RYU Controller Tutorial

This tutorial is intended for beginners to SDN application development for the RYU platform from NTT. RYU has support for several versions of OpenFlow, including OpenFlow versions 1.0 and 1.3 that have seen wide spread support from vendors.

For this tutorial, some Python knowledge will be useful, though it isn't absolutely necessary. This tutorial will help understand RYU's internals, steps to build a new application on top of RYU and get introduced to controller programming.

1. Quickstart

- Run Mininet on a terminal window using the following command. This starts a network emulation environment to emulate 1 switch with 3 hosts.

```
$ sudo mn --topo single,3 --mac --controller remote --switch ovsk
```

- The above command will spawn 1 switch that has support for both OpenFlow ver 1.0 and 1.3. Based on your needs, however, you can force a switch to support OpenFlow 1.3 by executing this command:

```
$ sudo ovs-vsctl set bridge s1 protocols=OpenFlow13
```

- The wireshark that is part of the VM can parse OpenFlow 1.3 messages. To start wireshark and view OpenFlow messages:

```
sudo wireshark &
```

- Next, start the RYU Controller. Assume that the main folder where ryu is installed is in /home/ubuntu/ryu, The below command starts the controller by initiating the OpenFlow Protocol Handler and Simple Switch 1.3 application.
  - Since the switch supports OpenFlow 1.0 and 1.3, while the application only supports 1.3, the system will auto-negotiate and choose to proceed will OpenFlow 1.3.

```
$ cd /home/ubuntu/ryu && ./bin/ryu-manager --verbose ryu/app/simple_switch_13.py
loading app ryu/app/simple_switch_13.py
loading app ryu.controller.ofp_handler
instantiating app ryu/app/simple_switch_13.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK SimpleSwitch13
    CONSUMES EventOFPPacketIn
    CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
    PROVIDES EventOFPPacketIn TO {'SimpleSwitch13': set(['main'])}
    PROVIDES EventOFPSwitchFeatures TO {'SimpleSwitch13': set(['config'])}
    CONSUMES EventOFPErrorMsg
    CONSUMES EventOFPSwitchFeatures
    CONSUMES EventOFPPortDescStatsReply
    CONSUMES EventOFPEchoRequest
    CONSUMES EventOFPHello
```

- Ensure that we start the correct ryu-manager if multiple versions of the controller is installed in the same system.
- Next, check if the hosts in the mininet topology can reach each other

```
mininet> h1 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_req=1 ttl=64 time=2.76 ms
64 bytes from 10.0.0.3: icmp_req=2 ttl=64 time=0.052 ms
64 bytes from 10.0.0.3: icmp_req=3 ttl=64 time=0.051 ms
```

2. RYU Code Structure

The main controller code is organized under the /ryu/ folder (In our VM – /home/ubuntu/ryu/ryu/). Here we discuss the functionalities of the key components. It is important to become familiar with them.

- **app**/ – Contains set of applications that run on-top of the controller.
- **base**/ – Contains the base class for RYU applications. The RyuApp class in the app_manager.py file is inherited when creating a new application.

- **controller**/ – Contains the required set of files to handle OpenFlow functions (e.g., packets from switches, generating flows, handling network events, gathering statistics etc).
- **lib**/ – Contains set of packet libraries to parse different protocol headers and a library for OFConfig. In addition, it includes parsers for Netflow and sFlow too.
- **ofproto**/ – Contains the OpenFlow protocol specific information and related parsers to support different versions of OF protocol (1.0, 1.2, 1.3, 1.4)
- **topology**/: Contains code that performs topology discovery related to OpenFlow switches and handles associated information (e.g., ports, links etc). Internally uses LLDP protocol.

3. RYU Controller Code Essentials

Most controller platforms expose some native features to allow these key features:

- Ability to listen to asynchronous events (e.g., PACKET_IN, FLOW_REMOVED) and to observe events using ryu.controller.handler.set_ev_cls decorator.
- Ability to parse incoming packets (e.g., ARP, ICMP, TCP) and fabricate packets to send out into the network
- Ability to create and send an OpenFlow/SDN message (e.g., PACKET_OUT, FLOW_MOD, STATS_REQUEST) to the programmable dataplane.

With RYU you can achieve all of those by invoking set of applications to handle network events, parse any switch request and react to network changes by installing new flows, if required. For instance, creating a new a application involves creating a subclass of RyuApp and building the required logic to listen for network events.

```
from ryu.base import app_manager

class L2Forwarding(app_manager.RyuApp):
    def __init__(self, *args, **kwargs):
        super(L2Forwarding, self).__init__(*args, **kwargs)
```

While the above code represents a valid RYU application, it doesn't have the logic to handle network events from OpenFlow switches. Next, to allow an application to receive packets sent by the switch to the controller, the class needs to implement a method which is decorated by EventOFPPacketIn.

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

The first argument of the decorator calls this function everytime a *packet_in* message is received. The second argument indicates the switch state. Any information about the switch and the protocol version supported by the switch can be deciphered using the following:

```
    msg = ev.msg               # Object representing a packet_in data structure.
    datapath = msg.datapath    # Switch Datapath ID
    ofproto = datapath.ofproto # OpenFlow Protocol version the entities negotiated. In our case OF1.3
```

Once the packet is received, you can decode the packet by importing the packet library under /ryu/lib:

```
    from ryu.lib.packet import packet
    from ryu.lib.packet import ethernet
```

We can inspect the packet headers for several packet types: ARP, Ethernet, ICMP, IPv4, IPv6, MPLS, OSPF, LLDP, TCP, UDP.

```
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)
```

I use the following two useful commands to extract Ether header details:

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

Similarly, the OFPPacketOut class can be used to build a packet_out message with the required information (e.g., Datapath ID, associated actions etc)

```
    out = ofp_parser.OFPPacketOut(datapath=dp,in_port=msg.in_port,actions=actions)#Generate the message
    dp.send_msg(out) #Send the message to the switch
```

Besides a PACKET_OUT, we can also perform a FLOW_MOD insertion into a switch. For this, we build the Match, Action, Instructions and generate the required Flow. Here is an example of how to create a match header where the in_port and eth_dst matches are extracted from the PACKET_IN:

```
    msg = ev.msg
    in_port = msg.match['in_port']
    # Get the destination ethernet address
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)
    dst = eth.dst
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
```

There are several other fields you can match. Here is an example of creating an action list for the flow.

```
    actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)] # Build the required action
```

OpenFlow 1.3 associates set of instructions with each flow entry such as handling actions to modify/forward packets, support pipeline processing instructions in the case of multi-table, metering instructions to rate-limit traffic etc. The previous set of actions defined in OpenFlow 1.0 are one type of instructions defined in OpenFlow 1.3.

Once the match rule and action list is formed, instructions are created as follows:

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

Given the above code, a Flow can be generated and added to a particular switch.

```
    mod = parser.OFPFlowMod(datapath=datapath, priority=0, match=match, instructions=inst)
    datapath.send_msg(mod)
```

4. Sample application: MAC Hub or Learning Switch

For the purposes of this tutorial, you should attempt to build a hub and/or a MAC learning switch using the above code snippets. For the main logic for hub and learning switch refer to **Pseudo-code.pdf**.

The implementation of the *simple_switch_13.py* application can be found under */usr/local/lib/python2.7/dist-packages/ryu/app*. Once you run this with the controller (as explained in the **QuickStart** section), you will see that mininet ping succeeds:

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
From 10.0.0.1 icmp_seq=4 Destination Host Unreachable
64 bytes from 10.0.0.2: icmp_req=5 ttl=64 time=0.529 ms
64 bytes from 10.0.0.2: icmp_req=6 ttl=64 time=0.133 ms
64 bytes from 10.0.0.2: icmp_req=7 ttl=64 time=0.047 ms
```

Contributor:

Sriram Natarajan, RYU Code Contributor
Any questions, email: natarajan(dot)sriram(at)gmail(dot)com