

A background network diagram featuring a complex web of interconnected nodes and lines. The nodes are represented by circles of varying sizes and colors, including light blue, green, and yellow. Some nodes are solid, while others are outlined. The lines connecting them are thin and light gray, creating a dense, organic network structure that fills the slide.

OpenFlow & Ryu

Setting up your first SDN with Mininet

Novella Bartolini
Federico Trombetti

OPTION 1:

- VirtualBox, or any virtual machine software
- This VM image: <https://bit.ly/3sE775n> (password: sdn)

OPTION 2:

- Ubuntu, or any Linux-based OS
- Wireshark: `sudo apt-get install wireshark`
- Mininet: `sudo apt-get install mininet`
- Mininet graphical interface located in `/lib/mininet/examples/`
- Python and pip module for installing packages
- Ryu controller: `python -m pip install ryu`

Note: If you're having problems starting your RYU application, try:
`python -m pip install eventlet==0.30.0`



Let's set up our first SDN network with Mininet!

Network set-up with Miniclient graphical interface

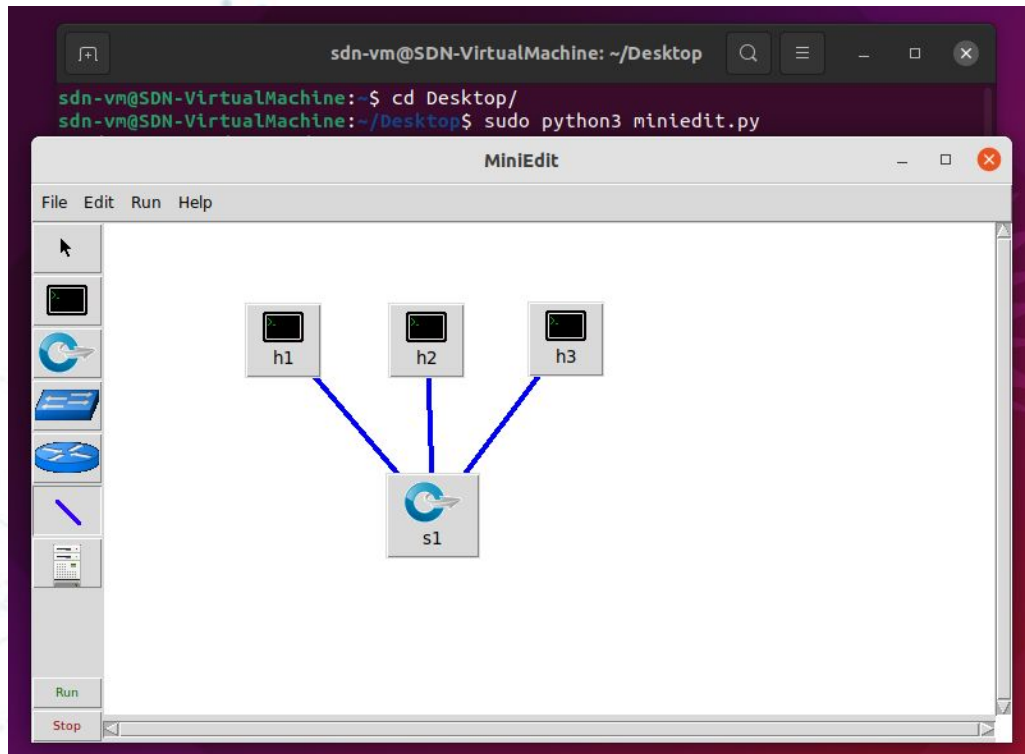
If you're using the VM provided, you should have a python script called ***miniedit.py*** on your Desktop.

Run that script with the command from your Desktop:

sudo python3 miniedit.py

Once you have executed the mininet graphic interface, try to spawn some hosts and clients to get some understanding on how the application works.

Lay down a very simple topology like the one in the image on the right, and let's now connect to the hosts' terminal!



Connecting to the hosts and switches' console

Under **Preferences**, check the option “Start CLI”.

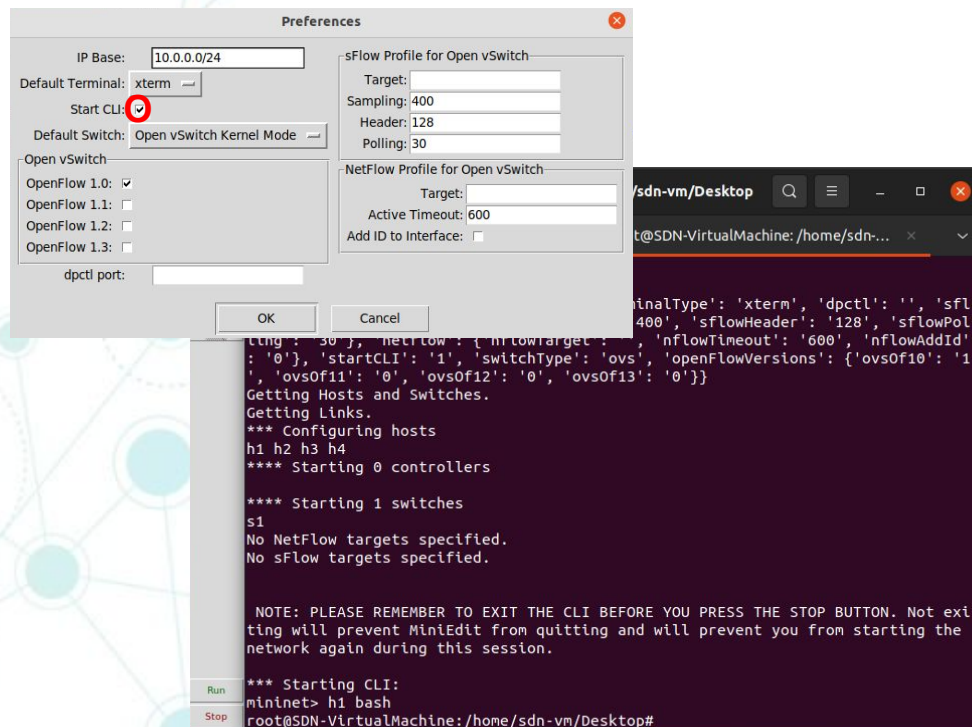
After that, just press the green button “**RUN**” on the Miniclient window.

If you go back on the terminal you used to run the graphical interface, you should have now access to the mininet console.

To connect to a node, just write **node_name bash**

That will prompt you to the node's console.

When closing the environment, remember to **EXIT** properly from the client before closing the simulation from the STOP button.



Setting up the OpenFlow switch for the controller

To see the current configuration of a switch, just type in the switch terminal:

ovs-vsctl show

This will prompt you with basic information of the virtual switch, such as the interfaces currently in use by the switch for the SDN network, and the status of the controller.

Currently, we have no controller running.

Let's tell the switch to listen for a controller on the local network by typing the command:

ovs-vsctl set-controller s1 tcp:localhost:6633

This will tell the virtual switch **s1** to listen for a controller on the local area network on port 6633 (the default for the OpenFlow protocol).

Let's start our first Ryu controller application!

```
root@SDN-VirtualMachine: /home/sdn-vm/Desktop
No sflow targets specified.

NOTE: PLEASE REMEMBER TO EXIT THE CLI BEFORE YOU PRESS THE STOP BUTTON. Not exiting will prevent MiniEdit from quitting and will prevent you from starting the network again during this session.

*** Starting CLI:
mininet> s1 bash
root@SDN-VirtualMachine:/home/sdn-vm/Desktop# ovs-vsctl show
25ba05d4-2494-4cbd-af64-bb50cbd2be36
    Bridge s1
        fail_mode: secure
        Port s1-eth2
            Interface s1-eth2
        Port s1
            Interface s1
                type: internal
        Port s1-eth1
            Interface s1-eth1
        Port s1-eth3
            Interface s1-eth3
    ovs_version: "2.16.0"
root@SDN-VirtualMachine:/home/sdn-vm/Desktop#
```

```
root@SDN-VirtualMachine: /home/sdn-vm/Desktop
    type: internal
    Port s1-eth1
        Interface s1-eth1
    Port s1-eth3
        Interface s1-eth3
    ovs_version: "2.16.0"
root@SDN-VirtualMachine:/home/sdn-vm/Desktop# ovs-vsctl set-controller s1 tcp:localhost:6633
root@SDN-VirtualMachine:/home/sdn-vm/Desktop# ovs-vsctl show
25ba05d4-2494-4cbd-af64-bb50cbd2be36
    Bridge s1
        Controller "tcp:localhost:6633"
        fail_mode: secure
        Port s1-eth2
            Interface s1-eth2
        Port s1
            Interface s1
                type: internal
        Port s1-eth1
            Interface s1-eth1
        Port s1-eth3
            Interface s1-eth3
    ovs_version: "2.16.0"
root@SDN-VirtualMachine:/home/sdn-vm/Desktop#
```


Starting your first Ryu controller application

On the Desktop of your VM, you should have a folder named ***ryu_examples***, inside, you will find different examples of Ryu controller applications.

To get our hands into programming a Ryu controller, we will start with a very basic implementation of a switch protocol, by running the script ***simple_switch.py***

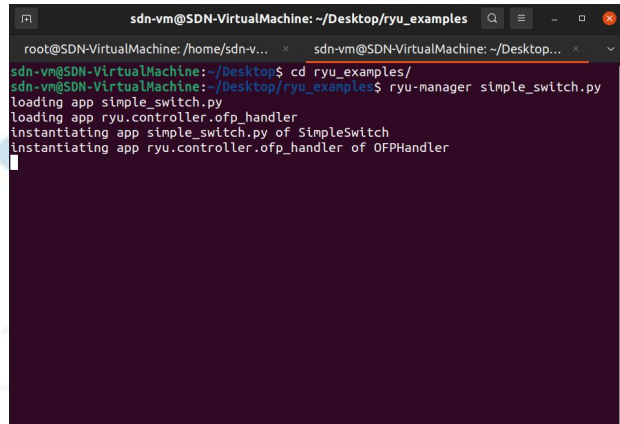
Make sure you're inside the ***ryu_examples*** folder, then run the following command:

ryu-manager simple_switch.py

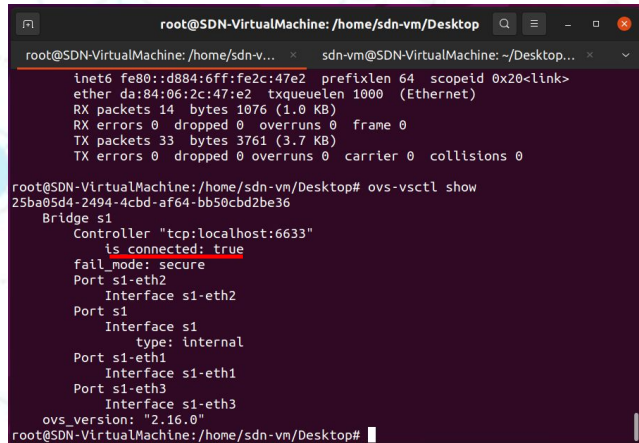
This will instruct Ryu to run ***simple_switch.py*** as the main controller application.

The controller should immediately communicate with the switch running in the Mininet simulation, as we have configured it before to listen to the controller on the local network interface.

By going back to the switch and checking the ovs configuration, we can now see that the switch is properly connected to the controller.



```
sdn-vm@SDN-VirtualMachine: ~/Desktop/ryu_examples
root@SDN-VirtualMachine: /home/sdn-v... x sdn-vm@SDN-VirtualMachine: ~/Desktop... x
sdn-vm@SDN-VirtualMachine: ~/Desktop$ cd ryu_examples/
sdn-vm@SDN-VirtualMachine: ~/Desktop/ryu_examples$ ryu-manager simple_switch.py
loading app simple_switch.py
loading app ryu.controller.ofp_handler
Instantiating app simple_switch.py of SimpleSwitch
Instantiating app ryu.controller.ofp_handler of OFPHandler
```



```
root@SDN-VirtualMachine: /home/sdn-vm/Desktop
root@SDN-VirtualMachine: /home/sdn-v... x sdn-vm@SDN-VirtualMachine: ~/Desktop... x
inet6 fe80::d884:6fff:fe2c:47e2 prefixlen 64 scopeid 0x20<link>
ether da:84:06:2c:47:e2 txqueuelen 1000 (Ethernet)
RX packets 14 bytes 1076 (1.0 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 33 bytes 3761 (3.7 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@SDN-VirtualMachine: /home/sdn-vm/Desktop# ovs-vsctl show
25ba05d4-2494-4cbd-af64-bb50cbd2be36
    Bridge s1
        Controller "tcp:localhost:6633"
            is connected: true
        fail_mode: secure
        Port s1-eth2
            Interface s1-eth2
        Port s1
            Interface s1
                type: internal
        Port s1-eth1
            Interface s1-eth1
        Port s1-eth3
            Interface s1-eth3
    ovs version: "2.16.0"
root@SDN-VirtualMachine: /home/sdn-vm/Desktop#
```

Verifying the connection with a ping

We can now verify if the OpenFlow switch is properly working, by pinging from one host to another.

To do that, just go back to the mininet client and type in:

h1 ping h2

You should get a response from the host h2, and, if you go back to the controller terminal, you should see the packets getting received by the controller. This happens until a flow rule is installed and the switch no longer needs to contact the controller to decide how to route the flows.

You can see all these messages (PACKET_IN, PACKET_OUT and FLOW_MOD), if you monitor your Lo (Loopback) interface with **Wireshark**.

```
sdn-vm@SDN-VirtualMachine: ~/Desktop/ryu_examples
root@SDN-VirtualMachine: /home/sdn-vm... x sdn-vm@SDN-VirtualMachine: ~/Desktop...
sdn-vm@SDN-VirtualMachine: ~/Desktop/ryu_examples$ ryu-manager simple_switch.py
loading app ryu.controller.ofp_handler
loading app ryu.controller.ofp_handler
instantiating app simple_switch.py of SimpleSwitch
instantiating app ryu.controller.ofp_handler of OFPHandler
packet in 1 ca:e7:dd:6f:04:e8 8a:39:12:0c:f8:a6 1
packet in 1 8a:39:12:0c:f8:a6 ca:e7:dd:6f:04:e8 2
packet in 1 ca:e7:dd:6f:04:e8 8a:39:12:0c:f8:a6 1

root@SDN-VirtualMachine: /home/sdn-vm/Desktop
25ba05d4-2494-4cbd-af64-bb50cbd2be36
Bridge s1
  Controller "tcp:localhost:6633"
    is_connected: true
    fail_mode: secure
    Port s1-eth2
      Interface s1-eth2
    Port s1
      Interface s1
        type: internal
    Port s1-eth1
      Interface s1-eth1
    Port s1-eth3
      Interface s1-eth3
    ovs_version: "2.16.0"
root@SDN-VirtualMachine: /home/sdn-vm/Desktop# exit
exit
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.68 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=1.37 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.119 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.041 ms
```

Capturing from Loopback: lo

No.	Time	Source	Destination	Protocol	Length	Info
41	29.523832277	127.0.0.1	127.0.0.1	TCP	66	6633 → 48686 [ACK] Seq=17 Ack=241 Win=44032 Len=0 TSval=2773212669 TSecr=2773212...
42	29.524318986	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
43	29.524323828	127.0.0.1	127.0.0.1	TCP	66	48686 → 6633 [ACK] Seq=241 Ack=25 Win=44032 Len=0 TSval=2773212670 TSecr=2773212...
44	31.570388196	ca:e7:dd:6f:04:e8	Broadcast	OpenFlow	126	Type: OFPT_PACKET_IN
45	31.570319757	127.0.0.1	127.0.0.1	TCP	66	6633 → 48686 [ACK] Seq=25 Ack=361 Win=44032 Len=0 TSval=2773214116 TSecr=2773214...
46	31.571184551	ca:e7:dd:6f:04:e8	Broadcast	OpenFlow	132	Type: OFPT_PACKET_OUT
47	31.571188878	127.0.0.1	127.0.0.1	TCP	66	48686 → 6633 [ACK] Seq=361 Ack=91 Win=44032 Len=0 TSval=2773214116 TSecr=2773214...
48	31.571285209	8a:39:12:0c:f8:a6	ca:e7:dd:6f:04:e8	OpenFlow	126	Type: OFPT_PACKET_IN
49	31.571287537	127.0.0.1	127.0.0.1	TCP	66	6633 → 48686 [ACK] Seq=91 Ack=361 Win=44032 Len=0 TSval=2773214117 TSecr=2773214...
50	31.571759972	127.0.0.1	127.0.0.1	OpenFlow	146	Type: OFPT_FLOW_MOD
51	31.571762721	127.0.0.1	127.0.0.1	TCP	66	48686 → 6633 [ACK] Seq=361 Ack=171 Win=44032 Len=0 TSval=2773214117 TSecr=277321...
52	31.571771960	8a:39:12:0c:f8:a6	ca:e7:dd:6f:04:e8	OpenFlow	132	Type: OFPT_PACKET_OUT
53	31.571773985	127.0.0.1	127.0.0.1	TCP	66	48686 → 6633 [ACK] Seq=361 Ack=237 Win=44032 Len=0 TSval=2773214117 TSecr=277321...
54	31.571957250	10.0.0.1	10.0.0.2	OpenFlow	182	Type: OFPT_PACKET_IN
55	31.571959537	127.0.0.1	127.0.0.1	TCP	66	6633 → 48686 [ACK] Seq=237 Ack=477 Win=44032 Len=0 TSval=2773214117 TSecr=277321...

Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 48678, Dst Port: 6633, Seq: 0, Len: 0

Checking for the installation of the flow rules

```
root@SDN-VirtualMachine: /home/sdn-vm/Desktop
No sFlow targets specified.

NOTE: PLEASE REMEMBER TO EXIT THE CLI BEFORE YOU PRESS THE STOP BUTTON. Not exiting will prevent MiniEdit from quitting and will prevent you from starting the network again during this session.

*** Starting CLI:
mininet>
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=3.17 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.120 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.036 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.036 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.032 ms
^C
--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4056ms
rtt min/avg/max/mdev = 0.032/0.679/3.173/1.247 ms
mininet> s1 bash
root@SDN-VirtualMachine: /home/sdn-vm/Desktop# ovs-ofctl dump-flows s1
cookie=0x0, duration=12.847s, table=0, n_packets=6, n_bytes=532, in_port="s1-eth2", dl_src=02:e5:03:02:4b:18, dl_dst=8a:8a:5c:da:7c:a7 actions=output:"s1-eth1"
cookie=0x0, duration=12.845s, table=0, n_packets=5, n_bytes=434, in_port="s1-eth1", dl_src=8a:8a:5c:da:7c:a7, dl_dst=02:e5:03:02:4b:18 actions=output:"s1-eth2"
root@SDN-VirtualMachine: /home/sdn-vm/Desktop#
```

To end with this simple test, we can verify the installation of the flow rules by going back to the **s1** terminal and typing:

ovs-ofctl dump-flows s1

This will prompt us with all the flows currently installed on the switch.

We can see that the switch has now installed two rules, one for each direction of the flow generated by the ping.

All the flows on a switch can be deleted anytime either by an instruction of the controller, or, in an easier way for a quick cleanup, with the command on the console of the switch:

ovs-ofctl del-flows s1

This ends our first test with a Ryu application.

Let's now try to understand how everything works under the hood.

Understanding the basics of a Ryu Application



Programming model of Ryu applications

Ryu controller applications are meant to be **event-driven**. With the controller and the switches actively communicating with each other during the whole execution of the application.

We distinguish messages that the controller and the switches exchange mainly in three categories:

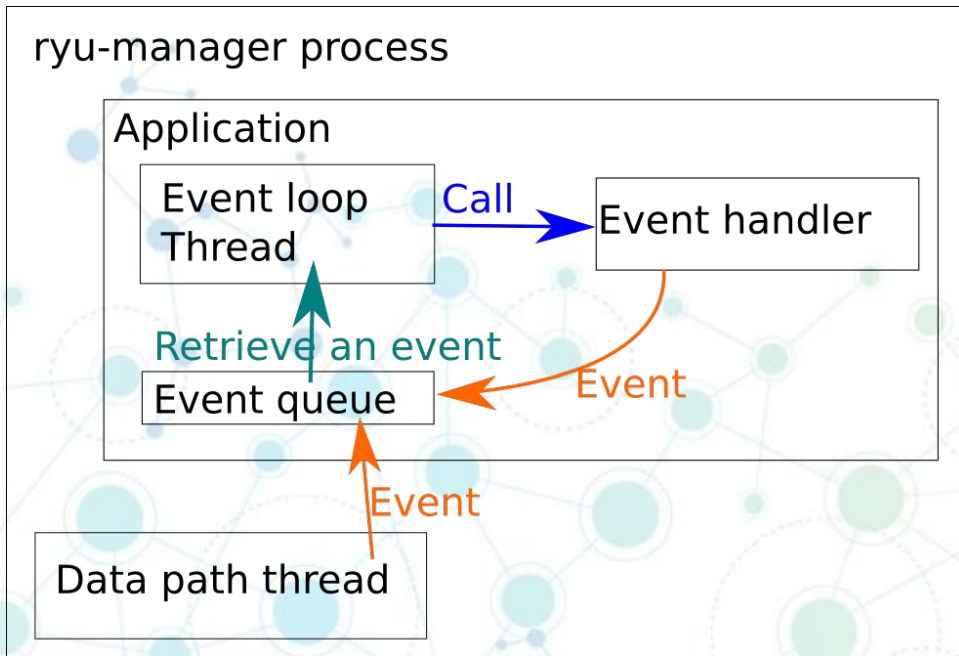
- Controller-to-Switch messages
- Symmetric messages
- Asynchronous messages

The first two type of messages will invoke a particular response from the switches and generate an event, which has to be captured from the controller with an handler. Event handlers can be declared in Ryu with the following Python decorator:

```
@set_ev_cls([event], [handler])
```

Asynchronous messages, on the other hand, are generated only by the switches, but they also have to be captured in the same manner.

You can see from most of the examples in **ryu_examples** how this decorator is used, and how the event object is passed to and used by the handler function.



For a complete reference of the OpenFlow messages in Ryu, refer to the official documentation:

https://ryu.readthedocs.io/en/latest/ofproto_ref.html

The simple_switch.py application

For a better understanding of a Ryu application, it may be wise to dig a little deeper in the code of *simple_switch.py* and see how everything happens under the hood.

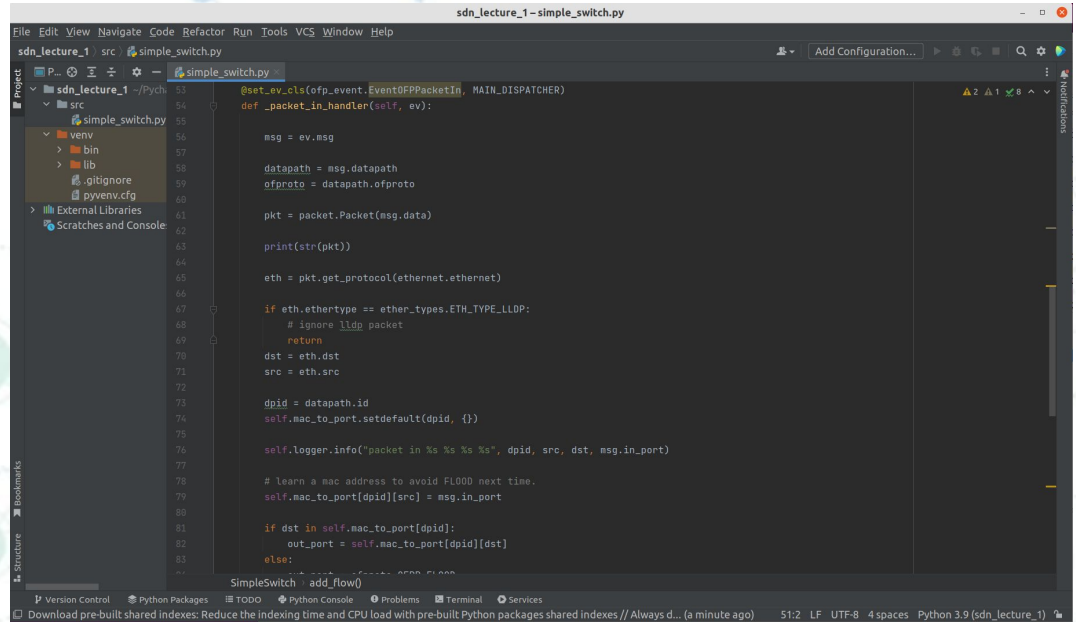
We import the *simple_switch.py* file inside a new PyCharm project.

By digging a little bit into the code, we can see how the application is implemented to handle **PACKET_IN** messages, which are generated by the switch when a new flow is encountered.

The application is configured so that a new rule is installed, as soon as both the source and the destination of the flow are known. This is done with the *add_flow(...)* function.

This function wraps the Ryu function which will generate a **FLOW_MOD** message to be sent to the switch, with the parameters for the flow rule correctly set.

If, instead, the flow destination is still not known (this happens because the host generating the ping doesn't yet know the mac address of the destination, which is required for an ethernet protocol communication), the controller will instruct the switch to output the packet to all the outgoing ports with a **FLOOD** mechanism. This will allow the packet to reach the destination, even if this is not known yet.



```
sdn_lecture_1 - simple_switch.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help
sdn_lecture_1 / src simple_switch.py
Project
  P...
  sdn_lecture_1
    src
      simple_switch.py
    venv
      bin
      lib
      gitignore
      pyvenv.cfg
    External Libraries
    Scratches and Console
sdn_lecture_1 - simple_switch.py
1: @set_ev_cls(ofp_event.EventOfPPacketIn, MAIN_DISPATCHER)
2: def _packet_in_handler(self, ev):
3:
4:     msg = ev.msg
5:
6:     datapath = msg.datapath
7:     ofproto = datapath.ofproto
8:
9:     pkt = packet.Packet(msg.data)
10:
11:     print(str(pkt))
12:
13:     eth = pkt.get_protocol(ethernet.ethernet)
14:
15:     if eth.ethertype == ether_types.ETH_TYPE_LLDP:
16:         # ignore lldp packet
17:         return
18:
19:     dst = eth.dst
20:     src = eth.src
21:
22:
23:     dpid = datapath.id
24:     self.mac_to_port.setdefault(dpid, {})
25:
26:     self.logger.info("packet in %s %s %s %s", dpid, src, dst, msg.in_port)
27:
28:     # learn a mac address to avoid FLOOD next time.
29:     self.mac_to_port[dpid][src] = msg.in_port
30:
31:     if dst in self.mac_to_port[dpid]:
32:         out_port = self.mac_to_port[dpid][dst]
33:     else:
34:         # Flood all the packet in case we don't know the output port
35:         self.logger.info("FLOOD")
36:         for port_id in self.datapath.get_ports():
37:             self.datapath.send(out_port=port_id, dst_mac=dst, src_mac=src, in_port=msg.in_port)
```

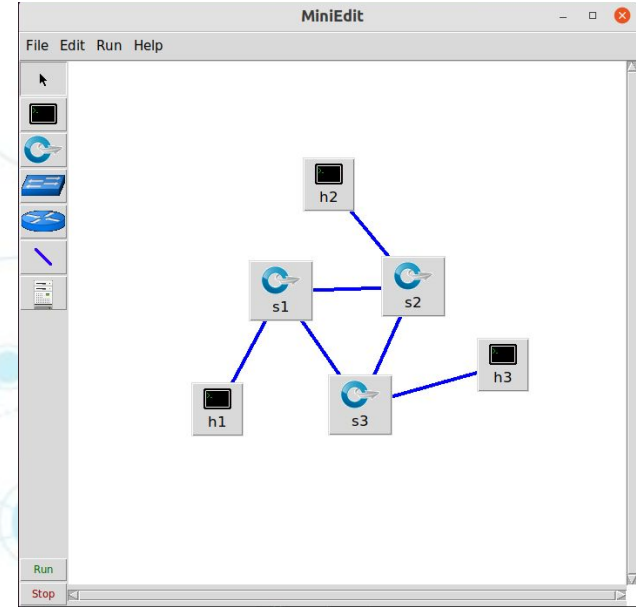
Deploying simple_switch.py to a different network

If you try to run the same application on a more complex network, like the one on the image on the right, you will notice that routing will work in a very inconsistent way. Most of the time, it won't work at all.

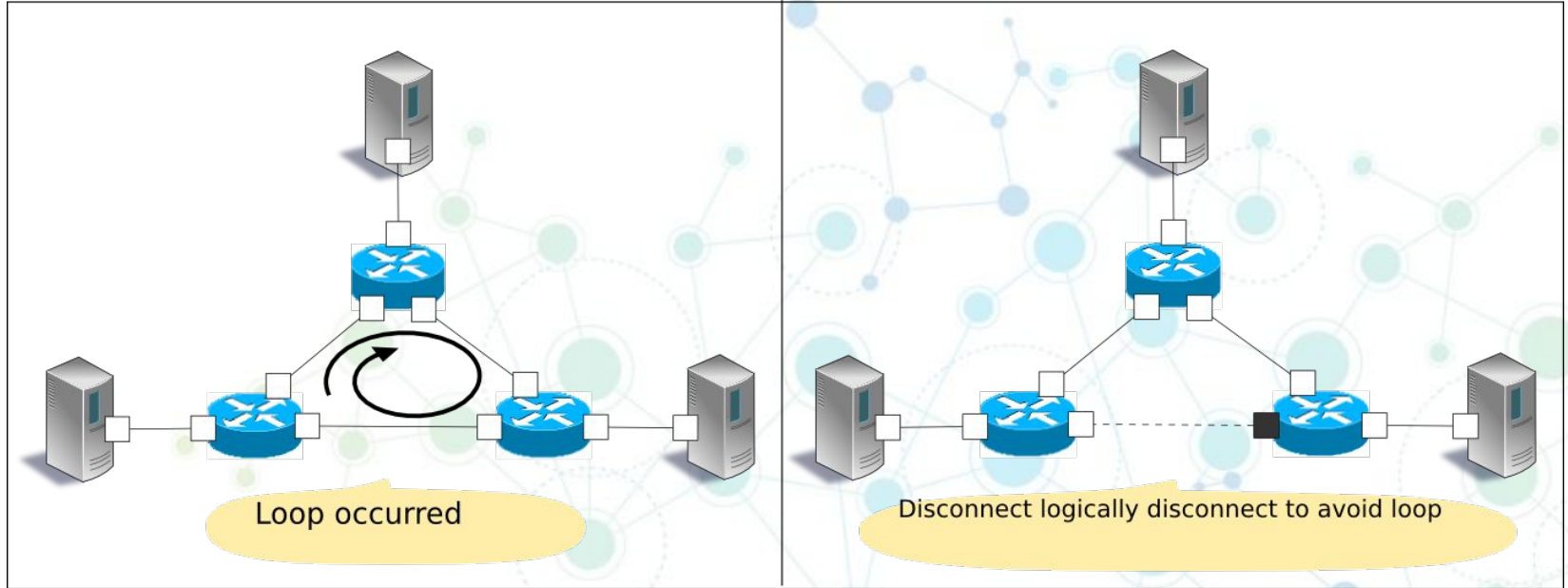
This happens due to the FLOOD mechanism that the controller uses to find out the destination mac address of a flow. If a loop is encountered by the FLOOD, an endless number of packets will be generated by the switches, and the communication between the hosts will mostly fail.

In order to prevent that, we will have either to logically disconnect one of the switches by another, or implement a more complex routing application.

This will be the main focus in the next lecture.



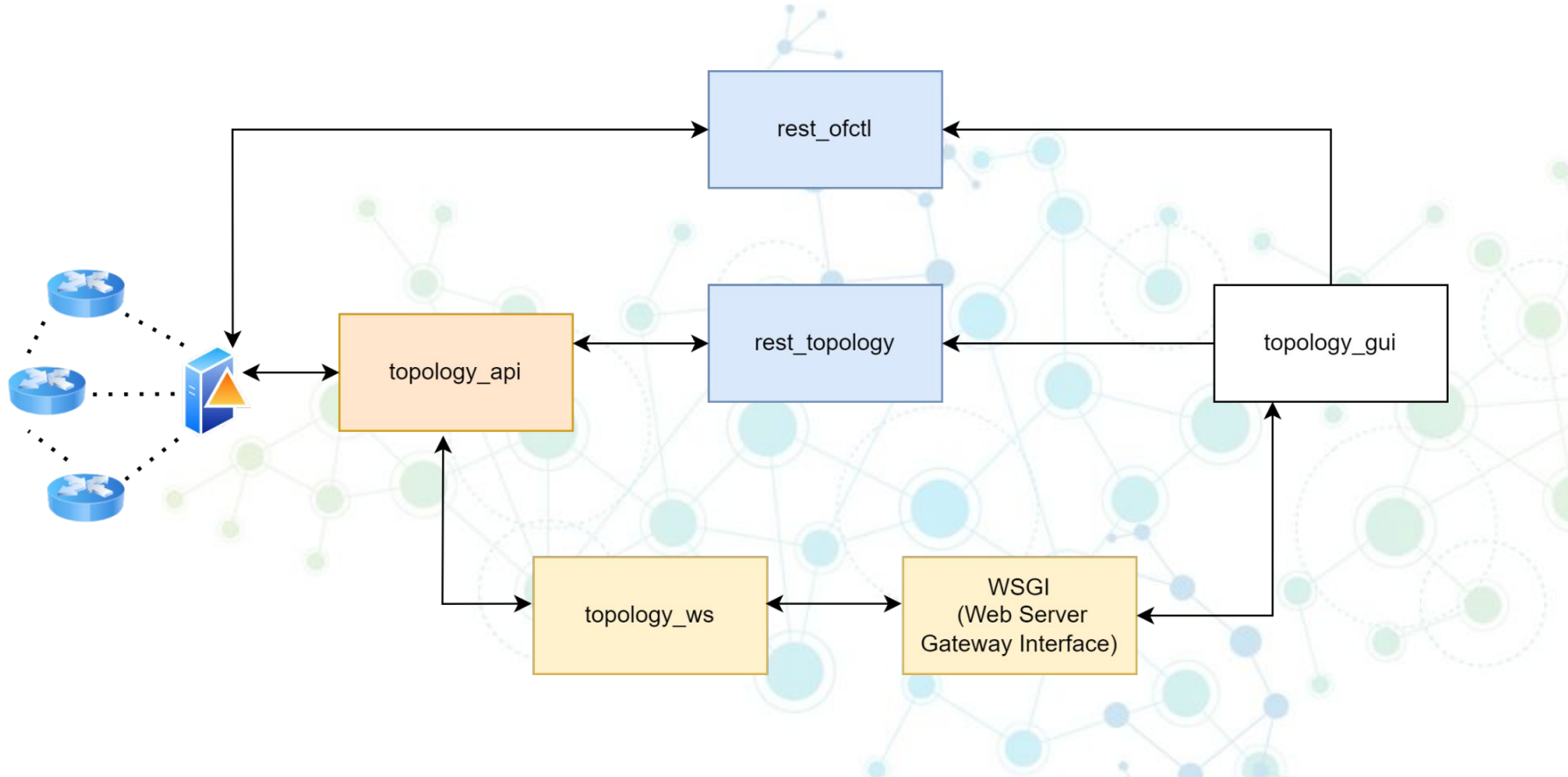
A possible workaround





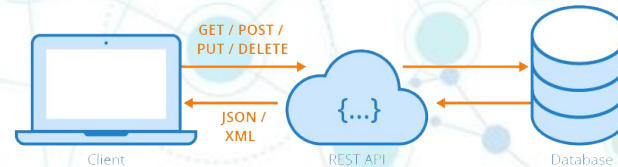
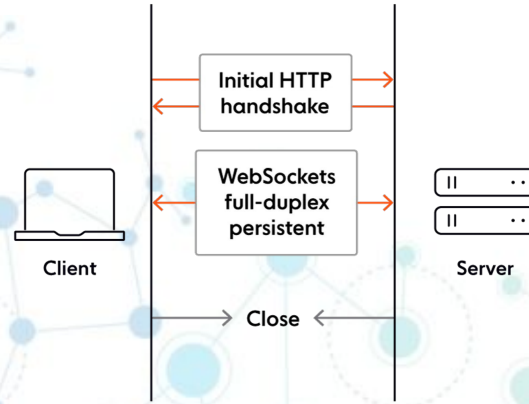
Moving to a more complex application

Topology GUI

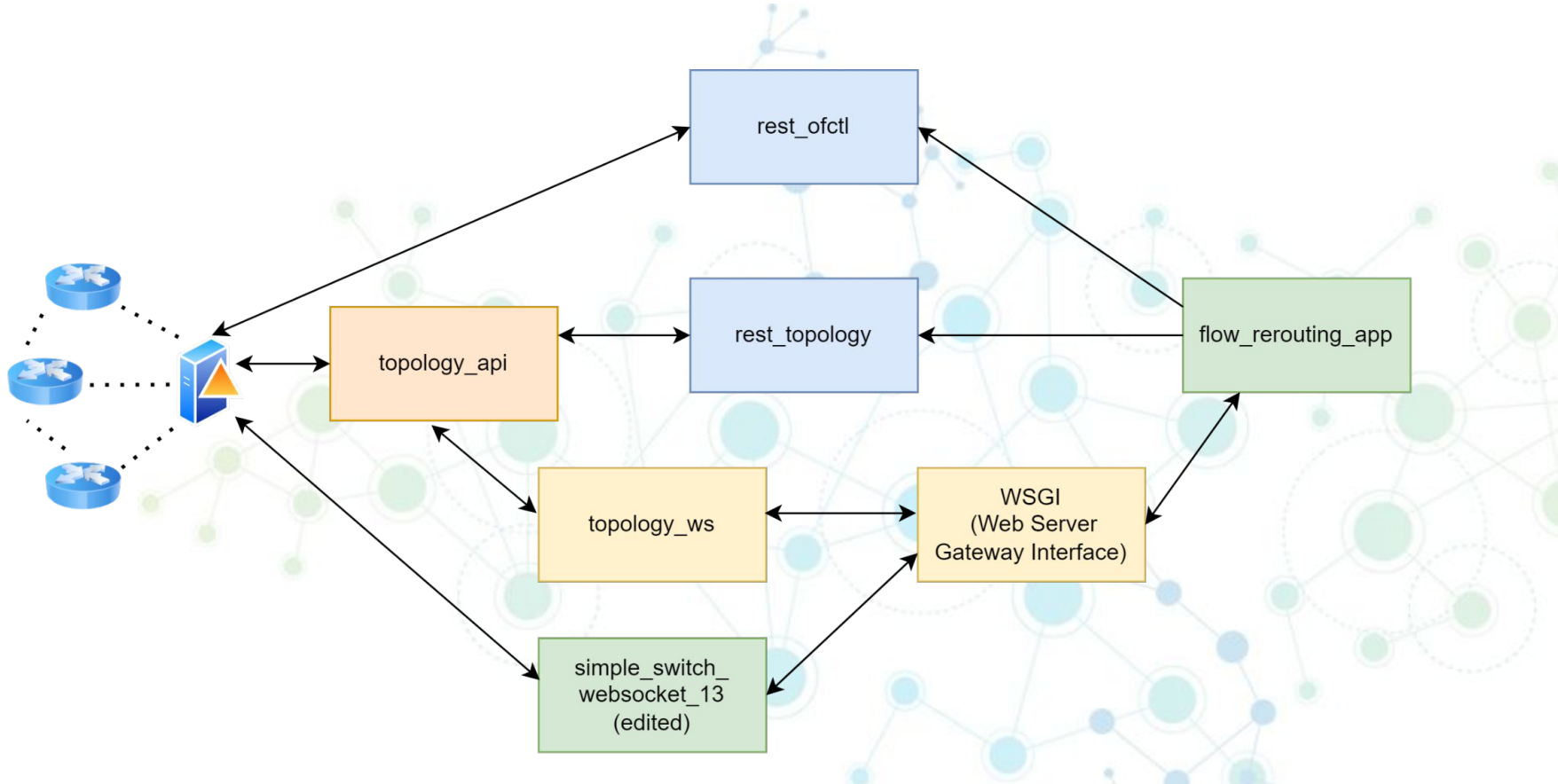


RESTful APIs and WebSockets

- **WebSocket** is a computer communications protocol, providing two-way communication channels over a single TCP connection.
- **REST** is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other.



Our Flow Rerouting Application



Alternative version without socket, simpler to implement

