



# SDN-Routing

1399.05.29

---

Negin Baghbanzadeh

810196599

Ali keramatipour

810196616

## کلاس کنترلر

به منظور پیاده‌سازی کنترلر، یک کلاس جدید از کلاس Ryu.application ارث‌بری می‌کنیم.

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

سپس ورژن پروتکل ارتباطی openflow را که می‌خواهیم از آن استفاده کنیم به OFP\_VERSION می‌دهیم.

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

## تابع init

برای استفاده از تابع تعریف شده در کلاس پدر، از تابع super استفاده می‌کنیم.

```
def __init__(self, *args, **kwargs):
    super(ProjectController, self).__init__(*args, **kwargs)
```

جدول mac address را تعریف می‌کنیم.

```
self.mac_to_port = {}
```

هنگامی که کنترلر به توپولوژی متصل می‌شود، سوئیچ‌ها و لینک‌های تعریف شده در توپولوژی، در کلاس کنترلر ذخیره می‌شوند. بنابراین با صدا زدن توابعی مانند get\_switch (که در ادامه می‌بینیم) روی خود کلاس کنترلر (self) می‌توانیم اطلاعات توپولوژی را بدست بیاوریم.

```
self.topology_api_app = self
```

در این لیست خصیصه datapath از کلاس switch، هر سوئیچ موجود در توپولوژی را ذخیره خواهیم کرد.

```
self.datapath_list=[]
```

## تابع switch\_features\_handler

در این تابع ابتدا handshake بین کنترلر و سوئیچ openflow انجام می‌شود و سپس برای اینکه سوئیچ‌های توپولوژی آماده دریافت packet in شود و سوئیچ‌ها بتوانند با یکدیگر ارتباط برقرار کنند ورودی‌های flow table-miss را به flow table اضافه می‌کنیم.

در ریزو هنگامی یک پیام openflow دریافت می‌شود یک instance از کلاس event متناسب با پیام دریافت شده ساخته می‌شود. سپس یک هندلر برای این event دریافت شده طراحی می‌کنیم.

Set\_ev\_cls به عنوان ورودی کلاس event ای را که از پیام دریافت شده پشتیبانی می‌کند و همچنین موقعیت سوئیچ openflow را دریافت می‌کند. وضعیت CONFIG\_DISPATCHER مشخص می‌کند که سوئیچ منتظر دریافت پیام switch features می‌باشد.

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures , CONFIG_DISPATCHER)
```

ev، event دریافت شده می‌باشد.

```
def switch_features_handler(self , ev):
```

هر سوئیچ دارای یک خصیصه از جنس کلاس ryu.controller.Datapath به نام datapath می‌باشد. کلاس Datapath وظیفه برقراری ارتباط میان سوئیچ‌های openflow را دارد. با استفاده از دستور زیر خصیصه datapath سوئیچ ورودی را دریافت می‌کنیم.

```
datapath = ev.msg.datapath
```

هر datapath دارای خصیصه‌ای به نام ofproto می‌باشد. می‌باشد که این خصیصه ماژول ofproto ای که openflow مورد استفاده آن را ساپورت می‌کند را مشخص می‌کند.

```
ofproto = datapath.ofproto
```

هر datapath دارای خصیصه دیگری به نام parse نیز می‌باشد. می‌باشد که این خصیصه ماژول ofproto\_parser ای که openflow مورد استفاده آن را ساپورت می‌کند را مشخص می‌کند.

```
parser = datapath.ofproto_parser
```

در قبل handshake بین کنترلر و سوئیچ را تعیین کردیم. حال به ساخت یک ورودی flow table miss برای اضافه کردن به flow table می‌پردازیم. flow table دارای کمترین اولویت می‌باشد. همچنین در ادامه هر پکتی که دریافت شود می‌تواند از این flow استفاده کند.

یک instance خالی از کلاس OFPMatch تولید می‌کنیم که این بدین معنی است که این flow مناسب برای استفاده تمامی پکت‌ها می‌باشد و محدودیت خاصی ندارد.

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

در ادامه تابع OFPActionOutput از کلاس parser را فراخوانی می‌کنیم. از این تابع به منظور فرستادن فرستادن پکت به یک مقصد خاص استفاده می‌شود. ورودی اول این تابع مقصد را مشخص می‌کند. ofproto.OFPP\_CONTROLLER به این معنی است که پکت به کنترلر فرستاده شود و ورودی دوم، OFPCLM\_NO\_BUFFER به این معنی است که تمامی پکت‌ها فرستاده شود و محدودیتی در تعداد پکت‌های ارسالی نباشد.

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

سپس تابع OFPInstructionAction از کلاس parser را فراخوانی می‌کنیم. این تابع مشخص می‌کند که چه عملیاتی روی لیست actions تعریف شده در خط بالا انجام شود. در کد زیر اکشن‌های موجود در لیست actions اجرا می‌شوند.

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

در ادامه یک instance از کلاس OFPFlowMod تعریف می‌کنیم. برای اضافه کردن یک flow جدید جنس ورودی تابع send\_msg از این کلاس است. با این کلاس می‌توانیم اطلاعاتی که می‌خواهیم انتقال دهیم را بسته‌بندی کنیم.

توضیح هر یک از ورودی‌های این تابع به شرح زیر است.

datapath: خصیصه datapath سوئیچی که event را به وجود آورده است.

match: توضیحات بالا

cookie: به عنوان یک فیلتر هنگام آپدیت و یا پاک کردن یک ورودی استفاده می‌شود.

command: چه عملیاتی انجام شود اگر ofproto.OFPPFC\_ADD به این معنا است که یک flow entry جدید ایجاد شود.

idle\_timeout: مدت زمان معتبر بودن یک ورودی به ثانیه. اگر یک ورودی مورد استفاده قرار گیرد، این بازه از اول آغاز می‌شود و اگر این بازه تمام شود، ورودی مورد نظر پاک خواهد شد. صفر به معنای هیچ وقت می‌باشد.

hard\_timeout: مدت زمان معتبر بودن یک ورودی به ثانیه. تفاوت این مورد با بالایی این است که اگر ورودی مورد استفاده قرار گیرد، زمان از اول شروع نمی‌شود.

priority: اولویت رسیدن به این flow را بیان می‌کند. هر چقدر بالاتر، اولویت بیشتر.

instructions: توضیحات بالا

```
mod = datapath.ofproto_parser.OFPFlowMod(datapath=datapath, match=match,
cookie=0, command=ofproto.OFPPFC_ADD, idle_timeout=0, hard_timeout=0, priority=0,
instructions=inst)
```

یکی از متدهای کلاس datapath، switch می‌باشد. این تابع یک پیام openflow را می‌فرستد.

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

## تابع `get_topology_data`

این تابع مانند یک `getter` عمل می‌کند و سوئیچ‌ها و لینک‌هایی که در کلاس در کنترلر ذخیره شده‌اند را به وسیله توابع تعریف شده می‌گیرد و آنها را در لیست‌هایی ذخیره می‌کند.

event زیر هنگامی رخ می‌دهد که یک سوئیچ جدید به توپولوژی اضافه شود.

```
@set_ev_cls(event.EventSwitchEnter)
```

برای هندل کردن event بالا تابع زیر را تعریف می‌کنیم. با استفاده از تابع `get_switch` که در کلاس `app_manager.ryuapp` تعریف شده است تمامی سوئیچ‌های موجود در توپولوژی را دریافت می‌کنیم و آنها را داخل `switch_list` ذخیره می‌کنیم.

```
def get_topology_data(self, ev):
    global switches
    switch_list = get_switch(self.topology_api_app, None)
```

سپس روی `switch_list` پیش می‌رویم و متغیر `id` مربوط به `datapath` هر سوئیچ را در لیست گلوبال `switches` ذخیره می‌کنیم.

```
switches=[switch.dp.id for switch in switch_list]
```

در ادامه رو `switch_list` پیش می‌رویم و خصیصه `datapath` هر سوئیچ را در لیست `datapath_list` که از متغیرهای کلاس کنترلر است ذخیره می‌کنیم.

```
self.datapath_list=[switch.dp for switch in switch_list]
```

با استفاده از تابع `get_link` که در کلاس `app_manager.ryuapp` تعریف شده است تمامی لینک‌های موجود در توپولوژی را دریافت می‌کنیم و آنها را داخل `links_list` ذخیره می‌کنیم.

```
links_list = get_link(self.topology_api_app, None)
```

سپس روی `links_list` پیش می‌رویم و به ازای هر لینک، `dpid` ورودی و خروجی آن و پورت ورودی و خروجی آن را در لیست `mylinks` ذخیره می‌کنیم.

```
mylinks=[(link.src.dpid, link.dst.dpid, link.src.port_no, link.dst.port_no)
for link in links_list]
```

در ادامه به پر کردن جدول مجاورت می‌پردازیم. هدف از این جدول این است که پورتهای که هر سوئیچ به آن وارد لینک شده است را بدانیم. بدین منظور روی لیست `mylinks` پیش می‌رویم. به ازای هر لینک موجود در این لیست دو خانه از `adjacency` را پر می‌کنیم. در ردیف `dpid` مبدأ و ستون `dpid` مقصد پورت مبدأ و در ستون `dpid` مقصد و ردیف `dpid` مبدأ پورت مقصد را قرار می‌دهیم. از این آرایه در الگوریتم دایکسترا استفاده می‌کنیم.

```
for s1,s2,port1,port2 in mylinks:
    adjacency[s1][s2]=port1
    adjacency[s2][s1]=port2
```

## تابع packet\_in\_handler

این تابع را برای هندل کردن ofp\_event.ofpEventPacketIn تعریف می‌کنیم. این event زمانی رخ می‌دهد که یک پکت جدید که یک flow با اولویت یک مخصوص مبدا و مقصد دریافت شده در flow table موجود نباشد دریافت شود.

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

مانند توضیحات بالا اطلاعات زیر را از event به وجود دریافت می‌کنیم.

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

سپس برای دریافت پورتهای پکت از آن پورت، دریافت شده است (پورت هاست و یا سوئیچ فرستنده) از دستور زیر استفاده می‌کنیم.

```
in_port = msg.match['in_port']
```

سپس یک instance از کلاس packet برای پیامی که از ev دریافت شده است می‌سازیم تا بتوانیم packet دریافت شده که باعث رخ دادن این event شده است را داشته باشیم.

```
pkt = packet.Packet(msg.data)
```

سپس پروتکل پکت دریافت شده را می‌گیریم.

```
eth = pkt.get_protocol(ethernet.ethernet)
```

برای جلوگیری از دریافت پکت‌های LLDP شرط زیر را اضافه می‌کنیم.

```
if eth.ethertype==35020:
    return
```

سپس id هاست یا سوئیچی که پکت از آن ارسال شده و id سوئیچ یا هاستی که مقصد پکت است را از پروتکل ethernet دریافت می‌کنیم.

```
dst = eth.dst
```

```
src = eth.src
```

سپس خصیصه id سوئیچ فرستنده پکت را ذخیره می‌کنیم.

```
dpid = datapath.id
```

متغیر mac\_to\_port که در این کد تعریف شده ولی از آن استفاده‌ای نشده است. Mac address table نیز در mymac ذخیره می‌شود.

```
self.mac_to_port.setdefault(dpid, {})
```

هدف استفاده از mac address table این است که پورتهای که هر سوئیچ (و یا هاست) با آن پیام انتقال می‌دهد را داشته باشیم. دلیل انجام این کار این است که برای فرستادن پیام، به پورت هاست مقصد احتیاج داریم بنابراین هر زمانی که یک پکت از طرف یک سوئیچ (و یا هاست) دریافت می‌شود، کنترلر با ذخیره کردن اطلاعات آن در mymac در مورد آن پورت یاد می‌گیرد تا بعداً هنگام فرستادن پیام به آن از پورت استفاده کند. بنابراین هرگاه که پکت جدیدی دریافت شود چک می‌کنیم که آیا قبلاً در مورد آن یاد گرفته‌ایم و اطلاعات آن داخل mymac هست یا خیر. اگر نبود در خانه src از mymac اطلاعات in\_port و dpid سوئیچ را ذخیره می‌کنیم.

```
if src not in mymac.keys():
    mymac[src] = (dpid, in_port)
```

در ادامه اگر dst در mymac موجود بود و قبلاً در مورد آن یاد گرفته بودیم تابع get\_path که توضیحات آن در ادامه آمده است را صدا می‌زنیم. این تابع کوتاهترین مسیر برای رسیدن به dst را به ما می‌دهد بنابراین out\_port ما که می‌شود پورتهای که باید پیام دریافت شده را به آن بفرستیم می‌شود سوئیچ بعدی داخل مسیر گرفته شده می‌باشد.

```
if dst in mymac.keys():
    print(mymac)
    p = get_path(mymac[src][0], mymac[dst][0], mymac[src][1],
mymac[dst][1])
```

تابع install\_path به اضافه کردن یک flow اختصاصی برای مسیر بین src و dst می‌پردازد که priority این flow یک است.

```
self.install_path(p, ev, src, dst)
```

خروجی تابع get\_path یک لیست از ست‌ها می‌باشد که هر ست دارای اطلاعات id و پورت سوئیچ داخل مسیر می‌باشد. پورت خروجی که می‌خواهیم این پیام را به آن سوئیچ بفرستیم می‌شود پورت سوئیچ بعدی داخل مسیر.

```
out_port = p[0][2]
```



اگر پورت خروجی از قبل شناخته شده نبود آن را یک instance از کلاس OFPP\_FLOOD قرار می‌دهیم. که OFPP\_FLOOD یعنی تمام پورت‌های فیزیکی به جز پورت ورودی و آن پورت‌هایی که توسط پروتکل spanning tree حذف شده‌اند.

```
out_port = ofproto.OFPP_FLOOD
```

سپس اکشن OFPActionOutPut را که می‌شود اکشن برای فرستادن خروجی را صدا می‌زنیم و پورت مقصد آن را out\_port قرار می‌دهیم.

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

کد زیر برای اضافه کردن یک flow جدید برای این مسیر می‌باشد که این کار در install\_path انجام شده است و نیازی به کد زیر نیست.

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

در ادامه data پیام دریافت شده از event را ذخیره می‌کنیم.

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

Out را که پیام نهایی است که می‌خواهیم ارسال کنیم را تعریف می‌کنیم. این متغیر یک instance از کلاس OFPPacketOut است که مخصوص فرستادن یک پکت به سوئیچ دیگر است. متغیرهای استفاده شده در آرگومان‌های این تابع در قبل توضیح داده شده‌اند.

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

سپس datapath سوئیچی که پیام از آن فرستاده شده بود، که وظیفه آن برقراری ارتباط با سوئیچ‌های دیگر است، out را ارسال می‌کند.

```
datapath.send_msg(out)
```

## تابع get\_path

ابتدا دیکشنری‌های distance (که در آن کوتاهترین فاصله هر سوئیچ از سوئیچ مبدا) و previous (که در آن سوئیچ قبلی هر سوئیچ در کوتاهترین مسیر از مبدا به آن سوئیچ) هستند را تعریف می‌کنیم. مقادیر اولیه distance تمامی سوئیچ‌ها به جز سوئیچ مبدا را برابر با بی‌نهایت قرار می‌دهیم. مقدار اولیه سوئیچ مبدا برابر با صفر خواهد بود. مقادیر اولیه previous تمامی سوئیچ‌ها را None قرار می‌دهیم.

```
distance = {}
previous = {}
for dpid in switches:
    distance[dpid] = float('Inf')
    previous[dpid] = None
distance[src]=0
```

ست سوئیچ‌ها را داخل متغیر Q قرار می‌دهیم.

```
Q=set(switches)
```

تابع minimum\_distance سوئیچی که در لیست Q کمترین فاصله با مبدا را دارد را برمی‌گرداند. آن سوئیچ را از Q حذف می‌کنیم تا آن سوئیچ دوباره به عنوان کمترین فاصله انتخاب نشود.

```
while len(Q)>0:
    u = minimum_distance(distance, Q)
    Q.remove(u)
```

سپس روی تمامی سوئیچ‌های شبکه پیش می‌رویم و آن‌هایی که یال با سوئیچ u (که در این مرحله نزدیکترین سوئیچ موجود در) و مقادیر distance آن‌ها را آپدیت می‌کنیم. به این صورت که اگر مجموع مقدار حال distance سوئیچ u و وزن یال بین u و آن سوئیچ کمتر از مقدار حال distance سوئیچ بود مقدار آن را به مجموع مقدار حال distance سوئیچ u و وزن یال بین u و آن سوئیچ آپدیت می‌کنیم. Previous این سوئیچ نیز به سوئیچ u تغییر می‌کند.

```
for p in switches:
    if adjacency[u][p] != None:
        w = 1
        if distance[u] + w < distance[p]:
            distance[p] = distance[u] + w
            previous[p] = u
```

دستورات نوشته شده در while را تا زمانی که Q خالی شود انجام می‌دهیم.

حال که `previous` هر سوئیچ را به طوری که آن سوئیچ دارای کوتاهترین فاصله از مبدا شود را داریم، از `dst` به عقب می‌رویم تا به سوئیچ `src` برسیم. از هر سوئیچ که می‌گذریم آن را در لیست `r` ذخیره می‌کنیم و به این ترتیب کوتاهترین مسیر از مقصد به مبدا را بدست می‌آوریم. در نهایت برای بدست آوردن مسیر از مبدا به مقصد، این لیست را برعکس می‌کنیم.

```
r=[]
p=dst
r.append(p)
q=previous[p]
while q is not None:
    if q == src:
        r.append(q)
        break
    p=q
    r.append(p)
    q=previous[p]
r.reverse()
```

اگر مبدا همان مقصد بود وارد `while` نمی‌شویم بنابراین مسیر همان `src` است.

```
if src==dst:
    path=[src]
else:
    path=r
```

`in_port` می‌شود پورته‌ای که یک سوئیچ با آن پورت با سوئیچ یا هاست قبلی خود در مسیر ارتباط دارد و `out_port` می‌شود پورته‌ای که یک سوئیچ با آن با سوئیچ و یا هاست بعدی خود در مسیر ارتباط دارد.

```
r = []
in_port = first_port
for s1,s2 in zip(path[:-1],path[1:]):
    out_port = adjacency[s1][s2]
    r.append((s1,in_port,out_port))
    in_port = adjacency[s2][s1]
r.append((dst,in_port,final_port))
return r
```

تابع `install_path`

هدف این تابع این است که هنگامی یک پکت مبدا و مقصد مشخص دریافت می‌شود و برای آن یک مسیر پیدا می‌کنیم، یک `flow` جدید با اولویت بیشتر برای آن مسیر به جدول `flow table` اضافه کنیم تا دفعه بعد که دوباره پکت با همین مبدا و مقصدی دریافت شد از این `flow` جدید استفاده کنیم. ورودی‌های این تابع به ترتیب کوتاهترین مسیر، `event` دریافت شده در تابع `packet_in_handler`، پورت سوئیچ مبدا و پورت سوئیچ مقصد می‌باشد.

```
def install_path(self, p, ev, src_mac, dst_mac):
```

توضیحات در بالا آمده است.

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

سپس روی تمامی سوئیچ‌های داخل مسیر به ترتیب پیش می‌رویم و عملیات‌های زیر را روی آن‌ها انجام می‌دهیم.

```
for sw, in_port, out_port in p:
```

یک `instance` از کلاس `OFPMatch` تولید می‌کنیم و به آن مشخصات پورت ورودی و خروجی مسیر را می‌دهیم. بنابراین تنها پکت‌هایی می‌توانند از `flow` استفاده کنند که دارای این پورت‌های ورودی و خروجی باشند.

```
match=parser.OFPMatch(in_port=in_port, eth_src=src_mac, eth_dst=dst_mac)
```

سپس اکشن `OFPACTIONOutput` را که می‌شود اکشن برای فرستادن خروجی را صدا می‌زنیم و پورت مقصد آن را `out_port` قرار می‌دهیم.

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

برای فرستادن پیام، به `datapath` سوئیچی که اکنون با آن داخل حلقه `for` هستیم نیازمندیم. در تابع `get_topology_data` تمامی `datapath` سوئیچ‌های موجود در شبکه را به ترتیب `id` آن سوئیچ ذخیره کردیم. بنابراین اگر `id` سوئیچی که اکنون با آن داخل مسیر هستیم را به `datapath_list` بدهیم، می‌توانیم `datapath` آن سوئیچ را دریافت کنیم. تنها تفاوت این است که `id` سوئیچ‌ها از یک شروع می‌شود در صورتی که برای `indexing` آرایه باید از صفر شروع کنیم. برای همین یک واحد از `id` کم می‌کنیم.

```
datapath=self.datapath_list[int(sw)-1]
```

سپس تابع `OFPIInstructionAction` از کلاس `parser` را فراخوانی می‌کنیم. این تابع مشخص می‌کند که چه عملیاتی روی لیست `actions` تعریف شده در خط بالا انجام شود. در کد زیر اکشن‌های موجود در لیست `actions` اجرا می‌شوند.

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

در ادامه یک `instance` از کلاس `OFPFFlowMod` تعریف می‌کنیم. برای اضافه کردن یک `flow` جدید جنس ورودی تابع `send_msg` از این کلاس است. با این کلاس می‌توانیم اطلاعاتی که می‌خواهیم انتقال دهیم را بسته‌بندی کنیم. توضیحات ورودی‌های این تابع در بالاتر آمده است. برای این `flow` جدید از `priority` یک استفاده می‌کنیم تا ابتدا `flow` های مخصوص هر مسیر بررسی شوند و اگر `flow` مخصوصی وجود نداشت به سراغ `flow miss table` برویم.

```
mod = datapath.ofproto_parser.OFPFFlowMod(datapath=datapath, match=match,
idle_timeout=0, hard_timeout=0,priority=1, instructions=inst)
```

یکی از متدهای کلاس `datapath` ، `switch` می‌باشد. این تابع یک پیام `openflow` را می‌فرستد.

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

## تابع add\_flow

تابع add\_flow در این کد استفاده نشده است و وظیفه آن در تابع install\_path انجام می‌شود. که توضیحات خط به خط آن در بالا آمده است. در واقع تابع add\_flow زیرمجموعه‌ای از کارهایی است که در تابع install\_path انجام می‌شود.

```
def add_flow(self, datapath, in_port, dst, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    match = datapath.ofproto_parser.OFPMatch(in_port=in_port, eth_dst=dst)
    inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
    mod = datapath.ofproto_parser.OFPFlowMod(datapath=datapath, match=match,
cookie=0, command=ofproto.OFPFC_ADD, idle_timeout=0,
hard_timeout=0, priority=ofproto.OFP_DEFAULT_PRIORITY, instructions=inst)
    datapath.send_msg(mod)
```

## خطاهای رفع شده

1. تابع `get_switch` استفاده شده در `get_topology_data` الزاما سوئیچ‌ها را بر اساس `dpid` آنها مرتب نمی‌کند، بنابراین در نتیجه `datapath_list` نیز بر اساس `dpid` مرتب نخواهد شد در صورتی که در خط زیر از تابع `install_path` فرض شده که `datapath_list` بر اساس `dpid` سوئیچ‌ها مرتب شده است زیرا از `id` سوئیچ داخل مسیر برای `indexing` در `datapath_list` استفاده شده است.

```
datapath=self.datapath_list[int(sw)-1]
```

بنابراین لازم است در تابع `get_topology_data` این لیست را بر اساس `dpid` مرتب کنیم. برای این کار خط زیر را بعد مقدار دهی `datapath_list` به کد اضافه می‌کنیم.

```
self.datapath_list = sorted(self.datapath_list, key=lambda x:x.id,
reverse=False)
```

2. در اجرای اولیه کد بسیاری خطای `spacing` داشت که با حذف و اضافه کردن `tab` این مشکل رفع شد.

## توپولوژی

ابتدا توپولوژی tree,3 را که در لینک داده شده در صورت پروژه بود اجرا کردیم. خروجی دستور pingall برای این توپولوژی به صورت زیر بود.

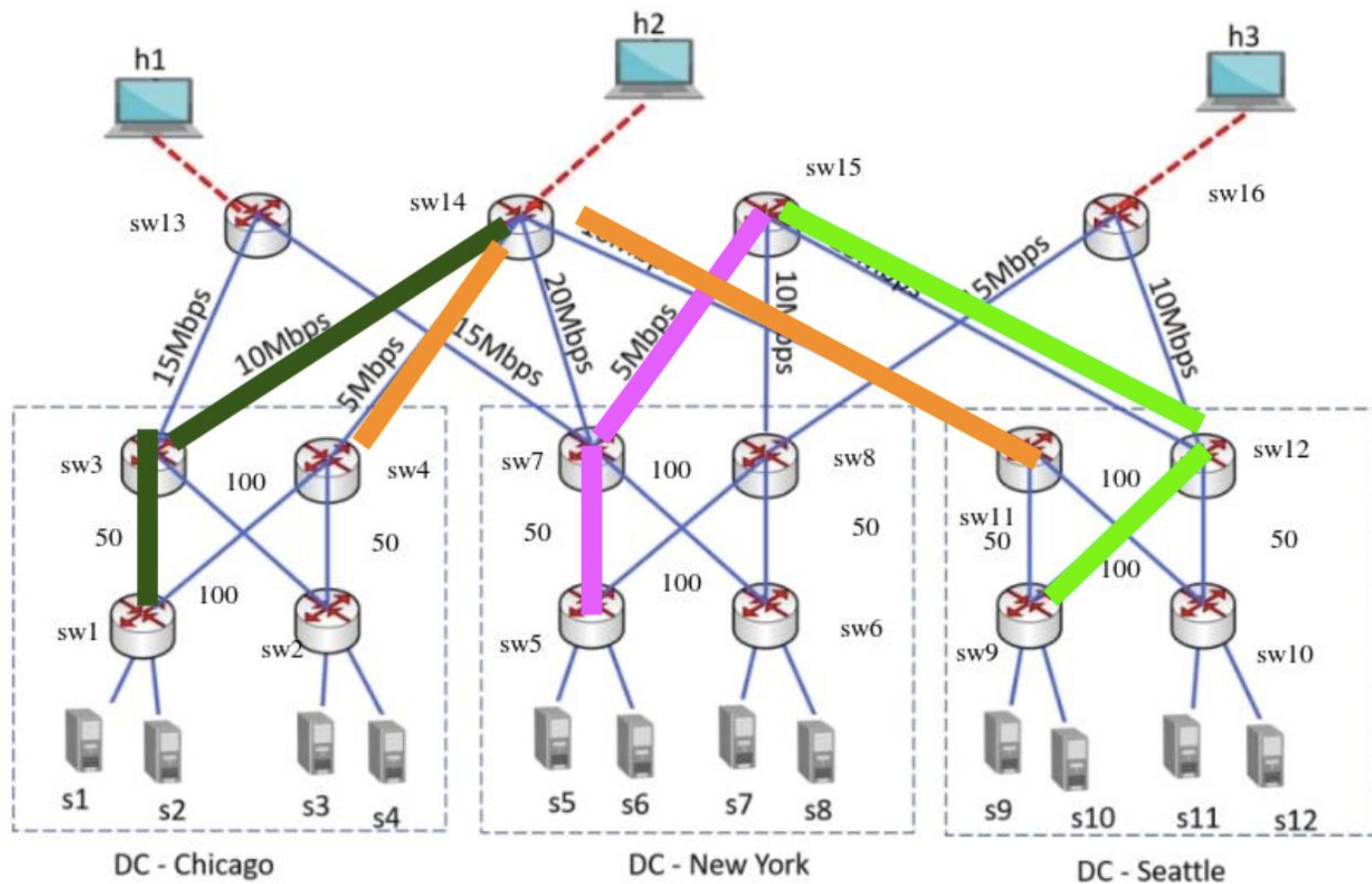
```

parallels@parallels-Parallels-Virtual-Platform: ~/Desktop/Parallels Shared Folders/Home/Desktop
File Edit View Search Terminal Help
pkill -9 -f .ssh/mn
rm -f ~/.ssh/mn/*
*** Cleanup complete.
parallels@parallels-Parallels-Virtual-Platform:~/Desktop/Parallels Shared Folders/Home/Desktop$ sudo mn --topo tree,3 --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7
*** Adding links:
(s1, s2) (s1, s5) (s2, s3) (s2, s4) (s3, h1) (s3, h2) (s4, h3) (s4, h4) (s5, s6) (s5, s7) (s6, h5) (s6, h6) (s7, h7) (s7, h8)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8
*** Starting controller
c0
*** Starting 7 switches
s1 s2 s3 s4 s5 s6 s7 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 X h4 h5 h6 h7 h8
h2 -> h1 h3 h4 h5 h6 X h8
h3 -> h1 h2 h4 h5 h6 h7 h8
h4 -> h1 h2 h3 h5 h6 h7 h8
h5 -> h1 h2 h3 h4 h6 X X
h6 -> h1 h2 h3 h4 h5 X X
h7 -> h1 h2 h3 h4 h5 h6 h8
h8 -> h1 h2 h3 h4 h5 h6 h7
*** Results: 10% dropped (50/56 received)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8
h2 -> h1 h3 h4 h5 h6 h7 h8
h3 -> h1 h2 h4 h5 h6 h7 h8
h4 -> h1 h2 h3 h5 h6 h7 h8
h5 -> h1 h2 h3 h4 h6 h7 h8
h6 -> h1 h2 h3 h4 h5 h7 h8
h7 -> h1 h2 h3 h4 h5 h6 h8
h8 -> h1 h2 h3 h4 h5 h6 h7
*** Results: 0% dropped (56/56 received)
mininet>

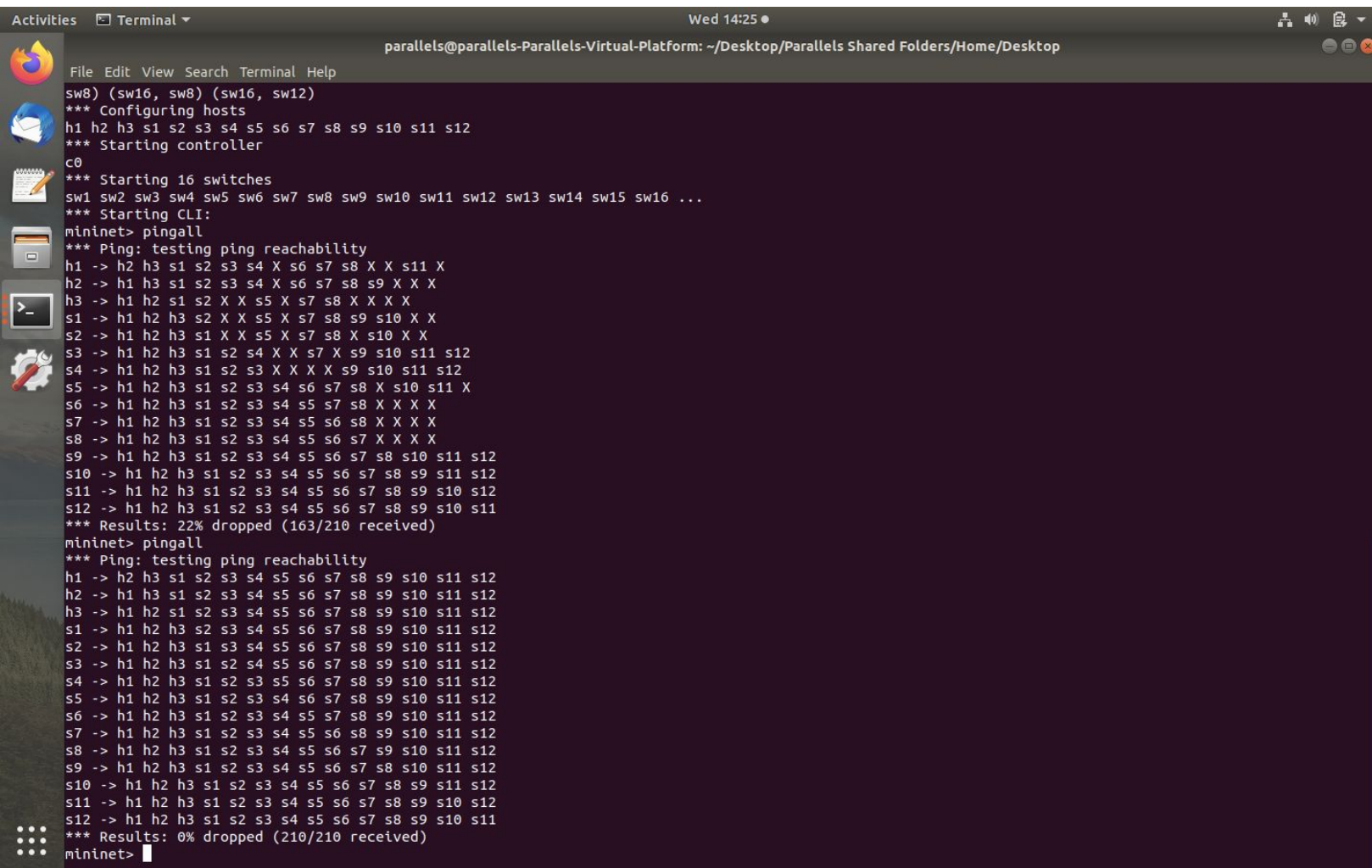
```

سپس برای تبدیل کردن توپولوژی داده شده در صورت پروژه به درخت، یال‌های مشخص شده در تصویر را حذف کردیم (یال‌های رنگی در تصویر یال‌های حذف شده می‌باشند). ۳۸ یال و ۳۱ راس داریم بنابراین برای تبدیل کردن گراف به درخت باید حداقل ۱ + (۳۸ - ۳۱) یال حذف کنیم).





سپس در فایل `topo.py` کد این توپولوژی را پیاده سازی می‌کنیم. بدین منظور یک کلاس با نام `MyTopo` از کلاس `Topo` ارث بری می‌کنیم و در تابع `init` آن سوئیچ، هاست و لینک‌های این توپولوژی درختی را اضافه می‌کنیم. برای این کار از توابع `addHost` و `addSwitch` استفاده می‌کنیم. در انتها نیز کلاس این کلاس را فراخوانی می‌کنیم. خروجی این توپولوژی به صورت زیر است.



```

Activities  Terminal  Wed 14:25  parallels@parallels-Parallels-Virtual-Platform: ~/Desktop/Parallels Shared Folders/Home/Desktop

File Edit View Search Terminal Help
sw8) (sw16, sw8) (sw16, sw12)
*** Configuring hosts
h1 h2 h3 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12
*** Starting controller
c0
*** Starting 16 switches
sw1 sw2 sw3 sw4 sw5 sw6 sw7 sw8 sw9 sw10 sw11 sw12 sw13 sw14 sw15 sw16 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 s1 s2 s3 s4 X s6 s7 s8 X X s11 X
h2 -> h1 h3 s1 s2 s3 s4 X s6 s7 s8 s9 X X X
h3 -> h1 h2 s1 s2 X X s5 X s7 s8 X X X
s1 -> h1 h2 h3 s2 X X s5 X s7 s8 s9 s10 X X
s2 -> h1 h2 h3 s1 X X s5 X s7 s8 X s10 X X
s3 -> h1 h2 h3 s1 s2 s4 X X s7 X s9 s10 s11 s12
s4 -> h1 h2 h3 s1 s2 s3 X X X s9 s10 s11 s12
s5 -> h1 h2 h3 s1 s2 s3 s4 s6 s7 s8 X s10 s11 X
s6 -> h1 h2 h3 s1 s2 s3 s4 s5 s7 s8 X X X
s7 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s8 X X X
s8 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s7 X X X
s9 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s7 s8 s10 s11 s12
s10 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s7 s8 s9 s11 s12
s11 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s12
s12 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11
*** Results: 22% dropped (163/210 received)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12
h2 -> h1 h3 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12
h3 -> h1 h2 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12
s1 -> h1 h2 h3 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12
s2 -> h1 h2 h3 s1 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12
s3 -> h1 h2 h3 s1 s2 s4 s5 s6 s7 s8 s9 s10 s11 s12
s4 -> h1 h2 h3 s1 s2 s3 s5 s6 s7 s8 s9 s10 s11 s12
s5 -> h1 h2 h3 s1 s2 s3 s4 s6 s7 s8 s9 s10 s11 s12
s6 -> h1 h2 h3 s1 s2 s3 s4 s5 s7 s8 s9 s10 s11 s12
s7 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s8 s9 s10 s11 s12
s8 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s7 s9 s10 s11 s12
s9 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s7 s8 s10 s11 s12
s10 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s7 s8 s9 s11 s12
s11 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s12
s12 -> h1 h2 h3 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11
*** Results: 0% dropped (210/210 received)
mininet>

```



## Switch\_features

```

Activities  Terminal  Wed 18:43  parallels@parallels-Parallels-Virtual-Platform: ~/Desktop/Parallels Shared Folders/Home/Desktop
File Edit View Search Terminal Help
parallels@parallels-Parallels-Virtual-Platform:~/Desktop/Parallels Shared Folders/Home/Desktop$ ryu-manager code.py --observe-links
loading app code.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app ryu.topology.switches of Switches
instantiating app code.py of ProjectController
init is called
instantiating app ryu.controller.ofp_handler of OFPHandler
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=15,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=8,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=14,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=2,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=7,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=13,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=4,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=9,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=10,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=11,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=1,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=5,n_buffers=0,n_tables=254))
msg is sent
switch_features_handler is called
("Event's msg: ", OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=3,n_buffers=0,n_tables=254))

```



The image shows a terminal window with a dark background and light-colored text. The title bar at the top reads "parallels@parallels-Parallels-Virtual-Platform: ~/Desktop/Parallels Shared Folders/Home/Desktop". On the left side, there is a vertical sidebar with icons for "Activities", "Terminal", and several application icons (Firefox, LibreOffice Writer, LibreOffice Calc, LibreOffice Impress, and a gear icon). The terminal content shows a script that repeatedly calls "get\_topology\_data" and prints out a large list of network links. The output is truncated on the right side of the image. The script appears to be a loop that calls "get\_topology\_data" and prints the result. The output shows a list of links in a specific format, including node IDs and link types. The output is truncated on the right side of the image.

