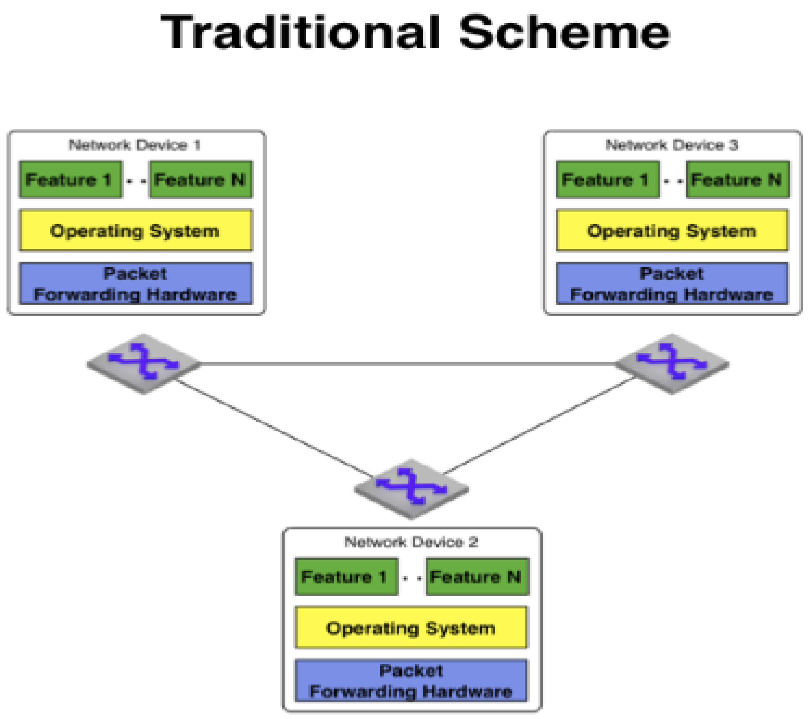
Ref: Internet of Things: A Hands-on Approach Book by Arshdeep Bahga and Vijay K. Madisetti
Ref: Software Defined Networks A Comprehensive Approach Paul Göransson Chuck Black

Software Defined Networking

Software-Defined Networking (SDN) is a networking architecture that separates the control plane from the data plane and centralizes the network controller.



- Figure above shows the conventional network architecture built with specialized hardware (switches, routers, etc.).
- Network devices in conventional network architectures are getting exceedingly **complex** with the **increasing number of distributed protocols** being implemented and the use of **proprietary hardware and interfaces**.

- In the conventional network architecture the control plane and data plane are coupled.
- **Control plane** is the part of the network that carries the **signaling and routing message** traffic while the **data plane** is the part of the network that carries the **payload data traffic**.

The limitations of the conventional network architectures are as follows:

- **Complex Network Devices**:Conventional networks are getting increasingly complex with **more and more protocols** being implemented to improve link speeds and reliability.

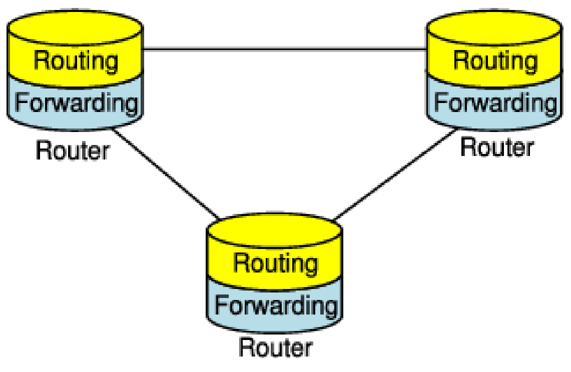
- **Limited Interoperability**:Interoperability is limited due to the lack of standard and open interfaces. Network devices use proprietary hardware and software and have slow product life-cycles limiting innovation.

- **Dynamic Traffic pattern:** The conventional networks were well suited for static traffic patterns and had a large number of protocols designed for specific applications. For IoT applications which are deployed in cloud computing environments, the traffic patterns are more dynamic. Due to the complexity of conventional network devices, making changes in the networks to meet the dynamic traffic patterns has become increasingly difficult.

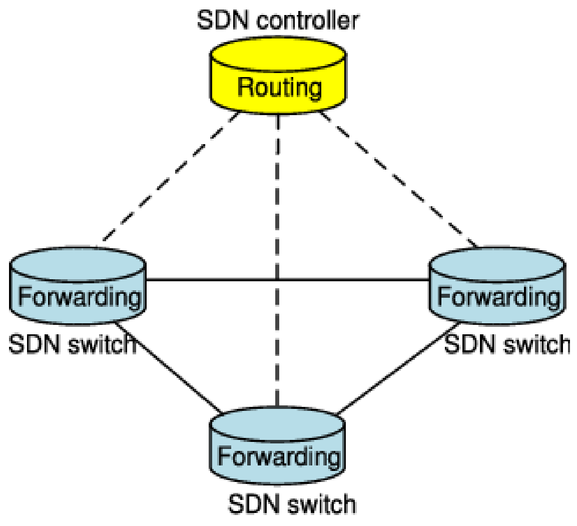
- **Management Overhead:** Conventional networks involve significant management overhead. Network managers find it increasingly difficult to manage **multiple network devices and interfaces** from multiple vendors. Upgradation of network **requires configuration changes in multiple devices** (switches, routers, firewalls, etc.)

- **Limited Scalability:** The virtualization technologies used in cloud computing environments has increased the **number of virtual hosts requiring network access**. IoT applications hosted in the cloud are distributed across multiple virtual machines that require exchange of traffic. The analytics components of lot applications run distributed algorithms on a large number of virtual machines that require huge amounts of data exchange between virtual machines. Such computing environments require highly scalable and easy to manage network architectures with **minimal manual configurations,** which is becoming **increasingly difficult with conventional networks.**

- SDN attempts to create network architectures that are simpler, inexpensive, scalable, agile and easy to manage.
- Figures below show the SDN architecture and the SDN layers in which the control and data planes are **decoupled and the network controller is centralized**.
- Software-based SDN controllers maintain a **unified view** of the network and make configuration, **management and provisioning simpler.**
- The **underlying infrastructure** in SDN uses simple **packet forwarding hardware** as opposed to specialized hardware in conventional networks.



Traditional network
(Routing and forwarding are coupled)



SDN network
(Routing and forwarding are decoupled)

- Network devices become simple with SDN as they do not require implementations of a large number of protocols.

- Network devices receive instructions from the SDN controller on how to forward the packets.

- These devices can be simpler and cost less as they can be built from standard hardware and software components.

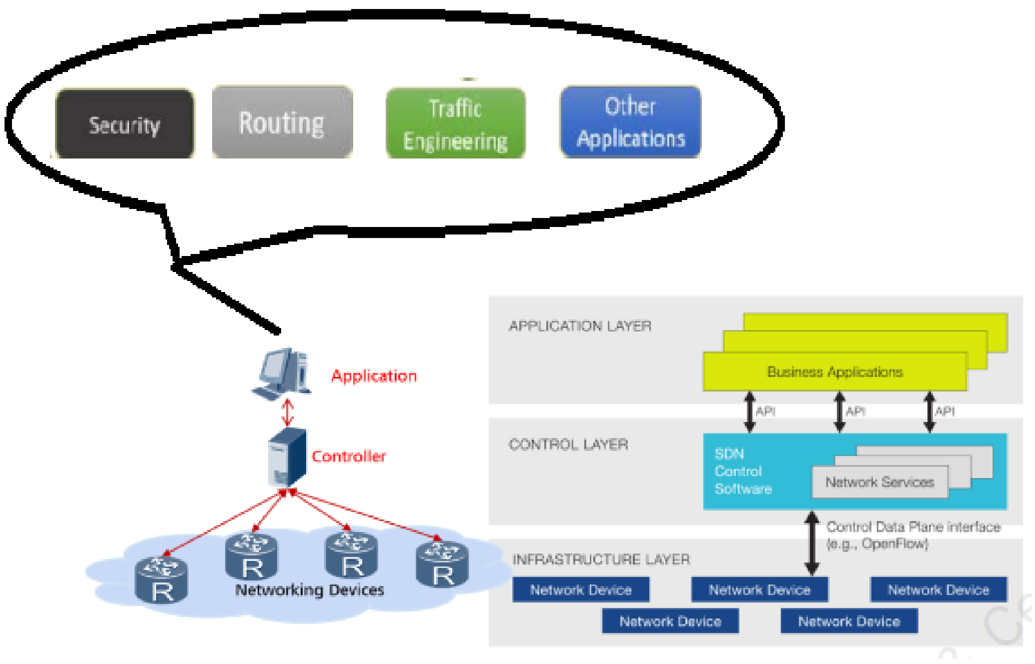
Key elements of SDN are as follows:

Centralized Network Controller:

- With decoupled control and data planes and centralized network controller, the network administrators can rapidly configure the network.
- SDN applications can be deployed through programmable open APIs. This speeds up innovation as the network administrators no longer need to wait for the device vendors to embed new features in their proprietary hardware.

Programmable Open APIs:

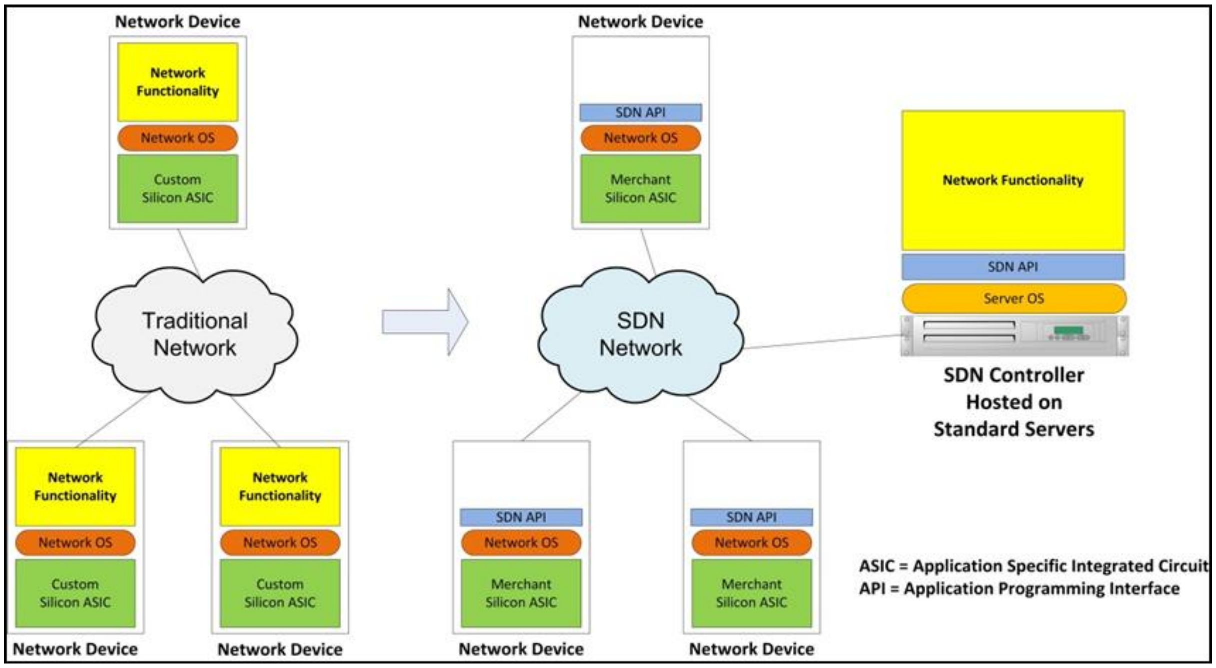
- SDN architecture supports programmable open APIs for interface between the SDN application and control layers (Northbound interface).
- With these open APIs various network services can be implemented, such as routing, quality of service (QoS), access control, etc.



The underlying network infrastructure is abstracted from the applications.

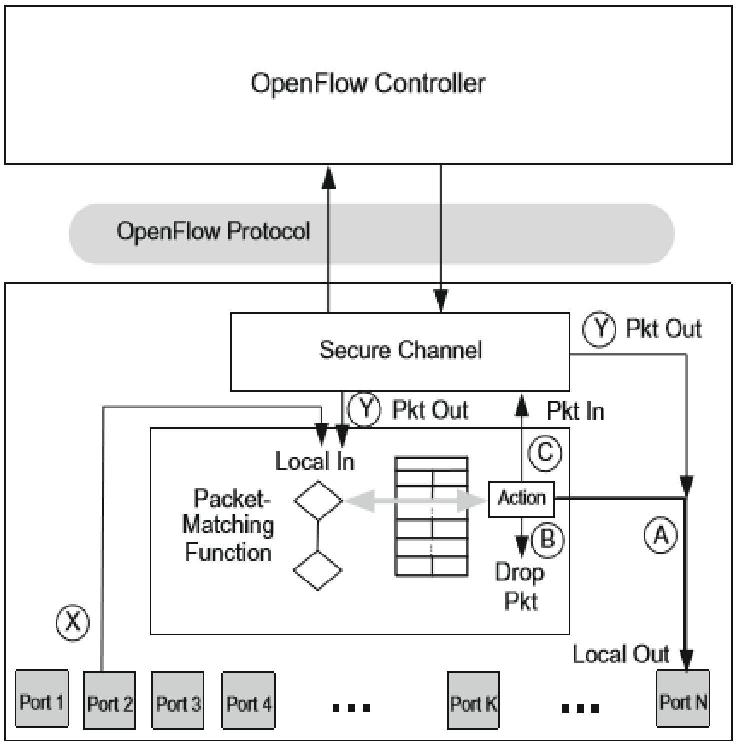
Standard Communication Interface (OpenFlow):

- SDN architecture uses a standard communication interface between the control and infrastructure layers (Southbound interface).
- OpenFlow, which is defined by the Open Networking Foundation (ONF) is the broadly accepted SDN protocol for the **Southbound interface**.
- With OpenFlow, the forwarding plane of the network devices can be directly accessed and manipulated.
- OpenFlow uses the concept of flows to identify network traffic based on pre-defined match rules.
- Flows can be programmed statically or dynamically by the SDN control software.



Openflow

The OpenFlow Switch



FIGURE

OpenFlow V.1.0 switch.

- Figure above 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.

- A unique aspect of the OpenFlow switch is embodied in the packet-matching function shown in Figure.

- The adjacent table is a flowtable, and we give separate treatment to this later. 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:
- Forward the packet out a local port, possibly modifying certain header fields first.
- Drop the packet.
- Pass the packet to the controller.
- These three fundamental packet paths are illustrated in Figure.

- 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.

- 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 left most path Y case, the controller indicates that it wants to defer the forwarding decision to the packet-matching logic.

- A given OpenFlow switch implementation is either OpenFlow-only or OpenFlow-hybrid.

- 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.

- One can view the hybrid case as an OpenFlow switch residing next to a completely independent traditional switch. Such a hybrid switch requires a preprocessing classification mechanism that directs packets to either OpenFlow processing or the traditional packet processing.

- It is probable that hybrid switches will be the norm during the migration to pure OpenFlow implementations.

The OpenFlow Controller

- 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. The fact that this can be carried out at line rates on multigigabit media is a technological wonder.
- The switching industry has long understood that not all functions on the switching data path 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 dataplane 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 100Gbps 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 three 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 controller is responsible for programming all the packet-matching and forwarding rules in the switch. Whereas a traditional router would run routing algorithms to determine how to program its forwarding table, that function or an equivalent replacement to it is now performed by the controller. Any changes that result in recomputing routes will be programmed onto the switch by the controller.

The OpenFlow Protocol

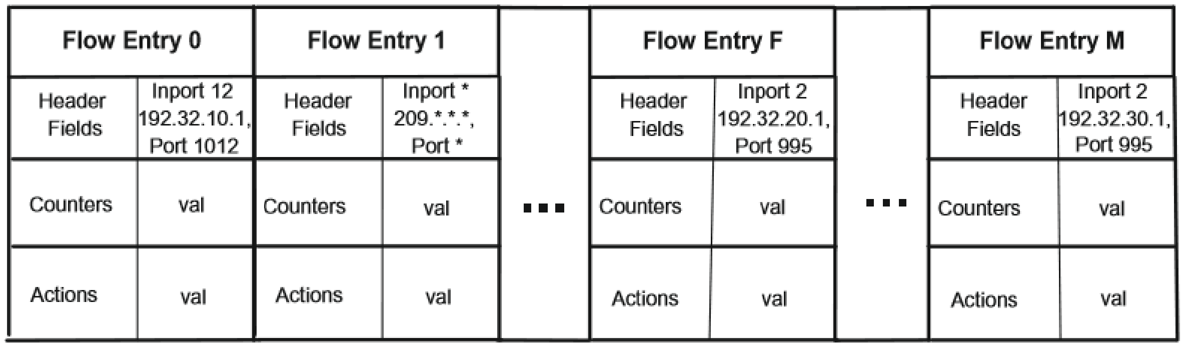
- As shown in Figure, the Open Flow protocol defines the communication between an OpenFlowcontroller and an OpenFlow switch.
- This protocol is what most uniquely identifies OpenFlow technology.
- At its essence, the protocol consists of a set of messages that are sent from the controller to the switch and a corresponding set of messages that are sent in the opposite direction.
- Collectively the messages allow the controller to program the switch so as to allow fine-grained control over the switching of user traffic. The most basic programming defines, modifies, and deletes flows.
- A flowis defined as a set of packets transferred from one network endpoint (or set of end points) to another endpoint (or set of endpoints).

- The end points may be defined as IPaddress-TCP/UDPportpairs, VLAN endpoints, layer three tunnel endpoints, or input ports, among other things.
- One set of rules describes the forwarding actions that the device should take for all packets belonging to that flow.
- When the controller defines a flow, it is providing the switch with the information it needs to know how to treat incoming packets that match that flow.
- The possibilities for treatment have grown more complex as the OpenFlow protocol has evolved, but the most basic prescriptions for treatment of an incoming packet are denoted by paths A, B, and C in Figure shown ago.
- These three options are to forward the packet out one or more output ports, drop the packet, or pass the packet to the controller for exception handling. The OpenFlow protocol has evolved significantly with each version of OpenFlow, so we cover the detailed messages of the protocol in the version-specific sections that follow.

OpenFlow 1.0 and OpenFlow Basics

Flow Table

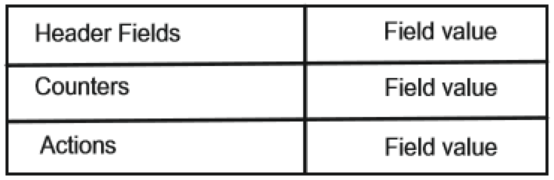
- The flowtable lies at the core of the definition of an Open Flow switch. We depict a generic flowtable in Figure below.
- A flowtable consists of flowentries, one of which is shown in second figure below. A flowentry consists of header fields, counters, and actions associated with that entry.



Flow Entry 0		Flow Entry 1			Flow Entry F			Flow Entry M	
Header Fields	Inport 12 192.32.10.1, Port 1012	Header Fields	Inport * 209.*.*.*, Port *		Header Fields	Inport 2 192.32.20.1, Port 995		Header Fields	Inport 2 192.32.30.1, Port 995
Counters	val	Counters	val	■■■	Counters	val	■■■	Counters	val
Actions	val	Actions	val		Actions	val		Actions	val

FIGURE

OpenFlow V.1.0 flow table.



Header Fields	Field value
Counters	Field value
Actions	Field value

FIGURE

Basic flow 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.

Packet Matching

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

• Switch input port
• VLAN ID
• VLAN priority
• Ethernet source address
• Ethernet destination address
• Ethernet frame type
• IP source address
• IP destination address
• IP protocol
• IPType 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 flowentry'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 flowtable. (We will see in subsequent versions of OpenFlow that there may be additional flowtables against which packet matching may continue.) For this reason, it is possible for there to be multiple matching flowentries for a packet to be present in a flowtable. Only the first flowentry to match is meaningful; the others will not be found, because packet matching stops upon the first match.

If the end of the flowtable 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. (Note that this is no longer strictly true in later versions.) If a matching flowentry is found, the actions associated with that flowentry determine how the packet is handled. The most basic action prescribed by an OpenFlow switch entry is how to forward this packet. The most common case is that the output action specifies a physical port on which the packet should be forwarded

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 are listed in Table 5.1. The messages fall into three general categories: symmetric, controller-switch, and async. We explain the categories of messages shown in Table 5.1 in the following paragraphs. We suggest that the reader refer to Figure 5.8 as we explain each of these messages. Figure 5.8 shows the most important of these messages in a normal context and illustrates whether it normally is used during the initialization, operational, or monitoring phases of the controller-switch dialogue.

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 connectionisstillaliveandtomeasurethecurrentlatencyorbandwidthoftheconnection.
- The VENDOR messages are available for vendor-specific experimentation or enhancements.

Async messages are sent 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.
- The switch can inform the controller that a flow entry is removed from the flow table via the FLOW_REMOVED 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.
- Finally, the switch uses the ERROR message to notify the controller of problems**.**

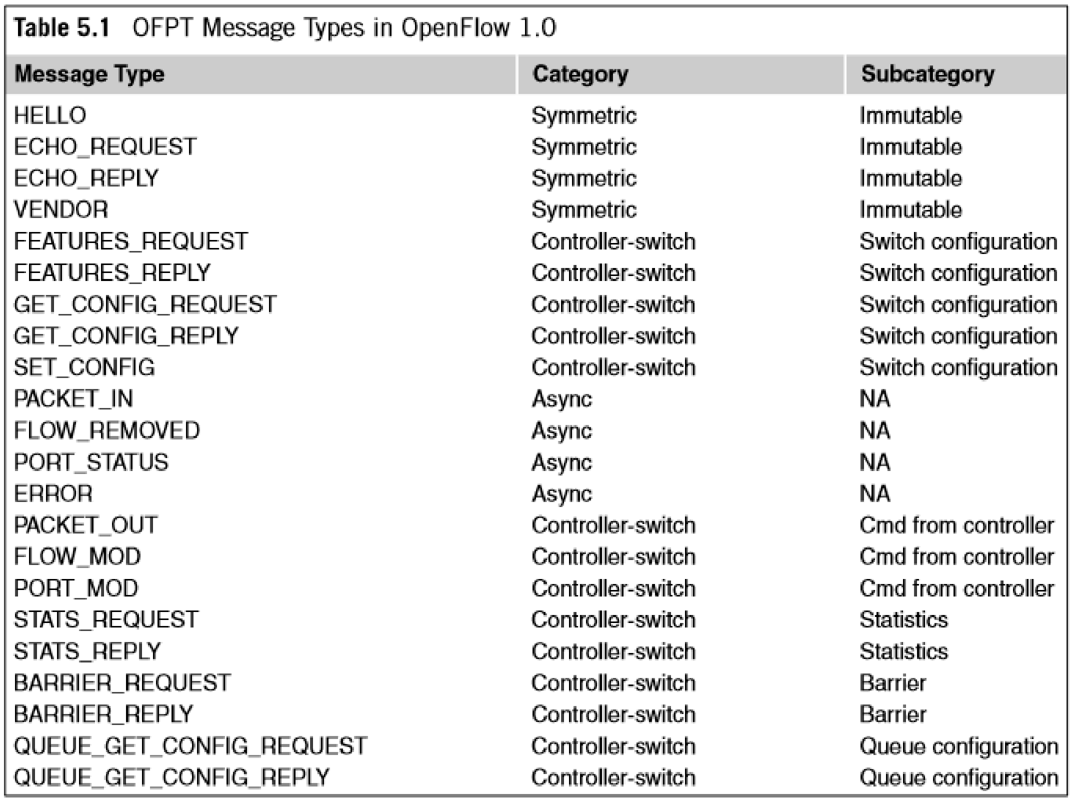
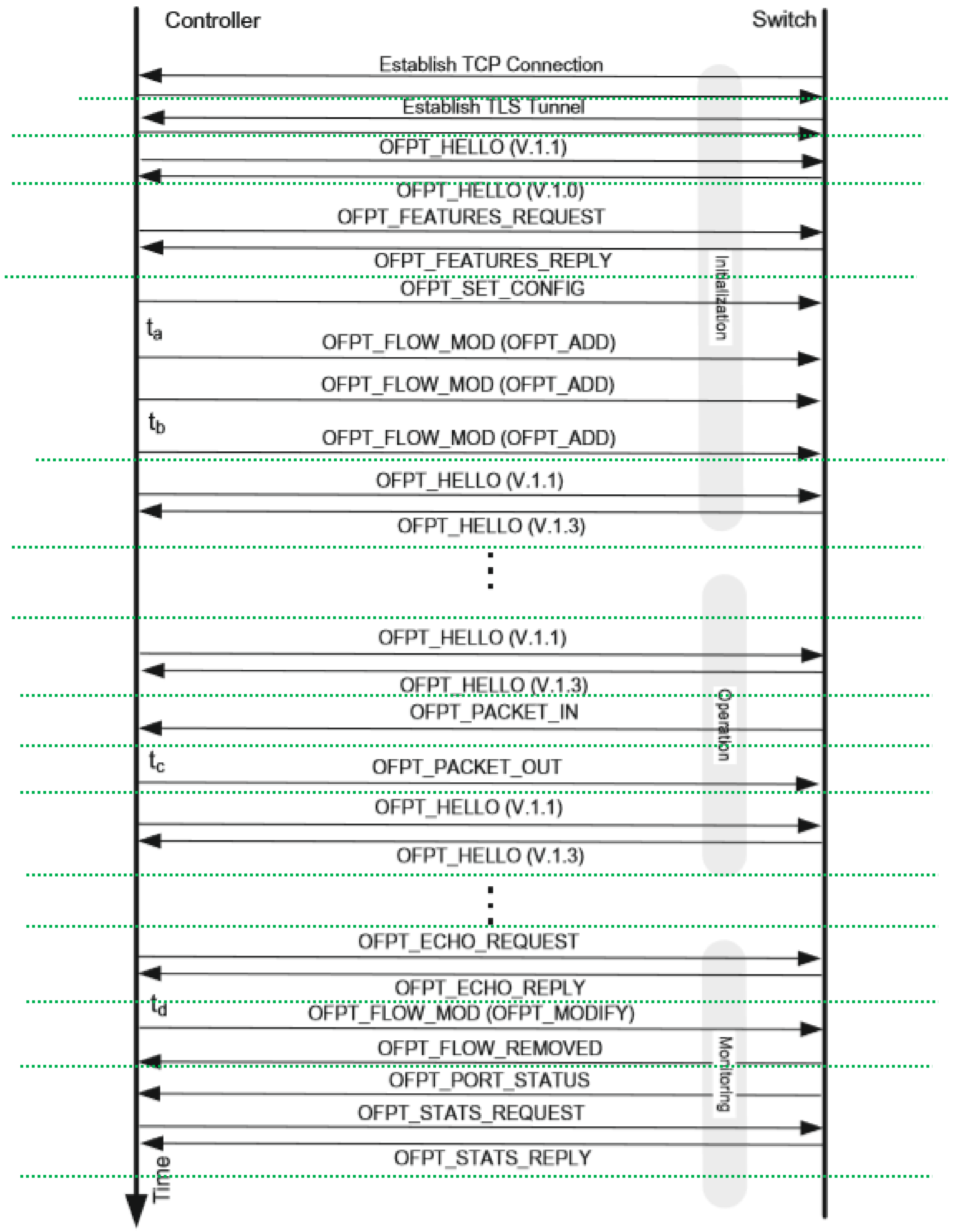


Table 5.1 OFPT Message Types in OpenFlow 1.0

Message Type	Category	Subcategory
HELLO	Symmetric	Immutable
ECHO_REQUEST	Symmetric	Immutable
ECHO_REPLY	Symmetric	Immutable
VENDOR	Symmetric	Immutable
FEATURES_REQUEST	Controller-switch	Switch configuration
FEATURES_REPLY	Controller-switch	Switch configuration
GET_CONFIG_REQUEST	Controller-switch	Switch configuration
GET_CONFIG_REPLY	Controller-switch	Switch configuration
SET_CONFIG	Controller-switch	Switch configuration
PACKET_IN	Async	NA
FLOW_REMOVED	Async	NA
PORT_STATUS	Async	NA
ERROR	Async	NA
PACKET_OUT	Controller-switch	Cmd from controller
FLOW_MOD	Controller-switch	Cmd from controller
PORT_MOD	Controller-switch	Cmd from controller
STATS_REQUEST	Controller-switch	Statistics
STATS_REPLY	Controller-switch	Statistics
BARRIER_REQUEST	Controller-switch	Barrier
BARRIER_REPLY	Controller-switch	Barrier
QUEUE_GET_CONFIG_REQUEST	Controller-switch	Queue configuration
QUEUE_GET_CONFIG_REPLY	Controller-switch	Queue configuration

Controller-switch is the broadest category of OpenFlow messages. In fact, as shown in Table 5.1, 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. In Figure 5.8 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.

There are three messages comprising the **command from controller** category. PACKET_OUT is the analog of the PACKET_IN mentioned above. The controller uses PACKET_OUT to send data packets to the switch for forwarding out through the dataplane. 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.

Statistics 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 configuration message pair is somewhat of a misnomer in that actual queue configuration is beyond the scope of the OpenFlow specification and is expected to be done by an unspecified outof-band mechanism. 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.

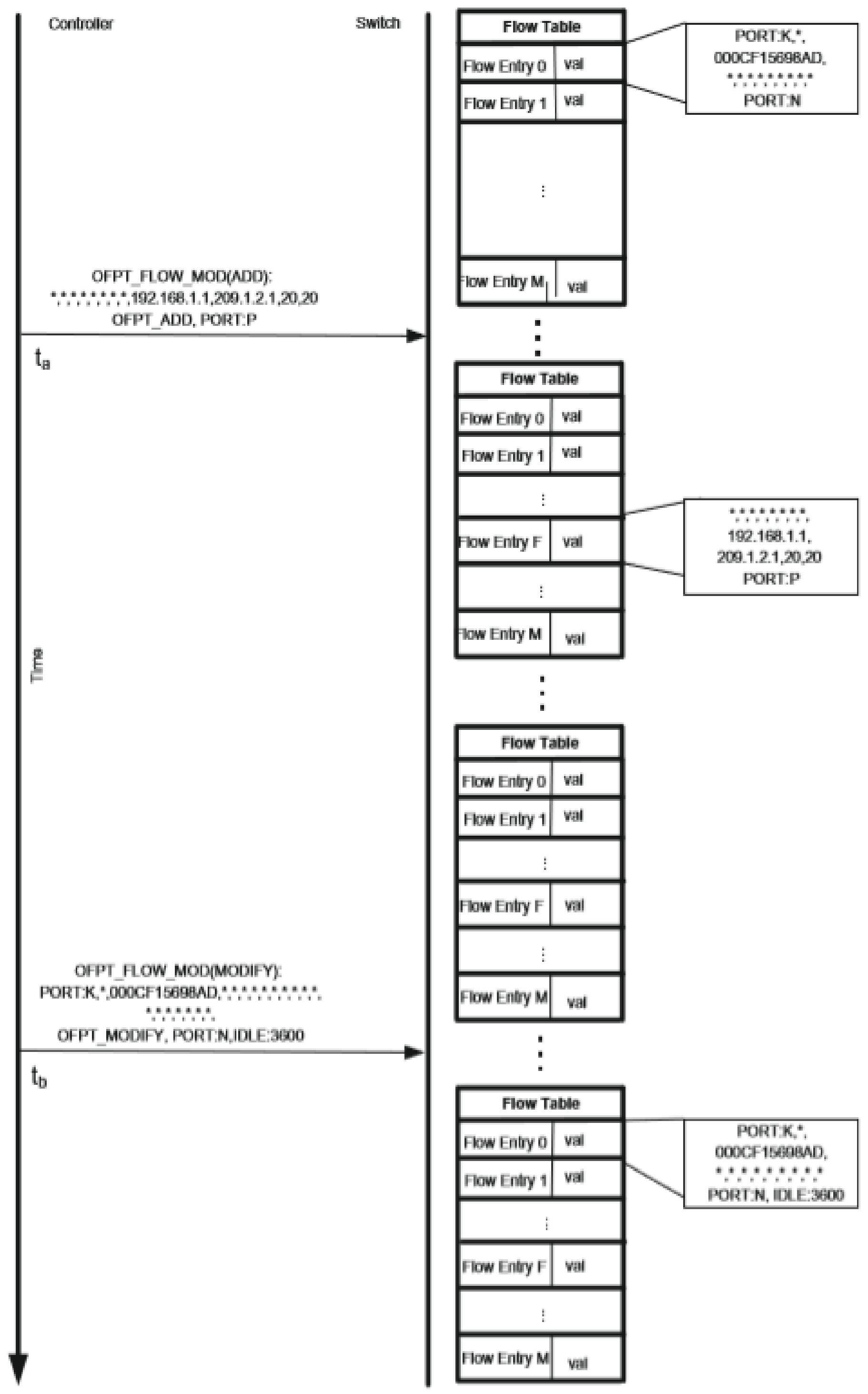
Note that immutable in this context means that the message types will not be changed in future releases of OpenFlow.

Example: Controller Programming Flow Table

In Figure 5.9 two simple OpenFlow V.1.0 flow table modifications are performed by the controller.
We see in the figure that the initial flow table has three flows.
In the quiescent state before time ta, we see an exploded version of flow entry zero. It shows that the flow entry specifies that all Ethernet frames entering the switch on input port K with a destination Ethernet address of 0x000CF15698AD should be output on output port N. All other match fields have been wildcarded, indicated by the asterisks in their respective match fields in Figure 5.9.

At time ta, the controller sends a FLOW_MOD (ADD) command to the switch, adding a flow for packets entering the switch on any port, with source IP addresses 192.168.1.1 and destination IP address 209.1.2.1, source TCP port 20, and destination port 20. All other match fields have been wild carded. The outport port is specified as P.We see that after this controller command is received and processed by the switch, the flow table contains a new flow entry F corresponding to that ADD message.

At time tb, the controller sends a FLOW_MOD (MODIFY) command for flow entry zero. The controller seeks to modify the corresponding flow entry such that there is a one-hour (3600-second) idle time on that flow. The figure shows that after the switch has processed this command, the original flow entry has been modified to reflect that new idle time. Note that idle time for a flow entry means that after that number of seconds of inactivity on that flow, the flow should be deleted by the switch. Looking back at Figure 5.8, we see an example of such a flow expiration just after time td. The FLOW_REMOVED message we see there indicates that the flow programmed at time tb in our examples in Figures 5.8 and 5.9 has expired. This controller had requested to be notified of such expiration when the flow was configured, and the FLOW_REMOVED messages serves this purpose.



Controller
Switch
Flow Table
Flow Entry 0
val
Flow Entry 1
val
Flow Entry M
val
PORT:K,*, 000CF15698AD, *********, PORT:N
OFPT_FLOW_MOD(ADD): *,*,*,*,*,*,*,*,192.168.1.1,209.1.2.1,20,20 OFPT_ADD, PORT:P
t_a
Flow Entry F
,,*,*,*,*,*,*, 192.168.1.1, 209.1.2.1,20,20 PORT:P
Time
OFPT_FLOW_MOD(MODIFY): PORT:K,*,000CF15698AD,*,*,*,*,*,*,*,*,*, *,*,*,*,*,*,*, OFPT_MODIFY, PORT:N,IDLE:3600
t_b
PORT:K,*, 000CF15698AD, *********, PORT:N, IDLE:3600

FIGURE 5.9

Controller programming flow entries in V.1.0.

Example: Basic Packet Forwarding

We illustrate the most basic case of OpenFlow V.1.0 packet forwarding in Figure 5.10. 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

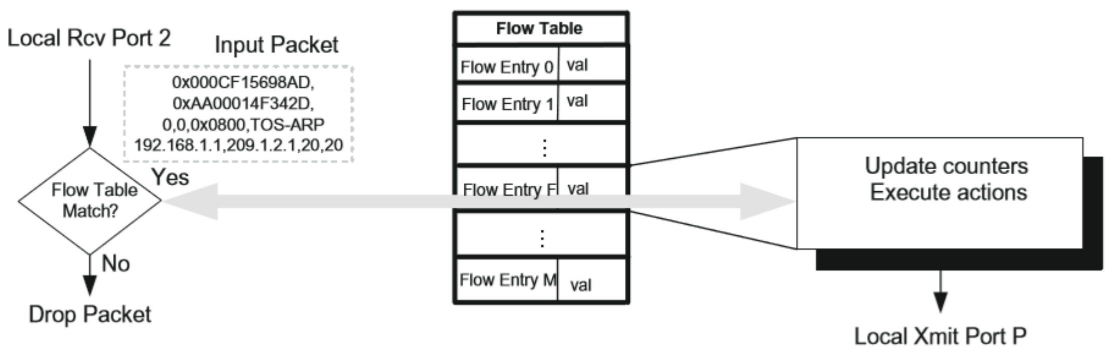


FIGURE 5.10

Packet matching function: basic packet forwarding V.1.0.

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. The semantics of the flow entries allow matching for a wide variety of protocol headers but a given switch will be programmed only for those that correspond to the role it plays in packet forwarding. 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.

Example: Switch Forwarding Packet to Controller

We showed in the last section an example of an OpenFlow switch forwarding an incoming data packet to a specified destination port. Another fundamental action of the OpenFlow V.1.0 switch is to forward 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. Obviously OFPR_NO_MATCH is used when no matching flow entry is found. OpenFlow retains the ability to specify that a particular matching flow entry should always be forwarded to the controller. In this case OFPR_ACTION is specified as the reason. An example is a control packet such as a routing protocol packet that always needs to be processed by the controller. In Figure 5.11 an example of an incoming data packet is shown that is an OSPF routing packet. 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

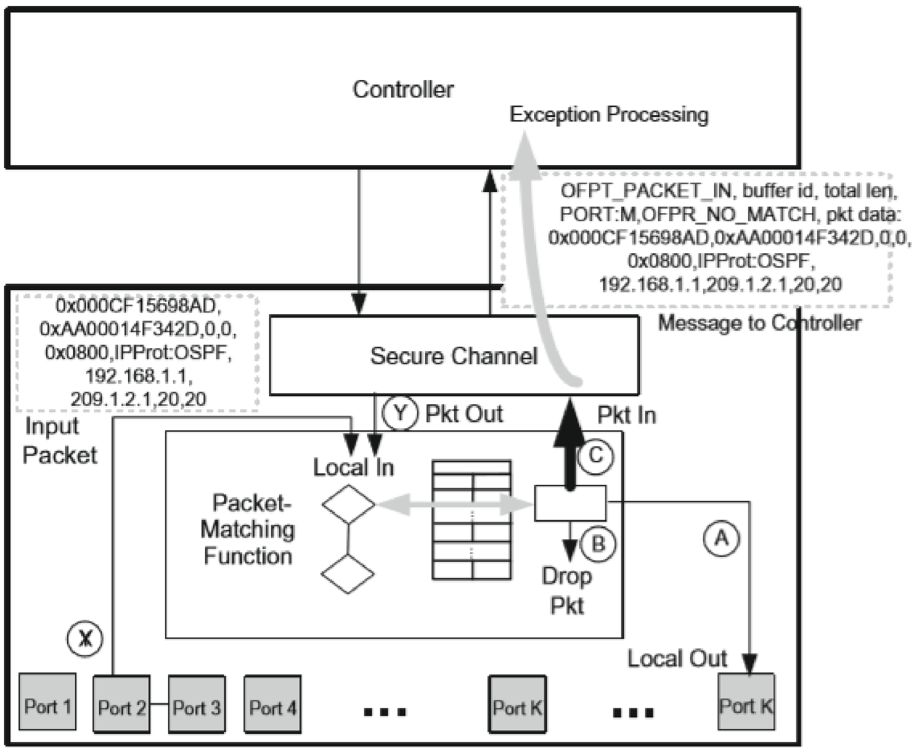

Controller
Exception Processing
OFPT_PACKET_IN, buffer id, total len, PORT:M,OFPR_NO_MATCH, pkt data: 0x000CF15698AD,0xAA00014F342D,0,0, 0x0800,IPProt:OSPF, 192.168.1.1,209.1.2.1,20,20
Message to Controller
0x000CF15698AD, 0xAA00014F342D,0,0, 0x0800,IPProt:OSPF, 192.168.1.1, 209.1.2.1,20,20
Secure Channel
Input Packet
Y
Pkt Out
Pkt In
C
Local In
Packet-Matching Function
B
Drop Pkt
A
X
Local Out
Port 1
Port 2
Port 3
Port 4
Port K
Port K

FIGURE 5.11

Switch forwarding incoming packet to controller.

Table change. At a minimum, the controller needs access to the packet header fields to determine its disposition of the packet. In many cases, though not all, it may need access to the entire packet. This would in fact be true in the case of the OSPF routing packet in this example. In the interest of efficiency, 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: (1) to flood the packet out all ports except the port on which it arrived, or (2) to drop the packet.