Cloud and Datacenter Networking

Università degli Studi di Napoli Federico II
Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione DIETI
Laurea Magistrale in Ingegneria Informatica

Prof. Roberto Canonico

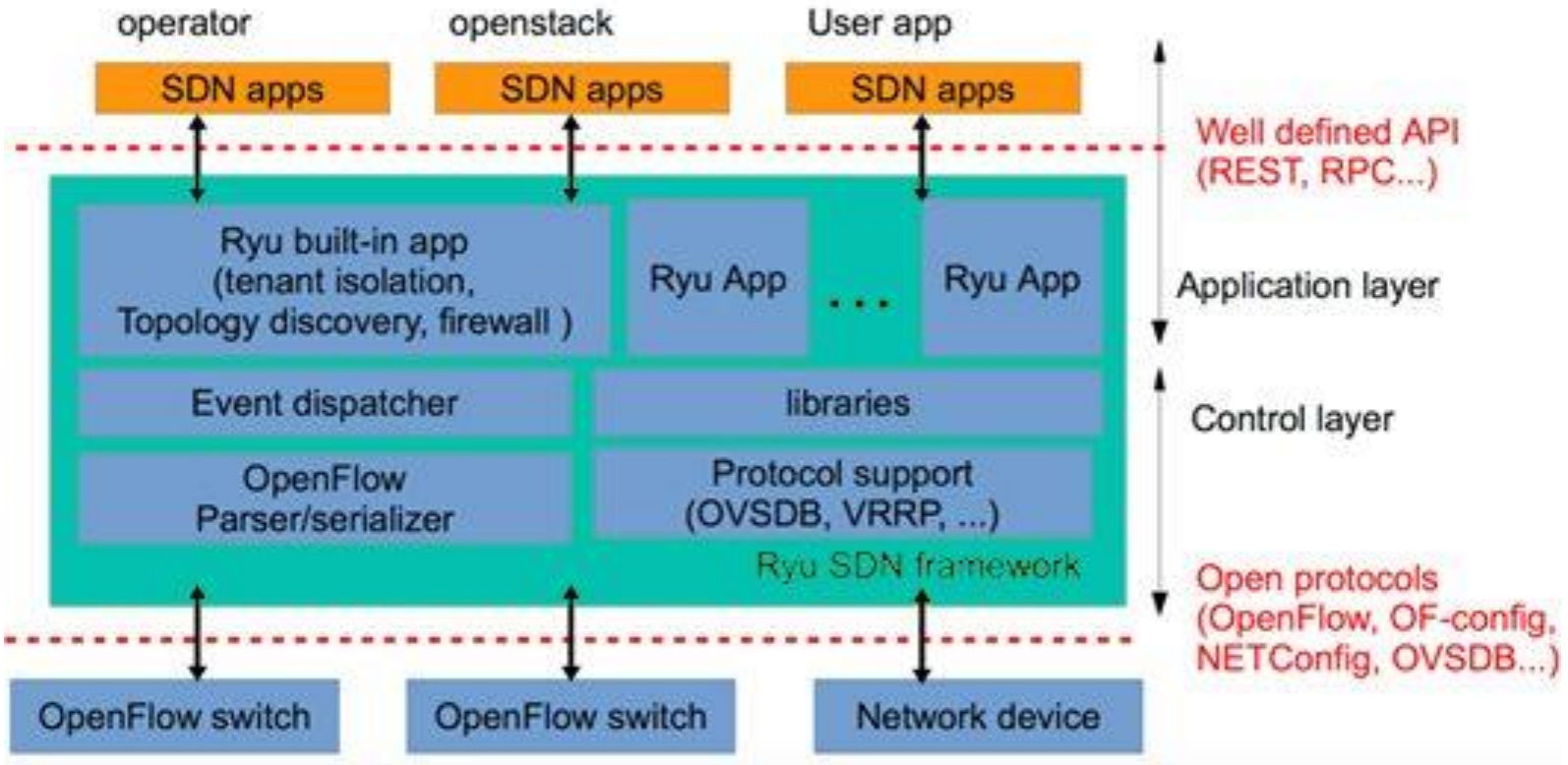
SDN controllers

The Ryu controller

- Ryu is a Python based controller
- More precisely, Ryu is a component-based SDN framework
- It provides software components with well defined API that make it easy for developers to create new network management and control applications
- Southbound: it supports multiple southbound protocols for managing devices, such as OpenFlow, NETCONF, OF-Config, and partial support of P4
- As most controller platforms, Ryu natively implements some basic features:
  - Ability to listen to asynchronous events (e.g., PACKET_IN, FLOW_REMOVED)
  - Ability to parse incoming packets 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

Ryu: an architectural view

- Southbound interfaces allow communication of SDN switches and controllers
- Ryu core supports basic functionalities (e.g. topology discovery, learning switch)
- External applications can deploy network policies to data planes via well-defined northbound APIs such as REST



Starting Ryu

- Ryu is executed by specifying a Python controller script as an argument
- A basic controller functionality is implemented in the `simple_switch_13.py` module:

```
$ ryu-manager --verbose ryu/app/simple_switch_13.py
```

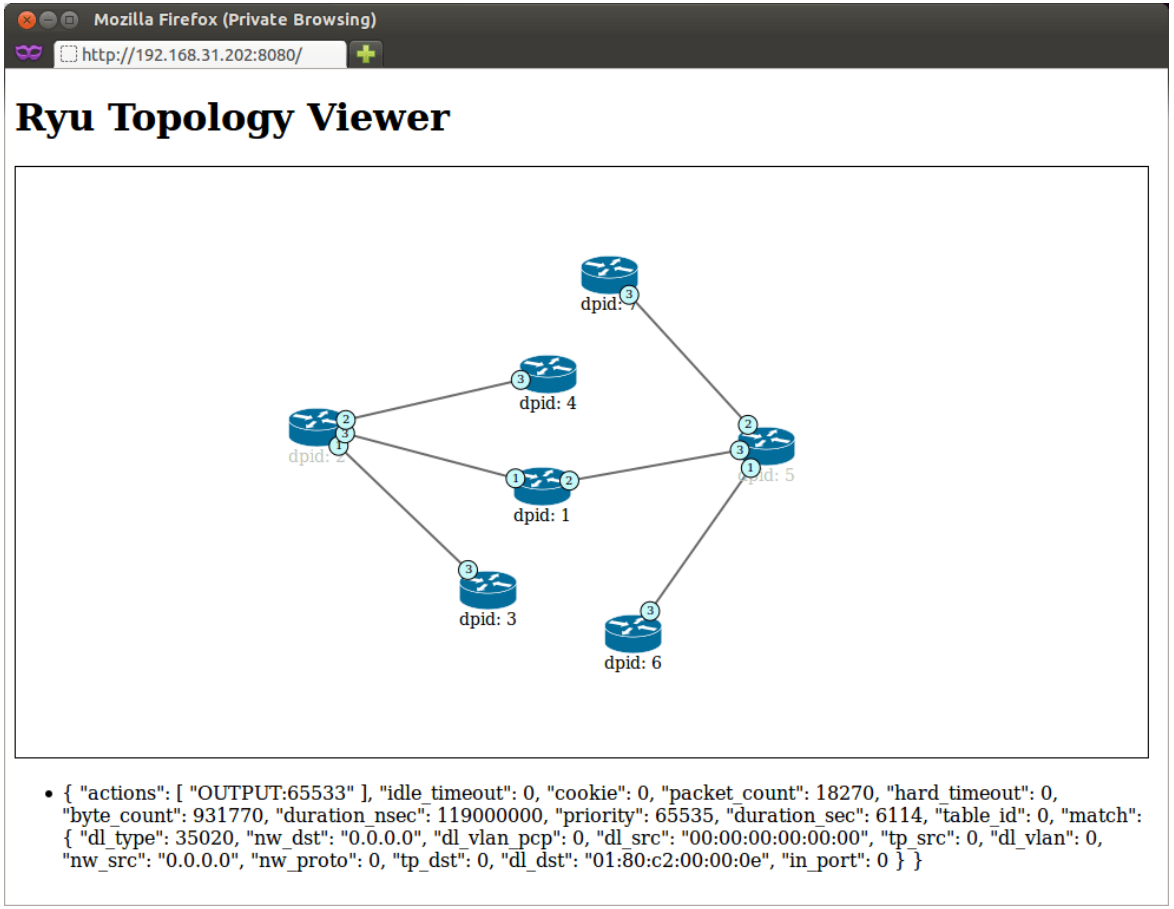
- The module implements the functionality of a learning switch in a set of OpenFlow 1.3 switches

Starting Ryu with a GUI module

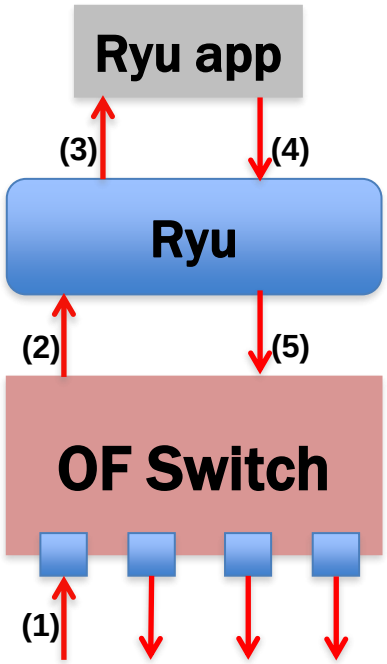
- Ryu provides a minimal web-based GUI that shows a graphical view of the network topology
- To start the GUI:

```
$ ryu-manager --verbose --observe-links
    ryu/app/gui_topology/gui_topology
    ryu/app/simple_switch_13.py
```

- The GUI is accessible with a browser on port 8080 of the host running the controller `http://controller_IP:8080`
- The GUI shows flow rules in each of the switches



Controller logic in Ryu: learning switch (1/3)



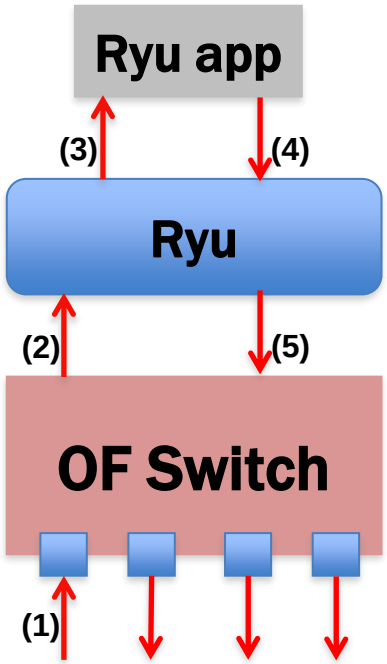
ryu/app/simple_switch_13.py

```
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

class ExampleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(ExampleSwitch13, self).__init__(*args, **kwargs)
        # initialize mac address table.
        self.mac_to_port = {}
    def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        # construct flow_mod message and send it.
        inst=[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)
        datapath.send_msg(mod)
        ...
```

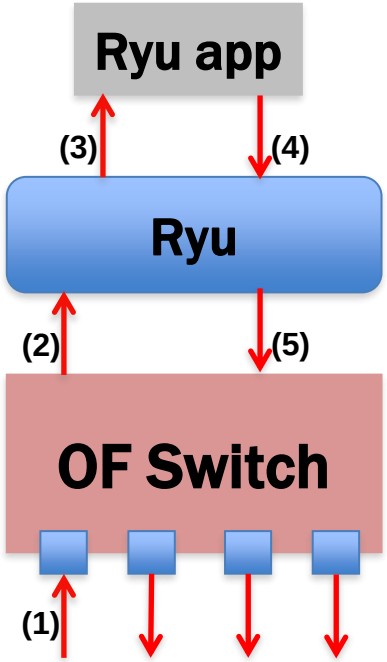
Controller logic in Ryu: learning switch (2/3)



ryu/app/simple_switch_13.py

```
...
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    # get Datapath ID to identify OpenFlow switches.
    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})
    # analyse the received packets using the packet library.
    pkt = packet.Packet(msg.data)
    eth_pkt = pkt.get_protocol(ethernet.ethernet)
    dst = eth_pkt.dst
    src = eth_pkt.src
    # get the received port number from packet_in message.
    in_port = msg.match['in_port']
    self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)
    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port
    # if the destination mac address is already learned,
    # decide which port to output the packet, otherwise FLOOD.
    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD
    ...
```

Controller logic in Ryu: learning switch (3/3)



ryu/app/simple_switch_13.py

```
        ...
        @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
        def _packet_in_handler(self, ev):
            ...
            ...
            ...
            ...
            # construct action list.
            actions = [parser.OFPActionOutput(out_port)]
            # install a flow to avoid packet_in next time.
            if out_port != ofproto.OFPP_FLOOD:
                match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
                self.add_flow(datapath, 1, match, actions)
            # construct packet_out message and send it.
            out = parser.OFPPacketOut(datapath=datapath,
                                      buffer_id=ofproto.OFP_NO_BUFFER,
                                      in_port=in_port, actions=actions,
                                      data=msg.data)
            datapath.send_msg(out)
```

Handling of events in Ryu

- The line

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
```

- before the `_packet_in_handler` function definition is a Python *decorator*
- Decorators are sort of wrapper functions (defined elsewhere in the code) that are executed when a function is invoked
- The `set_ev_cls` decorator in Ryu is used to register a handler function as associated to a specific event
- In particular, the decorator shown above is used to associate the function `_packet_in_handler` to the reception of OpenFlow PACKET_IN messages
- Each Ryu application has a first-in/first-out queue for handling events by preserving their order
- A Ryu application is single-threaded
- In Ryu, an OpenFlow message of type `<name>` is associated to an event that is instance of the class `ryu.controller.opf_event.EventOF<name>`

The Faucet controller

- Faucet is an open-source lightweight SDN Controller built on top of Ryu
  - developed by New Zeeland Research and Education (REANNZ), Waikato University and Victoria University
  - Written in Python with Apache 2 License

- Faucet supports:
  - OpenFlow v1.3 (multi table) switches (including optional table features),
  - Multiple datapaths
  - VLANs, mixed tagged/untagged ports
  - ACLs matching layer 2 and layer 3 fields
  - IPv4 and IPv6 routing, static and via BGP
  - Port statistics
  - Coexisting with other OpenFlow controllers

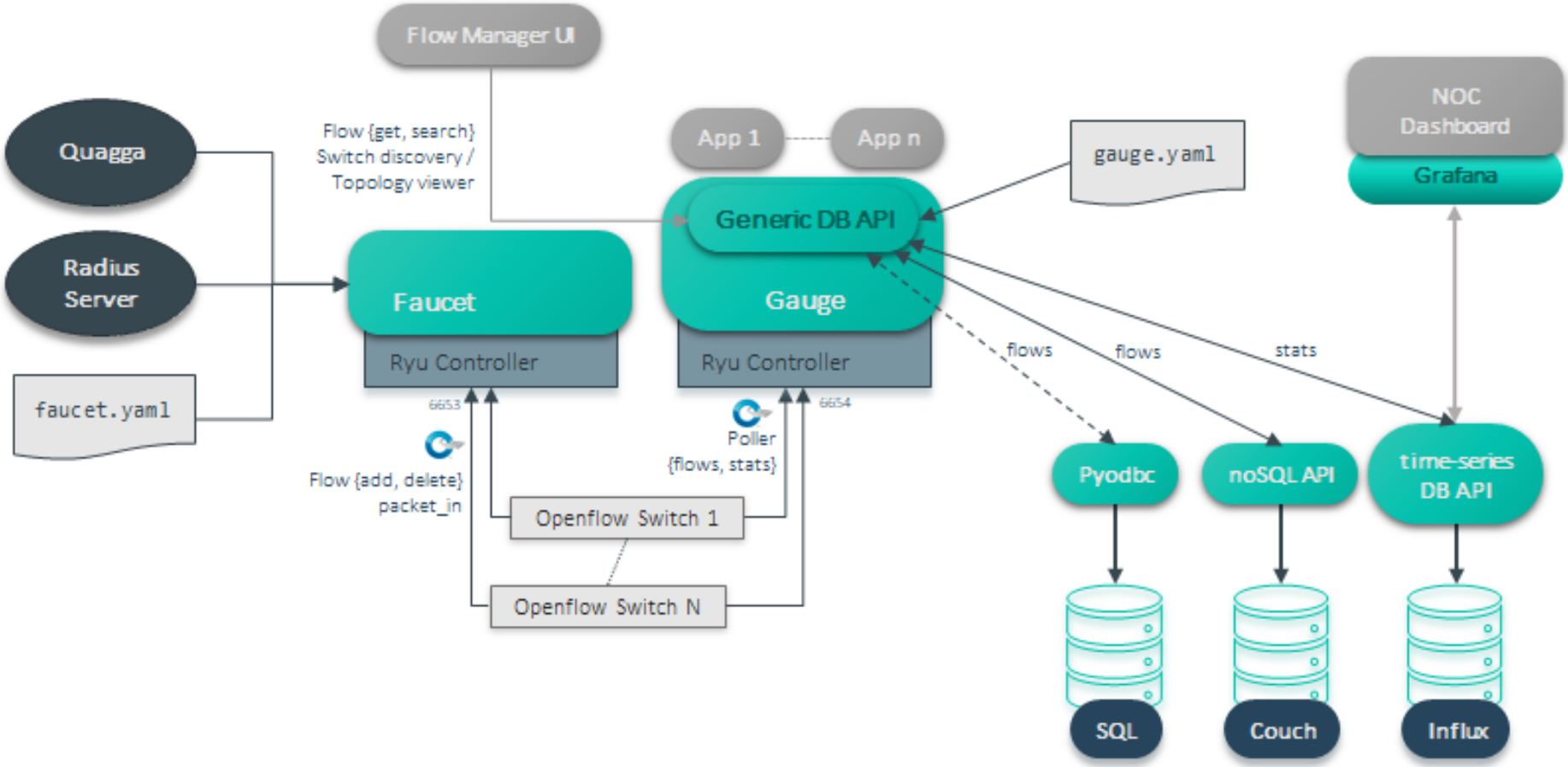
Faucet features

- Southbound
  - It supports OpenFlow v1.3 as a southbound protocol and has a support for feature such as VLANs, IPv4, IPv6, static and BGP routing, port mirroring, policy-based forwarding and ACLs matching
- Northbound
  - YAML configuration files track the intended system state instead of instantaneous API calls, requiring external tools for dynamically applying configuration
  - It opens SDN administration to well-understood CI/CD pipelines

- Faucet can export telemetry into Influxdb, Prometheus or flat text log files
- No inbuilt clustering mechanism, instead relying on external tools to maintain availability
  - High availability is achieved by running multiple, identically configured instances, or a single instance controlled by an external framework that detects and restarts failed nodes

The Faucet controller: architectural view

- Faucet has two Ryu running instances: one for control and configuration updates, the other (Gauge) is a read-only connection specifically for gathering, collating and transmitting state information to be processed elsewhere



Faucet scalability

- Faucet is designed to be deployed at scale such that each instance is close to the subset of switches under its control
- Each instance of Faucet is self-contained and can be deployed directly to server hardware or through containers
- Faucet contains no intrinsic clustering capability and requires external tools such as Zookeeper to distribute state if this is desired
  - Extra instances of the controller can be started independently as long as the backing configuration remains identical
- PCE functionality for these controllers could be pushed down to the instance in the form of modules, or implemented in a similar manner to OpenKilda