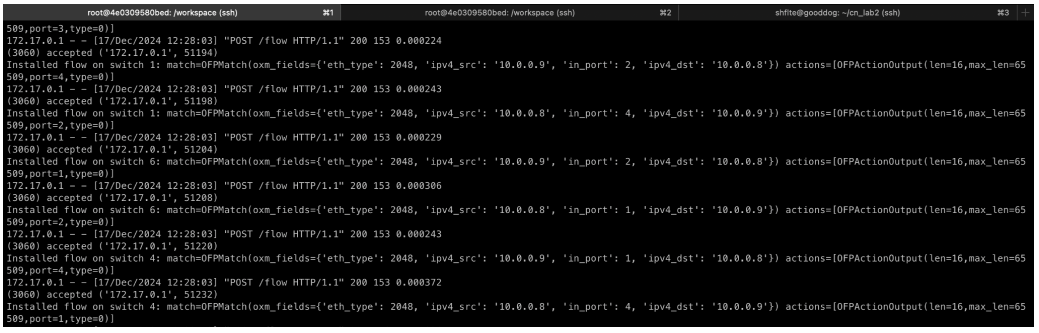
Computer Networks Lab2 Report

313552044 陳宇駿

Result

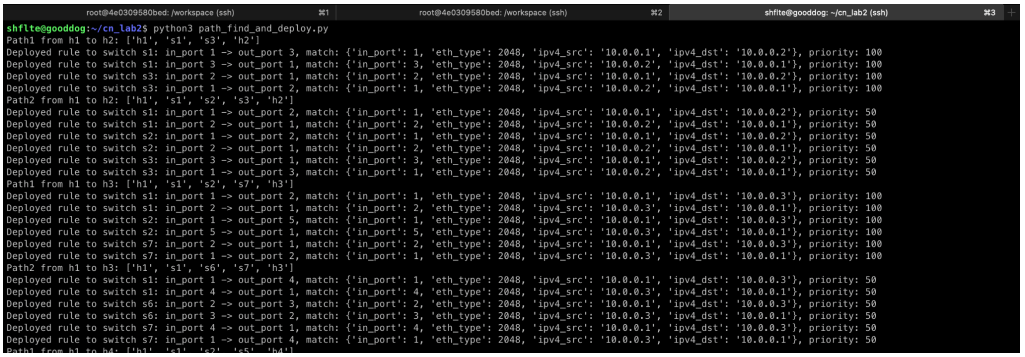
Ryu controller

The Ryu controller receives HTTP requests from the path calculator and deploys the flow rules to the switches accordingly.



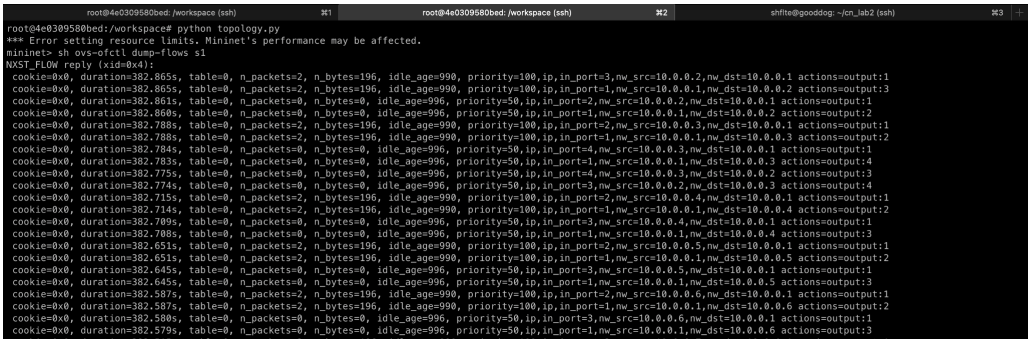
Path calculation and deployment

The path calculator finds the paths and sends the resulting flow rule requests to the controller for deployment.



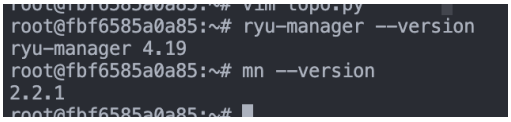
Mininet

The result of `sh ovs-ofctl dump-flows s1` in Mininet shows the deployed rules of switch s1.



Environment Setup

Utilize the docker image with preinstalled `ryu-manager` and `mininet` from the dockerhub iwaseyusuke/ryu-mininet.



Since the image works with Python2, which disables many convenient modern Python3 features, thus the path calculation is done outside the container.

In this lab, the following packages are utilized:

- `requests`: send HTTP requests to the ryu controller.
- `networkx`: shortest path calculation.

In this lab, the following source files are implemented:

- `topology.json`: topology settings.
- `topology.py`: construct environmental topology with mininet.
- `controller.py`: ryu controller with predefined RESTful API to accept HTTP requests and deploy switch forwarding rules accordingly.
- `path_find_and_deploy.py`: find shortest paths and deploy the rules by sending HTTP requests to the ryu controller.

Execution Steps

Retrieve the container:

```
docker pull iwaseyusuke/ryu-mininet
```

Start the container:

```
docker run -it --privileged -p 8080:8080 -v $(pwd):/workspace -e DISPLAY=$DISPLAY -w /workspace iwaseyusuke/ryu-mininet
```

Forward port 8080 for the path calculation and deployment script to access the REST API from the outside of the container.

In the container:

- Run the controller:
  `ryu-manager controller.py`
- Run the net:
  `python topology.py`

On the host:

- Install the utilized packages:
  `pip3 install requests; pip3 install networkx`
- Run the path calculation and deployment script:
  `python3 path_find_and_deploy.py`

Topology Setup

Since the topology is shared among multiple files, the basic settings of the topology is defined in the `topology.json` in JSON format.

```json
{
    "nodes": {
        "hosts": [
            {"name": "h1", "ip": "10.0.0.1", "mac": "00:00:00:00:00:01"},
            {"name": "h2", "ip": "10.0.0.2", "mac": "00:00:00:00:00:02"},
            {"name": "h3", "ip": "10.0.0.3", "mac": "00:00:00:00:00:03"},
            {"name": "h4", "ip": "10.0.0.4", "mac": "00:00:00:00:00:04"},
            {"name": "h5", "ip": "10.0.0.5", "mac": "00:00:00:00:00:05"},
            {"name": "h6", "ip": "10.0.0.6", "mac": "00:00:00:00:00:06"},
            {"name": "h7", "ip": "10.0.0.7", "mac": "00:00:00:00:00:07"},
            {"name": "h8", "ip": "10.0.0.8", "mac": "00:00:00:00:00:08"},
            {"name": "h9", "ip": "10.0.0.9", "mac": "00:00:00:00:00:09"}
        ],
        "switches": ["s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8"]
    },
    "links": [
        {
            "node1": "s1",
            "port1": 1,
            "node2": "h1",
            "port2": 1
        },
        {
            "node1": "s1",
            "port1": 2,
            "node2": "s2",
            "port2": 1
        },
```

In `topology.py`, the `topology.json` is loaded and parsed, then the topology is constructed with `mininet`.

```python
if __name__ == "__main__":
    topo_file = "topology.json"
    net = Mininet(controller=RemoteController, link=TCLink)
    controller = net.addController('c0', controller=RemoteController, ip='127.0.0.1', port=6653)

    with open(topo_file, "r") as file:
        topology = json.load(file)

    hosts = {}
    switches = {}

    # Add hosts
    for host in topology["nodes"]["hosts"]:
        hosts[host["name"]] = net.addHost(
            host["name"], ip=host["ip"], mac=host["mac"]
        )

    # Add switches
    for switch in topology["nodes"]["switches"]:
        switches[switch] = net.addSwitch(switch)
```

```python
    # Add links with ports
    for link in topology["links"]:
        node1 = link["node1"]
        port1 = link["port1"]
        node2 = link["node2"]
        port2 = link["port2"]

        if node1 in hosts:
            net.addLink(hosts[node1], switches[node2], port1=port1, port2=port2)
        elif node2 in hosts:
            net.addLink(switches[node1], hosts[node2], port1=port1, port2=port2)
        else:
            net.addLink(switches[node1], switches[node2], port1=port1, port2=port2)

    # Start the network
    net.start()
    net.staticArp()

    # Start CLI
    CLI(net)

    # Stop the network
    net.stop()
```

In `path_find_and_deploy.py`, the same topology is loaded and parsed, then a `networkx.Graph()` object is constructed accordingly for further path calculation.

```python
def load_topology(file_path):
    with open(file_path, "r") as f:
        return json.load(f)


def build_graph(topology):
    graph = nx.Graph()
    link_to_outport = {}
    for link in topology["links"]:
        graph.add_edge(
            link["node1"], link["node2"], port1=link["port1"], port2=link["port2"]
        )
        link_to_outport[(link["node1"], link["node2"])] = link["port2"]
        link_to_outport[(link["node2"], link["node1"])] = link["port1"]
    return graph, link_to_outport
```

Ryu SDN Controller

On the controller, the `FlowController` class is defined for basic controller functionalities, such as handling switch connections, setting default flow rules, and managing installed flows, while the `RestAPIController` is defined for exposing a RESTful API to deploy flow rules.

FlowController

`add_flow()` installs a flow rule on a given switch by setting its match conditions, actions, and priority.
`install_flow()` retrieves the switch datapath using its ID and calls `add_flow()` to apply the rule while logging the action for debugging purposes.

```
def add_flow(self, datapath, priority, match, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
    mod = parser.OFPFlowMod(
        datapath=datapath, priority=priority, match=match, instructions=inst
    )
    datapath.send_msg(mod)

def install_flow(self, datapath_id, match, actions, priority):
    datapath = self.switches.get(datapath_id)
    self.add_flow(datapath, priority, match, actions)
    self.logger.info(
        "Installed flow on switch {}: match={} actions={}".format(
            datapath_id, match, actions
        )
    )
```

`switch_features_handler()` handles the `SwitchFeatures` event, saving the connected switch's datapath and installing a default flow rule with `add_flow()`.

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    # Handler to set initial flow rules
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # Save connected switch
    self.switches[datapath.id] = datapath

    # Add default flow rule to send unmatched packets to the controller
    match = parser.OFPMatch()
    actions = [
        parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, ofproto.OFPCML_NO_BUFFER)
    ]
    self.add_flow(datapath, 0, match, actions)
```

RestAPIController

This method processes POST requests to the `/flow` API. It parses the request containing the switch ID, match fields, and actions, then builds an OpenFlow match and action list using the Ryu parser. Finally, it installs the flow on the specified switch and returns a success response.

Request format:

- Method: POST
- Header: "Content-Type: application/json"
- Payload:
    - switch: switch ID of the target switch to deploy rules.
    - match: the matching rules, including src/dst IP addr., eth. type, in port.
    - action: only specify the out port for this task.
    - priority: specified to differentiate the dual paths for the same src/dst pairs.

```python
@route("flow", rest_api_path, methods=["POST"])
def set_flow(self, req, **kwargs):
    try:
        # Parse incoming JSON request
        data = json.loads(req.body)

        datapath_id = data["switch"]
        match_fields = data["match"]
        actions = data["actions"]

        # Build match and actions for ryu
        parser = self.app.switches[datapath_id].ofproto_parser
        match = parser.OFPMatch(**match_fields)
        action_list = [parser.OFPActionOutput(a["port"]) for a in actions]

        # Install the flow rule
        self.app.install_flow(datapath_id, match, action_list)

        return Response(
            content_type="application/json",
            body=json.dumps({"status": "success"}),
            status=200,
        )
```

Path Calculate & Deploy

In `path_find_and_deploy.py`, both of the paths are calculated, and the forwarding rules are deployed to the ryu controller through the predefined RESTful API.

deploy_flow()

In `deploy_flow()`, the requests bound to the ryu controller for the forwarding rule deployment are constructed and sent.

```python
def deploy_flow(switch, in_port, out_port, src_ip, dst_ip, priority):
    url = "http://127.0.0.1:8080/flow"
    eth_type = 0x0800
    match = {"in_port": in_port, "eth_type": eth_type, "ipv4_src": src_ip, "ipv4_dst": dst_ip}

    data = {
        "switch": int(switch.replace("s", "")),
        "match": match,
        "actions": [{"port": out_port}],
        "priority": priority
    }

    response = requests.post(url, json=data)
```

path_find_and_deploy()

In `path_find_and_deploy()`, two shortest path searches are conducted to find two most disjoint paths. At the end of each search, the forwarding rules should be deployed to all switches along the calculated shortest path with `deploy_flow()`. Note that the priority of forwarding rules in each path are set differently in order to have the first main path be adopted first in the rule matching phase.

- First search:

```python
path1 = nx.shortest_path(graph, source=src["name"], target=dst["name"])
print(f"Path1 from {src['name']} to {dst['name']}: {path1}")

for i in range(1, len(path1) - 1):
    current_node = path1[i]
    next_node = path1[i + 1]
    prev_node = path1[i - 1]

    if not current_node.startswith("s"):
        continue

    switch = current_node
    in_port = link_to_outport[(prev_node, current_node)]
    out_port = link_to_outport[(next_node, current_node)]

    deploy_flow(
        switch, in_port, out_port, src_ip=src["ip"], dst_ip=dst["ip"], priority=100
    )
    deploy_flow(
        switch, out_port, in_port, src_ip=dst["ip"], dst_ip=src["ip"], priority=100
    )
```

- Second search:

```python
graph_copy = graph.copy()

for i in range(len(path1) - 1):
    u, v = path1[i], path1[i + 1]
    if graph_copy.has_edge(u, v):
        graph_copy[u][v]["weight"] = 1000

path2 = nx.shortest_path(
    graph_copy, source=src["name"], target=dst["name"], weight="weight"
)
print(f"Path2 from {src['name']} to {dst['name']}: {path2}")

for i in range(1, len(path2) - 1):
    current_node = path2[i]
    next_node = path2[i + 1]
    prev_node = path2[i - 1]

    if not current_node.startswith("s"):
        continue

    switch = current_node
    in_port = link_to_outport[(prev_node, current_node)]
    out_port = link_to_outport[(next_node, current_node)]

    deploy_flow(
        switch, in_port, out_port, src_ip=src["ip"], dst_ip=dst["ip"], priority=50
    )
    deploy_flow(
        switch, out_port, in_port, src_ip=dst["ip"], dst_ip=src["ip"], priority=50
    )
```