Computer Networks Lab 1 report

0. 作業目標

本次作業旨在使用軟體定義網路（SDN）技術實現以下策略，並在 Mininet 和 Ryu 環境中進行驗證：

1. 節點 A、B、C 可以自由通信。
2. 節點 D 僅能訪問節點 A 和 B 的 22 和 80 port，其餘 port 封鎖。
3. 節點 D 與節點 C 之間完全隔離，禁止任何通信。

我使用了：

- Mininet 自定義拓撲腳本創建網路結構。
- Ryu 控制器實現 OpenFlow 規則。
- 執行指令和封包流向分析來驗證策略的正確性。

1. `pingall` 測試結果截圖

以下為執行 `pingall` 測試的結果截圖：

```
root@richardserver:/mnt/hgfs# sudo mn --custom custom_topo.py --topo customtopo --controller=remote,
ip=127.0.0.1,port=6653
*** Creating network
*** Adding controller
*** Adding hosts:
A B C D
*** Adding switches:
s1 s2 s3
*** Adding links:
(A, s3) (B, s1) (C, s2) (D, s2) (s1, s2) (s1, s3) (s3, s2)
*** Configuring hosts
A B C D
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
A -> B C X
B -> A C X
C -> A B X
D -> X X X
*** Results: 50% dropped (6/12 received)
```

測試結果分析

- **封包流向解釋：**
    1. **A → B → C → X：**
        - 節點 A 成功與節點 B 和 C 通信，但無法與節點 D 通信。
        - **原因：**節點 A、B、C 的自由通信符合策略1，而節點 D 的通信限制（策略3）阻止了無授權的流量。
    2. **B → A → C → X：**

        - 節點 B 成功與節點 A 和 C 通信，但無法與節點 D 通信。
        - **原因：**策略1和策略3的規則同樣適用於節點 B。
    3. **C → A → B → X：**
        - 節點 C 成功與節點 A 和 B 通信，但無法與節點 D 通信。
        - **原因：**節點 C 與節點 D 完全隔離，符合策略3。
    4. **D → X → X → X：**
        - 節點 D 無法與其他節點通信，因其流量未滿足策略2的 port 規則。
        - **原因：**策略2僅允許 D 訪問 A 和 B 的 22 和 80 port，`ping` 使用的 ICMP 協議被阻止。

- **結論：** 測試結果驗證了策略的實現：
    - 節點 A、B、C 之間的自由通信（策略1）。
    - 節點 D 的流量限制和隔離（策略2 和策略3）。

---

2. 拓撲設計

該拓撲包含：

- **主機：**
    - 節點 A：IP `10.0.0.1`
    - 節點 B：IP `10.0.0.2`
    - 節點 C：IP `10.0.0.3`
    - 節點 D：IP `10.0.0.4`
- **交換機：**
    - s1、s2 和 s3
- **連接結構：**
    - 節點 A 與 s3 相連，使用 port 1。
    - 節點 B 與 s1 相連，使用 port 3。
    - 節點 C 與 s2 相連，使用 port 3。
    - 節點 D 與 s2 相連，使用 port 2。
    - 交換機之間形成三角形連接：
        - s1 的 port 1 與 s3 的 port 3 相連。
        - s1 的 port 2 與 s2 的 port 1 相連。
        - s2 的 port 4 與 s3 的 port 2 相連。
- **規則定義：**
    - 節點 A、B、C 之間通信應透過 s1、s2、s3 自由轉發。
    - 節點 D 僅能透過特定 port 訪問節點 A 和 B 的 22 和 80 port。
    - 節點 C 和 D 之間的封包禁止透過 s2 傳輸。

---

3. 策略詳細講解與程式碼分析

以下為每一項策略的實現邏輯與對應程式碼的詳細說明。

3.1 策略1：節點 A、B、C 之間自由通信

目標：

允許節點 A、B、C 之間無阻礙地進行通信，不限制任何協議或 port。

實現方式：

- 在 Ryu 控制器中，對於 A、B、C 的每一對節點，建立雙向的流表規則。
- 流表匹配條件包括：
  - 以 IPv4 為基礎的流量（`eth_type=0x0800`）。
  - 源 IP 和目標 IP 的具體匹配。

程式碼片段：

```
# 策略1：A、B、C 節點之間自由通信
def install_abc_policies(self, datapath, parser, ofproto):
    # A <-> B
    self.add_bidirectional_flow(datapath, self.IP_A, self.IP_B, priority=10)
    # A <-> C
    self.add_bidirectional_flow(datapath, self.IP_A, self.IP_C, priority=10)
    # B <-> C
    self.add_bidirectional_flow(datapath, self.IP_B, self.IP_C, priority=10)

def add_bidirectional_flow(self, datapath, src_ip, dst_ip, priority):
    parser = datapath.ofproto_parser
    # 流表1：從 src_ip 到 dst_ip
    match = parser.OFPMatch(eth_type=0x0800, ipv4_src=src_ip, ipv4_dst=dst_ip)
    actions = [parser.OFPActionOutput(datapath.ofproto.OFPP_NORMAL)]
    self.add_flow(datapath, priority, match, actions)

    # 流表2：從 dst_ip 到 src_ip
    match = parser.OFPMatch(eth_type=0x0800, ipv4_src=dst_ip, ipv4_dst=src_ip)
    self.add_flow(datapath, priority, match, actions)
```

程式碼解釋：

1. **`install_abc_policies` 函數：**
   - 呼叫 `add_bidirectional_flow` 函數，為 A、B、C 間的所有可能通信建立雙向規則。
2. **`add_bidirectional_flow` 函數：**
   - 配置兩條流表：
     - 一條匹配從 `src_ip` 到 `dst_ip` 的流量。
     - 另一條匹配從 `dst_ip` 到 `src_ip` 的流量。
   - 使用 `OFPP_NORMAL`，讓交換機使用默認行為轉發流量。

3.2 策略2：節點 D 僅能訪問節點 A 和 B 的 22 和 80 port

目標：

限制節點 D 的流量，僅允許其訪問節點 A 和 B 的 22 和 80 port，其他 port 一律封鎖。

實現方式：

- 在 Ryu 控制器中，針對 D 的流量，僅允許符合以下條件的 TCP 流量：
  - 目的 IP 是 A 或 B。
  - 目的 port 是 22 或 80。

程式碼片段：

```
# 策略2：D 節點訪問 A 和 B 的特定 port
def install_d_policies(self, datapath, parser, ofproto):
    for dst_ip in [self.IP_A, self.IP_B]:
        for port in [22, 80]:
            # 匹配 D 到 A/B 的特定 port 流量
            match = parser.OFPMatch(eth_type=0x0800, ip_proto=6,
                                    ipv4_src=self.IP_D, ipv4_dst=dst_ip,
tcp_dst=port)
            # 動作：根據目標主機的連接 port 轉發
            actions = [parser.OFPActionOutput(4 if dst_ip == self.IP_A else 1)]
            self.add_flow(datapath, 20, match, actions)
```

程式碼解釋：

1. **雙層迴圈遍歷目標 IP 和 port：**
   - 遍歷目標 IP（A 和 B ）與 port（22 和 80）。
2. **匹配條件：**
   - IPv4 流量（`eth_type=0x0800`）。
   - TCP 協議（`ip_proto=6`）。
   - 來源 IP 是 D，目標 IP 是 A 或 B。
   - 目標 port 是 22 或 80。
3. **動作設置：**
   - 如果目標主機是 A（IP 為 `10.0.0.1` ），則流量應轉發到 port 4。
   - 如果目標主機是 B（IP 為 `10.0.0.2` ），則流量應轉發到 port 1。
4. **流表優先級設置為 20：**
   - 確保此規則優先於策略1的規則。

3.3 策略3：阻止 D 與 C 的通信

目標：

完全阻止 D 與 C 之間的所有通信。

實現方式：

- 為 D 和 C 的流量添加阻止規則。
- 流表的優先級設置為最高，確保該規則優先執行。

程式碼片段：

```
# 策略3：D 和 C 節點之間完全隔離
def block_dc_communication(self, datapath, parser, ofproto):
    # 阻止 D 到 C
    match = parser.OFPMatch(eth_type=0x0800, ipv4_src=self.IP_D,
ipv4_dst=self.IP_C)
    self.add_flow(datapath, 30, match, [])
    # 阻止 C 到 D
    match = parser.OFPMatch(eth_type=0x0800, ipv4_src=self.IP_C,
ipv4_dst=self.IP_D)
    self.add_flow(datapath, 30, match, [])
```

程式碼解釋：

1. **阻止 D 到 C 的流量：**
    - 匹配來源 IP 為 D，目標 IP 為 C 的 IPv4 流量。
    - 動作設置為空（`[]`），表示丟棄匹配的流量。
    - 流表優先級設置為 30。
2. **阻止 C 到 D 的流量：**
    - 匹配來源 IP 為 C，目標 IP 為 D 的 IPv4 流量。
    - 動作設置為空（`[]`），表示丟棄匹配的流量。
    - 流表優先級設置為 30。

---

4. 程式碼實現

以下為自定義拓撲與 Ryu 控制器的程式碼，並說明其如何達成上述策略。

4.1 自定義拓撲 (`custom_topo.py`)

以下為自定義拓撲的程式碼：

```
from mininet.topo import Topo

class CustomTopo(Topo):
    def build(self):

        A = self.addHost('A', ip='10.0.0.1/24', mac='00:00:00:00:00:0A')
        B = self.addHost('B', ip='10.0.0.2/24', mac='00:00:00:00:00:0B')
        C = self.addHost('C', ip='10.0.0.3/24', mac='00:00:00:00:00:0C')
        D = self.addHost('D', ip='10.0.0.4/24', mac='00:00:00:00:00:0D')
```

```
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        s3 = self.addSwitch('s3')

        # 主機與交換機的連接
        self.addLink(A, s3, port1=1, port2=1)  # A <-> s3 的 port 1
        self.addLink(B, s1, port1=1, port2=3)  # B <-> s1 的 port 3
        self.addLink(C, s2, port1=1, port2=3)  # C <-> s2 的 port 3
        self.addLink(D, s2, port1=1, port2=2)  # D <-> s2 的 port 2

        # 交換機之間的連接
        self.addLink(s1, s3, port1=1, port2=3)  # s1 的 port 1 <-> s3 的 port 3
        self.addLink(s1, s2, port1=2, port2=1)  # s1 的 port 2 <-> s2 的 port 1
        self.addLink(s3, s2, port1=2, port2=4)  # s3 的 port 2 <-> s2 的 port 4
```

- **結構分析：**
  - **主機：**
    - A：10.0.0.1
    - B：10.0.0.2
    - C：10.0.0.3
    - D：10.0.0.4
  - **交換機：**
    - s1、s2 和 s3
  - **拓撲連接：**
    - 主機與交換機連接，交換機之間形成三角形拓撲，確保流量靈活轉發。

4.2 流表規則 (`policy_controller.py`)

以下為關鍵的 Ryu 控制器程式碼，用於實現網路策略：

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3

class PolicyController(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    # 定義主機 IP
    IP_A = '10.0.0.1'
    IP_B = '10.0.0.2'
    IP_C = '10.0.0.3'
    IP_D = '10.0.0.4'

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

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

```
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        parser = datapath.ofproto_parser
        ofproto = datapath.ofproto

        # 安裝默認流表
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

        # 安裝策略1：A、B、C 節點之間自由通信
        self.install_abc_policies(datapath, parser, ofproto)

        # 安裝策略2：D 節點訪問 A 和 B 的特定 port
        self.install_d_policies(datapath, parser, ofproto)

        # 安裝策略3：阻止 D 和 C 節點之間完全隔離
        self.block_dc_communication(datapath, parser, ofproto)

    def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                    priority=priority, match=match,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)

    # 策略1：A、B、C 節點之間自由通信
    def install_abc_policies(self, datapath, parser, ofproto):
        # A <-> B
        self.add_bidirectional_flow(datapath, self.IP_A, self.IP_B, priority=10)
        # A <-> C
        self.add_bidirectional_flow(datapath, self.IP_A, self.IP_C, priority=10)
        # B <-> C
        self.add_bidirectional_flow(datapath, self.IP_B, self.IP_C, priority=10)

    def add_bidirectional_flow(self, datapath, src_ip, dst_ip, priority):
        parser = datapath.ofproto_parser
        # 流表1：從 src_ip 到 dst_ip
        match = parser.OFPMatch(eth_type=0x0800, ipv4_src=src_ip, ipv4_dst=dst_ip)
        actions = [parser.OFPActionOutput(datapath.ofproto.OFPP_NORMAL)]
        self.add_flow(datapath, priority, match, actions)

        # 流表2：從 dst_ip 到 src_ip
        match = parser.OFPMatch(eth_type=0x0800, ipv4_src=dst_ip, ipv4_dst=src_ip)
        self.add_flow(datapath, priority, match, actions)
```

```
    # 策略2：D 節點訪問 A 和 B 的特定 port
    def install_d_policies(self, datapath, parser, ofproto):
        for dst_ip in [self.IP_A, self.IP_B]:
            for port in [22, 80]:
                # 匹配 D 到 A/B 的特定 port 流量
                match = parser.OFPMatch(eth_type=0x0800, ip_proto=6,
                                        ipv4_src=self.IP_D, ipv4_dst=dst_ip,
tcp_dst=port)
                # 動作：根據目標主機的連接 port 轉發
                actions = [parser.OFPActionOutput(4 if dst_ip == self.IP_A else
1)]
                self.add_flow(datapath, 20, match, actions)

    # 策略3：D 和 C 節點之間完全隔離
    def block_dc_communication(self, datapath, parser, ofproto):
        # 阻止 D 到 C
        match = parser.OFPMatch(eth_type=0x0800, ipv4_src=self.IP_D,
ipv4_dst=self.IP_C)
        self.add_flow(datapath, 30, match, [])
        # 阻止 C 到 D
        match = parser.OFPMatch(eth_type=0x0800, ipv4_src=self.IP_C,
ipv4_dst=self.IP_D)
        self.add_flow(datapath, 30, match, [])
```

程式碼解釋：

1. **初始化與事件處理：**
   - `PolicyController` 繼承自 `app_manager.RyuApp`，使用 OpenFlow 1.3 協議。
   - 在收到 `EventOFPSwitchFeatures` 事件時，安裝默認流表和各項策略的流表規則。
2. **`add_flow` 函數：**
   - 用於向交換機添加流表規則。
   - 接受優先級、匹配條件和動作，並發送 `OFPFlowMod` 訊息至交換機。
3. **策略1 的實現：**
   - 呼叫 `install_abc_policies`，為 A、B、C 之間的所有可能通信建立雙向規則。
   - 使用 `OFPP_NORMAL` 讓交換機按照正常流程轉發流量。
4. **策略2 的實現：**
   - 呼叫 `install_d_policies`，僅允許 D 訪問 A 和 B 的 22 和 80 port。
   - 根據目標主機的不同，將流量轉發至相應的 port。
5. **策略3 的實現：**
   - 呼叫 `block_dc_communication`，完全阻止 D 與 C 之間的通信。
   - 設定高優先級（30），確保此規則優先執行，並且不進行任何動作（丟棄流量）。

---

5. 環境設置與執行

1. 啟動 **Ryu** 控制器：

```
ryu-manager ryu.app.simple_switch_13
```

2. 啟動 **Mininet**：

```
sudo mn --topo single,3 --controller=remote,ip=127.0.0.1 --switch=ovsk
```