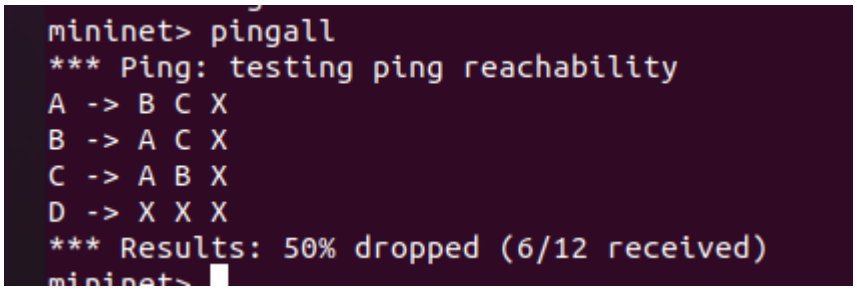
313551137 官劉翔

#Pingall 結果



A、B、C 可以自由連接

因為 ping 是使用 ICMP，而作業要求 Node D has access to A and B by tcp port 22 跟 80，所以使用 ping， D 無法連接到任何其他 host

C、D 不可互聯

實作細節:

實作可以分成這三個步驟

1 環境確認

1.1 確認 Ryu SDN 控制器已正確安裝且可運行

1.2 確認 Mininet 已正確安裝且可運行

2 建立網路拓樸：

2.1 需要建立 4 個節點(A、B、C、D)

2.2 需要建立 3 個交換機(S1、S2、S3)

2.3 需要將這些節點與交換機按照題目圖示連接

3 實現網路策略與修改 ryu：

3.1 策略 1：節點 A、B、C 之間可以自由通訊

3.2 策略 2：節點 D 只能訪問 A 和 B 的 22 端口(SSH)和 80 端口(HTTP)

3.3 策略 3：節點 D 和節點 C 之間禁止所有通訊

1. 環境確認

先按照助教提供的方法安裝 ryu 與 mininet

#啟動 ryu by ryu.app.simple_switch_13 (之後會換成 lab1_Controller.py)

ryu-manager ryu.app.simple_switch_13

啟動 mininet

sudo mn --custom=lab1_topo.py --topo=lab1_topo --controller=remote,ip=127.0.0.1,port=6653 --switch=ovsk,protocols=OpenFlow13

2. 建立網路拓樸

lab1_topology.py

```python
from mininet.topo import Topo

class Lab1_Topo(Topo):
    def __init__(self):
        # Initialize topology
        Topo.__init__(self)

        # Add hosts
        h1 = self.addHost('A')  # Node A
        h2 = self.addHost('B')  # Node B
        h3 = self.addHost('C')  # Node C
        h4 = self.addHost('D')  # Node D

        # Add switches
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        s3 = self.addSwitch('s3')

        # Add links
        self.addLink(h1, s3, port1=1, port2=1)
        self.addLink(h2, s1, port1=1, port2=3)
        self.addLink(h3, s2, port1=1, port2=3)
        self.addLink(h4, s2, port1=1, port2=2)
        self.addLink(s1, s2, port1=2, port2=4)
        self.addLink(s2, s3, port1=1, port2=2)
        self.addLink(s1, s3, port1=1, port2=3)


topos = {'lab1_topo': (lambda: Lab1_Topo())}
```

我使用 Python 建立了一個包含 4 個主機（A、B、C、D）和 3 個交換機（S1、S2、S3）的網路拓撲。

```
        # 添加主機
        h1 = self.addHost('A', ip='10.0.0.1/24', mac='00:00:00:00:00:01')
        h2 = self.addHost('B', ip='10.0.0.2/24', mac='00:00:00:00:00:02')
        h3 = self.addHost('C', ip='10.0.0.3/24', mac='00:00:00:00:00:03')
        h4 = self.addHost('D', ip='10.0.0.4/24', mac='00:00:00:00:00:04')
```

- 使用 addHost() 方法新增主機
- 為每個主機指定固定的 IP 和 MAC 地址
- 主機命名改為有意義的 A、B、C、D，而不是默認的 h1、h2、h3、h4

```
        # 添加交換機
        s1 = self.addSwitch('s1')
        # 添加 link
        self.addLink(h1, s3, port1=1, port2=1)   # A 連接到 S3
```

- 使用 addSwitch() 方法新增三個 OpenFlow 交換機
- 使用 addLink() 方法建立網路連接
- port1 和 port2 參數指定連接的端口號
- 連接配置形成了一個三角形的交換機拓撲，可能存在 cycle 問題

3. 實現網路策略與修改 ryu

為什麼 lab1 需要 STP？

在這個拓撲中，交換機 S1、S2、S3 形成了一個三角形 link：

- S1 ↔ S2
- S2 ↔ S3
- S1 ↔ S3

當我們發送 flood 訊息時，會產生**廣播風暴**

因此需要 STP（生成樹協議）來：

1. 阻塞冗餘鏈路
2. 防止網絡環路
3. 在主要鏈路故障時提供備份路徑

lab1_Controller.py 程式碼解釋

Import:

```
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 import dpid as dpid_lib
from ryu.lib import stplib
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ipv4
from ryu.lib.packet import tcp
from ryu.lib.packet import ether_types
```

我上網查每個模組：

- app_manager：這是 Ryu 應用程序的基礎，我們的控制器要繼承它
- ofp_event 和 set_ev_cls：用來處理 OpenFlow 的事件，比如當封包進入交換機時
- ofproto_v1_3：我們使用 OpenFlow 1.3 版本的協議
- stplib：因為我們的拓撲有環路，需要用 STP 來處理
- packet 相關模組：用來解析不同類型的網路封包

Init:

```python
    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.stp = kwargs['stplib']

        # 配置STP優先級
        config = {dpid_lib.str_to_dpid('0000000000000001'):
                     {'bridge': {'priority': 0x8000}},
                  dpid_lib.str_to_dpid('0000000000000002'):
                     {'bridge': {'priority': 0x9000}},
                  dpid_lib.str_to_dpid('0000000000000003'):
                     {'bridge': {'priority': 0xa000}}}
        self.stp.set_config(config)
```

這裡很重要的是：

1. 我們的控制器繼承了 RyuApp
2. OFP_VERSIONS 指定使用 OpenFlow 1.3
3. _CONTEXTS 啟用了 STP 功能
4. mac_to_port 字典用來記住每個交換機的 MAC 地址表

下一段設定了三個交換機的 STP 優先級。當時想了很久不懂為什麼要設這個，後來明白了：

- 優先級決定哪個交換機會成為 root bridge
- 數字越小優先級越高
- S1 的優先級最高（0x8000），所以它會成為 root bridge
- 這樣可以控制 STP 如何阻斷環路

實際測試時，我發現 STP 確實會阻斷一條鏈路，防止 cycle 發生。但剛開始時我很困惑為什麼網路不通，後來發現是因為 STP 在收斂，需要等待一下下。

Flow rule:

首先，我們需要一個方法來添加 flow rule。這個方法是所有策略實現的基礎：

```python
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)
```

一開始我對這個函數很困惑，後來理解了：

1. datapath 就是交換機
2. priority 決定規則的優先順序
3. match 定義了什麼樣的封包會匹配這條規則
4. actions 定義匹配後要做什麼

重要的一點是 priority（優先級）：較高的數字代表較高的優先級。這對實現我們的策略非常重要，因為我們需要確保某些規則能覆蓋其他 rule。

接下來在 switch_features_handler 底下加入新函數，來符合 lab1 的要求

設置這些規則時，優先級的設置很關鍵：

- 300：D 和 C 的阻止規則（最高優先級）
- 200：允許訪問特定端口的規則
- 100：默認規則
- 1：一般的 MAC 學習規則
- 0：table-miss 規則

```python
    def add_policy_rules(self, datapath):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # 策略2: Node D對ports 22和80的訪問控制
        if datapath.id == 2:  # S2
            # 允許D通過端口2訪問特定服務
            for port in [22, 80]:
                match = parser.OFPMatch(
                    eth_type=0x0800,
                    ip_proto=6,
                    tcp_dst=port
                )
                self.logger.info(f"Only port: {port} can access to D")
                # 允許轉發到其他端口
                actions = [parser.OFPActionOutput(ofproto.OFPP_NORMAL)]
                self.add_policy_flow(datapath, 200, match, actions)

            # 阻止D的其他訪問（除了已允許的端口）
            match = parser.OFPMatch(eth_dst='00:00:00:00:00:04')
            self.add_policy_flow(datapath, 100, match, [])  # 無動作表示丟棄

        # 策略3: 阻止D和C之間的通信
        if datapath.id == 2:  # S2
            # 阻止從D到C的通信
            match = parser.OFPMatch(in_port=2, eth_dst='00:00:00:00:00:03')  # 從D來的流量
            self.add_policy_flow(datapath, 300, match, [])

            # 阻止從C到D的通信
            match = parser.OFPMatch(in_port=3, eth_dst='00:00:00:00:00:04')  # 從C來的流量
            self.add_policy_flow(datapath, 300, match, [])
```

策略一:

A、B、C 可以自由通信

- 這個最簡單，因為默認的學習交換機行為就能實現
- 不需要特別的 flow rule

#策略二:

只要 C、D 兩個 host 使用 tcp port 22 與 80 就可以正常通過，並設置優先級 200

另外設一個 100 優先級的封鎖條件，就是當 D 不遵守限制，action 就為空

策略三:

這樣的優先級設置確保：

1. D 和 C 的通信一定會被阻止
2. 允許的端口訪問不會被阻止規則影響
3. 其他正常的通信可以進行

封包處理器（Packet-In Handler）

當交換機收到一個不知道如何處理的封包時，就會送到控制器，這時就會觸發這個處理器：

```
@set_ev_cls(stplib.EventPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]
```

因為我們使用了 STP，所以要改用 stplib.EventPacketIn。這個差別很重要，因為：

1. ofp_event.EventOFPPacketIn 是普通的封包進入事件
2. stplib.EventPacketIn 是經過 STP 處理後的封包事件

```
129         if eth.ethertype == ether_types.ETH_TYPE_LLDP:
130             # 忽略 LLDP 數據包
131             return
132 
133         dst = eth.dst
134         src = eth.src
135 
136         dpid = datapath.id
137         self.mac_to_port.setdefault(dpid, {})
138 
139         self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)
140 
141         # 學習 MAC 地址
142         self.mac_to_port[dpid][src] = in_port
143 
144         if dst in self.mac_to_port[dpid]:
145             out_port = self.mac_to_port[dpid][dst]
146         else:
147             out_port = ofproto.OFPP_FLOOD
148 
149         actions = [parser.OFPActionOutput(out_port)]
```

第一部分:

- 取得封包的源和目的 MAC 地址
- 創建交換機的 MAC 地址表
- 記錄源 MAC 地址來自哪個端口

第二部分:

- 查找目的 MAC 地址對應的端口
- 如果找到，從該端口轉發
- 如果找不到，泛洪到所有端口
- 最後安裝流表規則避免下次重新詢問控制器

```
151         # 安裝流表項以避免下次 packet_in
152         if out_port != ofproto.OFPP_FLOOD:
153             match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
154             self.add_flow(datapath, 1, match, actions)
155 
156         data = None
157         if msg.buffer_id == ofproto.OFP_NO_BUFFER:
158             data = msg.data
159 
160         out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
161                                   in_port=in_port, actions=actions, data=data)
162         datapath.send_msg(out)
163 
```

安裝流表項以避免下次 packet_in

STP 處理:

```python
@set_ev_cls(stplib.EventTopologyChange, MAIN_DISPATCHER)
def _topology_change_handler(self, ev):
    dp = ev.dp
    dpid_str = dpid_lib.dpid_to_str(dp.id)
    msg = 'Receive topology change event. Flush MAC table.'
    self.logger.debug("[dpid=%s] %s", dpid_str, msg)

    if dp.id in self.mac_to_port:
        self.delete_flow(dp)
        del self.mac_to_port[dp.id]
        self.add_policy_rules(dp)

@set_ev_cls(stplib.EventPortStateChange, MAIN_DISPATCHER)
def _port_state_change_handler(self, ev):
    dpid_str = dpid_lib.dpid_to_str(ev.dp.id)
    of_state = {stplib.PORT_STATE_DISABLE: 'DISABLE',
                stplib.PORT_STATE_BLOCK: 'BLOCK',
                stplib.PORT_STATE_LISTEN: 'LISTEN',
                stplib.PORT_STATE_LEARN: 'LEARN',
                stplib.PORT_STATE_FORWARD: 'FORWARD'}
    self.logger.debug("[dpid=%s][port=%d] state=%s",
                      dpid_str, ev.port_no, of_state[ev.port_state])
```

從 log 中我觀察到端口狀態的變化過程：

1. DISABLE → LISTEN：端口開始工作
2. LISTEN → LEARN：開始學習 MAC 地址
3. LEARN → FORWARD：可以轉發數據
4. 某些端口會變成 BLOCK 狀態，阻止環路

當 Topology 改變時（比如鏈路斷開）：

1. 刪除所有流表
2. 清空 MAC 地址表
3. 重新添加 flow rule