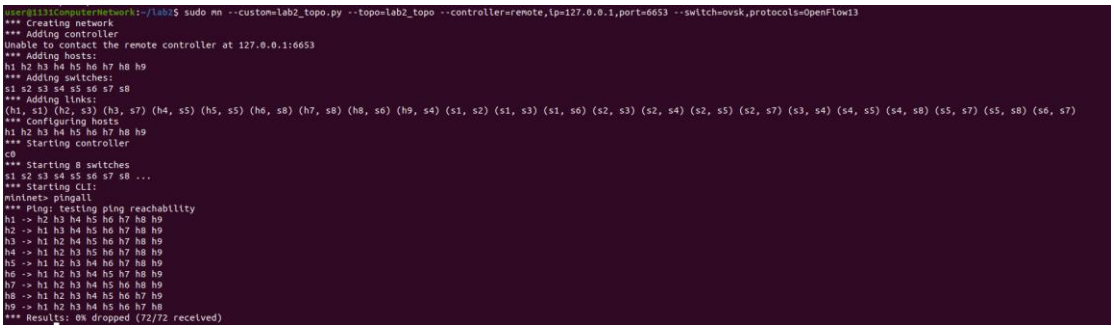
313551137 官劉翔

實作步驟&結果:

1. 要先開啟 mininet 再開啟 ryu (這樣 topology discover 才能正常運行)
2. 啟動 ryu 後，確定有 number of switch and link (lab2 的 topology 應該為 8 個 switch 與 26 條 link)，如果失敗就重啟 ryu (--observe-links 是必要的，否則 controller 無法得到 topology
   ryu-manager dual_disjoint_shortest_path.py --observe-links
3. 結果



實作流程:

我在這裡的實作流程為下

1. 先用 tree2,2 topology，確認 get_topology 可以順利讓 controller 收到 network topology
2. 使用 lab2 測試用的 topology，並將一開始的 ARP flood 的 broadcast storm 解決掉
3. 實作出 single_shortest path
4. 實作出 dual_shortest path
5. 實作出 dual & disjoint_shortest

實作細節 (程式碼講解):

首先，我看程式的整體架構。這次主要目的是實現具有雙路徑功能的最短路徑轉發。在網路中，有時候使用單一路徑可能會造成網路擁塞或者在路徑故障時無法快速恢復，因此我實現了一個可以同時使用兩條最不相交路徑的控制器。

在程式的初始化部分：

```
def __init__(self,*args,**kwargs):
    super(MyShortestForwarding,self).__init__(*args,**kwargs)
    self.network = nx.DiGraph()
    self.paths = {}
    self.topology_api_app = self
    self.switch_mac_matches = {}
    self.group_ids = {}
```

我使用了 NetworkX 的 DiGraph（有向圖）來儲存網路拓撲。這個選擇非常重要，因為在實際網路中，連接是有方向性的，每個方向可能有不同的端口號和特性。paths 字典用來快取已經計算過的路徑，這是一個重要的優化，因為路徑計算是比較耗時的操作。switch_mac_matches 用來追踪已經處理過的 MAC 地址配對，避免重複處理相同的流量，也避免廣播風暴。group_ids 則用於管理 OpenFlow 的群組表，這是實現多路徑轉發的關鍵。

在 Topology discover 部分，get_topology 函數扮演著關鍵角色：

```
@set_ev_cls(event.EventSwitchEnter,[CONFIG_DISPATCHER,MAIN_DISPATCHER])
def get_topology(self,ev):
    self.network.clear()
    switch_list = get_switch(self.topology_api_app,None)
    switches = [switch.dp.id for switch in switch_list]
    self.network.add_nodes_from(switches)
    ...
```

每當有新的交換機加入網路時，這個函數就會被調用。它使用 Ryu 的 API 獲取所有交換機和它們之間的連接信息，然後更新我的 NetworkX 圖。這個函數不僅僅是單純地收集信息，它還需要正確處理每條連接的雙向性質，因為在 SDN 中，我需要知道每個方向的具體端口號。

在**雙路徑的實現上**，find_disjoint_paths_from_shortest 函數：

```
def find_disjoint_paths_from_shortest(self, all_paths, k=2):
    if len(all_paths) <= k:
        return all_paths

    min_overlap = float('inf')
    best_path_pair = None

    for i in range(len(all_paths)):
        for j in range(i + 1, len(all_paths)):
            path1 = set(all_paths[i])
            path2 = set(all_paths[j])
            overlap = len(path1.intersection(path2))
            ...
```

這個函數的設計思路是，在所有最短路徑中找出重疊節點最少的兩條路徑。這樣做有幾個好處：首先，由於是在最短路徑中選擇，所以不會有路徑過長的問題；除此之外，選擇重疊最少的路徑可以提高網路的可靠性，因為如果一個節點故障，影響的路徑會更少。

在實際的封包處理中，packet_in_handler 函數扮演著核心角色：

```
@set_ev_cls(ofp_event.EventOFPPacketIn,MAIN_DISPATCHER)
def packet_in_handler(self,ev):
    msg = ev.msg
    datapath = msg.datapath
    in_port = msg.match['in_port']
    pkt = packet.Packet(msg.data)
    eth_pkt = pkt.get_protocol(ethernet.ethernet)
    eth_src = eth_pkt.src
    eth_dst = eth_pkt.dst
```

當交換機收到一個新的封包且不知道如何處理時，就會將它送到控制器，觸發 packet_in 事件。這個函數首先解析封包，獲取源 MAC 地址和目的 MAC 地址。這些信息對於後續的路徑選擇非常重要。

特別值得注意的是對 broadcast 封包的處理：

```
if (eth_dst == 'ff:ff:ff:ff:ff:ff'):
    arp_pkt = pkt.get_protocol(arp.arp)
    if arp_pkt:
        target_ip = arp_pkt.dst_ip
        if dpid not in self.switch_mac_matches:
            self.switch_mac_matches[dpid] = set()
        mac_match_key = (eth_src, eth_dst, target_ip)
```

這部分處理了 ARP 請求，這在網路中是非常重要的，因為它用於解析 IP 地址和 MAC 地址的對應關係。我使用 switch_mac_matches 來追踪已經處理過的 ARP 請求，避免產生廣播風暴。

最後，在 add_flow 部分：

```
if out_ports != ofproto.OFPP_FLOOD:
    if len(out_ports) == 1:
        actions = [ofp_parser.OFPActionOutput(out_ports[0])]
        match = ofp_parser.OFPMatch(in_port=in_port,eth_dst=eth_dst)
        self.add_flow(datapath,1,match,actions)
    else:
        group_id = self.add_dual_path_group(datapath, out_ports)
        actions = [ofp_parser.OFPActionGroup(group_id)]
```

這部分代碼根據計算出的路徑數量決定使用 add_flow 還是 add_dual_path_group。如果只有一條路徑，就直接 add_flow；如果有多條路徑，就創建 group 並讓流表指向這個 group。這樣的設計既保證了單路徑的簡單高效，又支持了多路徑的負載均衡。

在多路徑轉發的實現上 (add_dual_path_group)：

```
def add_dual_path_group(self, datapath, out_ports):
    buckets = []
    for port in out_ports:
        actions = [ofp_parser.OFPActionOutput(port)]
        bucket = ofp_parser.OFPBucket(
            weight=50,
            watch_port=port,
            watch_group=ofproto.OFPG_ANY,
            actions=actions
        )
        buckets.append(bucket)
```

這個函數使用 OpenFlow 的群組表功能來實現多路徑轉發。每條路徑被放入一個 bucket 中，並被賦予相同的權重(weight=50)，這樣可以實現負載均衡。watch_port 參數允許交換機監控端口狀態，如果某個端口出現問題，可以自動切換到其他可用路徑。

get_out_ports 函數則負責實際的路徑選擇：

```
def get_out_ports(self,datapath,src,dst,in_port):
    if dst in self.network:
        if dst not in self.paths[src]:
            all_paths = list(nx.all_shortest_paths(self.network,src,dst))
            paths = self.find_disjoint_paths_from_shortest(all_paths)
            self.paths[src][dst] = paths
```

這個函數首先檢查目的地是否在網路中，如果是的話就尋找所有可能的最短路徑，然後使用 find_disjoint_paths_from_shortest 選出最合適的路徑對。這個過程的結果會被快取在 paths 字典中，提高後續處理相同流量的效率。

我的設計不僅實現了基本的轉發功能，還加入了多路徑、負載均衡等進階特性，同時也考慮了效能優化和錯誤處理。