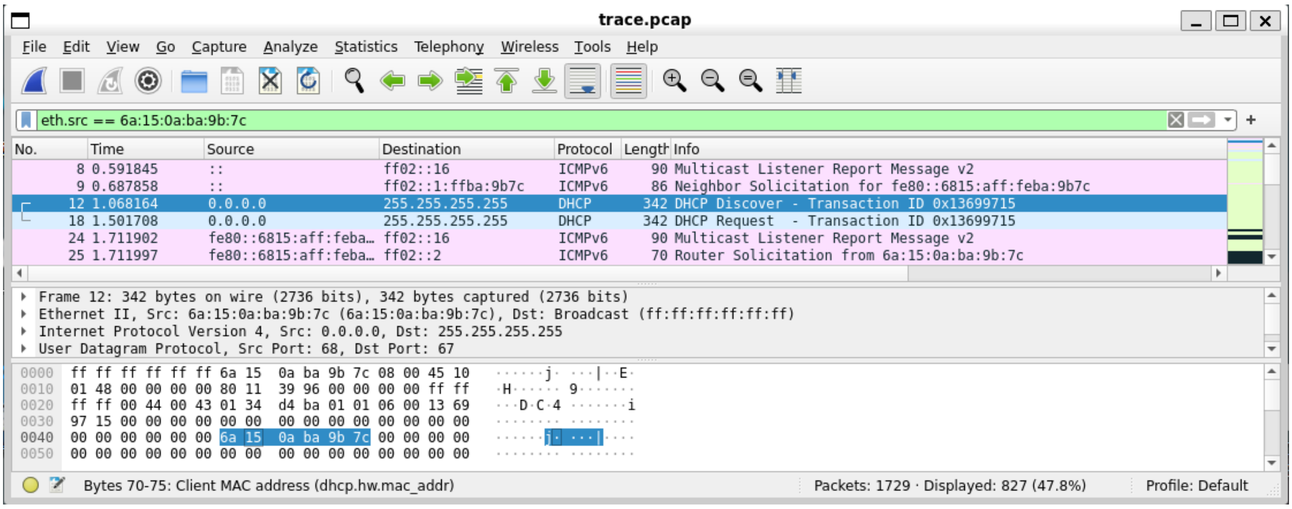
Computer Network (H) Lab1 report

Wenhan Ma 2100013124

WT1

Here is the result of opening `trace.pcap`.



Answer 1 There are **827** frames in the filtered result.

Answer 2 `ff:ff:ff:ff:ff:ff`, This means that the frame is a **broadcast frames**, indicating that the frame should be received and processed by all devices on the local network segment.

Answer 3 It is `0x15`.

PT1

I use C to work on this lab.

I have implemented basic device management methods. It can add devices using the `addDevice` function and find devices using the `findDevice` function. The devices are maintained through a linked list.

When we use the `addDevice` function to add a device, the function first checks if the device exists using the `pcap_findalldevs` function from the libpcap library, and if it is an Ethernet interfaces, and then get the MAC address. Next it is opened using `pcap_create`, followed by a series of configuration using libpcap functions. Finally we allocate a descriptor for it, and inserted it into the list.

When using `findDevice` to locate a device, we traverse this link list for the search.

For more detailed information, please refer to the source file `device.c` and `device.h`

PT2

I have implemented basic methods for sending and receiving frames.

For frame sending, I have implemented the `sendFrame` function. It first allocates some memory for the frame, then fill it with the frame header and data Finally, it sends the frame using the `pcap_sendpacket` function.

For receiving and handling frames, we have implemented `receiveFrame` and `handleFrame` functions to receive frames and process them using callback functions. The function call hierarchy is shown in the following figure.

User defined caller function	→	receiveFrame	→	pcap_dispatch	→	handleFrame	→	User defined callback function
(User) (or upper-layer)		(our)		(libpcap)		(our)		(User) (or upper-layer)

`receiveFrame` is called by the user (or the upper-layer protocol stack) and uses `pcap_dispatch` to receive frames. After frame is received, it calls `handleFrame`, which extracts the frame header and data, and call the user-defined callback functions to process the frame header and data. The user can call `setFrameReceiveCallback` to set the callback function. Our implementation acts as a bridge between the user and the libpcap library to handle Ethernet frames.

Please note that the meaning of the parameters in our callback functions is slightly different from what is provided in the lab document. Our callback functions are defined as:

```
typedef int (*frameReceiveCallback)(const void *data_load, uint32_t len,
    int device_id, struct EthHeader hdr);
```

Before calling the callback function, we first extract the frame into the frame header part and the data part in the `handleFrame` function, and store them in the fourth and first parameters, respectively. This is done for the convenience of the upper-layer protocol stack.

For more detailed information, please refer to the source file `frame.c` and `frame.h`

CP1

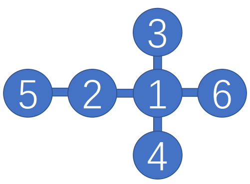
Execute all commands and programs in sudo mode, otherwise unexpected failures may occur.

We test if our implementation can add and find devices.

CP1: Automated testing

To execute this test, run `make` at the work folder. then run `sudo bash checkpoints/1/cp1.sh` . This script will do the following things:

1. Call `makeNet` to create a virtual network. This network has the following topology. This means the `host1` has four device, `veth1-2 veth1-3 veth1-4 veth1-6`



2. Execute program `detect` in the directory `build/test/lab1` on host1, the

program will try to find the following device: `veth1-2 veth1-3 veth1-4 veth1-5 veth1-6 eth0 123456789`. As a result, `veth1-2 veth1-3 veth1-4 veth1-6` will succeed while `veth1-5 eth0 123456789` will fail.

3. Call `removeNet` to remove the virtual network.

The output of the program is redirect to `checkpoints/1/stdout.log`. You can check this file after running the program, the content of the file is expected to be like this:

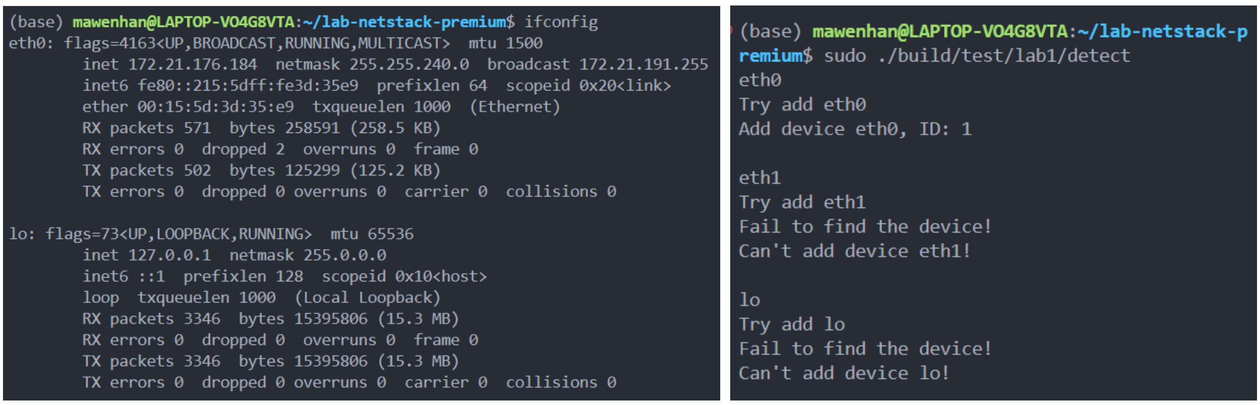
```
Try add veth1-2
Add device veth1-2, ID: 1

Try add veth1-3
Add device veth1-3, ID: 2

Try add veth1-4
Add device veth1-4, ID: 3

Try add veth1-5
Can't add device veth1-5!

Try add veth1-6
Add device veth1-6, ID: 4

Try add eth0
Can't add device eth0!

Try add 123456789
Can't add device 123456789!
...... (Too long, omitted)
```

CP1: Manual testing

Let's manually run the test program on the WSL. After run the command `ifconfig`, we know the system has one ethernet interface, `eth0` (`lo` is a Loopback Interface).

Run `sudo ./build/test/lab1/detect` in the shell, And input `eth0`, `eth1` and `lo`, we get the following result, show that the program can add the device `eth0` but can't add `lo` and `eth1`.

The following picture show our result.



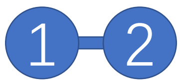
CP2

We test if our implementation can send and receive frames.

CP2: Automated testing

To execute this test, run `make` at the work folder. then run `sudo bash checkpoints/2/cp2.sh`. This script will do the following things:

1. Call `makeNet` to create a virtual network. This network has the following easy topology. We consider host1 as the sender and host2 as the receiver.



2. Execute program `sender` on host1 and `receiver` on host2 ,the two programs are in the directory `build/test/lab1`. The `sender` will input from `send_input.txt`, and output to `send_output.log`. The `receiver` will input from `receive_input.txt`, and output to `receive_output.log`.

The `sender` will prepend data with an Ethernet frame header, encapsulate it into a frame, and send it to the `receiver`. The `receiver` will extract the frame and output the dataload.

3. Wait for 5 seconds to ensure data sending is completed.
4. Call `removeNet` to remove the virtual network.
5. Exit, the receiver will also exit because of get the SIGHUP signal.

You can check the redirected output file at `checkpoints/1`, the content of `send_output.log` is expected to be like this:

```
Input the device name:
Device ID: 1
DesMacAddr: ff:ff:ff:ff:ff:ff
SrcMacAddr: be:36:15:9d:3e:6b

Input the data:
Succeed to send: Every day I imagine a future where I can be with you
Succeed to send: In my hand is a pen that will write a poem of me and you
Succeed to send: The ink flows down into a dark puddle
Succeed to send: Just move your hand write the way into his heart
Succeed to send: But in this world of infinite choices
Succeed to send: What will it take just to find that special day
```

the content of `receive_output.log` is expected to be like this. Other packets that are not sent by the sender may also be captured please ignore them.

```
Succeed to add device, ID: 1
Frame #00001
Data length: 53
Device ID: 1
Des Mac:ff:ff:ff:ff:ff:ff
Src Mac:3e:9c:6a:30:4c:9a
Type: 0x0800
```

```
String: Every day I imagine a future where I can be with you
Data:
45 76 65 72   79 20 64 61   79 20 49 20   69 6d 61 67
69 6e 65 20   61 20 66 75   74 75 72 65   20 77 68 65
72 65 20 49   20 63 61 6e   20 62 65 20   77 69 74 68
20 79 6f 75   00

Frame #00002
...... (Too long, omitted)
```

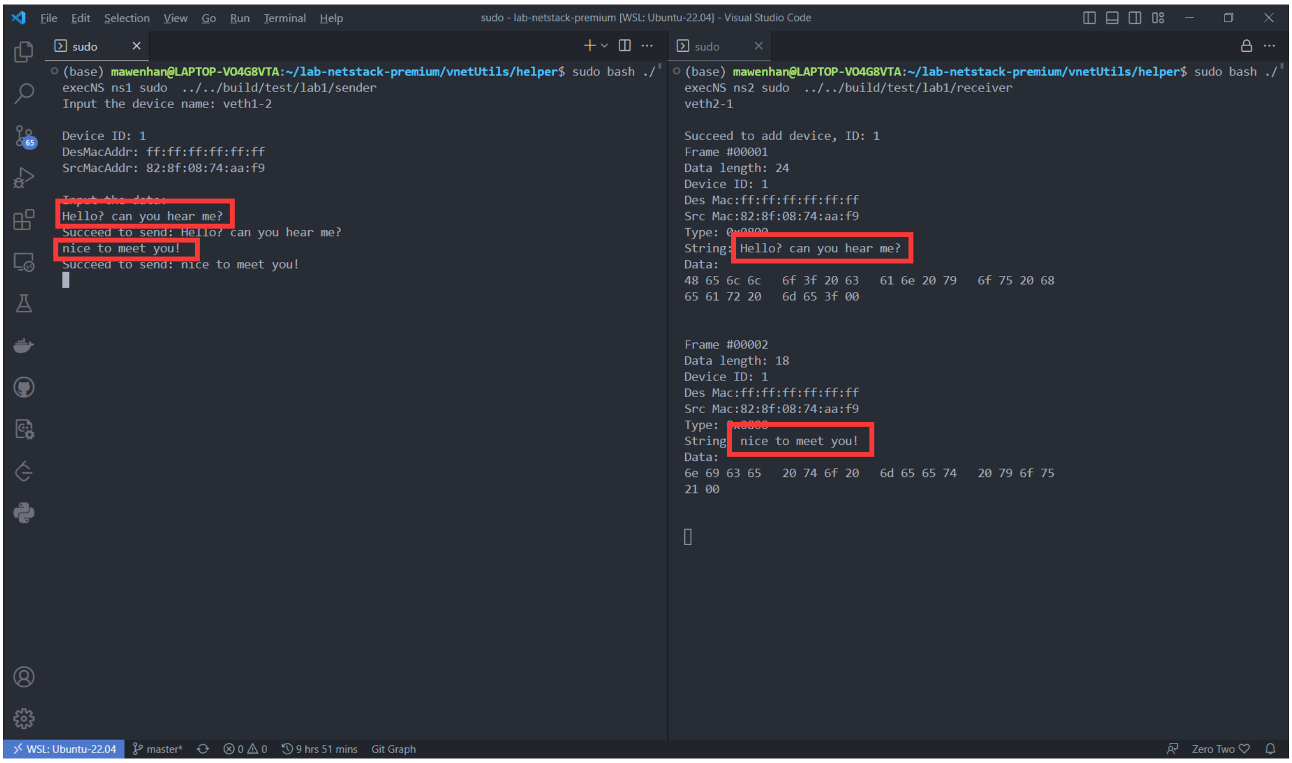
Please note that, according to our configuration, the printed `Data` does not include the frame header.

The result that our implementation can correctly send and receive the frame from the device.

CP2: Manual testing

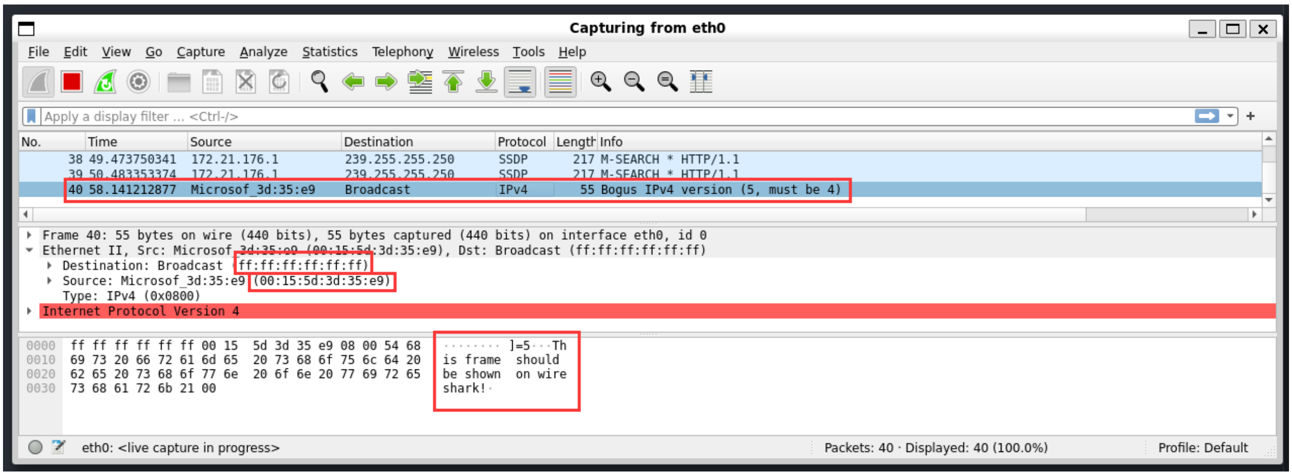
Test1

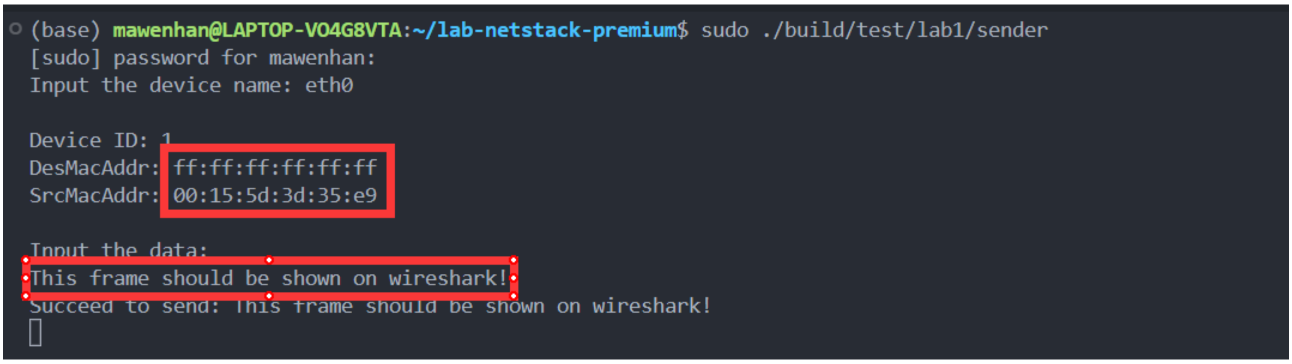
Let's play a manual test of this system. First, use `makeNet` to create a network connecting the two hosts mentioned earlier. Then, open two shells, one running `sender` on host1, and the other running `receiver` on host2. Manually input some message in the `sender`, and from the screenshot below, we can see that the information is received by the `receiver`.



Test2

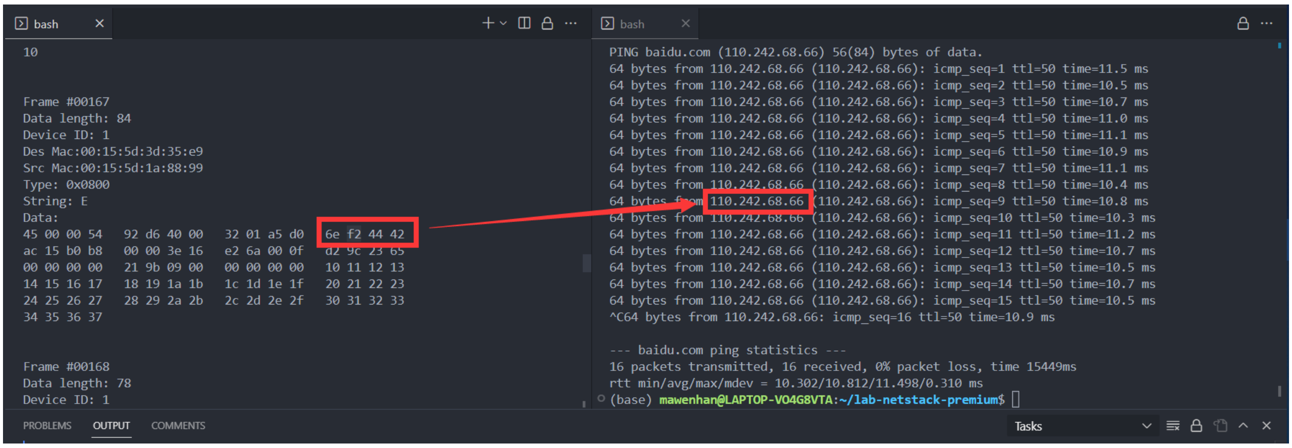
Let's test the `sender` separately in a real network environment. First, open wireshark, then use the `sender` to send a frame. We can find this frame in wireshark's capture list, and the information in it matches what we sent.





Test3

Finally, let's test the `receiver` separately. In a real network environment, we open the `receiver` and, at the same time, run `ping baidu.com` in another shell. The `receiver` immediately captures a large number of packets, and we can identify the IP address corresponding to the `ping` from the data segment of some of these frames.



These tests indicate that our implementation can correctly send and receive the frame from the device.

Code list

`link/device.c` `link/device.h`: Implement Programming Task 1, Support network interface management.

`link/frame.c` `link/frame.h`: Implement Programming Task 2, Support sending and receiving Ethernet frames.

`link/ethheader.c` `link/ethheader.h`: Define the struct of the header of an Ethernet frame and MAC address and some related method.

`link/link.c` `link/link.h`: Include a function that can initialize our link layer.

`test/lab1/detect.c` `checkpoints/1/cp1.sh`: Test Checkpoint 1

`test/lab1/sender.c` `test/lab1/receiver.c` `checkpoints/2/cp2.sh`: Test Checkpoint 2