109611066_Report

1. Setting.py

```
# Precondition: None
# Postcondition: Return the hosts in string, separated by space.
def get_hosts()

# Precondition: None
# Postcondition: Return the switches in string, separated by space.
def get_switches()

# Precondition: None
# Postcondition: Return the ip addresses of each host, using a dictionary. The key is the
host name, and the value is the ip address.
def get_ip()

# Precondition: None
# Postcondition: Return the mac addresses of each host, using a dictionary. The key is the
host name, and the value is the mac address.
def get_mac()

# Precondition: None
# Postcondition: Return the links between the hosts and switches. The format is 'a,b'
where a is the source node and b is the destination node.
def get_links()
```

2. 109611066.py

2.1. Custom Packet Classes

I defined some data structures to store fields in the packet:

```
# custom packet class
ETHERTYPE_ARP = 0x0806
ETHERTYPE_IP = 0x0800
class EthernetFrame:
    def __init__(self, src_mac, dst_mac, ethertype, payload):
        self.src_mac = src_mac
        self.dst_mac = dst_mac
        self.ethertype = ethertype
        self.payload = payload

ARP_HARDWARE_TYPE_ETHERNET = 1
ARP_PROTOCOL_TYPE_IP = 0x0800
ARP_OPCODE_REQUEST = 1
ARP_OPCODE_REPLY = 2
class ARPPacket:
    def __init__(self, hardware_type, protocol_type, opcode, sender_mac, sender_ip,
target_mac, target_ip):
        self.hardware_type = hardware_type
        self.protocol_type = protocol_type
        self.opcode = opcode
        self.sender_mac = sender_mac
        self.sender_ip = sender_ip
        self.target_mac = target_mac
```

```
        self.target_ip = target_ip

IP_PROTOCOL_ICMP = 1
class IPPacket:
    def __init__(self, src_ip, dst_ip, protocol, payload):
        self.src_ip = src_ip
        self.dst_ip = dst_ip
        self.protocol = protocol
        self.payload = payload

ICMP_TYPE_ECHO_REQUEST = 8
ICMP_TYPE_ECHO_REPLY = 0
class ICMPPacket:
    def __init__(self, type, payload):
        self.type = type
        self.payload = payload
```

2.2. Host Class

For the host class, I included the packet parsing in function `handle_packet` using nested if-else statements.

```
def handle_packet(self, packet, income_name):
        # handle incoming packets
        # Preconditions: packet is an object of class
        #                EthernetFrame.
        # Postconditions: packet is processed and appropriate
        #                 action is taken.
        if packet.ethertype == ETHERTYPE_ARP:
            arp_packet = packet.payload
            if arp_packet.opcode == ARP_OPCODE_REQUEST:
                if arp_packet.target_ip == self.ip:
                    # send ARP reply
            elif arp_packet.opcode == ARP_OPCODE_REPLY:
                # update ARP table
        elif packet.ethertype == ETHERTYPE_IP:
            ip_packet = packet.payload
            if ip_packet.dst_ip == self.ip:
                if ip_packet.protocol == IP_PROTOCOL_ICMP:
                    icmp_packet = ip_packet.payload
                    if icmp_packet.type == ICMP_TYPE_ECHO_REQUEST:
                        # send ICMP echo reply
```

The function `handle_packet` will call the function `send` for replying icmp echo request and arp request.

For function `ping`, I used `if` statement to check if destination ip is in the `arp_table`. If it is, then send the icmp echo request. Otherwise, send arp request first.

```
def ping(self, dst_ip):
    if self.arp_table.get(dst_ip) is not None:
        # send ICMP echo request
    elif self.ip == dst_ip:
        # handle ping to self
        self.update_arp([self.ip, self.mac])
    else:
        # send an ARP request
        if self.arp_table.get(dst_ip) is not None:
```

```
            # if the destination ip is found,
            # send ICMP echo request
            self.ping(dst_ip)
```

The `send` function is simple, it just calls the next node's `handle_packet` function.

```
def send(self, packet):
    node = self.port_to # get node connected to this host
    node.handle_packet(packet, self.name) # send packet to the connected node
```

2.3. Switch Class

For the switch class, similar `send` and `handle_packet` functions are implemented. The only difference is that in function `handle_packet`, the switch will find the input port using the `income_name`. Then it will flood the packet if the destination mac is not in the `mac_table`.

```
def send(self, idx, packet): # send to the specified port
    node = self.port_to[idx]
    node.handle_packet(packet, self.name)
def handle_packet(self, packet, income_name):
    # handle incoming packets
    # Preconditions: packet is an object of class
    #                EthernetFrame.
    # Postconditions: packet is processed and appropriate
    #                 action is taken.

    # determine the incoming port number from
    # the name of the incoming node
    income_port = None
    for i in range(self.port_n):
        if self.port_to[i].name == income_name:
            income_port = i
            break

    # update MAC table
    self.update_mac([packet.src_mac, income_port])

    # forward the packet or flood
    if packet.dst_mac in self.mac_table:
        self.send(self.mac_table[packet.dst_mac], packet)
    else:
        for i in range(self.port_n):
            if i != income_port:
                self.send(i, packet)
```

2.4. run_net in Main Function

In the main function, I used a `try` statement to handle the `EOFError` exception for testing purposes:

```
try:
    command_line = input(">> ")
except EOFError:
    return 0
```

I also added a `"show_all"` command to display the ARP table and MAC table of each node.

```
if command_line.strip() == 'show_all':
    show_table('all_hosts')
    show_table('all_switches')
```

3. Questions

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

Broadcasting is a method of sending a message to all nodes in a network. Flooding on the other hand, is also a method of sending a message to all nodes in a network, but it is used when the destination of the message is unknown, and the message will be received by only the intended recipient.

3.2. 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.

1. `h1` does not have the MAC address of `h7` in its ARP table, so it sends an ARP request to find the MAC address of `h7`.
2. The ARP request is received by the switch, which does not have the MAC address of `h1` in its MAC table. Therefore, the switch associates the MAC address of `h1` with the port `0`. The switch does not have the MAC address of `h7` in its MAC table, so it floods the ARP request to all ports except the port `0`.
3. The ARP request is received by `s2` and `h2`. `h2` ignores the ARP request because the destination IP address does not match its own IP address. `s2` does not have the MAC address of `h1` in its MAC table, so it associates the MAC address of `h1` with the port `0`. The switch does not have the MAC address of `h7` in its MAC table, so it floods the ARP request to all ports except the port `0`.
4. Switches `s3` and `s7` receive the ARP request. Same as the previous steps, `s3` knows `h1` is on port `2` and `s7` knows `h1` is on port `0`. The ARP requests flooded by `s3` are ignored by `h3` and `s4`. The ARP requests flooded by `s7` are sent to `s5`.
5. `s5` receives the ARP request and knows that `h1` is on port `2`. The ARP request is flooded to `s4` and `s6`.
6. `s4` receives the ARP request and knows that `h1` is on port `1`. The ARP requests are flooded to `h5` and `h6`, which are ignored. `s6` knows that `h1` is on port `2`, and the ARP request is flooded to `h7` and `h8`.
7. `h8` ignores the ARP request. `h7` on the other hand, receives the ARP request and sends an ARP reply to `h1`.
8. The switches already know the port for `h1`, so the ARP reply is sent to `h1` in the path `h7 -> s6 -> s5 -> s7 -> s2 -> s1 -> h1`. The switches in the path update their MAC tables accordingly.
9. `h1` now has the MAC address of `h7` in its ARP table, and it sends an ICMP echo request to `h7`.
10. The ICMP echo request is sent to `h7` in the path `h1 -> s1 -> s2 -> s7 -> s5 -> s6 -> h7`.
11. `h7` receives the ICMP echo request and sends an ICMP echo reply to `h1`. The reply is sent to `h1` in the path `h7 -> s6 -> s5 -> s7 -> s2 -> s1 -> h1`.

3.3. 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.)

When connecting `s2` and `s5` together, a switching loop is created. This can cause broadcast storms, where packets are endlessly forwarded between the switches.

The issue can be addressed using the Spanning Tree Protocol. It elects a root bridge and disables the links that are not part of the spanning tree. This prevents switching loops and broadcast storms.