testing program.

Types of conformance

Full conformance: meaning all 12 match fields are supported;

Layer two conformance: when only the layer two header field matching is supported;

Layer three conformance: when only layer three header field matching is supported.

**Actions and Packet Forwarding**

The required actions that must be supported by a flow entry are to either output (forward) or drop the matched packet. The most common case is that the output action specifies a physical port on which the packet should be forwarded. There are, however, **five** special virtual ports defined in V.1.0 that have special significance for the output action. They are **LOCAL, ALL, CONTROLLER, IN\_PORT, and TABLE**.

**LOCAL** dictates that the packet should be forwarded to the switch’s local OpenFlow control software, circumventing further OpenFlow pipeline processing. **LOCAL** is used when OpenFlow messages from the controller are received on a port that is receiving packets switched by the OpenFlow data plane. **LOCAL** indicates that the packet needs to be processed by the local OpenFlow control software.

**ALL** is used to flood a packet out all ports on the switch except the input port. This provides rudimentary broadcast capability to the OpenFlow switch.

**CONTROLLER** indicates that the switch should forward this packet to the OpenFlow controller.

**IN\_PORT** instructs the switch to forward the packet back out of the port on which it arrived. Effectively**, IN\_PORT** normally creates a loopback situation, which could be useful for certain scenarios.

**Two scenarios in which IN\_PORT is useful:**

1. 802.11 Wireless port: In the case of 802.11 wireless port, it is quite normal to receive a packet from that port from one host and to forward it to the receiving host via the same port. This needs to be done very carefully so as not to create unintended loopback situations. Thus the protocol requires explicit stipulation of this intent via this special virtual port.

2.Edge virtual bridging (EVB): EVB defines a reflective relay service between a physical switch in a data center and a lightweight virtual switch within the server known as a virtual edge port aggregator (VEPA). The standard IEEE 802.1Q bridge at the edge of the network will reflect packets back out the port on which they arrive to allow two virtual machines on the same server to talk to one another. This reflective relay service can be supported by the IN\_PORT destination in OpenFlow rules.

**TABLE** virtual port: Applies to packets that the controller sends to the switch. Such packets arrive as part of the PACKET\_OUT message from the controller, which includes an action list. This action list will generally contain an output action, which will specify a port number. The controller may want to directly specify the output port for this data packet, or, if it wants the output port to be determined by the normal OpenFlow packet-processing pipeline, it may do so by stipulating TABLE as the output port. These two options are depicted in the two Y paths shown in Figure 1.

**FLOOD**. In this case, the switch sends a copy of the packet out all ports except the ingress port.

Additional virtual ports, but support for these is optional in V.1.0.

* **NORMAL virtual port.** When the output action forwards a packet to the NORMAL virtual port, it sends the packet to the legacy forwarding logic of the switch.
* **LOCAL virtual port**, which designates that the packet be passed to the local OpenFlow control processing. Conversely, packets for which matching rules indicate NORMAL as the output port will remain in the ASIC to be looked up in other forwarding tables that are populated by the local (non-OpenFlow) control plane.

**Messaging between Controller and Switch**

The messaging between the controller and switch is transmitted over a secure channel. This secure channel is implemented via an initial TLS connection over TCP. (Subsequent versions of OpenFlow allow for multiple connections within one secure channel.) If the switch knows the IP address of the controller, the switch will initiate this connection. Each message between controller and switch starts with the OpenFlow header. This header specifies the OpenFlow version number, the message type, the length of the message, and the transaction ID of the message.

The various message types in V.1.0 fall into three general categories**: symmetric, controller-switch, and async.**

**Symmetric messages:** may be sent by either the controller or the switch without having been solicited by the other. The HELLO messages are exchanged after the secure channel has been established to determine the highest OpenFlow version number supported by the peers. The protocol specifies that the lower of the two versions is to be used for controller-switch communication over this secure channel instance. ECHO messages are used by either side during the life of the channel to ascertain that the connection is still alive and to measure the current latency or bandwidth of the connection. The VENDOR messages are available for vendor-specific experimentation or enhancements.

**Async messages: S**ent from the switch to the controller without having been solicited by the controller.

* The PACKET\_IN message is the way the switch passes data packets back to the controller for exception handling. Control plane traffic will usually be sent back to the controller via this message.
* FLOW\_REMOVED message : The switch can inform the controller that a flow entry is removed from the flow table via this message.
* PORT\_STATUS is used to communicate changes in port status, whether by direct user intervention or by a physical change in the communications medium itself.
* ERROR: Finally, the switch uses the ERROR message to notify the controller of problems.

**Controller-switch** is the broadest category of OpenFlow messages.

They can be divided into **five** subcategories: **switch configuration, command from controller, statistics, queue configuration, and barrier.**

The **switch configuration** messages consist of a unidirectional configuration message and two request-reply message pairs. The controller uses the unidirectional message, SET\_CONFIG to set configuration parameters in the switch. We see the SET\_CONFIG message sent during the initialization phase of the controller-switch dialogue. The controller uses the FEATURES message pair to interrogate the switch about which features it supports. Similarly, the GET\_CONFIG message pair is used to retrieve a switch’s configuration settings.

**Command from controller** category: **PACKET\_OUT** is the analog of the **PACKET\_IN**. The controller uses PACKET\_OUT to send data packets to the switch for forwarding out through the data plane. The controller modifies existing flow entries in the switch via the **FLOW\_MOD** message. **PORT\_MOD** is used to modify the status of an OpenFlow port.

S**tatistics** are obtained from the switch by the controller via the **STATS** message pair.

The **BARRIER** message pair is used by the controller to ensure that a particular OpenFlow command from the controller has finished executing on the switch. The switch must complete execution of all commands received prior to the **BARRIER\_REQUEST** before executing any commands received after it, and the switch notifies the controller of having completed such preceding commands via the **BARRIER\_REPLY** message sent back to the controller.

The **QUEUE\_GET\_CONFIG\_REQUEST** and **QUEUE\_GET\_CONFIG\_REPLY** message pair is the mechanism by which the controller learns from the switch how a given queue is configured. With this information, the controller can intelligently map certain flows to specific queues to achieve desired QoS levels.

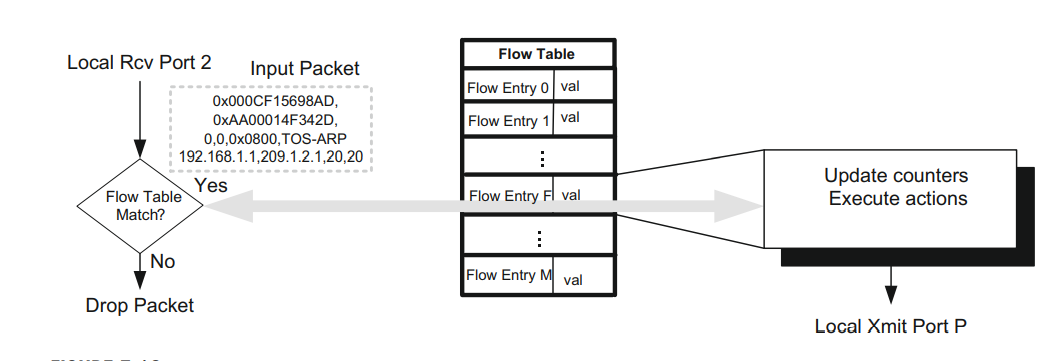
In the event that the **HELLO** protocol detects a loss of the connection between controller and switch, the V.1.0 specification prescribes that the switch should enter emergency mode and reset the TCP connection. At this time all flows are to be deleted except special flows that are marked as being part of the emergency flow cache. The only packet matching that is allowed in this mode is against those flows in that **emergency flow cache**.

**­­**

**Basic Packet forwarding**

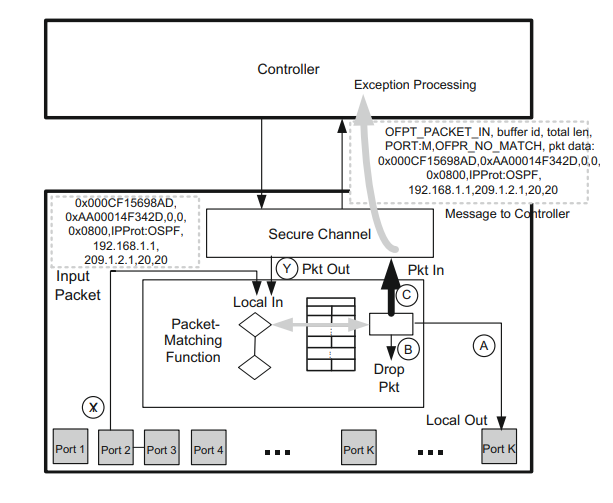
The figure depicts a packet arriving at the switch through port 2 with source IPv4 address of 192.168.1.1 and destination IPv4 address of 209.1.2.1. The packet-matching function scans the flow table starting at flow entry 0 and finds a match in flow entry F. Flow entry F stipulates that a matching packet should be forwarded out port P. The switch does this, completing this simple forwarding example.

An OpenFlow switch forwards packets based on the header fields it matches. The network programmer designates layer three switch behavior by programming the flow entries to try to match layer three headers such as IPv4. If it is a layer two switch, the flow entries will dictate matching on layer two headers. Whenever there is overlap in potential matches of flows, the priority the controller assigns the flow entry will determine which match takes precedence. For example, if a switch is both a layer two and a layer three switch, placing layer three header-matching flow entries at a higher priority would ensure that if possible, layer three switching will be done on that packet.

****

**Switch Forwarding Packet to Controller**

OpenFlow V.1.0 switch forwarding packets to the controller for exception handling. The two reasons for which the switch may forward a packet to the controller are OFPR\_NO\_MATCH and OFPR\_ACTION.

****

OFPR\_NO\_MATCH: is used when no matching flow entry is found.

OFPR\_ACTION: OpenFlow retains the ability to specify that a particular matching flow entry should always be forwarded to the controller. A control packet such as a OSPF routing protocol packet always needs to be processed by the controller. This particular matching flow entry should always be forwarded to the controller. OFPR\_ACTION is specified as the reason. This is shown in the figure above.

There is a matching table entry for this packet that specifies that the packet should be forwarded to the controller. We see in the figure that a PACKET\_IN message is sent via the secure channel to the controller, handing off this routing protocol update to the controller for exception processing. The processing that would likely take place on the controller is that the OSPF routing protocol would be run, potentially resulting in a change to the forwarding tables in the switch. The controller could then modify the forwarding tables via the brute-force approach of sending FLOW\_MOD commands to the switch modifying the output port for each flow in the switch affected by this routing table change.

OpenFlow allows the optional buffering of the full packet by the switch. In the event of a large number of packets being forwarded from the switch to the controller, for which the controller only needs to examine the packet header, significant bandwidth efficiency gains are achieved by buffering the full packet in the switch and only forwarding the header fields. Since the controller will sometimes need to see the balance of the packet, a buffer ID is communicated with the PACKET\_IN message. The controller may use this buffer ID to subsequently retrieve the full packet from the switch. The switch has the ability to age out old buffers that the switch has not retrieved.

There are other fundamental actions that the switch may take on an incoming packet: to flood the packet out all ports except the port on which it arrived, or to drop the packet.

**OpenFlow 1.3 Additions**

OpenFlow V.1.3 was released on April 13, 2012. This release was a major milestone.

The major new features added in this release are Refactor Capabilities Negotiation, More Flexible Table-Miss Support, Per-Flow Meters, Per Connection Event Filtering, Auxiliary Connections, Cookies in PACKET\_IN, Provider Backbone Bridging Tagging

**Refactor Capabilities Negotiation**

There is a new MULTIPART\_REQUEST/MULTIPART\_REPLY message pair in V.1.3. This replaces the READ\_STATE messaging in prior versions that used the STATS\_REQUEST/STATS\_REPLY to get this information. It is used for both reporting of statistics and capability information. The information is conveyed using a standard type-length-value (TLV) format. Some of the capabilities that can be reported in this new manner include next-table, table-miss flow entry, and experimenter.

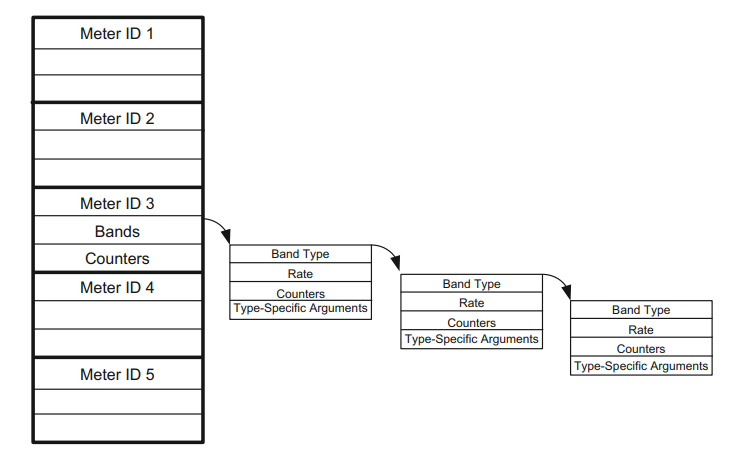
**More Flexible Table-Miss Support**

A table miss occurs when a packet does not match any flow entries in the current flow table in the pipeline. Formerly there were three configurable options for handling such a table miss. This included dropping the packet, forwarding it to the controller, or continuing packet matching at the next flow table. V.1.3 expands on this limited handling capability via the introduction of the table-miss flow entry. The controller programs a table-miss flow entry into a switch in much the same way it would program a normal flow entry. The table-miss flow entry is distinct in that it is by definition of lowest priority (zero) and all match fields are wildcards. The zero-priority characteristic guarantees that it is the last flow entry that can be matched in the table. The fact that all match fields are wildcards means that any packet being matched against the table miss will be a match.

**Per-Flow Meters**

V.1.3 introduces a flexible meter framework. Meters are defined on a per-flow basis and reside in a meter table. Figure shows the basic structure of a meter and how it is related to a specific flow.

A given meter is identified by its meter ID. V.1.3 instructions may direct packets to a meter identified by its meter ID. V.1.3 meters are only rate-limiting meters. As we see in Figure, there may be multiple meter bands attached to a given meter. Meter 3 in the example in the figure has three meter bands. Each meter band has a configured bandwidth rate and a type. Note that the units of bandwidth are not specified by OpenFlow. The type determines the action to take when that meter band is processed. When a packet is processed by a meter, at most one band is used.



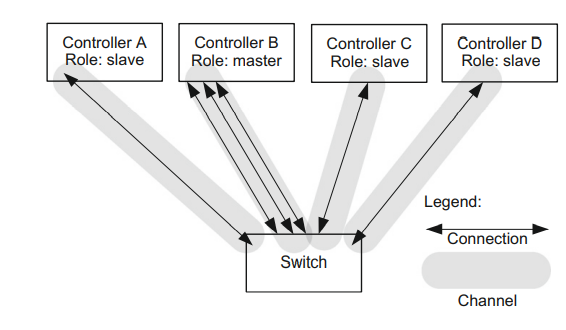
This band is selected based on the highest bandwidth rate band that is lower than the current measured bandwidth for that flow. If the current measured rate is lower than all bands, no band is selected and no action is taken. If a band is selected, the action taken is that prescribed by the band’s type field. As shown in Figure, there are meter-level counters and meter-band-level counters. The meter counters are updated for all packets processed by the meter, and the per-meter-band counters are updated only when that particular band is selected. The dual level of counters is key in that presumably the majority of the packets processed by the meter suffer no enforcement, but their passage through the meter must be recorded in order to track the current measured rate.

**Per Connection Event Filtering**

V.1.3 introduces a SET\_ASYNC message that allows the controller to specify the sorts of async messages it is willing to receive from a switch. It additionally allows the controller to filter out certain reason codes that it does not want to receive. A controller may use two different filters: one for the master/equal role and another for the slave role. This filter capability exists in addition to the ability to enable or disable asynchronous messages on a per-flow basis. This new capability is controller-oriented rather than flow-oriented.

**Auxiliary Connections**

V.1.3 introduces an additional layer of parallelism by allowing multiple connections per communications channel. That is, between a single controller and switch, multiple connections may exist. The figure shows that there are also multiple channels connecting the switch to different controllers. The advantage provided by the additional connections on a channel lies in achieving greater overall throughput between the switch and the controller. Because of the flow-control characteristics of a TCP connection, it is possible for the connection to be forced to quiesce (due to TCP window closing or packet loss) when there is actually bandwidth available on the physical path(s) between the switch and controller. Allowing multiple parallel connections allows the switch to take advantage of that.

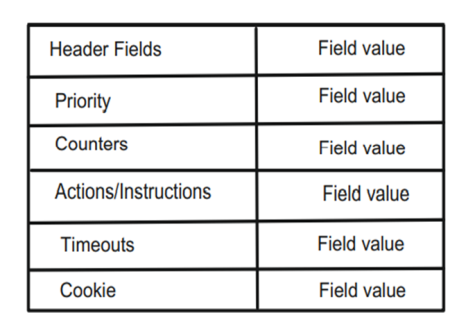
****

One example of the utility of auxiliary connections is that when a switch has many PACKET\_IN messages to send to the controller, this can create a bottleneck due to congestion. The delay due to this bottleneck could prevent important OpenFlow control messages, such as a BARRIER\_REPLY message, from reaching the controller. Sending PACKET\_IN data messages on the UDP auxiliary connection will obviate this situation. Another possible extension to this idea is that the OpenFlow pipeline could send UDP-based PACKET\_IN messages directly from the ASIC to the controller without burdening the control code on the switch CPU.

**V1.3 Flow Entry**

The timeouts field can be used to maintain two timeout clocks for this flow. Both an idle timer and a hard timeout may be set. If the controller wants the flow to be removed after a specified amount of time without matching any packets, then the idle timer is used.

If the controller wants the flow to exist only for a certain amount of time, regardless of the amount of traffic, the hard timeout is used and the switch will delete this flow automatically when the timeout expires.

****

**OpenFlow Limitations**

1. The currently defined match fields are limited to the packet header. Thus, deep packet inspection (DPI), whereby fields in the packet’s payload may be used to distinguish flows, is not supported in a standard OpenFlow.

2. Some OpenFlow abstractions may be too complex to implement directly in today’s silicon.

**Publish and Subscribe Interfaces**

Publish-Subscribe interfaces, or simply pub-sub as it is more commonly known, is a messaging pattern whereby senders of messages (called publishers) send messages to receivers (called subscribers

Senders do not program the messages to be sent directly to specific receivers but rather characterize published messages into classes.

In this model, subscribers express interest in one or more classes, and thereby only receive messages that are in the class of messages they are interested in.

It is called a messaging bus whereby messages are placed on the bus, and subscribers simply receive them.

This pattern provides greater network scalability and a more dynamic network topology than a point-to-point system would due to properties such as lower state management requirements.

Subscribers typically receive only a subset of the total messages published. It simultaneously lowers the burden of an application around message processing and lessens the overall system load of message delivery, maintenance, and accounting.

The process of selecting messages for reception and processing is called message filtering. The two common forms of filtering are called topic-based and content-based.

**Topic based system:** publishes messages to topics that represent logical sets or are analogous to logical channels in a broadcast system. Subscribers in a topic-based system will receive all messages published to the topics to which they subscribe, but no others.

**Content-based system:** only delivers messages to a subscriber if the attributes or content of those messages match constraints defined by the subscriber. This is analogous to setting up search filters for email messages with matches on certain fields of a message.

**Hybrid System:** Many messaging systems support a hybrid of the two approaches in that publishers can post messages to a topic, and subscribers may simultaneously register content-based subscriptions to topics

**Broker based:** publishers post messages to an intermediary message broker or event bus. The broker normally performs a store and forward function in order to buffer messages so that they are not lost in case of congestion

**Advantages and disadvantages of pub-sub model**

The pub-sub model provides the opportunity for better scalability than traditional tightly coupled client-server approaches in that parallel operation, message caching, and tree-based or network-based routing are possible within this system.

One of most serious problems with the pub-sub approach is the decoupling of the publisher from the subscriber. Pub-sub system so-designed cannot guarantee delivery of messages to any applications that might require such assured delivery. If publishers or subscribers are unaware of this limitation, then synchronization and other consistency issues might arise.

**XMPP**

**Extensible Messaging and Presence Protocol (XMPP)** is an example of a pub-sub protocol and has been used to implement a number of publish subscribe systems. XMPP is a communications protocol based on XML [Extensible Markup Language] .

The protocol can be used to provide near real-time instant messaging, presence information, or just about any information really that needs to be extended to a subscription group.XMPP is an open protocol standardized at the IETF. Many implementations have been developed and distributed that are in use, such as Jabber, Google Talk, and Facebook Messenger.

The server itself acts like the message broker we described in the pub-sub section. It handles all of the registration and message passing required. Publishers and subscribers all register with the server using a topic based approach in that they filter based on participating in what is effectively a group conversation. The architecture is decentralized by design in that there is no global authoritative server In the XMPP architecture, every user in the system has a unique Jabber ID. In the original specification, XMPP could use HTTP in two ways: either in polling mode or in a binding model.

**What is Network Orchestation?**

Network orchestration refers to actions a network controller performs in setting up devices, applications, and services in the network to achieve objectives. It's much like an orchestra conductor's role in directing individual musicians as they perform a piece of music together.

**What are the benefits of network orchestration?**

Network orchestration delivers the following benefits:

* It has a brain: the network controller. The controller translates business needs into network requirements, sets up the network to deliver on those requirements, and monitors it to help ensure that business needs are being met.
* It views the network as a whole and not as individual parts.
* It synchronizes all parts of the network to help achieve objectives.

### How is network orchestration different from network automation?

Network automation refers to performing discrete, fairly simple tasks without manual intervention. Examples of automation include uploading a new configuration file to a switch and updating the switch's software image—jobs that each achieve a single objective.

Orchestration refers to performing a series of related tasks to achieve a more-complex objective. A network controller executes automated tasks in a purposeful order and verifies the success of each task before performing the next one.

As an example, orchestrating a new wireless SSID might consist of identifying and reconfiguring the appropriate access points and wireless LAN controllers, and setting up proper credentials, security mechanisms, allowed bandwidth, etc., for the SSID.

### What is OpenStack?

OpenStack is a collection of open source software modules and tools that provides a framework to create and manage both public cloud and private cloud infrastructure.

OpenStack delivers infrastructure-as-a-service functionality -- it pools, provisions and manages large concentrations of compute, storage and network resources. These resources, which include bare metal hardware, virtual machines (VMs) and containers, are managed through application programming interfaces (APIs) as well as an OpenStack dashboard. Other OpenStack components provide orchestration, fault management and services intended to support reliable, high availability operations.

Businesses and service providers can deploy OpenStack on premises (in the data center to build a private cloud), in the cloud to enable or drive public cloud platforms, and at the

### What does OpenStack do?

To create a cloud computing environment, an organization typically builds off of its existing virtualized infrastructure, using a well-established hypervisor such as VMware vSphere, Microsoft Hyper-V or KVM. However, cloud computing offers more than just virtualization -- a public or private cloud provides extensive provisioning, lifecycle automation, user self-service, cost reporting and billing, orchestration and other features.

Installing OpenStack software on top of a virtualized environment forms a cloud operating system. An organization can use that to organize, provision and manage large pools of heterogeneous compute, storage and network resources. Whereas an IT administrator typically provisions and manages resources in a more traditional virtualized environment, OpenStack enables individual users to provision resources through management dashboards and an API.

This cloud-based infrastructure created through OpenStack supports an array of uses cases, including web hosting, big data projects, software-as-a-service delivery or container deployment.

### How does OpenStack work?

OpenStack is not an application in the traditional sense, but rather a platform composed of several dozen separate components, called *projects*, which interoperate with each other through APIs. Each component is complementary, but not all components are required to create a basic cloud. Organizations can install only select components that build the features and functionality in a desired cloud environment.

OpenStack also relies on two additional foundation technologies: a base operating system, such as Linux, and a virtualization platform, such as VMware or Citrix. The OS handles the commands and data exchanged from OpenStack, while the virtualization engine manages the virtualized hardware resources used by OpenStack projects.

Once the OS, virtualization platform and OpenStack components are deployed and configured properly, administrators can provision and manage the instanced resources that applications require. Actions and requests made through a dashboard produce a series of API calls, which are authenticated through a security service and delivered to the destination component, which executes the associated tasks.

OpenStack setups vary, but typically start with a handful of central components: **compute (Nova), VM images (Glance), networking (Neutron), storage (Cinder or Swift), identity management (Keystone) and resource management (Placement).**



OpenStack

OpenStack13 is a global collaboration whose aim is to produce the open standard cloud operating system for both public and private clouds. OpenStack is a freely available, Apache-licensed software system that can be used to build massively scalable cloud environments.

**Nova** is open source software designed to provision and manage large networks of virtual machines, creating a redundant and scalable cloud-computing platform. The software provides control panels and APIs required to orchestrate a cloud. Nova is written in Python

**Swift and Cinder** is open source software for creating redundant, scalable data storage using clusters of standard servers to store multiple blocks of accessible data. It is not a file system or real-time data system, but rather a long-term storage system for large amounts of static data that can be retrieved or updated. Object Storage uses a distributed architecture in order to not have a central point of failure. This also affords the user greater flexibility of deployment options, as well as the obvious scalability, redundancy, and performance.

**Glance** provides discovery, registration, and delivery services for virtual disk images. The Image Service API server provides a well-defined RESTful web services interface for querying information about virtual disk images. These disk images may be stored in a variety of backend stores, including OpenStack Object Storage. Clients can register new virtual disk images with the Image Service, query for information on publicly available disk images, and use the Image Service’s client library for streaming virtual disk images. These images can then be referenced later much in the way a menu of dishes can be made available to a diner in a restaurant.

**Quantum** provides the API that builds required network connectivity between Open‐ Stack physical nodes (i.e., between the vNICs managed by Openstack Nova—providing them network as a service functionality).

**What are the pros and cons of OpenStack?**

Many enterprises that deploy and maintain an OpenStack infrastructure enjoy several advantages, including that it is:

* Affordable. OpenStack is available freely as open source software released under the Apache 2.0 license. This means there is no upfront cost to acquire and use OpenStack.
* Reliable. With almost a decade of development and use, OpenStack provides a comprehensive and proven production-ready modular platform upon which an enterprise can build and operate a private or public cloud. Its rich set of capabilities includes scalable storage, good performance and high data security, and it enjoys broad acceptance across industries.
* Vendor-neutral. Because of OpenStack's open source nature, some organizations also see it as a way to avoid vendor lock-in, as an overall platform as well as its individual component functions.

But potential adopters must also consider some drawbacks, such as the following:

* Complexity. Because of its size and scope, OpenStack requires an IT staff with significant knowledge to deploy the platform and make it work. In some cases, an organization might require additional staff or a consulting firm to deploy OpenStack, which adds time and cost.
* Support. As open source software, OpenStack is not owned or directed by any one vendor or team. This can make it difficult to obtain support for the technology, beyond the open source community.
* Consistency. The OpenStack component suite is always in flux as new components are added and others are deprecated.

**CloudStack**

CloudStack14 is a Cloud Orchestration platform that pools computing resources to build public, private, and hybrid Infrastructure as a Service (IaaS) clouds. CloudStack is very similar to OpenStack in that it manages the network, storage, and compute nodes that make up a cloud infrastructure. A CloudStack cloud has a hierarchical structure that enables it to scale to manage large numbers of physical servers, all from a single management interface.

The CloudStack architecture is comprised of some basic elements:

**Pods**: A pod is hardware that has been configured to form clusters. A pod is most usually a data center rack containing one or more clusters and connectivity to a layer 2 switch that is shared by all clusters in that pod. It is important in the CloudStack architecture that end users are unaware of and have no visibility of pods.

**Clusters**: A cluster is a group of identical hosts running a common hypervisor. For example, a cluster could be a VMware cluster pre-configured in vCenter. Each cluster has a dedicated primary storage device. This storage device is where the virtual machine instances are hosted and launched from. With multiple hosts within a cluster, high availability and load balancing are standard features of a CloudStack deployment

The **secondary storage system:** is used to store virtual machine templates, ISO images, and snapshots. These are used later to launch instances of VMs. The storage is available to all pods in a zone. Storage can also be replicated between availability zones, thereby providing a common storage platform throughout the whole cloud. This can also be automated.

**Primary storage** is unique to each cluster and is used to host the virtual machine instances. Since primary storage is a critical component, it is often built on highperformance hardware with multiple high-speed disks that afford the system an element of both redundancy and higher-performance.

**Network Function Virtualization**

**What is network functions virtualization?**

Network functions virtualization (NFV) is the replacement of network appliance hardware with virtual machines. The virtual machines use a hypervisor to run networking software and processes such as routing and load balancing.

**Why network functions virtualization?**

NFV allows for the separation of communication services from dedicated hardware, such as routers and firewalls. This separation means network operations can provide new services dynamically and without installing new hardware. Deploying network components with network functions virtualization takes hours instead of months like with traditional networking. Also, the virtualized services can run on less expensive, generic servers instead of proprietary hardware.

Additional reasons to use network functions virtualization include:

Pay-as-you-go: Pay-as-you-go NFV models can reduce costs because businesses pay only for what they need.

Fewer appliances: Because NFV runs on virtual machines instead of physical machines, fewer appliances are necessary and operational costs are lower.

Scalability: Scaling the network architecture with virtual machines is faster and easier, and it does not require purchasing additional hardware.

**Benefits of network functions virtualization**

Many service providers feel that the benefits of network functions virtualization outweigh the risks. With traditional hardware-based networks, network managers have to purchase dedicated hardware devices and manually configure and connect them to build a network. This is time-consuming and requires specialized networking expertise.

NFV allows virtual network function to run on a standard generic server, controlled by a hypervisor, which is far less expensive than purchasing proprietary hardware devices. Network configuration and management is much simpler with a virtualized network. Best of all, network functionality can be changed or added on demand because the network runs on virtual machines that are easily provisioned and managed.

**NFV architecture**

In a traditional network architecture, individual proprietary hardware devices such as routers, switches, gateways, firewalls, load balancers and intrusion detection systems all carry out different networking tasks. A virtualized network replaces these pieces of equipment with software applications that run on virtual machines to perform networking tasks.

An NFV architecture consists of three parts:

**Centralized virtual network infrastructure:** An NFV infrastructure may be based on either a container management platform or a hypervisor thatabstracts the compute, storage and network resources.

**Software applications:** Software replaces the hardware components of a traditional network architecture to deliver the different types of network functionality (virtualized network functions).

**Framework:** A framework (often known as MANO – management, automation and network orchestration) is needed to manage the infrastructure and provision network functionality.

**Service Engineered Path**

In 2010, a proposal to decouple the Service and Network infrastructure (plane) was proposed by Jim Guichard, then a principal architect in the CTO Office at Juniper Networks.

The Service Engineered Path (SEP) concept was introduced as a potentially new means of service delivery for the Service Provider community—The problem statement addressed in SEP was that service providers were constrained in making new service offerings by the need to deploy the service appliances that would comprise the service offering.

Service offerings included firewalls, Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), load balancers, and SSL off-loaders that were run within the edge routers that would serve their projected markets.

These constraints made service introduction laborious and disruptive to network operation and put a premium on predicting the markets for a service

**Motivation for SEP**

Network devices that provide Service Enabling Technologies (SETs) would be transparent to the general network infrastructure.

Changes/additions/upgrades to one or more service instances wouldn’t affect routing in the network, providing a more stable service introduction environment.

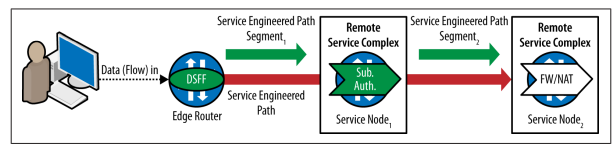
Provide flexible service SET and instance placement, and streamlined capacity planning of services.

Faster time to market for services from a design, upgrade, testing, and deployment perspective.

Edge routers need not be upgraded every time a new service is added or upgraded with new functionality.

Providing the ability to link together services of differing types, thereby enabling new and innovative bundled services

**Basic SEP Concept**



The construction of service was a hierarchy of the following components:

Service - A service function, application, or content used singularly or in collaboration with other SETs to enable a service.

SET Sequence Predetermined sequence of SETs that form the service.

Set Sequence Path As an instance of a SET may be available at multiple points in the network, there are potentially several combinations of Service Nodes that could form part of the SET sequence. The SET sequence path is a list of [Service Node, SET] combinations available that could be used to satisfy the service