RYU- Software-defined Networking Controller

Ryu is a component based, event-triggered software defined networking framework with well-defined APIs, which help developers to develop various network management and control applications seamlessly.

Rapid development of SDN controller based network applications is possible with the help of API provided by Ryu. Ryu's API allows the rapid development of controller application prototypes. Ryu controller code is Open Source under Apache license 2.0 and can be easily accessed for developing network applications.

Ryu controller makes use of OpenFlow (versions 1.0 - 1.5) and uses NetConf, OF-Config, etc., for developing high-level network control applications.

Basically, it is a component of the network that becomes a single point control and management tool. One of the main reason for its popularity is associated to its code readability.

Ryu Setup

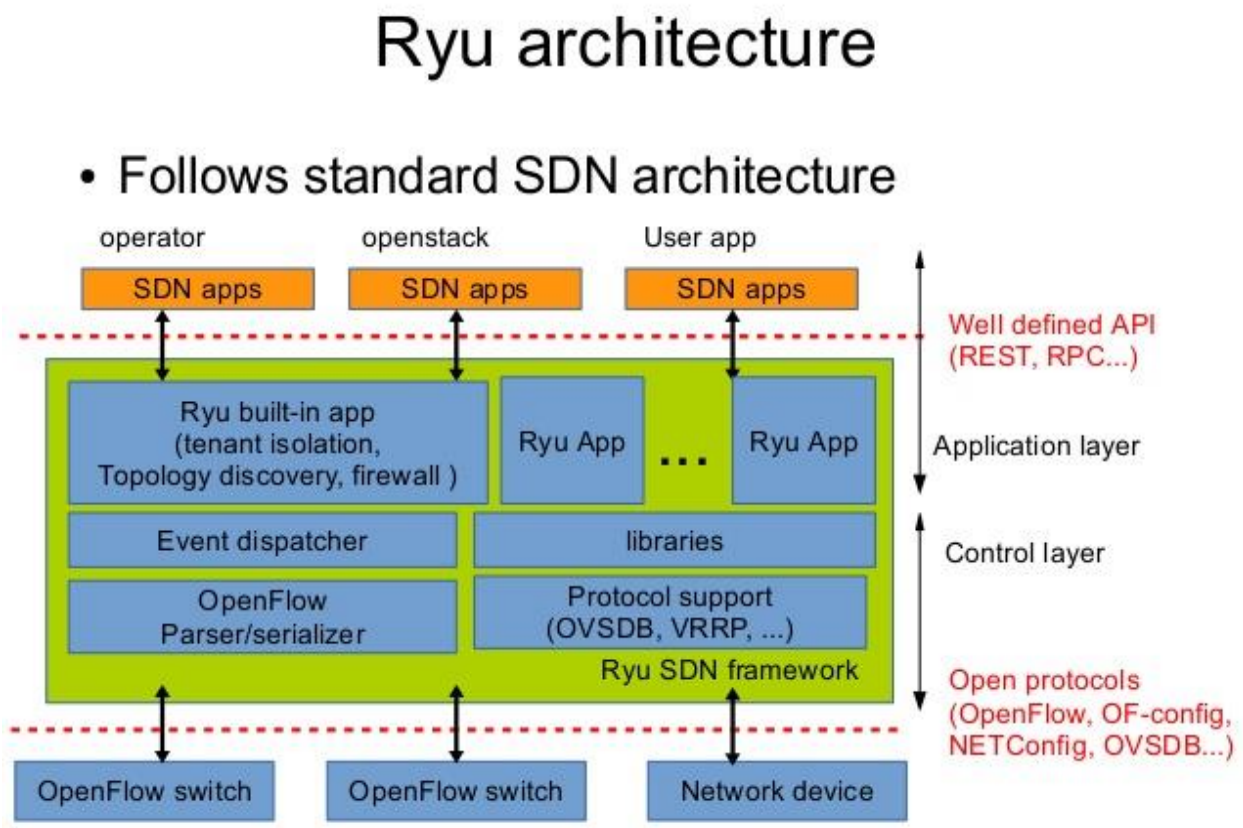
Option1: Setup by using pip:

$ pip install ryu

Option 2: Setup by using git:

$ git clone git://github.com/osrg/ryu.git

$ cd ryu; pip install .



Main components of the Ryu Controller

Before starting with the application development with the Ryu controller, it is best to understand these components:

1. controller/
2. base/
3. lib/
4. ofproto/
5. topology/
6. app/

Controller module

- This module contains set of required files to deal with the OpenFlow functions such as packets from switches, handling network events, etc.

- The controller contains many important files such as controller.py, event.py, ofp_handler.py, dpset.py, etc.
- controller.py is the main component of the OpenFlow Controller and defines the OpenFlowController base class, it handles connections from switches, generate and route events to appropriate entities like Ryu applications. event.py defines the base of all event classes, a Ryu application can define its own event type by creating a subclass. In dpset.py file, many messages, such as port status information, are defined for the switch to describe and operate the switch. Such as add port, delete port and other operations.

- This module contains the required set of files to handle OpenFlow functions (e.g., packets from switches, generating flows, handling network events, gathering statistics etc.).

- It is used to observe events as well as act when an event occurs. Being one of the core modules in the controller, most of the code in the controller module is focused on handling events. This component also happens to have the base class for all the events in the entire controller.

Events:
There are many things of importance in the controller's code, such as the Datapath class, and the handler submodule.

But the controller entirely functions on events, since there is a Packet In event, the flow table in the switch gets populated, since there is a OFP Change Event the parameters of the respective queue in the Datapath are altered and so on.

Events define the flow of control, making RYU a very dynamic environment.

Base module

- This module contains the base class for Ryu applications, the app manager class which resides in app_manager.py is inherited by every Ryu application. It handles the app initialization and communication with the other components of the controller.

  app_manager.py is the management center of RYU applications. The management is basically centered for:
  Loading Ryu applications.
  Providing contexts to Ryu applications.
  Routing messages among the Ryu applications.

- A RYU App is a collection of Contexts, event handlers and event observers. The lifetime of an application is decided by the number of events it must process. Based on that it dispatches the event to the many observers. These observers carry out further computation to manage the network.

- RYU is not just limited to a single application. It can execute multiple applications with different features at the same time. The base module in this case provides context to each of these applications. The context is an integral part of the application and one can understand the purpose of the application from it.

Lib module

- This module contains set of packet libraries to parse different protocol headers and a library for ofconfig. In addition to this it includes the parsers for NetFlow and sFlow.

- It defines the basic data structures such as DPID, MAC, IP, etc. In lib/packet directory we can find many network protocols such as ICMP, DHCP, etc. The class of each packet includes two functions namely parser and serialize which perform parsing and serialization of data packets.

- For example: Take OVS bridge for example, the lib module contains the implementation of an OVS Bridge which allows the developer to handle the QoS parameters specific to that bridge without having to worry about the internal details of how it works.

- The lib module provides functionality over the components of the network.

Ofproto module

- This module contains the OpenFlow protocol specific information and related parsers to support different versions of OpenFlow protocol. Implementations of the various versions and version specific features and events related to the protocol reside here. The details about which features from the specific versions of OpenFlow are incorporated in the controller can be found here.

Topology module

- This Module contains the code for performing topology discovery related to the OpenFlow switches and handles the information about ports, links, etc. associated with the OpenFlow Switches. It internally uses LLDP protocol.

- This module contains code that performs topology discovery related to OpenFlow switches and handles associated information.

- It defines the basic behavior of the ports and switches, the attributes which a port has, the attributes of a switch etc. This module contains another set of events. These events are more specific on the hardware related network events.

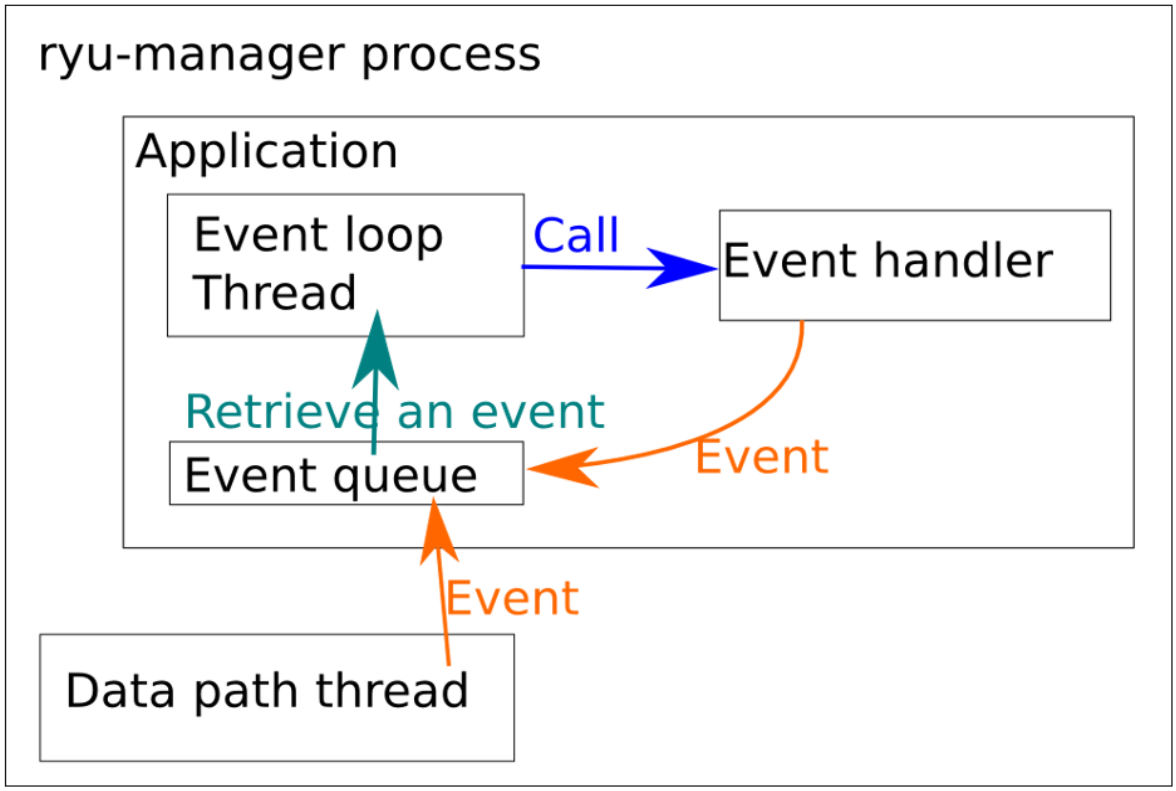
- Though the ports and hosts are logical entities, the basic events associated to these entities are defined here. These events help the developers write custom event handlers to respond to events intelligently.
- A few examples of such events are:
  Link Addition and Deletion
  Switch Enter and Switch exit from the network

App module

- This Module contains the set of applications provided by the Ryu.
- Examples include simple_switch_13.py, simple_monitor.py, rest_qos.py.
  simple_switch_13.py performs basic routing functionalities. simple_monitor.py take note of statistics, the total number of bytes exchanged. rest_qos.py to set and access the queues of the switches, etc.

  The main executable for Ryu code is bin/ryu-manager

Application Driven Ryu Architecture



- Every application on the RYU controller executes an infinite listening loop for all the events that happen across the controller.
- These events trigger event-specific handlers to invoke respective functions.
- For example, an OpenFlow HELLO message would trigger an OFPEvent.

Application

Application is a class which inherits ryu.base.app_manager.RyuApp. User can describe its logic as an application.

Event

Events are class objects that inherit ryu.controller.event.EventBase. Communication between applications is performed by transmitting and receiving events.

Event Queue

Each application has a single queue for receiving events.

Threads

Ryu runs in multi-thread using eventlets.

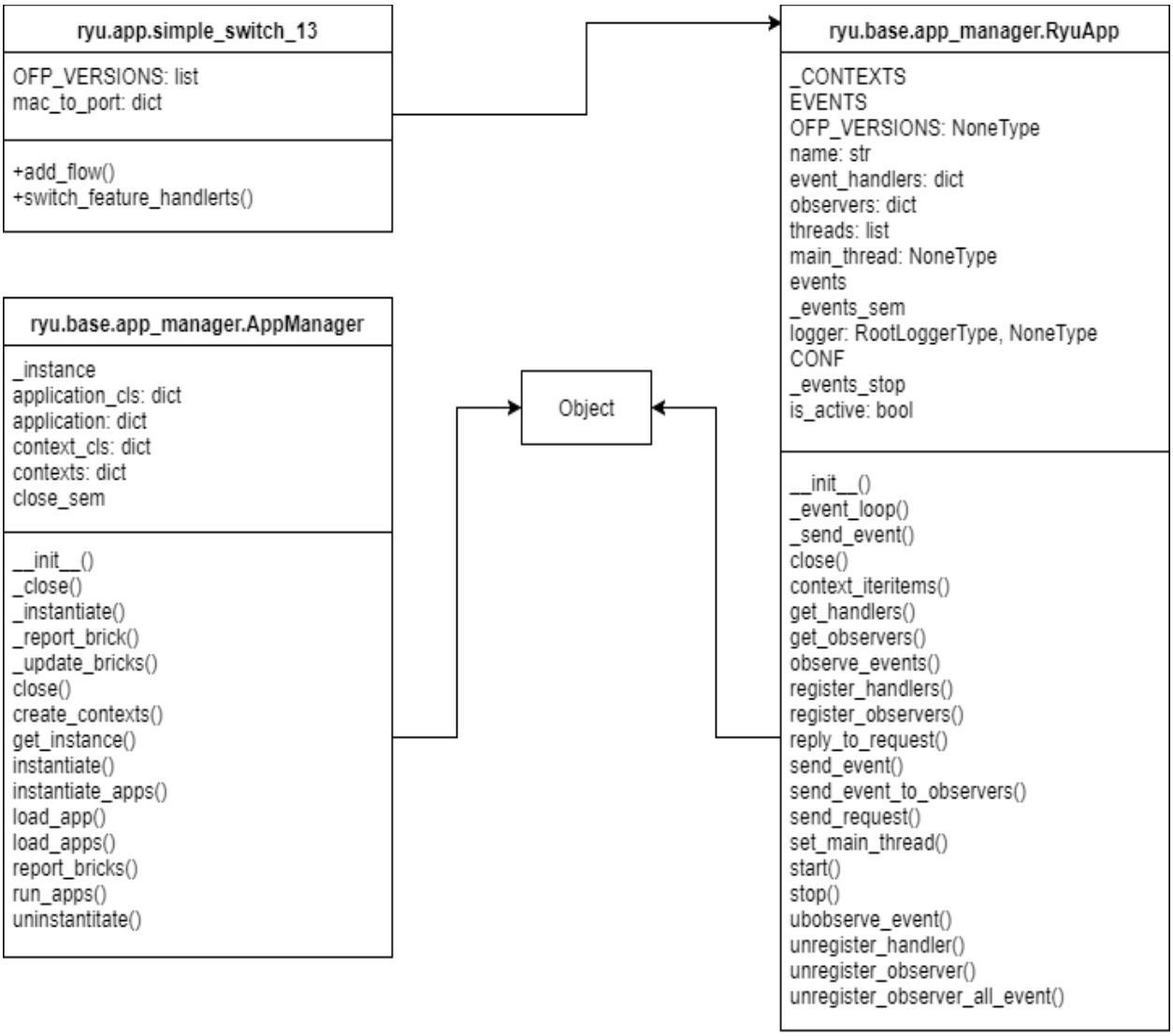
Event Loops

One thread is automatically created for each application. This thread runs an event loop. If there is an event in the event queue, the event loop will load the event and call the corresponding event handler.

Event Handlers

An event handler can be defined by decorating application class method with an ryu.controller.handler.set_ev_cls decorator. When an event of the specified type occurs, the event handler is called from the application's event loop.

UML Diagram for simple_switch_13.py file



Code analysis for simple_switch_13.py file

Required Imports and libraries

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
```

The libraries in use by Simple Switch consist of the following:

ryu – The Ryu package itself. (Module)
base – Provides base classes for Ryu Controller Applications. (Module)
app_manager – Provides the RyuApp base class used by Simple Switch. (Module)

ryu – The Ryu package itself. (Module)
controller - Contains modules for the base Ryu Controller. (Module)
ofp_event - OpenFlow event definitions. (Module)

ryu – The Ryu package itself. (Module)
controller - Contains modules for the base Ryu Controller. (Module)
handler - Event handling tools. (Module)
CONFIG_DISPATCHER - One of the OpenFlow data path states. (Attribute)
MAIN_DISPATCHER - One of the OpenFlow data path states. (Attribute)

ryu – The Ryu package itself. (Module)
controller - Contains modules for the base Ryu Controller. (Module)
handler - Event handling tools. (Module)
set_ev_cls – Method as event handler for the specified class. (Method)

ryu – The Ryu package itself. (Module)
ofproto - OpenFlow protocol definitions and parsers. (Module)
ofproto_v1_3 - Definitions for the 1.3 version of OpenFlow. (Module)

ryu – The Ryu package itself. (Module)
lib - Utility libraries. (Module)

packet - Packet parsing and serializing libraries. (Module)
packet - Utilities for parsing and serializing packets. (Module)

ryu – The Ryu package itself. (Module)
lib - Utility libraries. (Module)
packet - Packet parsing and serializing libraries. (Module)
ethernet - Ethernet protocol utilities. (Module)

ryu – The Ryu package itself. (Module)
lib - Utility libraries. (Module)
packet - Packet parsing and serializing libraries. (Module)
ether_types - Defines several Ethertypes. (Module)

Controller app registration and initialization

Steps involved are:

- Creating the Ryu Application.
- Specifying the OpenFlow protocol version for that application.
- Initializing the internal MAC-to-Port table.

```
class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
```

```
class SimpleSwitch13(app_manager.RyuApp):
```

Here a new class is created, sub classing ryu.base.app_manager.RyuApp, which registers the application and gets instantiated by ryu-manager. Any Ryu controller application must have at least one subclass of RyuApp. When passed to ryu-manager, all RyuApp subclasses imported or defined on the specified module will also be started. Since this Simple Switch controller exists under the Ryu code base at ryu/app/simple_switch_13.py, it may be referred to as ryu.app.simple_switch_13, as the Ryu code base is available in the PYTHONPATH. If you wish to start your controller app via a module name rather than by file name, make sure your module is available in the PYTHONPATH.

```
OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
```

OFP_VERSIONS define a list of supported versions of OpenFlow for the controller application. If this was not specified, the application would be marked as compatible with all versions of OpenFlow that Ryu supports. Since this version of Simple Switch is designed for OpenFlow v1.3, the OFP_VERSION constant from ofproto_v1_3 is provided as the only value in the list. It is a good idea for your applications to support a wide range of OpenFlow protocol versions if possible.

```
def __init__(self, *args, **kwargs):
    super(SimpleSwitch13, self).__init__(*args, **kwargs)
    self.mac_to_port = {}
```

The self.__init__ method follows the standard convention for overriding the initialization of a superclass. Specifically, self.__init__ will accept any number of arguments or keyword arguments, then pass them to the superclass RyuApp.__init__ method to ensure the superclass is initialized properly. Finally, the application-wide MAC-to-Port table (self.mac_to_port) is initialized as an instance variable defined as a dict. This will later be referenced as a two-dimensional dictionary with the first key set to the DPID of the switch in question and the second key set to a MAC address to find or set the associated learned port (dictionary<dictionary<integer>>)

Event Handler for New Switches

The purpose of this code is to have it run any time a switch is added to the controller and install a catch-all (or table-miss) flow entry in the switch, which allows the switch to send packets to the controller.

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # install table-miss flow entry
    #
    # We specify NO BUFFER to max_len of the output action due to
    # OVS bug. At this moment, if we specify a lesser number, e.g.,
    # 128, OVS will send Packet-In with invalid buffer_id and
    # truncated packet data. In that case, we cannot output packets
    # correctly.  The bug has been fixed in OVS v2.1.0.
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                      ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 0, match, actions)
```

Registering an Event Handler for the New Switch:

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
```

The first line is the decorator statement. The decorator @set_ev_cls (..) registers the attached function as an event handler for the specified event type, within the specified dispatcher. In this case, the instance method switch_features_handler (self, ev) will be called any time Ryu processes an event of type EventOFPSwitchFeatures during the CONFIG_DISPATCHER negotiation phase. While a switch connects to the Ryu controller, the switch connection goes through different negotiation phases. During the CONFIG_DISPATCHER negotiation phase, Ryu asks the switch for its features (via OFPFeaturesRequest). The switch will respond with a features reply message, which is interpreted as an OFPSwitchFeatures message. This message is handled by Ryu for its own purposes and emits an EventOFPSwitchFeatures event. In general, any message received by the switch will result in an event that is similarly named. The second line starts the instance method that will be called with the event as ev when the event is triggered.

```
datapath = ev.msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
```

Now the event is passed in as ev (Argument), which is of type EventOFPSwitchFeatures, itself a subclass of EventOFPMsgBase, and references extracted from it. Here is an object property tree for the properties used in this method

ev - Instance of ryu.controller.ofp_event.EventOFPSwitchFeatures (EventOFPMsgBase) - The event itself.

msg – Instance of ryu.ofproto.ofproto_v1_3_parser.OFPSwitchFeatures - The message parsed by Ryu, which triggered the event.

datapath – Instance of ryu.controller.controller.Datapath - Datapath object instance representing the switch that sent the message. Locally referenced as datapath.

ofproto – ryu.ofproto.ofproto_v1_3 - A reference to the definitions library for the version of the OpenFlow protocol used in communicating between the controller and this datapath. Locally referenced as ofproto.

ofproto_parser – ryu.ofproto.ofproto_v1_3_parser - The message parsing library for the version of OpenFlow protocol used here. Locally referenced as parser.

```
match = parser.OFPMatch()
```

To create the flow entry, Simple Switch must first create a match condition and a set of actions that will be followed if those match conditions are met. In this case, the match is an empty parser.OFPMatch instance, which means that the flow entry will match any packet.

```
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                  ofproto.OFPCML_NO_BUFFER)]
```

Now the actions are defined as a list containing a parser.OFPActionOutput, which is used to output the matched packet out a specific port, in this case ofproto.OFPP_CONTROLLER. This instructs the switch to send the match packet to the controller via a Packet-In message that will be handled as ofp_event.EventOFPPacketIn, later in the controller code.

The ofproto.OFPCML_NO_BUFFER flag instructs the switch to not buffer the packet, but instead always send the entire packet to the controller. There are other modes of operation that might be more efficient if the switch supports it. In this case, this was added due to a bug in Open vSwitch that was later fixed.

```
self.add_flow(datapath, 0, match, actions)
```

The event handler asks for a flow entry to be added to the switch through the helper instance method self.add_flow. In this case, it instructs a flow entry to be added to the datapath (switch) with a priority of 0. The flow entry will also include the match and actions elements.

Helper method for adding flow entry

As most of the flow entries that will be added are going to have similar structure, a helper method is defined to construct and send the final flow entry.

```
def add_flow(self, datapath, priority, match, actions, buffer_id=None):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                         actions)]
    if buffer_id:
        mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                priority=priority, match=match,
                                instructions=inst)
    else:
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)
    datapath.send_msg(mod)
```

Defining the Helper function:

```
def add_flow(self, datapath, priority, match, actions, buffer_id=None):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
```

self.switch_features_handler called a helper instance method self.add_flow. This method adds flow entries with predefined options to reduce the code required elsewhere in the controller application.

self.add function takes the below arguments:

datapath - The ryu.controller.datapath.Datapath, instance to add the flow entry to.
priority - An integer representing the priority that the flow entry should be added with.
match - An parser.OFPMatch instance containing the match conditions for the flow entry.
actions - A list of parser.OFPAction subclassed entities that will be sent along with the flow mod.
buffer_id - An optional integer that points to a buffered packet on the switch to immediately apply the flow entry to.

Additionally, the ofproto and ofproto_parser are extracted from the datapath and assigned to ofproto and parser respectively. As before, these are extracted from the datapath so the appropriate version of OpenFlow is used while constructing the OFPFlowMod message.

Constructing the Instruction:

```
inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                     actions)]
```

As this is working with OpenFlow above v1.2, a flow-mod must include a set of instructions rather than just actions. This is due to the additional abilities added such as writing metadata or, more commonly, instructing the switch to continue processing the packet on a different table. OpenFlow v1.3 adds an additional instruction for applying a meter.

Here, the inst variable is set to a list with a single parser.OFPInstructionActions instance set to ofproto.OFPIT_APPLY_ACTIONS provided in actions.

Constructing the Flow mod:

```
if buffer_id:
    mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                            priority=priority, match=match,
                            instructions=inst)
else:
    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                            match=match, instructions=inst)
```

Depending on if buffer_id was provided, a parser.OFPFlowMod is instantiated with or without that buffer_id.

In either case, it is passed with the same datapath passed to the self.add_flow method and with the same priority and match arguments. The only customized argument is for the flow instructions passed in as inst, which was created above. The result is the final parser.OFPFlowMod message.

Sending the Flow mod:

```
datapath.send_msg(mod)
```

Sending the flow mod is as easy as passing it to the datapath.send_msg method. Ryu will take care of the rest to ensure the message is properly encoded and sent to the switch.

Packet-In Handler and Packet dissection

This section starts with the main logic in the controller application. A handler is defined for ryu.controller.ofp_event.EventOFPPacketIn, which is called any time the switch sends a packet to the controller. This only occurs if the switch doesn't already know where to send the packet and the table-miss flow entry is matched. The first part of the handler extracts vital information about the message and the packet sent to the controller.

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    # If you hit this you might want to increase
    # the "miss_send_length" of your switch
    if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug("packet truncated: only %s of %s bytes",
                          ev.msg.msg_len, ev.msg.total_len)
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    if eth.ethertype == ether_types.ETH_TYPE_LLDP:
        # ignore lldp packet
        return
```

Packet In Handler:

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
```

As with the self.switch_features_handler, the self._packet_in_handler is registered to be called any time a certain event is fired. In this case, the @set_ev_cls decorator is applied to register this method to be called on any ofp_event.EventOFPPacketIn event is fired in the MAIN_DISPATCHER.

As before, this method is called with the current event as ev. The following attributes of ev are used directly in this method

ev - instance of ryu.controller.ofp_event.EventOFPPacketIn - The event fired.

msg – instance of ryu.ofproto.ofproto_v1_3_parser.OFPPacketIn - The message that triggered the event.

buffer_id - an integer identifying the buffer on the switch that holds the packet, if the packet was buffered.

data – raw bytearray containing the data portion of the message.

datapath – Instance of ryu.controller.controller.Datapath that represents the switch.

id - 64-bit int of the datapath ID.

ofproto – ryu.ofproto.ofproto_v1_3src, reference to the currently used OpenFlow protocol definition module.

OFP_NO_BUFFER - integer, special buffer ID to indicate “no buffer”.

OFPP_FLOOD - integer, special port no. to flood using non-OpenFlow pipeline.

ofproto_parser - ryu.ofproto.ofproto_v1_3_parser, reference to the currently used OpenFlow protocol parser module.

OFPActionOutput - creates an action message segment to indicate a packet should be output on a specific port.

OFPMatch - creates a match message segment to indicate a packet should be matched on the specified parameters.

send_msg - ends the specified message to this datapath.

match - dictionary, dictionary of key, values provided in the match message portion.

msg_len - integer, length of the message received by the switch.

total_len - integer, total length of the message, may be larger than msg_len if the message was truncated.

Sanity Checking for Message Length:

```
if ev.msg.msg_len < ev.msg.total_len:
    self.logger.debug("packet truncated: only %s of %s bytes",
                      ev.msg.msg_len, ev.msg.total_len)
```

The method starts of by checking the message length to ensure that the entire message was received by the switch, otherwise a log message is written explaining that the packet was truncated, which may cause issues routing the package.

Pulling Important Data:

```
msg = ev.msg
datapath = msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
in_port = msg.match['in_port']
```

Several attributes are assigned to local variables for more compact code. Of interest is the in_port variable, which is set to the value retrieved from msg.match, a dict, with the key of in_port. This provides the physical port number where the packet was received on the switch that sent the packet-in message. It is used later to learn the location of devices on the network.

Parsing the Submitted Packet:

```
pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]
```

Since this is an L2 learning switch, the devices on the network will be referred to by their L2 addresses. The pkt variable is set to an instance of packet. Packet, which is passed the msg.data portion of the OpenFlow message. This class instantiation automatically parses the first header in the data as an ethernet frame. The next line sets eth to the first ethernet frame by first calling pkt.get_protocols with ethernet.ethernet as the argument. The return value of that call is a list of ethernet headers in that data. Since the L2 learning code is only interested in the outermost ethernet frame, the first value of the returned list is referenced when setting eth. The following is a list of attributes used from the decoded packet

eth - instance of ryu.lib.packet.ethernet.ethernet - The decoded ethernet frame
dst - string – string representation of the destination MAC, like 'ff:ff:ff:ff:ff:ff'
ethertype - integer - The 16-bit ethertype of the packet
src – string - string representation of the source MAC, like '00:00:00:00:00:01'

Ignoring LLDP Packets:

```
if eth.ethertype == ether_types.ETH_TYPE_LLDP:
    # ignore lldp packet
    return
```

Simple Switch ignores packets using the Link Layer Discovery Protocol (LLDP) and this code effectively stops processing of packet if eth.ethertype matches ether_types.ETH_TYPE_LLDP.

This prevents the forwarding of LLDP traffic as only the controller is allowed to flood packets and processing is stopped before that action can be given.

Learning the MAC address and Associated port:

Once the essentials are extracted, the MAC-to-Port table for the DPID of the current switch is created if it does not already exist. The packet information is logged and the MAC-to-Port table is finally updated with the source address of the packet associated with the port it arrived on.

```
dst = eth.dst
src = eth.src

dpid = datapath.id
self.mac_to_port.setdefault(dpid, {})
```

The dst, src, and dpid variables are set to their appropriate values and the application's MAC-to-Port table is prepared to learn on the current switch, if it was not already set up before.

```
self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)
```

The vital information for the packet being processed is logged to help with debugging both the controller application and the topology.

```
self.mac_to_port[dpid][src] = in_port
```

The self.mac_to_port (dictionary <dictionary<integer>>) is updated by using the dpid and src variables as keys and setting it to the in_port.

Mac-to-Port and Packet Destination Lookup:

```
    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD
```

The same table is then used to look up the physical port assigned to the destination MAC address. This check is performed by asking if the dst is in the self.mac_to_port[dpid] keys. If there is a hit, the out_port is set to the physical port at that location in the table. Otherwise, the out_port is set to ofproto.OFPP_FLOOD.

```
actions = [parser.OFPActionOutput(out_port)]
```

Actions are set to a list with an instance of parser.OFPActionOutput given out_port as the port to send the packet to.

Adding a Flow Entry for a Learned Destination:

```
if out_port != ofproto.OFPP_FLOOD:
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_src=src)
```

If the destination MAC address is known, the controller application needs to add a new flow entry to the switch so the packet can be directly forwarded rather than depending on the controller application to forward it, which is a much slower path.

A check is performed to ensure that the out_port is not set to ofproto.OFPP_FLOOD, which means the destination physical port is known. The match conditions for a new flow are specified by setting match to an instance of parser.OFPMatch with arguments as in_port, eth_dst and eth_src.

```
if msg.buffer_id != ofproto.OFP_NO_BUFFER:
    self.add_flow(datapath, 1, match, actions, msg.buffer_id)
    return
```

If the packet-in message contained a buffer ID, then the switch has buffered the packet and is waiting for the controller to tell it where to send it. This means that the flow entry must be added while referencing that buffer, causing the switch to immediately forward the buffered packet while the flow entry is added to the switch. A priority is set to a higher number than the table-miss flow entry so this new learned-path flow entry will be processed before the table-miss flow entry. Since the packet was buffered and the switch was told to forward the buffered packet using the flow added, the method returns immediately.

```
else:
    self.add_flow(datapath, 1, match, actions)
```

If the packet was not buffered, the flow entry is still added, but does not reference a buffer. This simply adds the flow entry to the switch and new matching packets will be forwarded directly. The original packet must still be sent, however.

<u>Forwarding Packet received by the controller:</u>

```
data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
    data = msg.data
```

If the code execution has gotten this far, that means that either the destination port is unknown or a buffer was not specified in the packet-in message. In either case, the original packet must be sent somewhere.

The data argument for the later parser.OFPPacketOut call is prepared by setting data to None. If the message did not reference a buffer, then the switch did not buffer the packet so the data in the packet-in message sent to the controller must be supplied in the packet-out message.

```
out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                          in_port=in_port, actions=actions, data=data)
datapath.send_msg(out)
```

The parser.OFPPacketOut message is prepared with the variables prepared so far. The in_port argument is provided so the switch knows what port the packet should not be forwarded to, if the out_port is set to OFPP_FLOOD. The actions prepared are used both for this packet-out message and for the added flow entry, if one was added.
Finally, now that the destination set to flood or a learned port, the switch is instructed to send out the packet that was received by the controller so the packet is not lost during the learning process.

Writing Ryu Application

Writing a Ryu application is as good as writing a script in python. Please follow the guide http://ryu.readthedocs.io/en/latest/writing_ryu_app.html for ryu application.

Test Environment

SDN Hub provides a VM which has inbuilt SDN controllers like POX, Ryu, Floodlight, etc. Make sure you update the Ryu controller provided by SDN hub or remove the existing Ryu controller and clone the latest Ryu repository from the GitHub with the steps as given below

SDN Hub URL

http://sdnhub.org/releases/sdn-starter-kit-ryu/

Ryu Development Community

Send Ryu-devel mailing list submissions to
ryu-devel@lists.sourceforge.net

To subscribe or unsubscribe via the World Wide Web, visit
https://lists.sourceforge.net/lists/listinfo/ryu-devel

or, via email, send a message with subject or body 'help' to
ryu-devel-request@lists.sourceforge.net

You can reach the person managing the list at
ryu-devel-owner@lists.sourceforge.net

References

Ryu application API. (n.d.). Retrieved September 10, 2017, from http://ryu.readthedocs.io/en/latest/ryu_app_api.html

The First Application. (n.d.). Retrieved August 2, 2017, from http://ryu.readthedocs.io/en/latest/writing_ryu_app.html

(n.d.). Retrieved June 18, 2017, from https://osrg.github.io/ryu-book/en/Ryubook.pdf

Goswami, K. (n.d.). Slides on Ryu

O. (2016, June 30). Osrg/ryu. Retrieved September 19, 2017, from https://github.com/osrg/ryu/tree/v4.4

Understanding the Ryu API: Dissecting Simple Switch. (2016, October 12). Retrieved May 14, 2017, from https://inside-openflow.com/2016/07/21/ryu-api-dissecting-simple-switch/