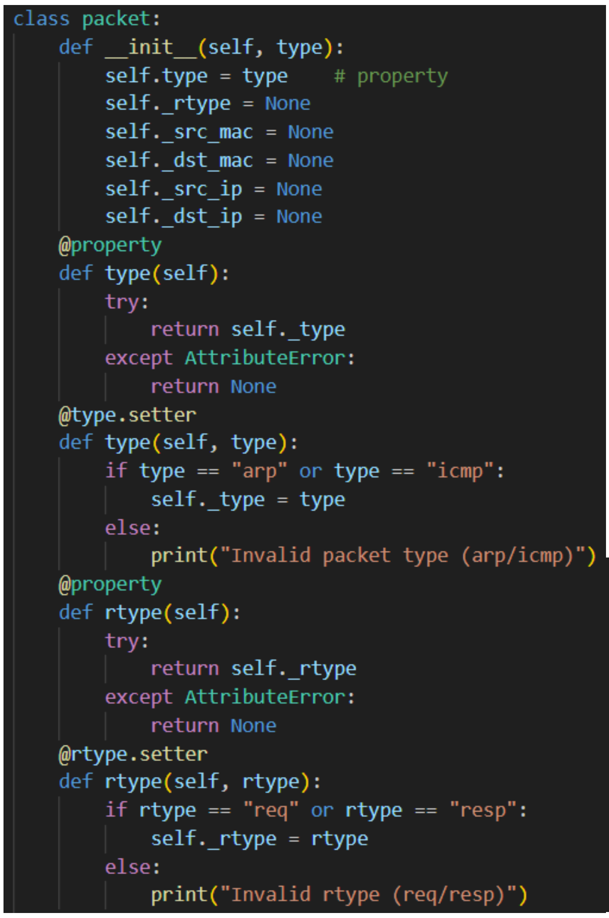
ID: 109550194
Name: 龍偉亮

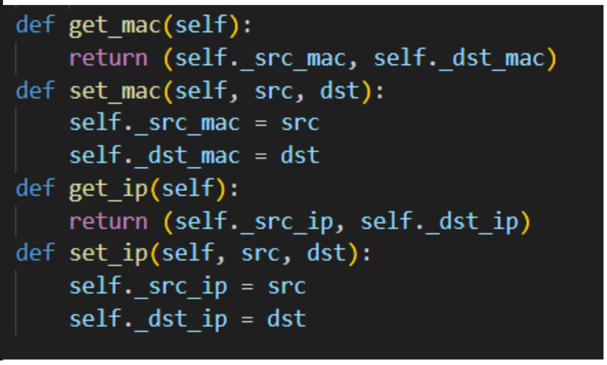
Homework 2 Report

Code Explanation

class packet



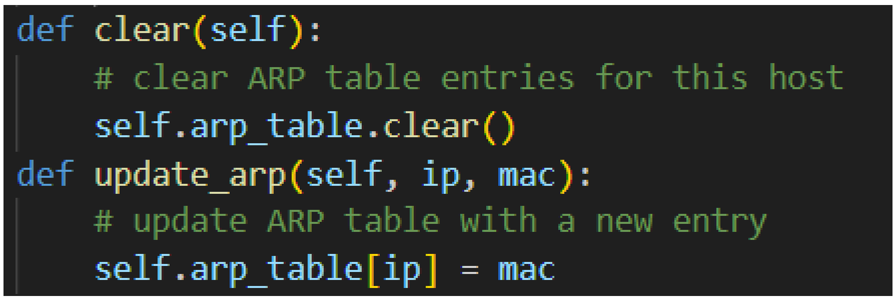
```python
class packet:
    def __init__(self, type):
        self.type = type    # property
        self._rtype = None
        self._src_mac = None
        self._dst_mac = None
        self._src_ip = None
        self._dst_ip = None
    @property
    def type(self):
        try:
            return self._type
        except AttributeError:
            return None
    @type.setter
    def type(self, type):
        if type == "arp" or type == "icmp":
            self._type = type
        else:
            print("Invalid packet type (arp/icmp)")
    @property
    def rtype(self):
        try:
            return self._rtype
        except AttributeError:
            return None
    @rtype.setter
    def rtype(self, rtype):
        if rtype == "req" or rtype == "resp":
            self._rtype = rtype
        else:
            print("Invalid rtype (req/resp)")
```



```python
def get_mac(self):
    return (self._src_mac, self._dst_mac)
def set_mac(self, src, dst):
    self._src_mac = src
    self._dst_mac = dst
def get_ip(self):
    return (self._src_ip, self._dst_ip)
def set_ip(self, src, dst):
    self._src_ip = src
    self._dst_ip = dst
```

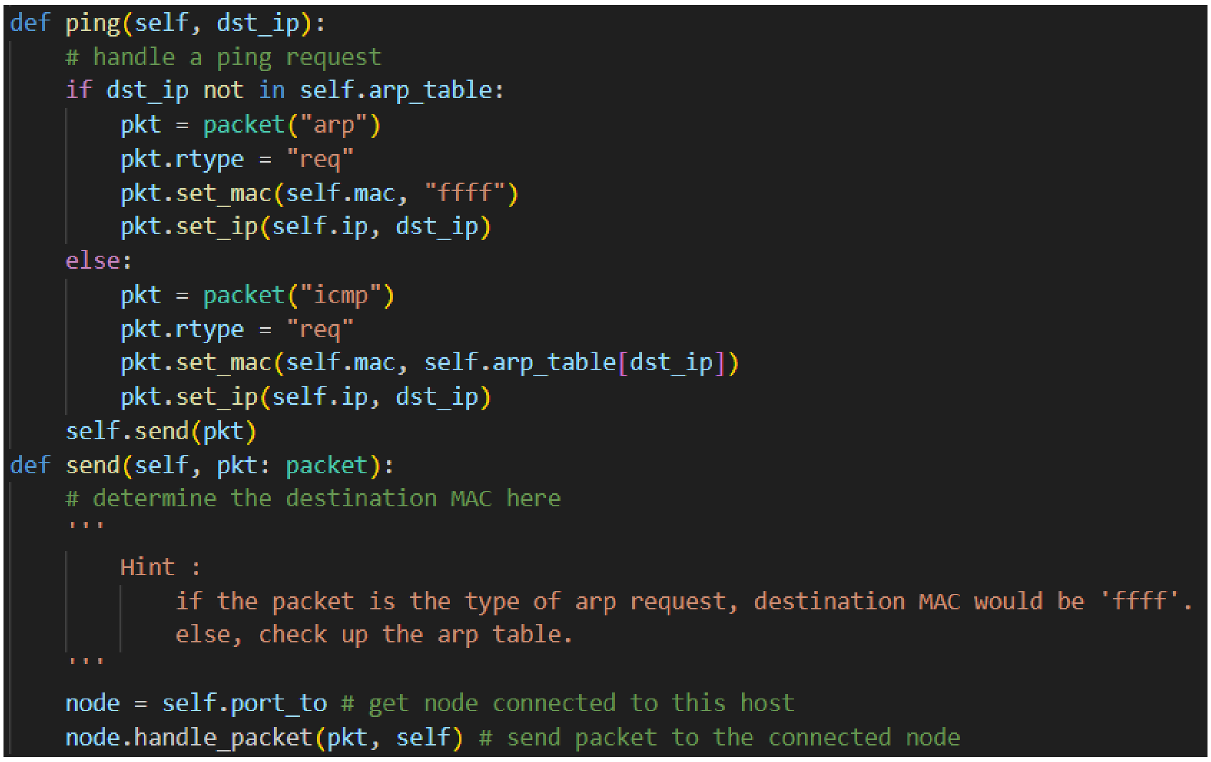
I wrote a class packet to help with storing required info like packet type (arp / ip), rtype (request / response), source and destination MAC and ip addresses.
The property decorators and functions are just getter and setters for their corresponding data.

class host



```python
def clear(self):
    # clear ARP table entries for this host
    self.arp_table.clear()
def update_arp(self, ip, mac):
    # update ARP table with a new entry
    self.arp_table[ip] = mac
```

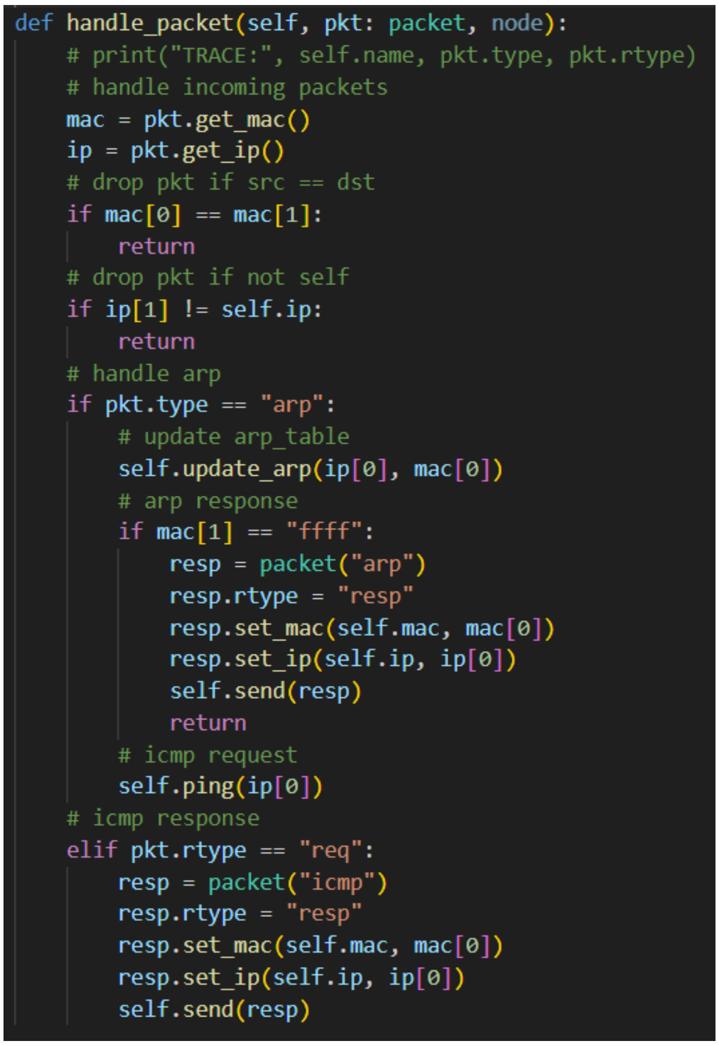
clear() and update_arp() are pretty self-explanatory, clear() clears the ARP table while update_arp() sets a key and value pair in it.



```python
def ping(self, dst_ip):
    # handle a ping request
    if dst_ip not in self.arp_table:
        pkt = packet("arp")
        pkt.rtype = "req"
        pkt.set_mac(self.mac, "ffff")
        pkt.set_ip(self.ip, dst_ip)
    else:
        pkt = packet("icmp")
        pkt.rtype = "req"
        pkt.set_mac(self.mac, self.arp_table[dst_ip])
        pkt.set_ip(self.ip, dst_ip)
    self.send(pkt)
def send(self, pkt: packet):
    # 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.
    '''
    node = self.port_to # get node connected to this host
    node.handle_packet(pkt, self) # send packet to the connected node
```

ping() handles ping request such that if destination IP address is not in the ARP table, an ARP request packet with the host's source MAC and IP addresses and destination MAC address of "ffff" and ip address is created and sent.

Or if such IP address corresponds to a MAC address in the ARP table, an ICMP request packet is made and sent.



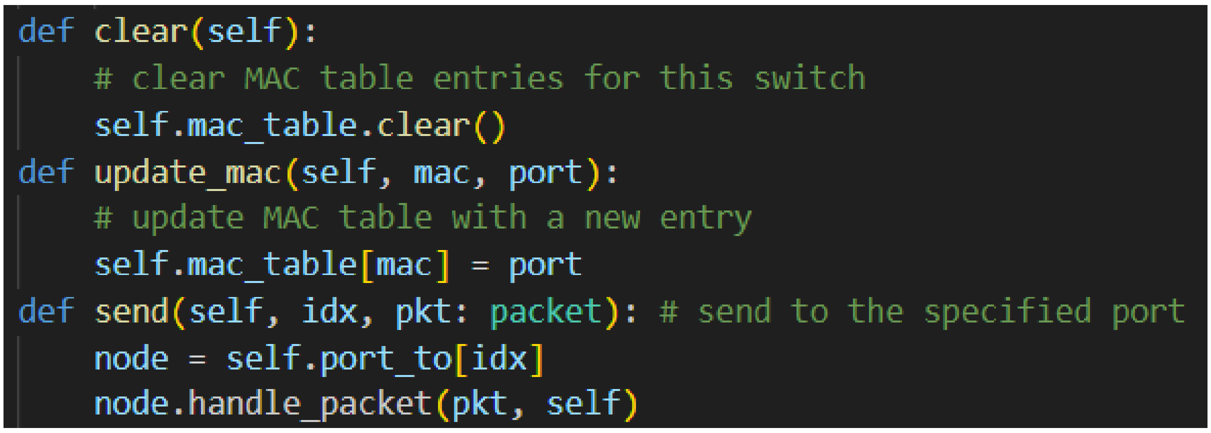
```python
def handle_packet(self, pkt: packet, node):
    # print("TRACE:", self.name, pkt.type, pkt.rtype)
    # handle incoming packets
    mac = pkt.get_mac()
    ip = pkt.get_ip()
    # drop pkt if src == dst
    if mac[0] == mac[1]:
        return
    # drop pkt if not self
    if ip[1] != self.ip:
        return
    # handle arp
    if pkt.type == "arp":
        # update arp_table
        self.update_arp(ip[0], mac[0])
        # arp response
        if mac[1] == "ffff":
            resp = packet("arp")
            resp.rtype = "resp"
            resp.set_mac(self.mac, mac[0])
            resp.set_ip(self.ip, ip[0])
            self.send(resp)
            return
        # icmp request
        self.ping(ip[0])
    # icmp response
    elif pkt.rtype == "req":
        resp = packet("icmp")
        resp.rtype = "resp"
        resp.set_mac(self.mac, mac[0])
        resp.set_ip(self.ip, ip[0])
        self.send(resp)
```

handle_packet() first determines if the source and destination MAC addresses are the same, drop packet if true. Then checks to see if the destination IP address to see if it is the intended recipient.

If the packet is an ARP packet, it will first update its ARP table with the source address. Then it determines the ARP packet is a request if the destination MAC address is “ffff”, it will create and send an ARP response. But if the ARP packet is a response from another host, it will proceed to send an ICMP request with ping().

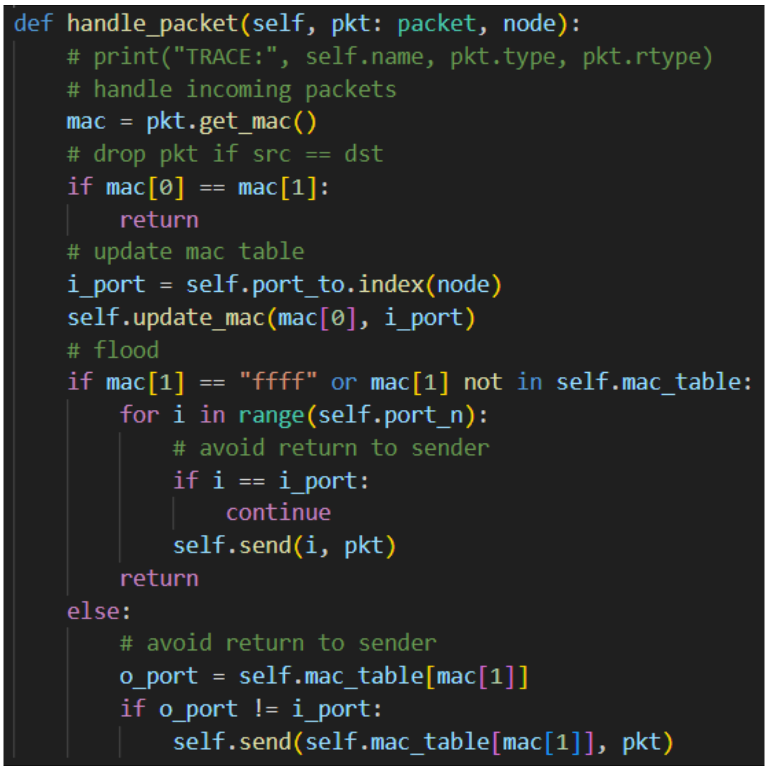
And if the packet is an ICMP request, a response will be made and sent.

class switch



```python
def clear(self):
    # clear MAC table entries for this switch
    self.mac_table.clear()
def update_mac(self, mac, port):
    # update MAC table with a new entry
    self.mac_table[mac] = port
def send(self, idx, pkt: packet): # send to the specified port
    node = self.port_to[idx]
    node.handle_packet(pkt, self)
```

clear() clears the MAC table, update_mac() sets the key and value pair in the table, send() sends a packet by calling the corresponding node on port idx to handle packet.



```python
def handle_packet(self, pkt: packet, node):
    # print("TRACE:", self.name, pkt.type, pkt.rtype)
    # handle incoming packets
    mac = pkt.get_mac()
    # drop pkt if src == dst
    if mac[0] == mac[1]:
        return
    # update mac table
    i_port = self.port_to.index(node)
    self.update_mac(mac[0], i_port)
    # flood
    if mac[1] == "ffff" or mac[1] not in self.mac_table:
        for i in range(self.port_n):
            # avoid return to sender
            if i == i_port:
                continue
            self.send(i, pkt)
        return
    else:
        # avoid return to sender
        o_port = self.mac_table[mac[1]]
        if o_port != i_port:
            self.send(self.mac_table[mac[1]], pkt)
```

The switch will first determine if source and destination MAC addresses are the same, and drop packet if true. Then obtain the source MAC address and incoming port number to update its MAC table.

If destination MAC address is “ffff” or if the address doesn’t correspond to any port number in the MAC table, proceed to flood every port other than the incoming port.

But if the destination MAC address has a record in the MAC table, forward the packet to the next node but avoid sending back to incoming port.

Questions

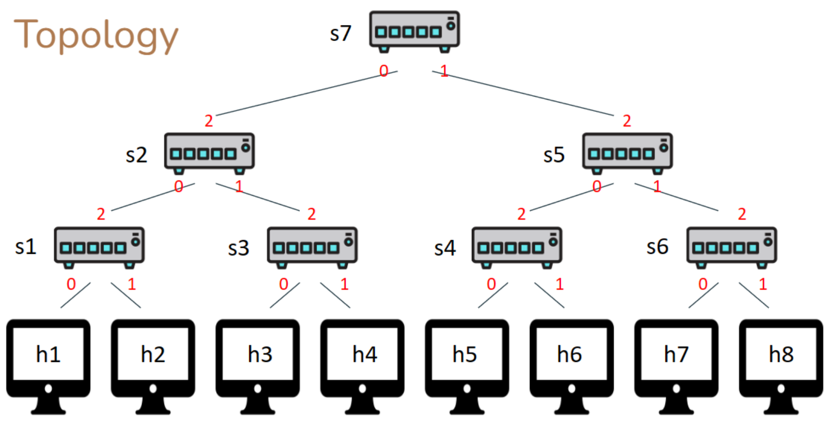
1. *What is the difference between broadcasting and flooding in a network?*

From my understanding, broadcasting and flooding have very similar meanings and outcome. But flooding is an action taken when a switch encounters a broadcast frame ffff.ffff.ffff, it will flood to every one of its ports other than the incoming port.

And broadcast refers to a particularly addressed frame or a packet, rather than the action / process as stated by a Cisco employee in 2012 (source). But it seems the two terms are very commonly used interchangeably.

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.*

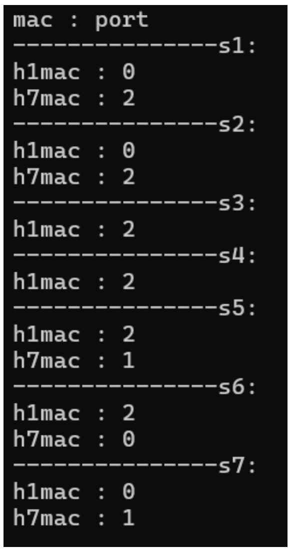
When there are no entries in any switch's MAC table or any host's ARP table, "h1 ping h7" starts at h1 attempting to send a packet to the ip of h7, which is "h7ip" but it doesn't have h7's MAC address. So it generates an ARP request with its own MAC and IP addresses as source, "ffff" and "h7ip" as the destination MAC and IP address respectively.



The above shows the topology for this homework, so h1 sends an ARP request packet to s1. s1 updates its MAC table with the incoming source MAC address and port number, then it finds out the destination MAC is "ffff" and proceeds to flood the packet out of every port other than the incoming port. This then happens to the switches s2, s3, s7, s5, s4, s6.

The hosts h2, h3, h4, h5, h6, h8 will receive the ARP request from the flooding but they will all discard the packet when the destination IP address doesn't match their own. As for h7, it receives the ARP request and updates its ARP table with the source MAC and IP addresses, then sends an ARP response back to h1.

At this point, every switch in the network now knows where h1 is coming from, as shown in the screenshot below,



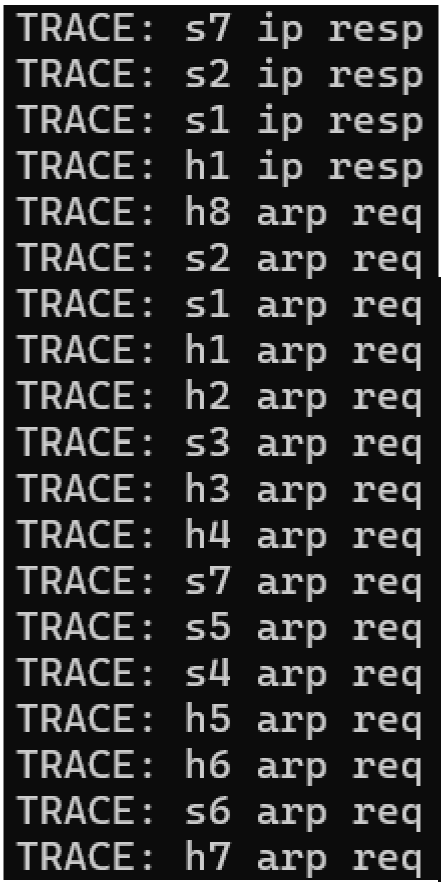
```
mac : port
---------------s1:
h1mac : 0
h7mac : 2
---------------s2:
h1mac : 0
h7mac : 2
---------------s3:
h1mac : 2
---------------s4:
h1mac : 2
---------------s5:
h1mac : 2
h7mac : 1
---------------s6:
h1mac : 2
h7mac : 0
---------------s7:
h1mac : 0
h7mac : 1
```

So the ARP response from h7 to h1 will follow the path of
h7 -> s6 -> s5 -> s7 -> s2 -> s1 -> h1
without the need of flooding. And when h1 receives an ARP response from h7, it updates its ARP table with the MAC and IP addresses of h7, then proceeds to ping h7 with an ICMP request to get a response from h7.

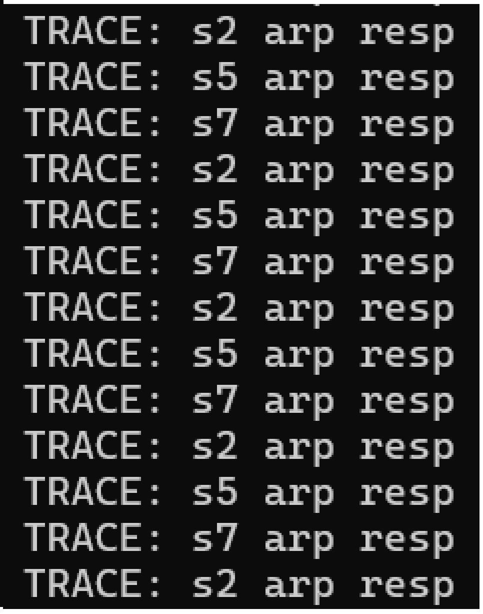
3. *What problem can arise when connecting s2 and s5 together and thus creating a switching loop? How can this issue be addressed? (You should mention the specific algorithm or protocol used.)*

Connecting s2 and s5 creates a switching loop and problems like packet duplication and MAC flapping can occur. MAC flapping happens when the same MAC address comes in rapidly from different ports. One way to address this issue is with the spanning tree protocol (STP) whereby the network is trimmed automatically into a spanning tree without loops.

In my experiment with “h1 ping h7”,



```
TRACE: s7 ip resp
TRACE: s2 ip resp
TRACE: s1 ip resp
TRACE: h1 ip resp
TRACE: h8 arp req
TRACE: s2 arp req
TRACE: s1 arp req
TRACE: h1 arp req
TRACE: h2 arp req
TRACE: s3 arp req
TRACE: h3 arp req
TRACE: h4 arp req
TRACE: s7 arp req
TRACE: s5 arp req
TRACE: s4 arp req
TRACE: h5 arp req
TRACE: h6 arp req
TRACE: s6 arp req
TRACE: h7 arp req
```



```
TRACE: s2 arp resp
TRACE: s5 arp resp
TRACE: s7 arp resp
TRACE: s2 arp resp
TRACE: s5 arp resp
TRACE: s7 arp resp
TRACE: s2 arp resp
TRACE: s5 arp resp
TRACE: s7 arp resp
TRACE: s2 arp resp
TRACE: s5 arp resp
TRACE: s7 arp resp
TRACE: s2 arp resp
```

The left screenshot shows that even after an ARP request, response, ICMP request, response were completed due to the nature of the function calling sequence. The ARP request packet was duplicated at s5, flooding to all of its ports including to s2 again.

Then Python raises
RecursionError: maximum recursion depth exceeded while calling a Python object when the ARP response packet gets stuck in a s2 -> s5 -> s7 -> s2 loop.