# Part1

There are 6 distinct “OFPT_FLOW_MOD” headers during the experiment.

| Match fields | Actions | Timeout values |
|---|---|---|
| OFPXMT_OFB_ETH_TYPE=IPv4(0x0800) | Type=OFPAT_OUTPUT<br>Port=OFPP_CONTROLLER(4294967293) | 0 |
| OFPXMT_OFB_ETH_TYPE=ARP(0x0806) | Type=OFPAT_OUTPUT<br>Port=OFPP_CONTROLLER(4294967293) | 0 |
| OFPXMT_OFB_ETH_TYPE=LLDP(0x88cc) | Type=OFPAT_OUTPUT<br>Port=OFPP_CONTROLLER(4294967293) | 0 |
| OFPXMT_OFB_ETH_TYPE=unknown(0x8942) | Type=OFPAT_OUTPUT<br>Port=OFPP_CONTROLLER(4294967293) | 0 |
| OFPXMT_OFB_IN_PORT=2<br>OFPXMT_OFB_ETH_DST=32:aa:fb:35:0d:a9<br>OFPXMT_OFB_ETH_SRC=06:e1:0a:36:39:fa | Type=OFPAT_OUTPUT<br>Port=1 | 0 |
| OFPXMT_OFB_IN_PORT=1<br>OFPXMT_OFB_ETH_DST=06:e1:0a:36:39:fa<br>OFPXMT_OFB_ETH_SRC=32:aa:fb:35:0d:a9 | Type=OFPAT_OUTPUT<br>Port=2 | 0 |

# Part2

## Flow Rules List

onos Open Network Operating System ? onos

Flows for Device of:0000000000000001 (7 Total)

Search | All Fields

| STATE | PACKETS | DURATION | FLOW PRIORITY ▾ | TABLE NAME | SELECTOR | TREATMENT | APP NAME |
|---|---|---|---|---|---|---|---|
| Added | 15 | 1,156 | 50002 | 0 | ETH_TYPE:ipv4, IPV4_DST:10.0.0.2/32 | imm[OUTPUT:2], cleared:false | *rest |
| Added | 15 | 1,159 | 50001 | 0 | ETH_TYPE:ipv4, IPV4_DST:10.0.0.1/32 | imm[OUTPUT:1], cleared:false | *rest |
| Added | 86 | 2,050 | 50000 | 0 | ETH_TYPE:arp | imm[OUTPUT:ALL], cleared:false | *rest |
| Added | 3 | 8,540 | 40000 | 0 | ETH_TYPE:arp | imm[OUTPUT:CONTROLLER], cleared:true | *core |
| Added | 0 | 8,540 | 40000 | 0 | ETH_TYPE:lldp | imm[OUTPUT:CONTROLLER], cleared:true | *core |
| Added | 0 | 8,540 | 40000 | 0 | ETH_TYPE:bddp | imm[OUTPUT:CONTROLLER], cleared:true | *core |
| Added | 0 | 8,539 | 5 | 0 | ETH_TYPE:ipv4 | imm[OUTPUT:CONTROLLER], cleared:true | *core |

## Activate Apps

```
lspss95207@root > apps -a -s
*   3 org.onosproject.hostprovider        2.7.0    Host Location Provider
*   6 org.onosproject.lldpprovider        2.7.0    LLDP Link Provider
*   7 org.onosproject.optical-model       2.7.0    Optical Network Model
*   8 org.onosproject.openflow-base       2.7.0    OpenFlow Base Provider
*   9 org.onosproject.openflow            2.7.0    OpenFlow Provider Suite
*  14 org.onosproject.drivers             2.7.0    Default Drivers
* 122 org.onosproject.gui2                2.7.0    ONOS GUI2
```

## ARP

```
mininet> h1 arping h2 -c 5
ARPING 10.0.0.2
42 bytes from 8a:88:18:e6:45:26 (10.0.0.2): index=0 time=706.286 usec
42 bytes from 8a:88:18:e6:45:26 (10.0.0.2): index=1 time=75.198 usec
42 bytes from 8a:88:18:e6:45:26 (10.0.0.2): index=2 time=69.598 usec
42 bytes from 8a:88:18:e6:45:26 (10.0.0.2): index=3 time=68.698 usec
42 bytes from 8a:88:18:e6:45:26 (10.0.0.2): index=4 time=75.799 usec

--- 10.0.0.2 statistics ---
5 packets transmitted, 5 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 0.069/0.199/0.706/0.254 ms
```

```
mininet> h2 arping h1 -c 5
ARPING 10.0.0.1
42 bytes from c6:39:d1:1d:2e:54 (10.0.0.1): index=0 time=850.284 usec
42 bytes from c6:39:d1:1d:2e:54 (10.0.0.1): index=1 time=68.099 usec
42 bytes from c6:39:d1:1d:2e:54 (10.0.0.1): index=2 time=70.499 usec
42 bytes from c6:39:d1:1d:2e:54 (10.0.0.1): index=3 time=66.998 usec
42 bytes from c6:39:d1:1d:2e:54 (10.0.0.1): index=4 time=73.499 usec

--- 10.0.0.1 statistics ---
5 packets transmitted, 5 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 0.067/0.226/0.850/0.312 ms
```

ICMP Pings

```
mininet> h1 ping h2 -c 5
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.806 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.105 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.069 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.076 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.073 ms

--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4081ms
rtt min/avg/max/mdev = 0.069/0.225/0.806/0.290 ms
```

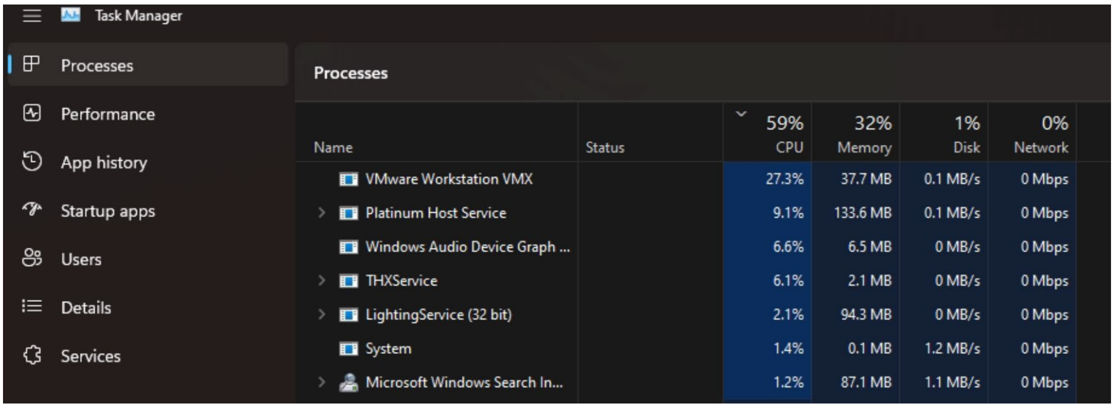
```
mininet> h2 ping h1 -c 5
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.691 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.137 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.086 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.076 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.079 ms

--- 10.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4102ms
rtt min/avg/max/mdev = 0.076/0.213/0.691/0.239 ms
```

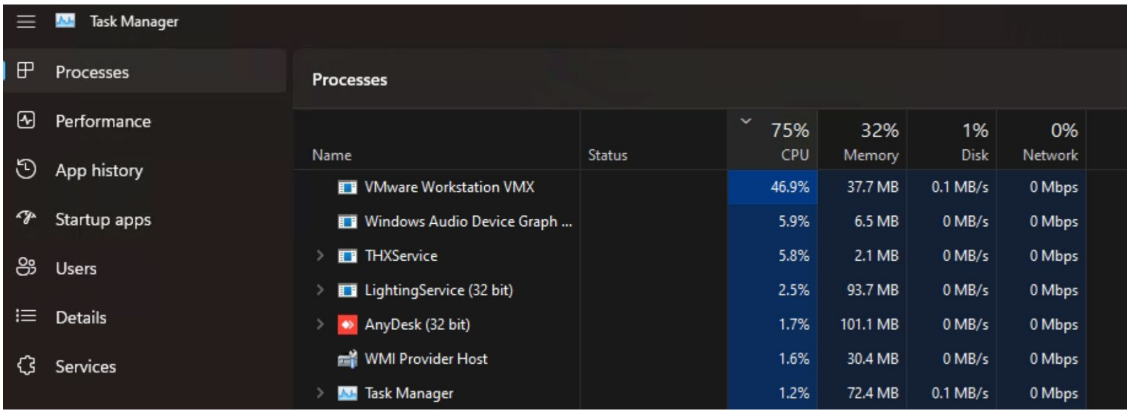
# Part3

## Outside VM

### Normal – CPU 27.3%

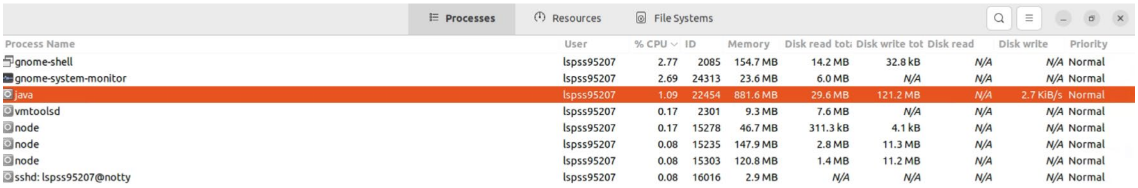



### When Broadcast Storm happened – CPU 46.9%

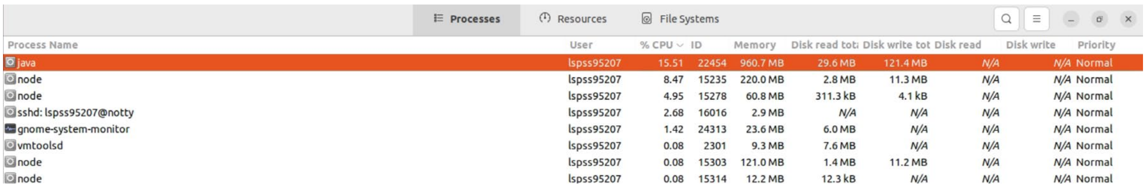



## Inside VM

### Normal – CPU 1.09%



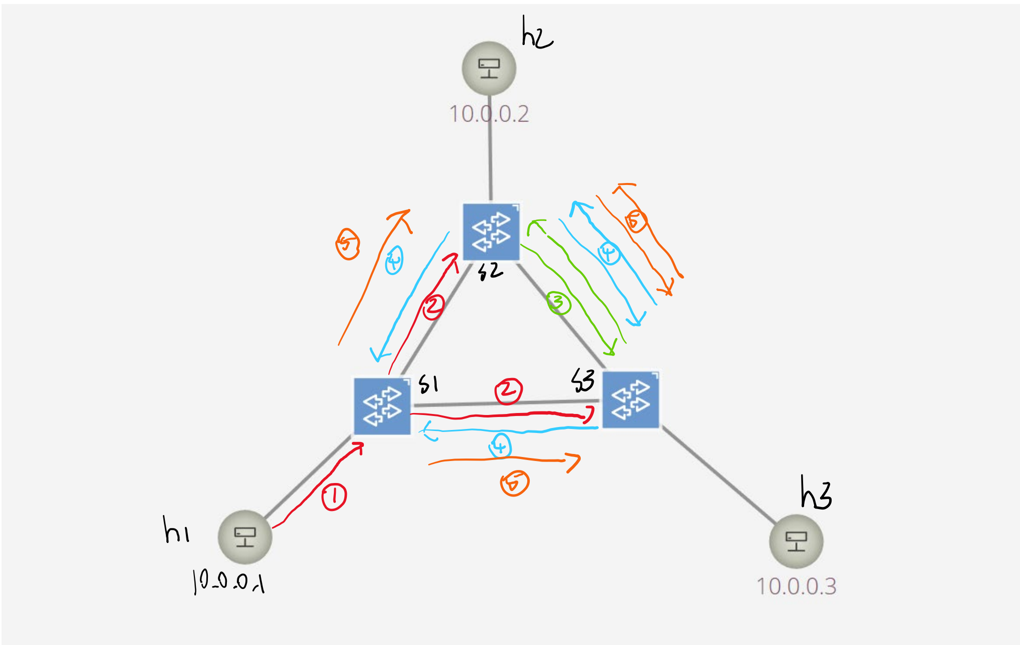

### When Broadcast Storm happened – CPU 15.51%



我發現當我架設好會產生 Broadcast Storm 的拓譜跟 flow 後，進行 h1 arping h2 時就算有用-c 控制產生 packet 數量，還是會無限的讀到 packet，如下圖。

```
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22873 time=4.044 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22874 time=4.044 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22875 time=4.045 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22876 time=4.045 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22877 time=4.046 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22878 time=4.046 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22879 time=4.046 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22880 time=4.047 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22881 time=4.047 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22882 time=4.048 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22883 time=4.048 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22884 time=4.049 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22885 time=4.049 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22886 time=4.049 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22887 time=4.049 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22888 time=4.050 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22889 time=4.050 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22890 time=4.050 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22891 time=4.051 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22892 time=4.051 sec
42 bytes from 42:e0:1a:7e:91:5a (10.0.0.2): index=22893 time=4.051 sec
```

由下圖所示，我使用的拓譜是一個簡單的 ring，而所下的 flow rule 是如同 part2 的 arp broadcast。因此當 host h1 送出 arp packet 時，switch s1 會先 broadcast arp packet 複製給 switch s2 和 s3，接著 s2 收到 packet 時會 broadcast arp packet 給 s3、s3 會 broadcast arp packet 給 s2，然後 s3 給 s1、s2 給 s1，然後 s1 又將 s2 傳來的送至 s3、將 s3 傳來的送至 s2。加上 arp packet 是 L2 packet 沒有 ttl 機制，因此這些 packets 就會一直在 ring 中繞來繞去不會消失，電腦就要一直花資源處理 packet，造成 VM 的 CPU 使用率維持很高，這就是所謂的 broadcast storm。



# Part4

Data Plane:

1. 確定是 IPv4 或 IPv6 的 packet，如果 match 到 flow 目的地，packet 就直接照 flow action 傳送。如果 match 不到 flow 目的地，packet 就傳給 controller 產生 packet-in
2. 當收到 controller 回傳 packet-out 時，把存在 buffer 中的 packet 由 packet-out 中 output port 傳出去
3. 當收到 controller 回傳 flow_mod 時，將這個新的 flow 安裝起來，以後 match 到 flow 紀錄的目的地時就可以直接由 flow 的 action 做處理、轉傳

Control Plane:

1. 讀到 data plane 送來 controller 的 packet-in packet
2. 如果 packet 符合以下任意條件就將 packet 丟棄
   - 已經被 handle
   - 不是 Ethernet Packet
   - 是 Control Packet
   - IPv6 forward disable 而且是 multicast
   - 是 LLDP
   - MAC 是 multicast
3. 如果知道 packet 目的地而且目的地與傳來 packet-in 的 switch 直接連結，就用 flow_mod 安裝一個 rule 將所有送往該目的地的 packet 都送往 switch 連接目的地的 port，並將此 port 放到 packet-out 中，最後將 packet-out 與 flow_mod 傳回去，並結束處理過程。
4. 如果知道 packet 目的地而且找得到一條到達目的地的 path，就用 flow_mod 安裝一個 rule 將所有送往該目的地的 packet 都送往 switch 連接 path 的 port，並將此 port 放到 packet-out 中，最後將 packet-out 與 flow_mod 傳回去，並結束處理過程。
5. 如果沒有目的地紀錄、目的地與 switch 沒有連結、也找不到 path，就 flood 把 packet 往 switch 的每一個 port 送，交給下一個連接裝置處理。

## What I've learned or solved

在 part1 中我學會利用 wireshark 監聽一個 network interface 中的 packets，並利用他來觀察 org.onosproject.fwd 下了那些 flow_mod 給 switch，來了解 org.onosproject.fwd 的功能，我發現 org.onosproject.fwd 讓 switch 遇到沒遇過的 IPv4、ARP、LLDP、0x8942 packets 時都會送往 controller，遇到學過的 mac 位置與 port 對應的 IPv4 packet 後就會直接將 packet 送到對應的 port 去。

在 part2 中我學會如何透過 onos 提供的 REST API 來安裝 openflow flow rules 給 switch，並成功使用 REST API 來安裝自己寫的 ARP broadcast 與 IPv4 Routing flow rules，使 arping 與 ping 能成功運行。我自己也寫了幾個 script 來安裝、讀取、刪除 flow rules，就可以不用每次都要打一大串 curl 指令。

在 part3 中我就由查資料得知了什麼事 broadcast storm，並且利用 ring 拓譜與 arp broadcast 來產生 broadcast storm，最後成功觀察到 VM CPU 使用率的上升，也了解到避免 broadcast storm 的重要性。

在 part4 中透過 trace org.onosproject.fwd 的原始碼，我學會了如何建構一個基礎的建立 L2 routing table 的方式。了解 switch 在完全沒有 routing 資訊時，如何將 packet 傳給 controller，而 controller 透過 flood 的辦法學到目的地的 routing，再將 routing 資訊傳回給 switch 的過程。

透過這次的實驗，我對於 SDN 中 control plane 與 data plane 之間的關係以及各自的角色有更多的了解，也對 wireshark、curl 等實用網路工具更為熟悉。