

مسیریابی در شبکه SDN به کمک RYU

1399.05.29

شماره دانشجویی

810196667

810196531

نام و نام خانوادگی

سپهر رزمیار

نعیم قهرمانپور

مقدمه

در این پروژه سعی شده است که با استفاده از الگوریتم `dijkstra` و `RYU` یک کنترلر بنویسیم و با استفاده از `MININET` یک شبکه تشکیل دهیم و آن کنترلر را در آن شبکه تست کنیم.

اهداف پروژه

1. بررسی نحوه ی تشکیل توپولوژی شبکه
2. آشنایی با نحوه ی برقراری ارتباط میان کنترلر `RYU` و `MININET`
3. یافتن کوتاه ترین مسیر برای ارسال بسته از یک نود به نود دیگر

فرض گرفته شده در پروژه

فرض کردیم که وزن مسیر ها در شبکه با هم برابر و مقدار آنان یک می باشد زیرا با وجود درخت بودن توپولوژی داشتن وزن در یال ها بی معنی می باشد.

شرح کدهای پروژه

ا. تابع `minimum_distance`:

ورودی های تابع:

- **Distance** ← لیستی از فاصله هر راس نسبت به راس مبدا می باشد.
- **Q** ← لیستی از راس های دیده نشده توسط راس مبدا در شبکه.

خروجی تابع:

- **node** ← راسی که در لیست راس های دیده نشده کمترین فاصله را نسبت به راس مبدا داشته باشد

در این تابع راسی را پیدا می کنیم که کمترین فاصله را نسبت به راس مبدا داشته باشد.

ا.ا. تابع `get_path`:

ورودی های تابع:

- **src** ← راس مبدايي که بسته از آن ارسال می شود.
- **dst** ← راس نهايي که بسته به آن فرستاده می شود.

- **first_port** ← پورت ورودی سوئیچ که هاست بسته را به سوئیچ متصل به خود ارسال می کند.
- **final_port** ← پورت نهایی که سوئیچ مقصد بسته را به هاست مقصد از آن ارسال می کند.

خروجی های تابع:

- **r** ← لیستی از آیدی سوئیچ ها و پورت ورودی و خروجی آنان که نشان دهنده کوتاهترین مسیر از مبدا به مقصد می باشد.

در این تابع با توجه به راس های (سوئیچ های) ورودی و خروجی الگوریتم دایکسترا را بر روی راس های (سوئیچ های) شبکه اجرا کرده و در نهایت بعد از اینکه به کمک الگوریتم دایکسترا فاصله از سوئیچ را نسبت به راس مبدا پیدا کردیم، کوتاهترین مسیر از مبدا به مقصد را بدست می آوریم و بعد پورت ورودی و خروجی را به آن اضافه می کنیم.

III. کلاس ProjectController:

این کلاس، کنترلرمان را تشکیل می دهد، و از `app_manager.RyuApp` ارث می برد. توابع این کلاس را توضیح خواهیم داد. این کلاس یک ممبر دارد به نام `OF_VERSIONS`، که مشخص می کند چه ورژن هایی از `OpenFlow` را کنترلر ما پشتیبانی می کند.

ما فرض کردیم که برای هر نودی، `uniDirectional Neighbors` هم عضو همسایه ها آن نود است چون هر همسایه `uniDirectional` پس از مدتی یا به همسایه `biDirectional` تبدیل می شود یا پس از مدتی چون پکتی از آن نود د

IV. تابع `ProjectController._init`:

در این کانستراکتر، ابتدا کانستراکتر `app_manager.RyuApp` را صدا می زنیم تا کلاسمان `initialize` شود، سپس سه `object member` تعریف می کنیم و مقدار اولیه به آن ها می دهیم.

- **Mac_to_port** ← یک `dictionary` است که درون آن یک `dictionary` دیگر است که `value` آن `Int` است. به عنوان مثال `mac_to_port[2][00:00:00:00:00:01] = 2` به این معنی است که در سوئیچ شماره ۲، `host` با `mac address 00:00:00:00:00:01` به پرت ۲ این سوئیچ وصل است.
- **Topology_api_app** ← از این برای گرفتن `topology` استفاده می شود.
- **Datapath_dict** ← یک `dictionary` است که `key` آن شماره سوئیچ است و `value` آن `object` آن سوئیچ است.

V. تابع `ProjectController.switch_features_handler`:

ورودی های تابع:

- **ev** ← همان `event` می باشد که یک نمونه از `ryu.controller.ofp_event.EventOFPSwitchFeatures`

این تابع زمانی صدا زده می شود که (Switch Features(Features Reply به دست کنترلر می رسد، چون

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

تعیین می کند که پس از اینکه سوئیچ به کنترلر وصل شد، کنترلر منتظر Features های آن سوئیچ می باشد. CONFIG_DISPATCHER به این معنی است که کنترلر منتظر Features های آن سوئیچ می باشد).

پس از طریق ev، سوئیچ ای که به کنترلر وصل شده است را در می یابیم و سپس می خواهیم در flow table این سوئیچ یک flow اضافه کنیم، که این flow زمانی به کار می رود که پکتان با هیچ یک از flow entry های سوئیچ مان match نشود، از این رو EventOFPPacketIn ایجاد می شود (که بعدا توضیح داده می شود) و این پکت به کنترلرمان می رود تا مسیر به مقصد مشخص شود.

برای اینکه این flow مان همه ی پکت ها را match کند به parser.OFPMatch هیچ پارامتری نمی دهیم و برای اینکه پکت به کنترلر برود به parser.OFPACTIONOutput پارامتر ofproto.OFPP_CONTROLLER می دهیم و برای اینکه کل پکت یک جا فرستاده شود، پارامتر ofproto.OFPCML_NO_BUFFER را نیز می دهیم. سپس از تابع کمکی add_flow استفاده می کنیم. به این flow کمترین الویت را می دهیم تا اگر flow دیگری بود که می توانست آن پکت را match کند، بتواند match کند.

VI. تابع ProjectController.add_flow:

ورودی های تابع:

- **Datapath** ← سوئیچی که می خواهیم به آن flow را اضافه کنیم.
- **priority** ← الویت flow ای که می خواهیم اضافه کنیم.
- **Match** ← مشخص می کند چه پکت هایی را match می کند این flow.
- **Actions** ← مشخص می کند چه action هایی باید انجام شود در این flow، مثلا به چه پرتی پکت برود.
- **Buffer_id** ← همیشه none است.

چون در OpenFlow ها با ورژن بالاتر از v1.2 نیاز است action ها را به instructions تبدیل کنیم، این کار را به کمک

```
parser.OFPIInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)
```

انجام می دهیم، سپس با کمک parser.OFPFlowMod، پیغام flowmod را می سازیم و به سوئیچ ارسال می کنیم.

VII. تابع ProjectController.packet_in_handler:

ورودی های تابع:

- **ev** ← همان event می باشد که یک نمونه از ryu.controller.ofp_event.EventOFPPacketIn

همان طور که توضیح داده شده بود، زمانی که پکتی که سوئیچ مقصد را نداند، به کنترلر می رود (توسط flow ای که به سوئیچ ها در تابع switch_features_handler اضافه کردیم). و EventOFPPacketIn ایجاد می شود و این تابع صدا زده می شود. چون

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

تعیین می کند که پس از اینکه ofp_event.EventOFPPacketIn ایجاد شد، این تابع صدا زده می شود. (MAIN_DISPATCHER یعنی وضعیت نرمال است و MAIN_DISPATCHER بعد از CONFIG_DISPATCHER است).

پس از طریق ev، سوئیچ ای که به کنترلر وصل شده است را در می یابیم و سپس پرت ورودی پکت را می گیریم. از طریق پارس کردن پکت می توان نوع ethertype پکت را دریافت و اگر LLDP بود، پکت را ایگنور می کنیم (چون فقط کنترلر باید Flood کند). اگر mac_to_port برای این سوئیچ خالی بود آن را initialize می کنیم و مقدار self.mac_to_port[dpid][src] = in_port می دهیم (dpid همان id سوئیچ است و src همان mac آدرس host مبدا است). اگر mymac کلیدی از host مبدا نداشت، به آن مقدار

```
mymac[src] = (dpid, in_port)
```

می دهیم.

اکنون باید چک کنیم آیا برای این سوئیچ می دانیم که host مقصد به چه پرتی از سوئیچ راه دارد، اگر ندانیم که Flood می کنیم، اگر بدانیم باید کوتاه ترین مسیر به مقصد را پیدا کنیم و سپس در flow table های این سوئیچ ها flow مناسب را قرار دهیم (flow هایی که پکت ها را به درستی match کنند و به درستی forward کنند). اکنون باید به کمک mac_to_port، پرتی که host مقصد به آن سوئیچ راه دارد را به عنوان out_port تعیین می کنیم (چون توپولوژیمان درخت است، بین هر دو راس یک مسیر است، پس mac_to_port همیشه پرت درستی می دهد). سپس به parser.OFPActionOutput

پارامتر out_port را می دهیم تا به این پرت (ها) forward شود.

چون ما از buffering استفاده نکردیم، باید یک جا msg.data را به parser.OFPPacketOut بدهیم و سپس پکت را ارسال کنیم.

VIII. تابع ProjectController.install_path:

ورودی های تابع:

- p ← لیست کوتاهترین مسیر از راس مبدا به راس مقصد.
- ev ← همان event می باشد که یک نمونه از
- ryu.controller.ofp_event.EventOFPSwitchFeaturesr
- src_mac ← آدرس MAC سوئیچ فرستنده.
- dst_mac ← آدرس MAC سوئیچ گیرنده.

این تابع بعد از اینکه مسیر کوتاه از راس فرستنده به گیرنده مشخص شد صدا زده می شود تا اطلاعات پورت خروجی و ورودی برای این MAC آدرس ها برای هر سوئیچ مشخص شود یعنی ما از آنجایی که کوتاهترین مسیر را برای این MAC آدرس ها داریم، این اطلاعات را در جدول flow ذخیره می کنیم با صدا زدن تابع add_flow.

IX. تابع ProjectController._packet_in_handler:

ورودی های تابع:

• **ev** ← همان event می باشد که یک نمونه از `.ryu.controller.ofp_event.EventOFPSwitchFeaturesr`

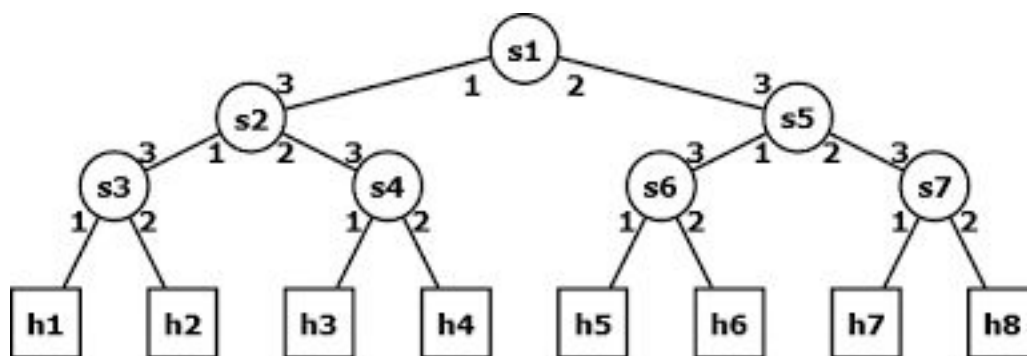
X. تابع `ProjectController.get_topology_data`:

ورودی های تابع:

• **ev** ← همان event می باشد که یک نمونه از `.ryu.controller.ofp_event.EventOFPSwitchFeaturesr`

این تابع زمانی صدا زده می شود که `EventSwitchEnter` اتفاق بیفتد در این تابع به کمک توابع `get_switch` و `get_link` اطلاعات در مورد شبکه (به ترتیب لیستی از سوئیچ ها شبکه و لیستی از اتصالات مربوط به شبکه) را از `MININET` گرفته و یک ماتریس مجاورت می سازیم که نشان دهنده این است که سوئیچ با کدام پورت به سوئیچ همسایه با پورت مشخص متصل می شود.

توپولوژی شبکه بررسی شده در این پروژه



در این شکل تنها تفاوتی که با توپولوژی اصلی در شبکه وجود دارد این می باشد که در سوئیچ شماره هفت جای پورت های یک و دو با شکل جابه جا می باشد.

تحلیل اجرای پروژه

برای اجرای پروژه و دیباگ کردن آن از دستورات زیر استفاده می کنیم:

• `sudo mn --topo tree,3 --mac --switch ovsk --controller remote -x`

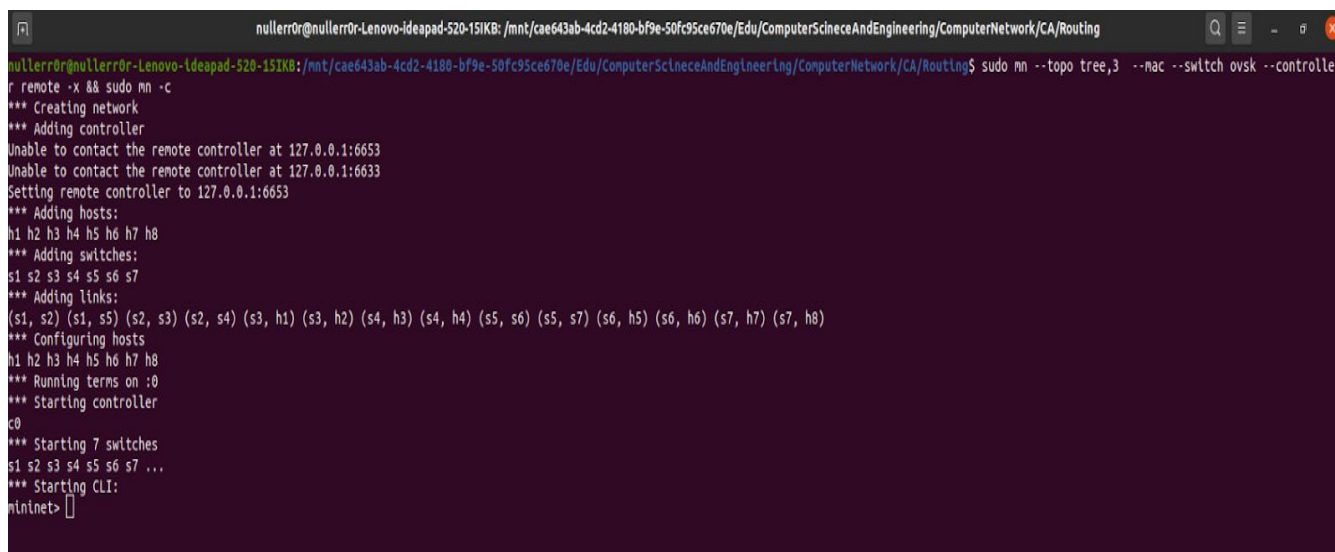
در این دستور MININET را اجرا می کنیم و با استفاده از option های topo tree -- توپولوژی شبکه را به صورت درخت درمی آوریم که شکل بالا حاصل می شود (البته عدد نوشته شده در دستور نشان دهنده تعداد لول ها در درخت می باشد.) در دستور بالا مقدار controller remote -- نشان دهنده این است که از یک کنترلر خارجی استفاده می کنیم. x- نشان دهنده این است که به هنگام اجرای MININET، برای هر یک از سوئیچ ها و هاست ها و کنترلر یک پنجره xterm باز می شود.

در صورتی که بخواهیم از توپولوژی که خودمان ساخته ایم استفاده کنیم می توان به جای دستور بالا از دستور زیر استفاده کرد:

```
sudo mn --custom ./customTopo.py --mac --switch ovsk --controller remote -x
ryu-manager --verbose ryu/app/controller.py --observe-links •
```

این دستور را در پنجره controller:c0 اجرا می کنیم و با این دستور کنترلر RYU را اجرا می کنیم که همان فایل controloer.py می باشد.

شکل های زیر نتایج گفته شده در بالا می باشند.



```

nullerr0r@nullerr0r-Lenovo-ideapad-520-151KB: /mnt/cae643ab-4cd2-4180-bf9e-50fc95ce670e/Edu/ComputerScienceAndEngineering/ComputerNetwork/CA/Networking$ sudo mn --topo tree,3 --mac --switch ovsk --controller remote -x && sudo mn -c
*** 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
*** Running terms on :0
*** Starting controller
c0
*** Starting 7 switches
s1 s2 s3 s4 s5 s6 s7 ...
*** Starting CLI:
mininet>

```



```

root@nullr0r-Lenovo-ideapad-520-151KB:/mnt/cae643ab-4cd2-4180-bf9e-50fc95ce670e/ComputerScienceAndEngineering/ComputerNetwork/CA/Networking# cd ryu
root@nullr0r-Lenovo-ideapad-520-151KB:/mnt/cae643ab-4cd2-4180-bf9e-50fc95ce670e/ComputerScienceAndEngineering/ComputerNetwork/CA/Networking/ryu# ryu-manager --verbose ryu/app/controller.py --observe-links
loading app ryu/app/controller.py
require_app: ryu.topology.switches is required by controller
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app ryu/app/controller.py of ProjectController
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK ProjectController
  CONSUMES EventOFPPacketIn
  CONSUMES EventSwitchEnter
  CONSUMES EventOFPSwitchFeatures
BRICK switches
  PROVIDES EventSwitchEnter TO {'ProjectController': 'set()'}
  CONSUMES EventOFPPacketIn
  CONSUMES EventHostRequest
  CONSUMES EventLinkRequest
  CONSUMES EventOFPPortStatus
  CONSUMES EventOFPSwitchFeatures
  CONSUMES EventSwitchRequest
BRICK ofp_event
  PROVIDES EventOFPPacketIn TO {'ProjectController': {'main'}, 'switches': {'main'}}
  PROVIDES EventOFPSwitchFeatures TO {'ProjectController': {'config'}}
  PROVIDES EventOFPPortStatus TO {'switches': {'main'}}
  PROVIDES EventOFPSwitchFeatures TO {'switches': {'dead', 'main'}}
  CONSUMES EventOFPEchoReply
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPErrormsg
  CONSUMES EventOFPHello
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPPortStatus
  CONSUMES EventOFPSwitchFeatures

```

بعد از اتصال کنترلر RYU به MININET لینک ها و سوئیچ ها را از شبکه کنترلر می گیرد و ماتریس مجاورت را بر می کند که شکل زیر این مطلب را نشان می دهد:

[illegible]

حال بعد از گرفتن این اطلاعات سعی می کنیم که چهار بسته از هاست h1 به هاست h8 ارسال کنیم: (به کمک دستور زیر)

```
mininet> h1 ping -c4 h8
```

که اگر در آن لحظه کمی صبر کنیم خروجی زیر را در MININET مشاهده می کنیم:

```
mininet> h1 ping -c4 h8
PING 10.0.0.8 (10.0.0.8) 56(84) bytes of data.
64 bytes from 10.0.0.8: icmp_seq=1 ttl=64 time=54.5 ms
64 bytes from 10.0.0.8: icmp_seq=2 ttl=64 time=0.579 ms
64 bytes from 10.0.0.8: icmp_seq=3 ttl=64 time=0.204 ms
64 bytes from 10.0.0.8: icmp_seq=4 ttl=64 time=0.085 ms

--- 10.0.0.8 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3055ms
```

اگر به شکل نگاه کنیم در میابیم که پکت اول مدت زمانی که طی می کند تا به مقصد برسد بسیار بیشتر از بکت های دیگر می باشد. دلیل این امر این می باشد که در لحظه اول که هاست h1 می خواهد پکت به هاست h8 ارسال نماید در ابتدا MAC آدرس h8 و مسیر را نمی داند به همین دلیل ابتدا یک بسته با نوع ARP که نوع eth.ethertype عدد 2054 می باشد را broadCast می کند و این بسته به همه هاست ها در شبکه ارسال می شود و هاست مورد نظر بعد از دریافت این بسته یک بسته پاسخ با نوع ARP به هاست h1 ارسال می کند. بعد از دریافت این MAC آدرس و با اجرای الگوریتم دایکسترا مسیر را فهمیده در هر سوئیچ در جدول خود ثبت می کند به همین دلیل برای دفعات بعدی زمان ارسال بسته خیلی کم می باشد.

و خط آخر در شکل نشان دهنده تعداد بسته ها ارسالی و تعداد بسته های دریافت شده و میزان packet loss را نشان می دهد.

شکل زیر نشان دهنده برخورد کنترلر با دستور بالا می باشد:

```
EVENT off_event->ProjectController EventOFFPacketIn
EVENT off_event->switches EventOFFPacketIn
_packet_in_handler(self, ev) was called!
packet in 0000000000000007 00:00:00:00:00:08 00:00:00:00:00:01 2
----> 2054 00:00:00:00:00:08 00:00:00:00:00:01
get_path is called!
get_path is called, src= 7 dst= 3 first_port= 2 final_port= 1
Q= {1, 2, 3, 4, 5, 6, 7}
u= 7
port: adjacency from u to p = 3
u= 5
port: adjacency from u to p = 3
port: adjacency from u to p = 1
port: adjacency from u to p = 2
u= 1
port: adjacency from u to p = 2
port: adjacency from u to p = 1
u= 6
port: adjacency from u to p = 3
u= 2
port: adjacency from u to p = 3
port: adjacency from u to p = 2
port: adjacency from u to p = 1
u= 3
port: adjacency from u to p = 3
u= 4
port: adjacency from u to p = 3
[(7, 2, 3), (5, 2, 3), (1, 2, 1), (2, 3, 1), (3, 3, 1)]
install_path is called
00:00:00:00:00:08 -> 00:00:00:00:00:01 via 7 in_port= 2 out_port= 3
add_flow is called!
00:00:00:00:00:08 -> 00:00:00:00:00:01 via 5 in_port= 2 out_port= 3
add_flow is called!
00:00:00:00:00:08 -> 00:00:00:00:00:01 via 1 in_port= 2 out_port= 1
add_flow is called!
00:00:00:00:00:08 -> 00:00:00:00:00:01 via 2 in_port= 3 out_port= 1
add_flow is called!
00:00:00:00:00:08 -> 00:00:00:00:00:01 via 3 in_port= 3 out_port= 1
add_flow is called!
EVENT off_event->ProjectController EventOFFPacketIn
EVENT off_event->switches EventOFFPacketIn
```

h8 → h1 ARP Packet

← shortest Path from h8 to h1

} adding to flowTable

اگر به شکل نگاه کنیم طبق آنچه که در شکل نشان داده شده است کد 2054 نشان داده شده است که مبدا بسته هاست h8 و مقصد آن h1 می باشد که طبق توضیحات بالا این بسته همان بسته پاسخ به بسته ARP ایست که broadcast می شود و چون می خواهیم کوتاهترین مسیر را پیدا کنیم الگوریتم دایکسترا انجام می شود و مسیر کوتاه نشان داده می شود و این سوئیچ ها به همراه پورت های ورودی و خروجی در جدول flow ذخیره می شوند. که اگر دقت کنیم این بسته از سوئیچ شماره هفت که از پورت ورودی دو بسته را گرفته و در نهایت بسته توسط سوئیچ سه و پورت خروجی 1 آن را هاست h1 تحویل می دهد.

بعد از گرفتن بسته پاسخ ARP توسط هاست h1 که شامل MAC آدرس h8 می باشد حال هاست h1 تصمیم به ارسال بسته مورد نظر خود می باشد که شکل زیر نشان دهنده نحوه ی رفتار کنترلر با این موضوع می باشد:

```
EVENT ofp_event->ProjectController EventOFFPacketIn
EVENT ofp_event->switches EventOFFPacketIn
_packet_in_handler(self, ev) was called!
packet in 0000000000000005 00:00:00:00:00:01 00:00:00:00:00:08 3
----> 2048 00:00:00:00:00:01 00:00:00:00:00:08
get_path is called!
get_path is called, src= 3 dst= 7 first_port= 1 final_port= 2
0= {1, 2, 3, 4, 5, 6, 7}
u= 3
port: adjacency from u to p = 3
u= 2
port: adjacency from u to p = 3
port: adjacency from u to p = 2
port: adjacency from u to p = 1
u= 1
port: adjacency from u to p = 2
port: adjacency from u to p = 1
u= 4
port: adjacency from u to p = 3
u= 5
port: adjacency from u to p = 3
port: adjacency from u to p = 1
port: adjacency from u to p = 2
u= 6
port: adjacency from u to p = 3
u= 7
port: adjacency from u to p = 3
[[3, 1, 3), (2, 1, 3), (1, 1, 2), (5, 3, 2), (7, 3, 2)]
install_path is called
00:00:00:00:00:01 -> 00:00:00:00:00:08 via 3 in_port= 1 out_port= 3
add_flow is called!
00:00:00:00:00:01 -> 00:00:00:00:00:08 via 2 in_port= 1 out_port= 3
add_flow is called!
00:00:00:00:00:01 -> 00:00:00:00:00:08 via 1 in_port= 1 out_port= 2
add_flow is called!
00:00:00:00:00:01 -> 00:00:00:00:00:08 via 5 in_port= 3 out_port= 2
add_flow is called!
00:00:00:00:00:01 -> 00:00:00:00:00:08 via 7 in_port= 3 out_port= 2
add_flow is called!
EVENT ofp_event->ProjectController EventOFFPacketIn
```

→ ICMP Packet from h1 to h8

→ Shortest Path

} adding to flow Table

که در شکل بالا کد بسته ارسال از h1 به h8 برابر 2048 می باشد که نشان دهنده بسته ICMP می باشد که هاست h1 آن را به سوئیچ s3 از طریق پورت یک داده و در نهایت سوئیچ s7 بسته را از طریق پورت خروجی دو به هاست h8 ارسال می کند. اگر به قسمت add to flow table توجه کنیم در می یابیم که کنترلر برای این دو MAC آدرس این مسیر را در جدول flow ذخیره می کند.

حال به بررسی برخی از سوئیچ های در مسیر می پردازیم که ببینیم با این دستور چگونه برخورد می کنند:
با استفاده از دستور

```
ovs-ofctl dump-flows <SWITCH NUM>
```

می توان اطلاعات در مورد پکت های ارسال توسط سوکت مورد نظر دریافت کرد.

برای سوئیچ s3 شکل زیر حاصل می شود بعد از دستور ping:

مشخصات 10.0.0.8 می باشد یک پاسخ در نوع ARP به هاست h1 ارسال می کند و بعد از آن هاست h1 یک بسته از نوع ICMP به هاست مقصد که الان MAC address آن را می داند ارسال می کند و در بسته دریافتی می بینیم که h8 یک بسته از نوع ICMP پاسخ به هاست h1 ارسال کرده است که یعنی من بسته را دریافت کردم.