NFV-Dual Path Routing Report

1. 概要

NFV-Dual Path Routing工程旨在設計和實現一個dual path routing機制，適用於多個 OpenFlow switch 和 host 組成的網路拓撲。主要目的是在每對源和目的主機間找到兩條完全分離的最短路徑，以保證通信的備份與穩定性。系統涵蓋動態網路拓撲管理、RESTful API 支援，以及 ARP (Address Resolution Protocol) 廣播處理等功能。

2. 目標

1. **雙路徑路由實現：**
    - 計算兩條最短且完全分離的路徑。
    - 若無法找到完全分離的路徑，提供退化為single path的機制。
2. **動態拓撲管理：**
    - 實時監控網路拓撲變化，更新路由信息。
    - 使用生成樹避免環路問題，減少廣播風暴。
3. **流表與群組表管理：**
    - 動態安裝高效且不重複的流表。
4. **RESTful API 支援：**
    - 提供主機信息和路徑查詢的功能。

3. 網路結構

3.1 拓撲組成

- **交換機：** 8 個 OpenFlow 交換機 (S1-S8)。
- **主機：** 9 個主機 (H1-H9)。

3.2 節點連接

主機到交換機的連接

- H1 → S1
- H2 → S3
- H3 → S6
- H4, H5 → S5
- H6, H7 → S8
- H8 → S6
- H9 → S4

交換機之間的連接

- S1 ↔ S2, S1 ↔ S3
- S2 ↔ S3, S2 ↔ S5, S2 ↔ S6
- S3 ↔ S4, S4 ↔ S5, S4 ↔ S8
- S5 ↔ S7, S5 ↔ S8
- S6 ↔ S7, S7 ↔ S8

4. 實作

4.1 網路拓撲的構建 (Custom Topology Script)

該部分程式碼用於構建 Mininet 中的網路拓撲，為dual path calculation提供基礎。

定義交換機與主機

以下生成後的拓樸截圖以及程式碼講解：

```
richard@ubuntu:/mnt/hgfs/internet/nfv$ sudo mn --custom custom_topology.py --to
po customtopo --controller remote --mac
*** 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 h9
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8
*** Adding links:
(h1, s1) (h2, s3) (h3, s6) (h4, s5) (h5, s5) (h6, s8) (h7, s8) (h8, s6) (h9, s4
) (s1, s2) (s1, s3) (s2, s3) (s2, s5) (s2, s6) (s3, s4) (s4, s5) (s4, s8) (s5,
s7) (s5, s8) (s6, s7) (s7, s8)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Starting controller
c0
*** Starting 8 switches
s1 s2 s3 s4 s5 s6 s7 s8 ...
*** Starting CLI:
mininet>
```

```
for i in range(1, 9):
    switches['S{}'.format(i)] = self.addSwitch('s{}'.format(i))

for i in range(1, 10):
    hosts['H{}'.format(i)] = self.addHost('h{}'.format(i))
```

- **交換機 (Switch)：**
    - 使用 `for` 迴圈從 1 到 8 創建 8 個交換機，名稱為 S1 至 S8。
    - `self.addSwitch('s{}'.format(i))`：調用 Mininet 的 `addSwitch` 方法，動態生成交換機名稱並添加到拓撲中。
    - `switches` 字典用於存儲所有交換機對象，以便後續建立連接。

- **主機 (Host)：**
  - 使用 `for` 迴圈從 1 到 9 創建 9 個主機，名稱為 H1 至 H9。
  - `self.addHost('h{}'.format(i))`：調用 Mininet 的 `addHost` 方法，動態生成主機名稱並添加到拓撲中。
  - `hosts` 字典用於存儲所有主機對象，以便後續建立連接。

連接節點

主機到交換機的連接：

```
self.addLink(hosts['H1'], switches['S1'])
self.addLink(hosts['H2'], switches['S3'])
```

- **功能：**
  - 使用 `addLink` 方法建立主機與交換機之間的物理連接。
  - 例如，`self.addLink(hosts['H1'], switches['S1'])` 將主機 H1 連接到交換機 S1。
  - 確保每個主機能夠透過其對應的交換機進行網路通信。

交換機之間的連接：

```
self.addLink(switches['S1'], switches['S2'])
self.addLink(switches['S2'], switches['S3'])
```

- **功能：**
  - 使用 `addLink` 方法建立交換機之間的連接，形成多路徑結構。
  - 例如，`self.addLink(switches['S1'], switches['S2'])` 將交換機 S1 和 S2 直接連接。
  - 這些連接構成了網路的骨幹，支持雙路徑路由的實現。

4.2 控制器功能實作 (Ryu Controller)

以下是 Ryu 控制器啟動成功之截圖以及程式碼片段說明:

```
richard@ubuntu:/mnt/hgfs/internet/nfv$ ryu-manager --observe-links dual_path_ro
uting.py
loading app dual_path_routing.py
loading app ryu.topology.switches
loading app ryu.controller.dpset
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app dual_path_routing.py of DualPathRouting
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.dpset of DPSet
instantiating app ryu.controller.ofp_handler of OFPHandler
(8524) wsgi starting up on http://0.0.0.0:8080
當前節點: {1, 2, 3, 4, 5, 6, 7, 8}
當前邊: set()
當前節點: {1, 2, 3, 4, 5, 6, 7, 8}
當前邊: {(3, 1), (1, 3)}
當前節點: {1, 2, 3, 4, 5, 6, 7, 8}
當前邊: {(6, 2), (1, 2), (2, 1), (3, 1), (2, 6), (1, 3)}
當前節點: {1, 2, 3, 4, 5, 6, 7, 8}
當前邊: {(6, 2), (1, 2), (2, 1), (3, 4), (4, 3), (3, 1), (6, 7), (7, 6), (2, 6)
, (1, 3)}
當前節點: {1, 2, 3, 4, 5, 6, 7, 8}
當前邊: {(6, 2), (1, 2), (2, 1), (3, 4), (4, 3), (3, 1), (6, 7), (7, 6), (2, 6)
, (2, 5), (1, 3), (5, 2)}
當前節點: {1, 2, 3, 4, 5, 6, 7, 8}
當前邊: {(3, 4), (4, 3), (3, 1), (2, 5), (1, 3), (6, 2), (4, 8), (8, 5), (1, 2)
, (2, 1), (6, 7), (7, 6), (3, 2), (5, 2), (8, 4), (5, 8), (8, 7), (2, 3), (2, 6
), (7, 8)}
當前節點: {1, 2, 3, 4, 5, 6, 7, 8}
當前邊: {(3, 4), (4, 3), (3, 1), (5, 4), (5, 7), (2, 5), (1, 3), (6, 2), (4, 5)
, (4, 8), (8, 5), (1, 2), (2, 1), (6, 7), (7, 6), (3, 2), (5, 2), (8, 4), (5, 8
), (8, 7), (2, 3), (2, 6), (7, 5), (7, 8)}
```

初始化與交換機特性處理

控制器初始化時，為每個交換機設置預設流表：

```
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(dp.ofproto.OFPP_CONTROLLER)]
self.add_flow(dp, priority=0, match=match, actions=actions)
```

- **功能說明：**
  - **匹配條件 (`match`)：** `parser.OFPMatch()` 創建一個空的匹配條件，表示匹配所有封包。
  - **動作 (`actions`)：** `[parser.OFPActionOutput(dp.ofproto.OFPP_CONTROLLER)]` 將所有匹配的封包輸出到控制器，實現封包的轉發至控制器進行處理。
  - **流表安裝 (`add_flow`)：**
    - `self.add_flow(dp, priority=0, match=match, actions=actions)`：為每個交換機添加一條預設流表，優先級為 0，確保所有未知封包都能被轉發至控制器。
    - **參數說明：**
      - `dp`：表示交換機的數據平面（Data Plane）。
      - `priority=0`：設置流表的優先級為最低。
      - `match`：匹配所有封包。

        - `actions`：指定動作為將封包送至控制器。

- **作用：**
    - 確保所有未匹配的封包都會被控制器接收，以便控制器根據網路狀況動態下發流表。

拓撲更新處理

動態監控拓撲變化，更新網路結構：

```
switches = get_switch(self, None)
links = get_link(self, None)
for link in links:
    self.net.add_edge(link.src.dpid, link.dst.dpid)
```

- **功能說明：**
    - **獲取交換機和鏈路信息：**
        - `get_switch(self, None)`：調用 Ryu 的 `get_switch` 函數獲取當前網路中的所有交換機。
        - `get_link(self, None)`：調用 Ryu 的 `get_link` 函數獲取當前網路中的所有鏈路（交換機之間的連接）。
    - **更新網路圖 (`self.net`)：**
        - `self.net.add_edge(link.src.dpid, link.dst.dpid)`：使用 NetworkX 庫將交換機之間的鏈路添加到網路圖中，`dpid` 是交換機的唯一標識符。
        - 此步驟確保控制器擁有最新的網路拓撲信息，以便進行路由計算。
- **作用：**
    - 實時更新網路拓撲，反映網路中的變化，如交換機的新增或鏈路的變動，確保路由計算基於最新的拓撲結構。

雙路徑計算

該部分計算兩條最短且完全分離的路徑：

```
path1 = nx.shortest_path(self.net, src_mac, dst_mac)
path2 = nx.disjoint_paths(self.net, src_mac, dst_mac)[1]
```

- **功能說明：**
    - **Path1：最短路徑計算：**
        - `nx.shortest_path(self.net, src_mac, dst_mac)`：使用 NetworkX 的 `shortest_path` 函數計算從源 MAC (`src_mac`) 到目的 MAC (`dst_mac`) 的最短路徑。
        - **參數：**
            - `self.net`：表示當前的網路拓撲圖。
            - `src_mac` 和 `dst_mac`：源主機和目的主機的 MAC 地址，作為路徑計算的節點標識。

    - **Path2：完全分離路徑計算：**
        - `nx.disjoint_paths(self.net, src_mac, dst_mac)[1]`：使用 NetworkX 的 `disjoint_paths` 函數計算從源 MAC 到目的 MAC 的完全分離路徑，並選取第二條路徑（假設第一條為最短路徑）。
        - **功能：** `disjoint_paths` 返回所有節點不重疊的路徑集合，選取第二條路徑確保與第一條路徑完全分離。

- **作用：**
    - 計算兩條互不干擾的路徑，增強網路的冗餘性和可靠性。如果無法找到兩條完全分離的路徑，則需實現退化機制使用單一路徑。

添加流表

根據計算結果，動態安裝流表：

```
for i in range(len(path)-1):
    match = parser.OFPMatch(eth_src=src_mac, eth_dst=dst_mac)
    actions = [parser.OFPActionOutput(out_port)]
    self.add_flow(dp, priority=10, match=match, actions=actions)
```

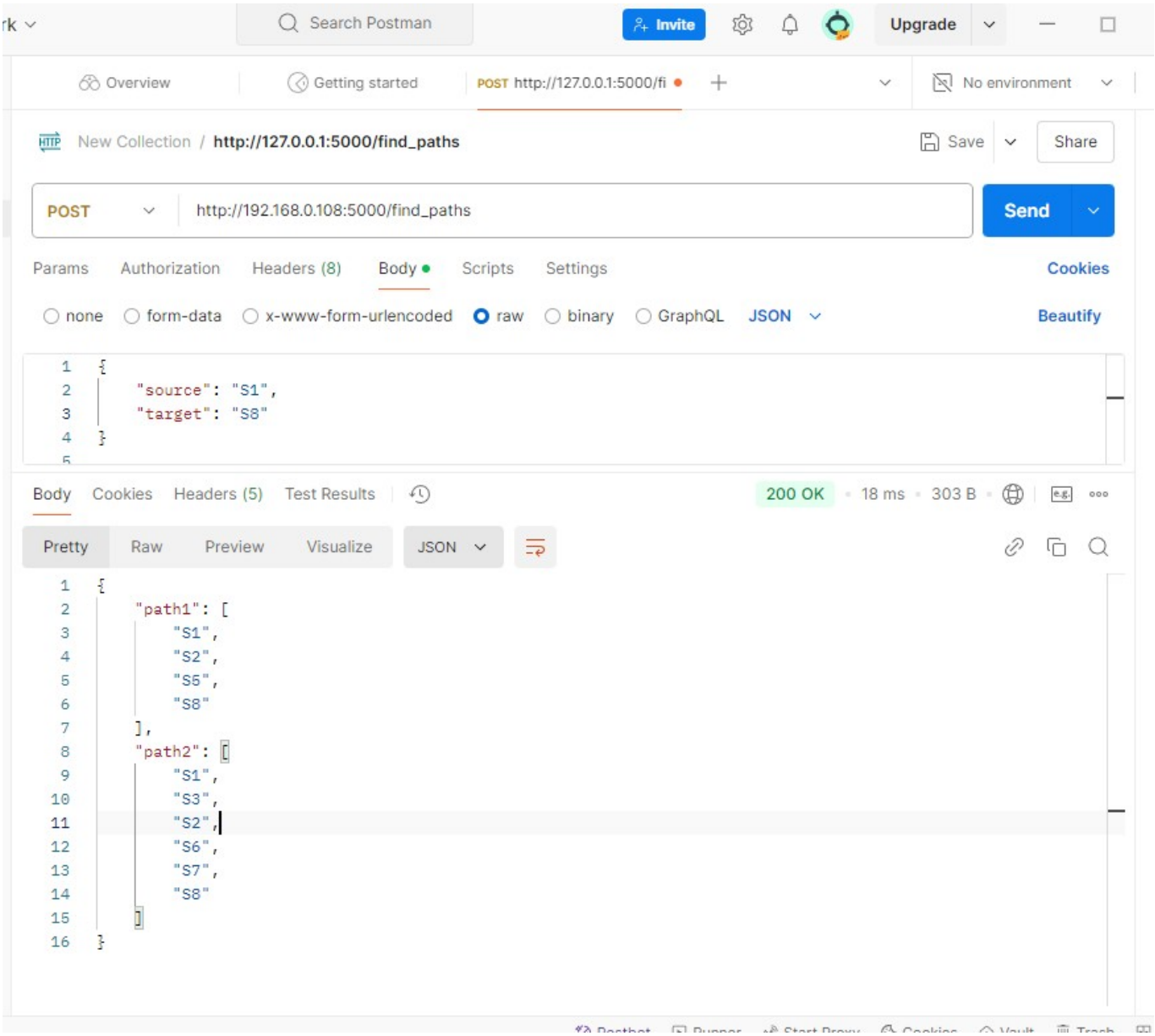
- **功能說明：**
    - **迴圈遍歷路徑中的每一個節點對：**
        - `for i in range(len(path)-1)`：遍歷路徑中的每一個交換機節點，為每一跳添加相應的流表。
    - **匹配條件 (`match`)：**
        - `parser.OFPMatch(eth_src=src_mac, eth_dst=dst_mac)`：匹配來源 MAC 和目的 MAC 地址，確保流量轉發的正確性。
    - **動作 (`actions`)：**
        - `[parser.OFPActionOutput(out_port)]`：指定將匹配的封包輸出到下一跳的端口。
        - `out_port`：根據路徑計算結果，確定應該將封包轉發到哪一個端口。
    - **流表安裝 (`add_flow`)：**
        - `self.add_flow(dp, priority=10, match=match, actions=actions)`：為交換機安裝具體的流表，優先級設置為 10，高於預設流表，確保優先匹配這些流量。

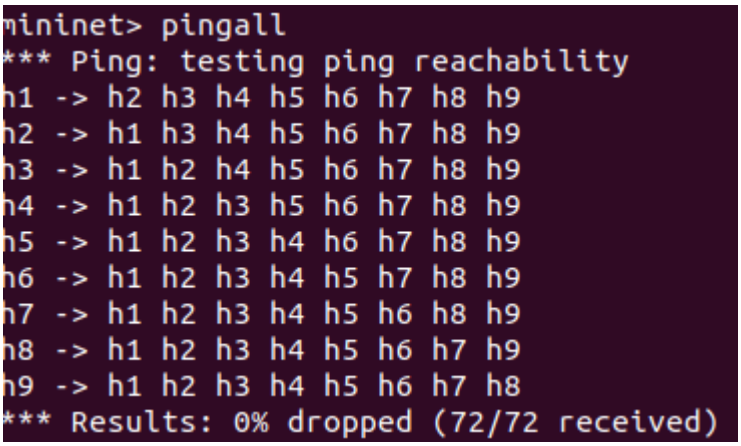
- **作用：**
    - 根據計算出的路徑動態安裝流表，確保流量能夠按照雙路徑路由機制正確轉發，實現負載均衡和冗餘備份。

4.3 RESTful API 的實作 以及拓樸測試

以下是API實作及拓樸測試的截圖，並且講解API的程式碼部分

路徑查詢及拓樸測試：






```python
@route('dualpath', '/dualpath/route/{src_mac}/{dst_mac}', methods=['GET'])
def get_route(self, req, **kwargs):
    src_mac = kwargs['src_mac']
    dst_mac = kwargs['dst_mac']
    path1, path2 = self.compute_two_disjoint_paths(src_mac, dst_mac)
```

```
    return Response(content_type='application/json', body=json.dumps({'path1':
path1, 'path2': path2}))
```

- **功能說明：**
  - **路由裝飾器 (`@route`)：**
    - `@route('dualpath', '/dualpath/route/{src_mac}/{dst_mac}', methods=['GET'])`：定義一個 RESTful API 路徑，當接收到對應的 GET 請求時，調用 `get_route` 函數處理。
    - **參數：**
      - `'dualpath'`：API 名稱。
      - `'/dualpath/route/{src_mac}/{dst_mac}'`：API 的 URL 模式，包含源 MAC 和目的 MAC 作為參數。
      - `methods=['GET']`：指定 HTTP 方法為 GET。
  - **函數定義 (`get_route`)：**
    - `def get_route(self, req, **kwargs)`：定義處理函數，接受請求對象和關鍵字參數。
    - **參數提取：**
      - `src_mac = kwargs['src_mac']`：提取 URL 中的源 MAC 地址。
      - `dst_mac = kwargs['dst_mac']`：提取 URL 中的目的 MAC 地址。
    - **路徑計算：**
      - `path1, path2 = self.compute_two_disjoint_paths(src_mac, dst_mac)`：調用自定義函數 `compute_two_disjoint_paths` 計算兩條完全分離的路徑。
    - **響應生成：**
      - `return Response(content_type='application/json', body=json.dumps({'path1': path1, 'path2': path2}))`：將計算結果以 JSON 格式返回給客戶端。
      - **JSON 格式：**

```
{
    "path1": ["S1", "S2", "S3"],
    "path2": ["S1", "S4", "S5"]
}
```

- **作用：**
  - 提供一個 RESTful API 介面，允許用戶通過指定源和目的主機的 MAC 地址來查詢雙路徑路由結果。
  - 方便網管人員或自動化工具進行路由查詢和網路狀態監控。

`compute_two_disjoint_paths` 函數詳解

為了更清楚地了解雙路徑計算的實作細節，以下對 `compute_two_disjoint_paths` 函數進行詳細說明：

```
def compute_two_disjoint_paths(self, src_mac, dst_mac):
    src = self.mac_to_switch[src_mac]
    dst = self.mac_to_switch[dst_mac]
```

```
        try:
            path1 = nx.shortest_path(self.net, src, dst)
            path2 = nx.shortest_path(self.net, src, dst, weight='weight')
            # 確保 path2 與 path1 完全分離
            if set(path1).isdisjoint(set(path2)):
                return path1, path2
        except nx.NetworkXNoPath:
            pass
        # 無法找到兩條分離路徑，退化為單一路徑
        path = nx.shortest_path(self.net, src, dst)
        return path, []
```

- **功能說明：**
  - **參數：**
    - `src_mac`：源主機的 MAC 地址。
    - `dst_mac`：目的主機的 MAC 地址。
  - **MAC 地址到交換機映射：**
    - `src = self.mac_to_switch[src_mac]`：將源 MAC 地址映射到對應的交換機。
    - `dst = self.mac_to_switch[dst_mac]`：將目的 MAC 地址映射到對應的交換機。
  - **路徑計算：**
    - `path1 = nx.shortest_path(self.net, src, dst)`：計算從源交換機到目的交換機的最短路徑。
    - `path2 = nx.shortest_path(self.net, src, dst, weight='weight')`：再次計算路徑，這裡假設可以通過設定不同的權重來嘗試找到另一條不同的路徑。
  - **路徑分離性檢查：**
    - `if set(path1).isdisjoint(set(path2))`：檢查兩條路徑是否完全分離，即沒有任何共同的交換機節點。
    - **返回結果：** 如果兩條路徑完全分離，則返回這兩條路徑。
  - **異常處理：**
    - `except nx.NetworkXNoPath`：如果無法找到任何路徑，則捕捉異常並進行處理。
  - **退化機制：**
    - 如果無法找到兩條完全分離的路徑，則僅返回單一路徑，並將第二條路徑設為空列表。
- **作用：**
  - 確保在網路拓撲允許的情況下，提供兩條冗餘的路徑以增強網路的可靠性。
  - 當網路拓撲限制無法提供雙路徑時，能夠自動退化為單一路徑，保證基本的通信功能。

5. 結論

在這次作業中，我學習並實作了 NFV-Dual Path Routing 系統，涵蓋了網路拓撲的構建、控制器的開發以及 RESTful API 的實現。使用 Mininet 工具讓我熟悉了如何自訂義網路拓撲，並理解了交換機與主機之間的連接方式和多路徑結構的設計原理。在控制器開發部分，我掌握了 Ryu 控制器的基本操作，包括如何安裝和管理流表，並學會了動態監控網路拓撲的變化，及時更新路由信息，確保路由計算基於最新的網路狀態。 通過使用 NetworkX 庫，我了解了雙路徑路由算法的實現方法，能夠計算兩條完全分離的最短路徑，並在無法找到雙路徑時實現退化機制，確保基本的通信功能。在實作過程中，我遇到了路徑重疊的問題，通過研究路徑算法並調整參數設置，成功解決了這些挑戰。此外，我還學會了如何高效管理流表，避免流表的重複安裝，從而提升了系統的運行效率。 在開發 RESTful API 的過程中，我學習了如何設計和實作 API，提供主機信息和路徑查詢的功

能，這不僅提升了系統的可操作性和靈活性，也讓我理解了 API 與網路管理工具整合的重要性。這次作業讓我對軟體定義網路（SDN）和網路功能虛擬化（NFV）有了更深入的理解，並增強了解決實際問題的能力。