NSCAP HW2

Code Explaining

> Only show the **modified** part

Main Idea of this LAB: Making uniform packets

```
|---My packet format----|

| ARP_pkt = {"type":"ARP", "operation":"", "src ip":"", "dst ip":"", "src mac":"", "dst mac":""}
|
| ICMP_pkt = {"type":"ICMP", "operation":"", "src ip":"", "dst ip":"", "src mac":"", "dst mac":""}
|
| operation : request / reply
```

- I set the packet format for all messages propagation in this LAB and the **unity** gave the convenience for progarming analyze what were we doing then.

- Following part we will show how the program work with these packets.

1. Class host

```
"""
host.clear()
"""
    def clear(self):
        # clear ARP table entries for this host
        self.arp_table.clear()
```

- Use `clear()` of dictionary in python to clear ARP table of host

```
"""
Host.send(pkt)
"""
    def send(self, pkt): # host will not help propagate others's pkt
        # print(self.name, "send pkt", pkt)
        node = self.port_to # get node connected to this host
        node.handle_packet(pkt) # send packet to the connected node

```

- I made the function of host.send() to do simplier action: just **pass to the next target**

```python
"""
Host handle received pkt
"""
    def handle_packet(self, pkt):
        # determine the destination MAC here
        '''
            Hint :
                if the packet is the type of arp request, destination MAC would be 'ffff'.
                else, check up the arp table.
        '''
        # print(self.name, "recv pkt", pkt)
        # handle incoming packets
        if pkt["dst ip"] != self.ip:
            return #drop

        # dst ip == self.ip
        if pkt["type"] == "ARP" and pkt["operation"] == "request":
            if pkt["src ip"] not in self.arp_table:
                self.update_arp(pkt["src ip"], pkt["src mac"])
            reply_pkt = {
                            "type":"ARP",
                            "operation":"reply",
                            "src mac":self.mac, # the meaning of reply is replying own mac addr
                            "src ip":self.ip,
                            "dst mac":pkt["src mac"],
                            "dst ip":pkt["src ip"]
                        }
            self.send(reply_pkt)
        elif pkt["type"] == "ARP" and pkt["operation"] == "reply":
            if pkt["src ip"] not in self.arp_table:
                self.update_arp(pkt["src ip"], pkt["src mac"])
            # ICMP request
            request_pkt = {
                            "type":"ICMP",
                            "operation":"request",
                            "src ip":self.ip,
                            "dst ip":pkt["src ip"],
                            "src mac":self.mac,
                            "dst mac":pkt["src mac"]
                        }
            self.send(request_pkt)

        elif pkt["type"] == "ICMP" and pkt["operation"] == "request":
            reply_pkt = {
                            "type":"ICMP",
                            "operation":"reply",
                            "src ip":self.ip,
                            "dst ip":pkt["src ip"],
                            "src mac":self.mac,
                            "dst mac":pkt["src mac"]
                        }
            self.send(reply_pkt)
        elif pkt["type"] == "ICMP" and pkt["operation"] == "reply":
            pass
            # print("ICMP success after", self.name, "pinging.")

```

- handlePKT do following actions

1. If the `dst_ip` in the packet is **not** the ip of receiving host ➡️ DROP

   (We will receive the pkt which [dst_ip] == our [own ip])

2. If the pkt is **ARP request** pkt, we need to send back to the source host as a feeback.

3. If **ARP reply**, we need to make an `ICMP request` as a feeback.

> Note: This is what `ping` do when (pinged host) in (pinging host) ARP table.

4. If **ICMP request**, we need to make an `ICMP reply` as a feeback.

5. If **ICMP reply**, the host will do nothing.

```
"""
host.ping(dst_ip)
"""
    def ping(self, dst_ip):
        # handle a ping request
        if dst_ip in self.arp_table:
            request_pkt = {
                            "type"  :"ICMP",
                            "operation" : "request",
                            "src ip":self.ip,
                            "dst ip":dst_ip,
                            "src mac":self.mac,
                            "dst mac":self.arp_table[dst_ip]
                          }
            self.send(request_pkt)
        else:
            # broad an ARP request to all host by setting dst = 'ffff'
            request_pkt = {
                            "type"   :"ARP",
                            "operation" : "request",
                            "src mac":self.mac,
                            "src ip" :self.ip,
                            "dst mac" :'ffff',
                            "dst ip"  : dst_ip
                          }
            self.send(request_pkt)
```

1. If pinged host **IN** pinging host ARP table
   ➡️ Send `ICMP request` to pinged host.
2. Else
   ➡️ Broadcast by sending `ARP request` with [dst_mac] is "ffff".

```
"""
Update ARP Table
"""
    def update_arp(self, newHostIp, newHostMac):
        # update ARP table with a new entry
        for host in host_dict: # Search in all hosts
            if host_dict[host].ip == newHostIp:
                """Don't use global information"""
                # self.arp_table[newHostIp] = host_dict[host].mac
                self.arp_table[newHostIp] = newHostMac
                return
```

1. I have a principle to the local device in this LAB:
   Don't use **GLOBAL information**; that is, we will not access address information in `host_dict` or `switch_dict` to revise our **LOCAL** table.

2. Class switch

```
class switch:
"""
Switch clear up mac table
"""
    def clear(self):
        # clear MAC table entries for this switch
        self.mac_table.clear()
```

- Use dictionary.clear() to clean up mac table of the switch

```
"""
Switch send pkt
"""
    def send(self, idx, pkt): # send to the specified port
        # idx: target port
        # node may be host or switch
        # print(self.name, "port", idx, "send pkt", pkt)

        node = self.port_to[idx]
        node.handle_packet(pkt) # pass pkt to "port_to" node

```

- idx: target port
- node can be `host` or `switch`
- We can get the **target device** by the member list **self.port_to** and target port `idx`
- self.port_to[`idx`] is the target device
- After all, we can **pass the pkt** to the target device.

```
"""
Switch flood to all ports other than incoming port
"""
    def flood(self, pkt, inPort):
        # flood to all ports other than inPort
        for idx in range(self.port_n):
            if (idx == inPort): continue
            self.send(idx, pkt)
```

- the parameter `inPort` is the port `pkt` coming from.
- Use for loop to send the pkt for (port `idx`) except `inPort`

```python
def update_mac(self, mac):
        # update MAC table with a new entry
        """How to NOT use GLOBAL information"""
        """We can use incoming port? """

        """NO, We need to take the empty port for a switch. Since"""
        """outPort for a switch is not the inPort of another switch"""

        """NO concept which called "empty port", we need to use device
           to give the ports.
        """
        idx = 0
        for dev in self.port_to:
            if (dev.name in host_dict) and (dev.mac == mac):
                idx = self.port_to.index(dev)
                break
            elif (dev.name in switch_dict) and (mac in dev.mac_table):
                idx = self.port_to.index(dev)
                # if we have another one mac in this dev's mac_table, we assign the same port
                break

        self.mac_table[mac] = idx
```

- My main idea is that **use device** to allocate port, not the mac address.
- The diveces are in the **self.port_to** list, so we can imply the port for new adding mac address in two cases.

> Case 1: if the device is a **host** && parameter `mac` equal to device's MAC ➡️ assign the **index** of device in self.port_to list as the **port** in MAC table.
> (i.e. self.port_to.**index**(dev) = the **port** of new adding MAC address)

> Case 2: if the device is a **switch** && parameter `mac` **in** device's MAC table➡️ we can assign the index to be the port in the same way.

```
"""
Switch handle received pkt
"""
    def handle_packet(self, pkt):
        # handle incoming packets
        # print(self.name, "recv pkt", pkt)

        if pkt["type"] == "ARP" and pkt["operation"] == "request":
            # Learning incoming port for mac table
            if pkt["src mac"] not in self.mac_table:
                self.update_mac(pkt["src mac"])

            if pkt["dst mac"] == 'ffff': # flood
                self.flood(pkt, self.mac_table[pkt["src mac"]])
            elif pkt["dst mac"] in self.mac_table: # searching
                # searching mac table will get the port of learned port
                self.send(self.mac_table[pkt["dst mac"]], pkt)
            else:
                # searching failed, just flood
                self.flood(pkt, self.mac_table[pkt["src mac"]])

        else: # hold for ARP request, ICMP request, ICMP reply
            # Learning incoming port for mac table
            if pkt["src mac"] not in self.mac_table:
                self.update_mac(pkt["src mac"])


            if pkt["dst mac"] in self.mac_table: # searching
                # searching mac table will get the port of learned port

                self.send(self.mac_table[pkt["dst mac"]], pkt)
            else:
                # searching failed, just flood
                self.flood(pkt, inPort = self.mac_table["src mac"])

```

- handlePKT do following actions

1. If the pkt is **ARP request** pkt
    - update MAC table if the src_mac not in the table
    - if the `dst_mac` is "ffff" ➡️ **flood**
    - if the `dst_mac` in MAC table ➡️ **send** pkt to the port in MAC table
    - if the `dst_mac` **not** in MAC table ➡️ **flood**

2. Else case: (ARP reply, ICMP request, ICMP reply)
    - update MAC table if the src_mac not in the table
    - if the `dst_mac` in MAC table ➡️ **send** pkt to the port in MAC table

- if the `dst_mac` **not** in MAC table ➡️ **flood**

> Note: we need to check "ffff" case only

other modified

- In the bottom of function `def set_topology():`

  I change the switch.**port_n** into the number of **connected devices** of the switch since it is more reasonable for the meaning of port_n —— **number of ports** on this switch

> i.e. number of ports = number of connected devices

Answer Questions

1. What is the difference between broadcasting and flooding in a network?

> Broacasting: For switch, broadcasting will let **all the devices** know some information in this LAN.
>
> Flood: send to all ports except incoming port, we **don't need to promise** all devices should receive the packet(information).

> In essense, **flooding** when switch doesn't know the device's port; however, **broadcasting** will send to all devices with destination MAC address "ffff".

2. Explain the steps involved in the process of h1 ping h7 when there are no entries in the switch's MAC table and the host's ARP table.

- packet: ARP request

  1. h1 -> s1: ARP request with dst_mac = "ffff"

  2. s1 -> h2, s2: flooding the packet

     - s1 update MAC table
     - h2 **DROP** packets

  3. s2 -> s3, s7: flooding the packet

     - s2 update MAC table
     - s3 -> h3, h4 (flooding) -> h3, h4 **DROP** packets
       - s3 update MAC table

  4. s7 -> s5: flooding the packet

     - s7 update MAC table

5. s5 -> s4, s6: flooding the packet
    - s5 update MAC table
    - s4 -> h5, h6 (flooding) -> h5, h6 **DROP** packets
        - s4 update MAC table
    - s6 -> h7, h8 (flooding)
      -> h8 **DROP** packets
      -> h7 receive the packets and make a reply with ARP reply packet -> h7 **update** ARP table
        - s6 update MAC table

- packet: ARP reply with dst_mac = h1mac
    1. h7 -> s6
        - s6 update MAC table
    2. s6 -> s5 (checking MAC table of h1mac)
        - s5 update MAC table
    3. s5 -> s7 (checking MAC table of h1mac)
        - s7 update MAC table
    4. s7 -> s2 (checking MAC table of h1mac)
        - s2 update MAC table
    5. s2 -> s1 (checking MAC table of h1mac)
        - s1 update MAC table
    6. s1 -> h1 (checking MAC table of h1mac)
    7. h1 receive packet
        - **update** ARP table
        - make ICMP request to h2

- packet: ICMP request with dst_mac = h2mac, dst_ip = h1ip
    1. h1 -> s1
    2. s1 -> s2 (checking MAC table of h2mac)
    3. s2 -> s7 (checking MAC table of h2mac)
    4. s7 -> s5 (checking MAC table of h2mac)
    5. s5 -> s6 (checking MAC table of h2mac)
    6. s6 -> h7 (checking MAC table of h2mac)
    7. h6 receive ICMP request

- END of this ping

3. What problem can arise when connecting s2 and s5 together and thus creating a switching loop? How can this issue be addressed? (You should mention the specific algorithm or protocol used.)

- We will form a loop **spawning** infinite packets and crash the LAN.

- The spanning tree protocol