README for LAB #2

Yuyang Zhou
2000013061

1 Implemented

The features implemented are `PT1, PT2, PT3`. and this document include the answer of `WT2` $\sim$ `WT4`, `CP3` $\sim$ `CP6`

2 WT2

(1) The following in ARP Reply is the same as the Sender MAC address in ARP Request:

- the Receiver MAC address in ARP Reply.
- the destination MAC address in the Ethernet Header in the last Hop.

(2) 1674 non-fragment packets in total.
(3) The length of `IPv6` header is 40 bytes, while length of `IPv4` header is 20 bytes.

3 WT3 & WT4

The answer of these two problems are shown below, because the answer are highly related to each other. I use an ARP-Like protocol to implement routing, and find the MAC address of the next hop. The detail is shown below.

- when I use ARP in server A, trying to find the information about server D, I need to broadcast ARP-request to all devices which directly connected to server A. and when server B receive an ARP-request from MAC address C, server B knows that:

  * If I want to send any message to server A, I can send along the route that server send an ARP-request to be backwards, and the route is surely exists if all servers works well.

- So server B can directly set the following rule: If I want to send a message to A, the next hop address is C. The correctness is shown above. The same holds for ARP-reply request.

- If server B contains the next hop to the destination, or server B itself is just the destination, it can simply send an ARP-reply back to server A. Also the send-back procedure doesn't require any ARP-request. This is because that on the route server A send the ARP-request to it, the next hop to server A has already been set-up accroding to the rule above, so every next-hop-query is success along the route.

- Otherwise, server B will try to broadcast the ARP-request to all its neighbors. but it may cause infinity ARP requests if we doesn't handle it well. So I design the following broadcast rule:

  * If server A have not send any message to me, or I have not broadcast any information about server D, I will broadcast the message; otherwise, I will not try to broadcast it, and there is no reply.

- The rule above works well if there are no host broke down, and no new servers add in. and it will not cause request loop. Also it could give a really low delay(lower than 100ms in testing) even we try to send a message to a new server.
- One of ways to improve it is to restart server in some specific situations, which could help to solve the broke down server, or the new added server. But it may be a bit complex, so I doesn't implement it.

Using the ARP-like protocol shown above, I can always find the MAC address for the next-hop, and the MAC address of the destination, if the network is connected and static. and these imformations can be used in `sendFrames`.

4 Code Arrangement

macro.h

define the macros used in the following programs.

- `swap(T &a, T &b)`
  * Swap two items

constant.h

define the constants used in the following programs.

type.h

define the specific types used in the following programs.

name2addr.h

find the Mac Address(actually hardware address) & IP Address of specific device

- `findMac(const char* device, uint8_t* mac_addr)`
  * find the Mac Address of specific device
  * @param device the name of the device request for Mac address
  * @param mac_addr the pointer to the memory used to save the Mac address
  * @return 0 on success, 1 on failure.

- `findIP(const char* device, struct in_addr &ip_addr)`

  * find the IP Address of specific device

  * @param device the name of the device request for Mac address

  * @param ip_addr the pointer to the memory used to save the IP address

  * @return 0 on success, -1 on failure.

device.h

Library supporting network device management.

- `checkDevice(const char* device)`

  * Check whether the device name exists in the network.

  * @param device Name of network device to check.

  * @return True on success , False on error.

- `addDevice(const char* device)`

  * Add a device to the library for sending / receiving packets .

  * @param device Name of network device to send / receive packet on .

  * @return A non - negative _device - ID_ on success , -1 on error .

- `findDevice(const char* device)`

  * Find a device added by 'addDevice'.

  * @param device Name of the network device .

  * @return A non - negative _device - ID_ on success , -1 if no such device was found .

In order to manage this, the code contains two structures `DeviceNode` and `DeviceManager`, and implements the following methods

- `DeviceNode`

  - `char* device_names` The name of the device
  - `pcap_t* receive_handler` The handler used to receive messages
  - `pcap_t* send_handler` The handler used to send messages
  - `frameReceiveCallback callback` The default link layer callback function
  - `IPPacketReceiveCallback callback` The default IP Layer callback function
  - `uint8_t mac_addr[8]` Mac address of the device
  - `struct in_addr ip_addr` IP address of the device
  - `int index` The index of the device

  - `bool isEqualDevice(const char* device)`
    * Check if the name of the device is equal to 'device'
    * @param device The name of device to check out
    * @return True on same, False on different
  - `void setCallback(frameReceiveCallback __callback__)`
    * Set up the link layer callback function of the device
    * @param ___callback___ the pointer of callback function to set up
  - `void setIPCallback(IPPacketReceiveCallback __callback__)`
    * Set up the IP layer callback function of the device
    * @param ___callback___ the pointer of callback function to set up
  - `int setDevice(const char* device)`
    * Initalize the deviceNode with name device
    * @param device The name of device used to set up
    * @return 0 on success, -1 on failure.

- `DeviceManager`
  - `DeviceNode** device_list` The pointer to the piece of memory, to save the pointer of `DeviceNode`s
  - `int device_count` The number of devices in the manager
  - `int device_bound` The maximum number of devices can be saved now
  - `DeviceNode* operator [](const int index)`
    * Return the pointer of the index-th DeviceNode
    * @param index The index of device to find
  - `int addDevice(const char* device)`
    * Add a device to the library for sending / receiving packets .
    * @param device Name of network device to send / receive packet on .
    * @return A non - negative _device - ID_ on success , -1 on error .
  - `findDevice(const char* device)`
    * Find a device added by 'addDevice'.
    * @param device Name of the network device .
    * @return A non - negative _device - ID_ on success , -1 if no such device was found .
  - `count()`
    * @return The number of devices in the manager.

packetio.h

Library supporting sending / receiving Ethernet II frames.

- `int sendFrame (const void * buf, int len, int ethtype, const void *destmac, int id)`

  * Encapsulate some data into an Ethernet II frame and send it .

  * @param buf Pointer to the payload .

  * @param len Length of the payload .

  * @param ethtype EtherType field value of this frame .

  * @param destmac MAC address of the destination .

  * @param id ID of the device (returned by ”addDevice”) to send on .

  * @return 0 on success , -1 on error .

- `int setFrameReceiveCallback(frameReceiveCallback callback, int id)`

  * Register a callback function to be called each time an Ethernet II frame was received .

  *@param callback the callback function.

  * @return 0 on success , -1 on error.

- `int LinkHandInPacket(struct pcap_pkthdr* pkt_header, const u_char* framebuf, int index)`

  * After receive a packet captureed on specific device, try to handle it using the default function, and print the raw message if the function is not found.

  * @param pkt_header the header of the packet captured.

  * @param framebuf the buffer of the packet captured.

  * @param index the index of the packet captured.

  * @return 0 on success , -1 on error.

- `int receiveAllFrame(int id, int frame_count)`

  * try to receive specific number of Ethernet frames from device ID id.

  * @param id The Index of device to receive the package.

  * @param frame_count A number,-1 represents receiving until error occurs, 0-65535 represents the number of packet expected to receive.

  * @return the number of packages received,

iptable.h

Library for a data structure which gives the mapping between IP address and the information about IP address.

We implemented following structure to maintain it.

- `IPTableNode`
  - `class T value` The value saved in the trie node.
  - `int child[4]` The index of the 4 childs of the node.
- `IPTable`
  - `IPTableNode<class T>* mem` the pointer to the memory, which save the nodes in the trie
  - `int node_released` the number of nodes in the trie
  - `int node_count` the maximum number of nodes mem can save.
  - `find(const uint32_t &addr)` * find if the information about IP address
    * @param addr the IP address to be checked
    * @return 1 on information exist, 0 on not found
  - `class T& operator [](const uint32_t &addr)`
    * find if the information about IP address, and set the piece of memory if the information is not found
    * @param addr the IP address to be checked
    * @return the information

routing.h

Library for a data structure which gives the mapping between IP address and the information about IP address.

We implemented following structure to maintain it.

- `RoutingTableNode`
  - `bool rule` if the node contains a routing rule
  - `std::pair<macAddress, int> value` the routing rule in (next hop address, next hop device index) format
  - `int child[2]` the index of child node.
- `RoutingTable`
  - `IPTableNode* mem` the pointer to the memory, which save the nodes in the trie
  - `int node_released` the number of nodes in the trie

  - `int node_count` the maximum number of nodes mem can save.
  - `void setNextHopMac(uint32_t dst, struct in_addr mask, std::pair<macAddress,int> value)`
    * set the given routing rule
    * @param dst the destination
    * @param mask the mask of the destination
    * @param value the (next hop address, next hop device) pair
  - `int queryNextHopMac(uint32_t dst, std::pair<macAddress,int> *value)`
    * @brief find the (next hop address, next hop device) pair according to the rule set above
    * @param dst the destination v
    * @param value the pointer of (next hop address, next hop device) pair
    * @return 0 on at least one matching rules, -1 on no matching tules.

mytime.h

Library for microsecond timer.

- `gettime()`
  * get the current time in microsecond(us)
  * @return the current time in microsecond(us)

arp.h

Library for my ARP-Like Routing algorithm.

- `int ARPCallback(const void* __buffer, const void* __mac_addr, int len, int index)`

  The callback function for my ARP-Like routing algorithm. The function is called if the device receive a ARP-Like packet.
  * @param ___buffer the buffer of the ARP-Like Header
  * @param ___mac_addr the device which send the ARP-Header
  * @param len the length of the buffer
  * @param index the index of the device which receive the ARP-Header.
  * @return 0 on success, 1 on failure.

- `int getNextHopMac(struct in_addr dst_ipaddr, void* nextHopMac, int &index)`
  * find the Mac address for the next hop
  * @param dst_ipaddr the ip address of the device which packet will be send.
  * @param nextHopMac the piece of memory to save the Mac address of the nextHop
  * @param index the index of device to retransmit the packet.
  * @return 0 on success, -1 on failure.

callback.h

Library for callback functions.

- `egLinkCallback(const void* __buffer, const void* __mac_addr, int len, int index))`

  * Example LinkLayer Callback function.

  * @param ___buffer the message from the packet.

  * @param ___mac_addr the mac address of the source of the packet.

  * @param len the length of ___buffer

  * @param the index of device which receive the packet.

  * @return 0 on success, 1 on failure.

- `egLinkCallback(const void* __buffer, const void* __mac_addr, int len, int index, uint16_t proto))`

  * Link Layer Callback function used in 5-layer netstack model

  * @param ___buffer the message from the packet.

  * @param ___mac_addr the mac address of the source of the packet.

  * @param len the length of ___buffer

  * @param the index of device which receive the packet.

  * @return 0 on success, 1 on failure.

  * @param proto the protocol used in the packet

- `egIPCallback(const void* __buffer, struct IPHeader header, int len, int index))`

  * Example IP Layer Callback function.

  * @param ___buffer the message from the packet.

  * @param header the header of the IP Packet

  * @param len the length of ___buffer

  * @param the index of device which receive the packet.

  * @return 0 on success, 1 on failure.

ip.h

Library supporting sending / receiving IP packets encapsulated in an Ethernet II frame .

- `int sendIPPacket(const struct in_addr src ,const struct in_addr dest ,int proto , const void * buf , int len)`

  * Send an IP packet to specified host .

  * @param src Source IP address .

* @param dest Destination IP address .

* @param proto Value of ‘protocol ‘field in IP header .

* @param buf pointer to IP payload

* @param len Length of IP payload

* @return 0 on success , -1 on error .

- `void setIPPacketReceiveCallback(IPPacketReceiveCallback callback, int index)`

  * Register a callback function to be called each time an IP packet was received .

  * @param callback The callback function .

  * @return 0 on success , -1 on error .

- `int IPHandInPacket(const void* __buffer, int len)`

  * Handle the IPPacket, retransmit it if the destination of the packet is not the device, and call the callback function if it is the destination

  * @param ___buffer raw IPPacket including information and header

  * @param len the length of ___buffer .

  * @return 0 on success , -1 on error .

- `int setRoutingTable(const struct in_addr dest, const struct in_addr mask, const void* nextHopMac, const char* device)`

  * Manully add an item to routing table . Useful when talking with real Linux machines . * * @param dest The destination IP prefix .

  * @param mask The subnet mask of the destination IP prefix .

  * @param nextHopMAC MAC address of the next hop .

  * @param device Name of device to send packets on .

  * @return 0 on success , -1 on error

main.cpp

- `msg` test message to send.

I write 4 piece of testing function, which can helps to test information deliver(in the same time and different time), disconnect judgement, routing rules, and distance table. I write them in `main.cpp`.

5 Usage

The executable file is generated using `cmake`.

```
mkdir build && cd build
cmake ..
make
```

Then executable file is saved in `/build/src/lab1`, and you can use the following command to run it.

```
sudo ./src/lab1
```

6 CheckPoint

CP3

The following image shows one of the frame we try to retransmit on device `ns2`, while the network looks like `ns1 -- ns2 -- ns3 -- ns4`.

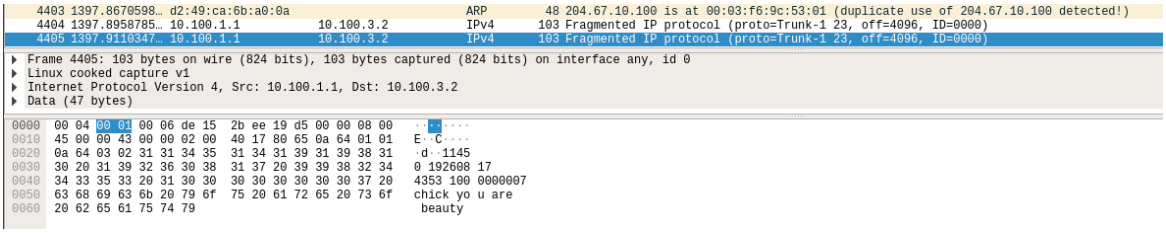

Figure 1: CP-3: The frame we send on device `ns2`

The IP Header locates in bit `0x0010` $\sim$ `0x0023`. and the meaning is shown in the following table.

- the higher 4 bits of `0x0010`: the version of IP protocol, 4 because we use `IPv4`.
- the lower 4 bits of `0x0010`: the length of IP Header, 5 because we use $5 \times 4 = 20$ bytes.
- `0x0011`: the type of service, we done need to set up this one, so it is filled 0.
- `0x0012` $\sim$ `0x0013`: the length of the information, `0x43` because exactly 81 bytes follows the header.
- `0x0014` $\sim$ `0x0015`: the identification of the packet, because the information may be fragmented. We done need to set up this one, so it is filled 0.
- the higher 3 bits of `0x0016`: the identification of the fragment, We done need to set up this one.
- the lower 13 bits of `0x0016` $\sim$ `0x0017`: the offset of the fragment, used while the information is fragmented. We done need to set up this one.
- `0x0018`: the time it will survive in the server, we done need to set up this one, so it is filled `0x17` like in `pcap.trace`).
- `0x0019`: the protocol used, we done need to set up this one, so it is filled `0x17`.

- `0x001a` ∼ `0x001b`: the checksum of packet, calculated after other terms are filled.
- `0x001c` ∼ `0x001f`: the IP address of source, `10.100.1.1` in the network.
- `0x0020` ∼ `0x0023`: the IP address of destination, `10.100.3.2` in the network.

CP4

Set UP the virtual network saved in `network1.txt`. and then launch 4 bashs, each runs the server using the command

```
sudo ./src/lab1 2 name
```

where name represents the name of the server locate in, like `ns1`.

First, launch the 4 servers at the same time, and server `ns1` will try to send a message to `ns4` after about 30 seconds. `ns4` receive the message and it print the raw message like follow.(`ns1` on the top left, `ns4` on the bottom right, the same arrangement is used in `CP4-2,CP4-3`).

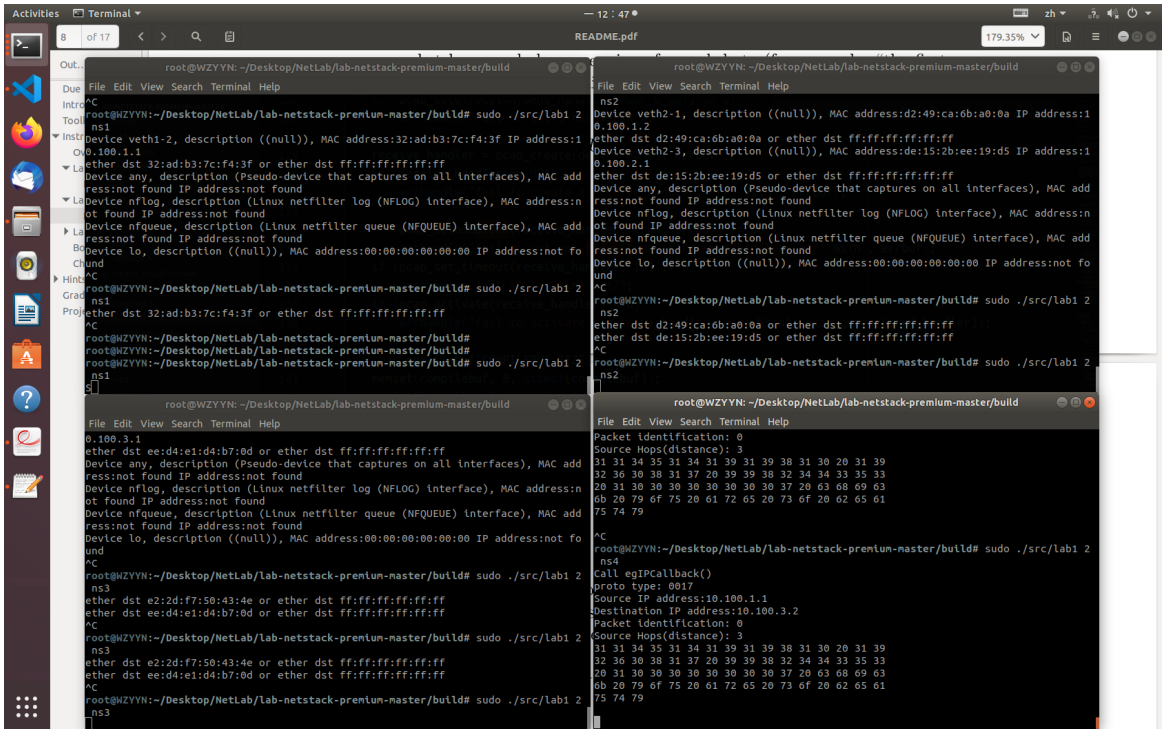

Figure 2: CP4-1: `ns4` receive the message from `ns1`

Then we close `ns2`, and launch `ns1` again. It will show that `ns4` is disconnected because it could not find the route to it. The image I cut is `CP4-2`.

Finally, we launch `ns2` again. This time `ns1` could found `ns4`, and the message is successfully delivered. The image I cut is `CP4-4`.

CP5

Set UP the virtual network saved in `network1.txt`. and then launch 6 bashs, each runs the server using the command

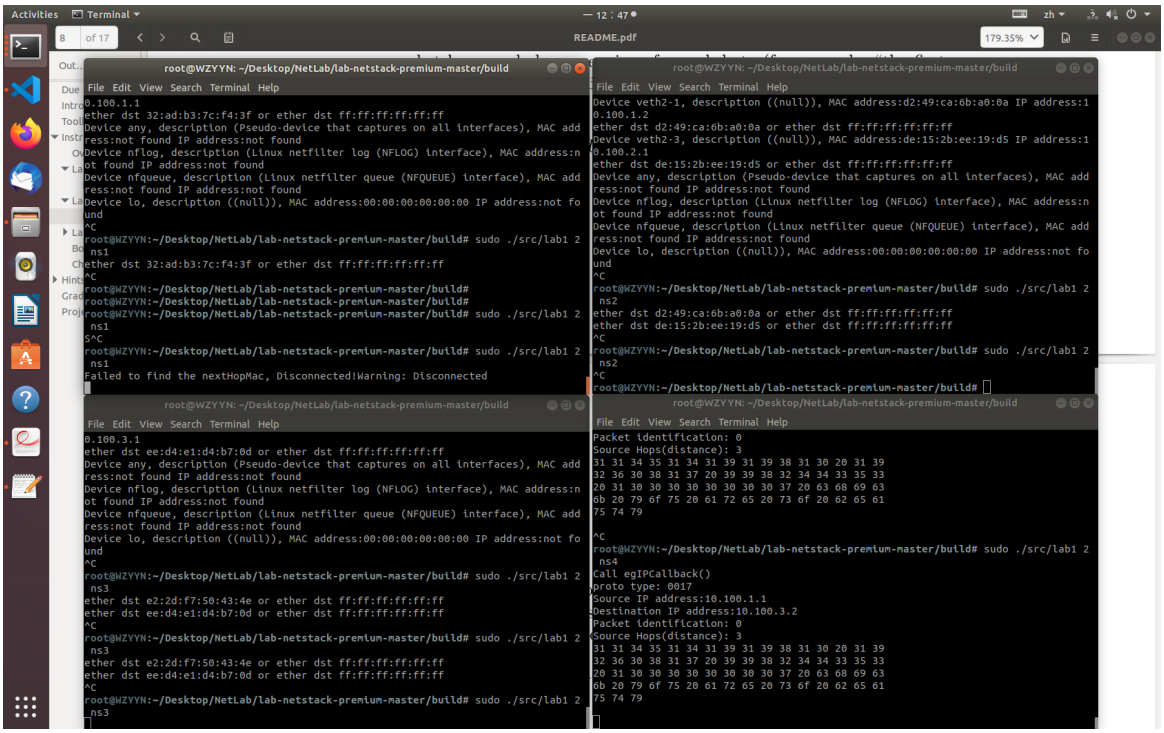

Figure 3: CP4-2: `ns1` shows that `ns4` is disconnected.

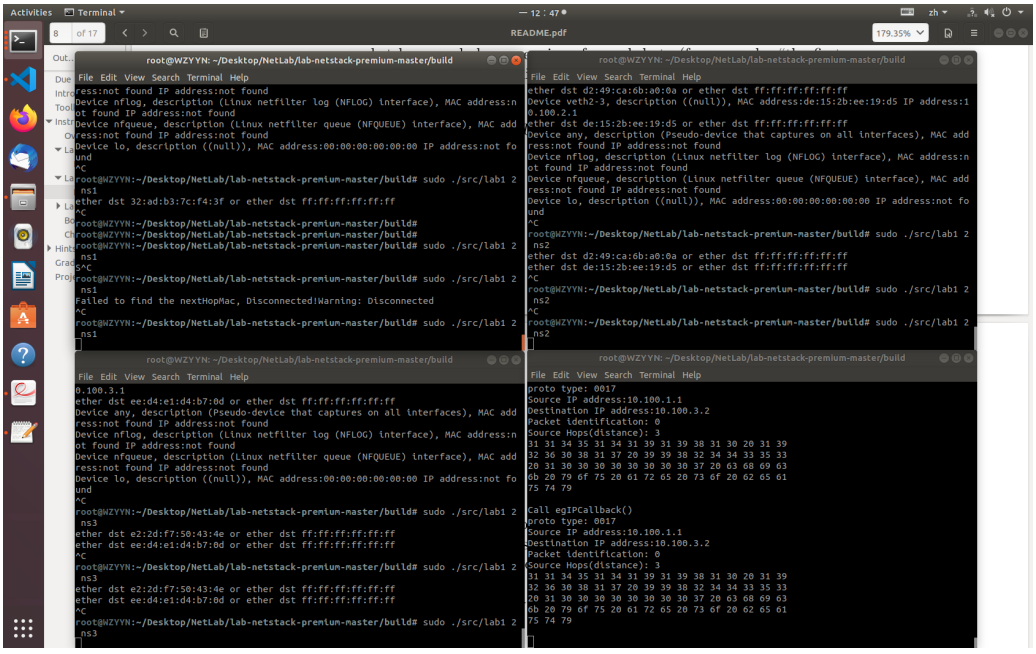

Figure 4: CP4-3:`ns4` receive the message from `ns1`

```
sudo ./src/lab1 4 name
```

First, launch the 6 servers at the same time, and each server will use my ARP-like protocol to find the distance to each other servers. If it could not found the server, the distance if set up to -1 represent non-exist. The output will like follow:(`ns1` on the top left, `ns6` on the bottom right, the same arrangement is used in `CP5-2`).

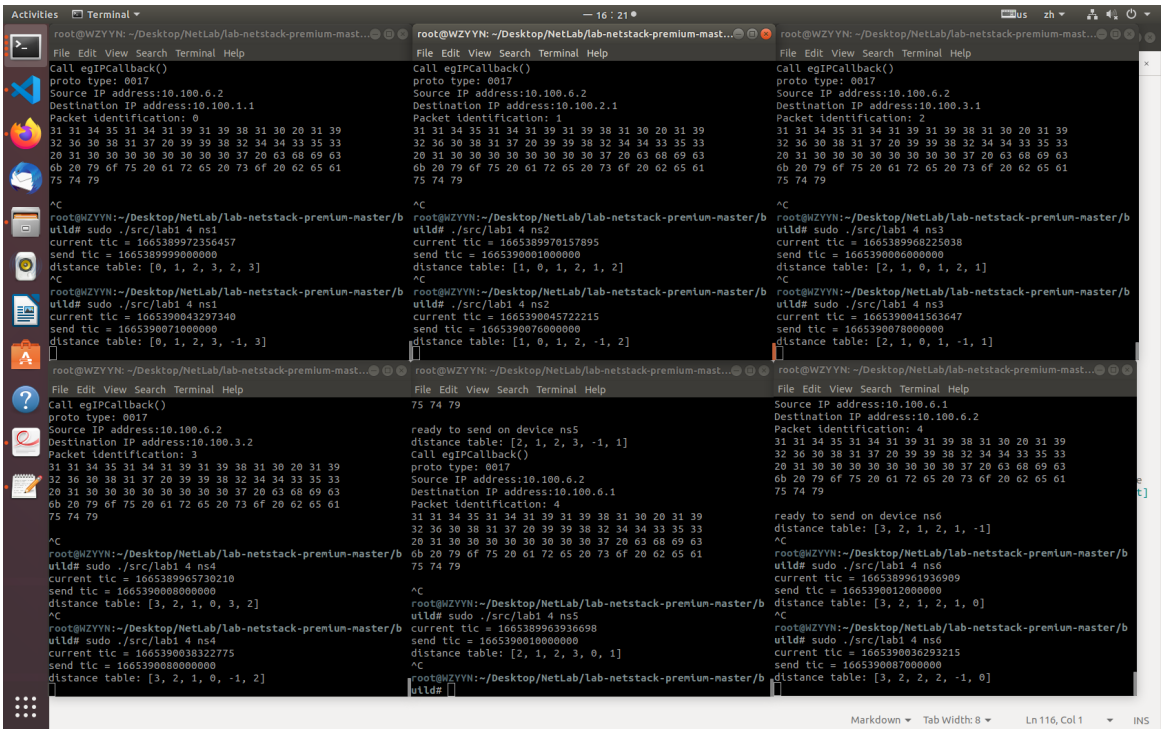

Figure 5: CP5-1: the distance for each pair of servers.

so the distance table look like follows:

/	1	2	3	4	5	6
1	0	1	2	3	2	3
2	1	0	1	2	1	2
3	2	1	0	1	2	1
4	3	2	1	0	3	2
5	2	1	2	3	0	1
6	3	2	1	2	1	0

After that, we only launch `ns1` $\sim$ `ns4, ns6` at the same time. then the output will look like follows:

so the distance table look like follows:

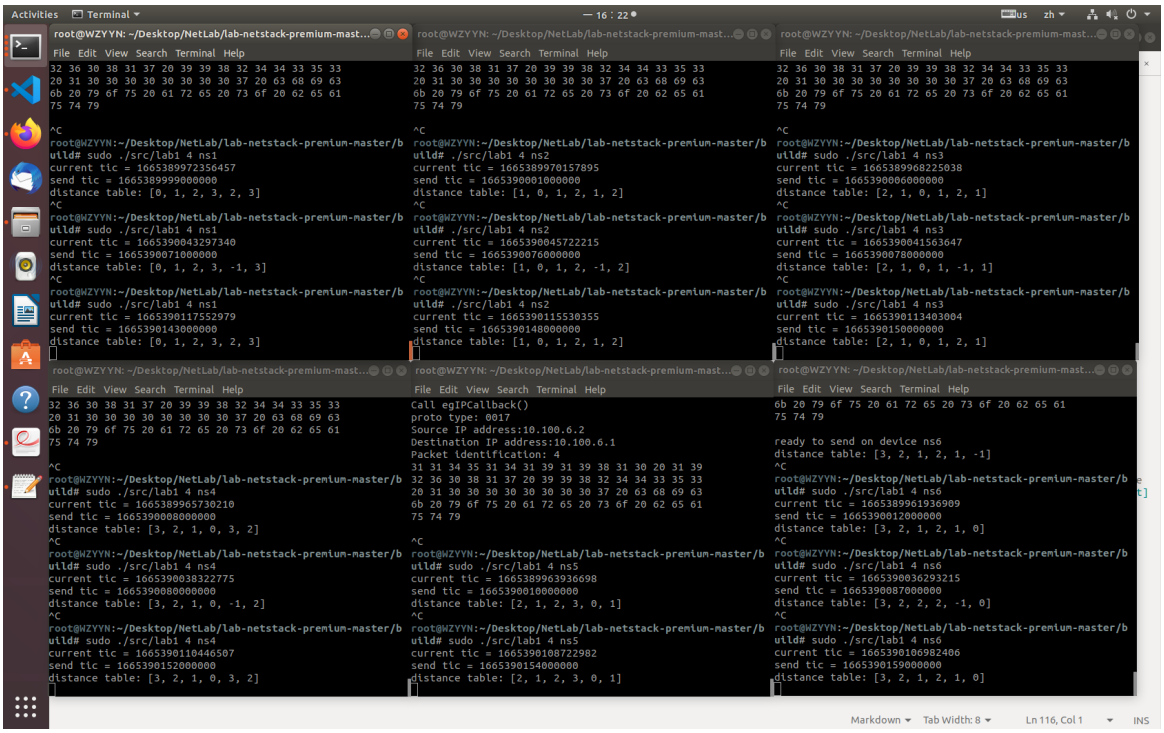

Figure 6: CP5-2: the distance for each pair of servers if `ns5` is not launched.

/	1	2	3	4	5	6
1	0	1	2	3	-1	3
2	1	0	1	2	-1	2
3	2	1	0	1	-1	1
4	3	2	1	0	-1	2
5	-1	-1	-1	-1	-1	-1
6	3	2	1	2	-1	0

CP6

Launch one bash, and runs the following command

```
sudo ./src/lab1 3 ns1
```

the server will try to set up 2 routing rules, where the one have mask length 16, the other have 24.

In the first query, only the first rule fit, so the router return the macaddress of the first rule; however, the second query fit both rule, so the router returns the second rule, because it have a longer fitting length.

the picture is shown below.

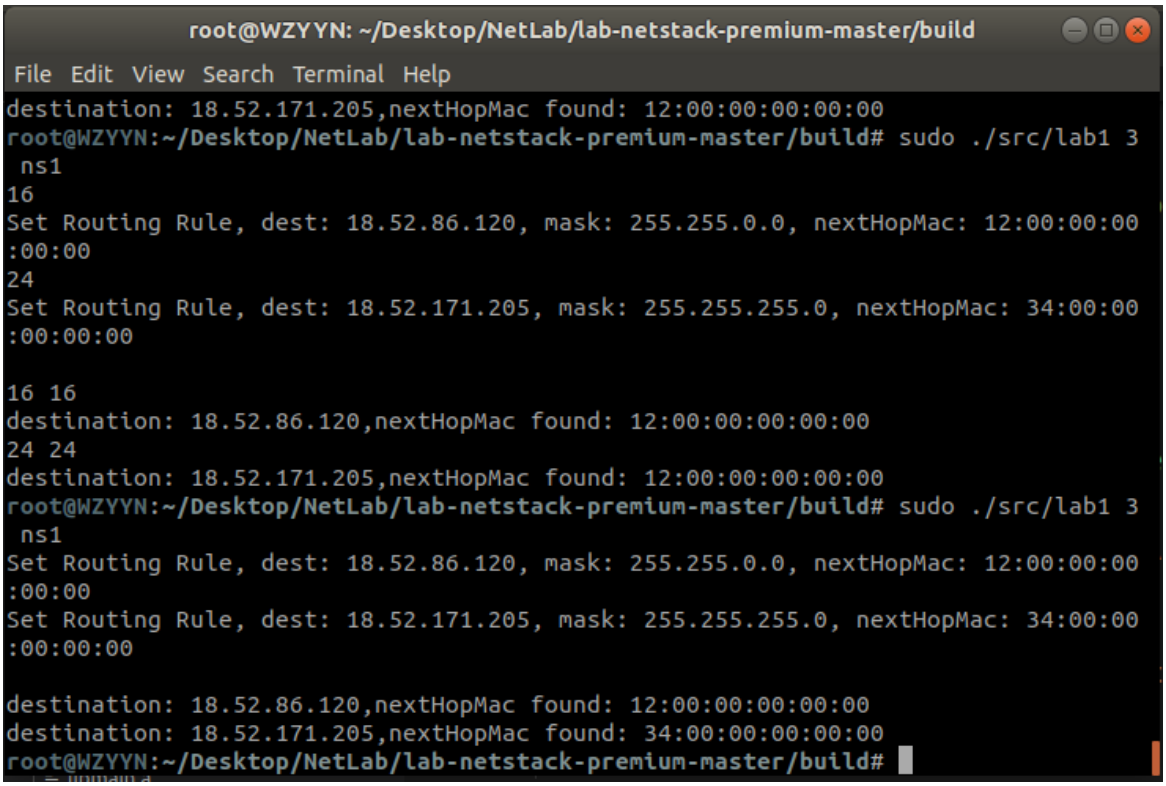

Figure 7: CP6: Example router reply.