PKU-CompNet (H) Fall’23 Lab Assignment (Premium): Protocol Stack

寿晨宸 2100012945

1 Lab 1: Link-layer

1.1 Writing Task 1(WT1).

> Open trace.pcap with Wireshark. First set filter to eth.src == 6a:15:0a:ba:9b:7c to only reserve Ethernet frames with source address 6a:15:0a:ba:9b:7c. Find the third frame in the filtered results and answer the following questions.
>
> 1. How many frames are there in the filtered results? (Hint: see the status bar)
> 2. What is the destination address of this Ethernet frame and what makes this address special?
> 3. What is the 71th byte (count from 0) of this frame?

The third frame in the filtered results is:

No.	Time	Source	Destination	Protocol	Length	Info
12	1.068164	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0x13699715

1. There are 827 frames in the filtered results.
2. ff:ff:ff:ff:ff:ff, the broadcast address.

   > Frames are addressed to reach every computer on a given LAN segment if they are addressed to MAC address FF:FF:FF:FF:FF:FF. Ethernet frames that contain IP broadcast packages are usually sent to this address. ***by Wikipedia***

3. 0x15.

1.2 Programming Task 1(PT1).

I have implemented network device management functionality in the files `src/lib/device.cpp` and `src/include/device.h`. In these files, I have abstracted two classes named `Device` and `DeviceManager`. The `Device` class contains various information about network devices, such as name, MAC address, device number, etc., and functions for operating on the device. The `DeviceManager` class contains a list of `Device` objects and implements various operations for managing these devices.

For this programming task, which is the implementation of network device management, specifically, I have implemented the following functionalities (all are member functions of `DeviceManager`):

1. `dev_id addDevice(const char* dev_name)`: Adds a device to the list of devices based on the given device name. This function completes the initial setup for the device, such as recording the device's MAC address, setting up a thread dedicated to handling receive callbacks, and so on. It returns a device identifier, which is the device's index in the device list (linearly increasing).

2. `Device* findDevice(const char* dev_name)`: Retrieves the device object based on the given device name (equivalent to getting the device by its ID; it does not affect functionality).
3. `Device* getDevice(dev_id id)`: Retrieves the device object based on the given device identifier.
4. `void printAllValidDevice() const`: A utility function for printing information about all currently valid devices owned by the network namespace.
5. `void printAllAddedDevice() const`: A utility function for printing information about all devices that have been added.

1.3 Programming Task 2(PT2).

I have implemented the functionality for sending and receiving Ethernet II frames in the files `src/lib/device.cpp`, `src/include/device.h`, `src/include/type.h`, `src/include/callback.h`, and `src/lib/callback.cpp`.

In `type.h`, I have provided some commonly used type definitions. In `callback.h` and `callback.cpp`, I have implemented a reusable receive callback function as an auxiliary function (in actual test programs, I have redesigned new functions based on specific scenarios).

For this task, I have primarily implemented the following functions, all of which are related to the `Device` class (member or friend functions):

1. `Device(char* dev_name, u_char* mac, dev_id id, DeviceManager* manager)`: Constructor function. Sets various members and starts a new thread for listening to messages.
2. `int sendFrame(const void* buf, int len, int ethtype, const void* destmac)`: Constructs an Ethernet II frame and sends the message.
3. `int setFrameReceiveCallback(frameReceiveCallback callback)`: Sets the callback function.
4. `friend void deviceRecvFrame(Device* device)`: Function run by the thread listening to messages, running `pcap_loop` inside it.
5. `friend void deviceRecvPcapHandler(u_char* args, const pcap_pkthdr* head, const u_char* packet)`: Program run by `pcap_loop`. When a message is received, it runs the callback function if it is not empty.
6. `int stopRecv()`: An auxiliary function to stop listening and set the callback function to NULL.
7. `const char* getDeviceName() const`: An auxiliary function to get the device name.
8. `const int getDeviceID() const`: An auxiliary function to get the device ID.
9. `const u_char* getDeviceMac() const`: An auxiliary function to get the device's MAC address.
10. `~Device()`: Destructor function that joins the listening thread and releases allocated resources.

In all of the above operations, I have used mutex to ensure thread safety.

1.4 Checkpoint 1(CP1).

For Checkpoint 1 (CP1), the implementation code is located in `checkpoints/CP1/{cp1.sh and cp1.txt}` and `src/tests/ethernet/test_device_manager.cpp`. The main idea is to perform a series of device management operations in a network built based on `vnetUtils/examples` within ns3, and record their outputs.

1. `test_device_manager <input_file>`: This is a testing tool provided for CP1, which takes one parameter representing an input file. If the parameter is empty, it reads from standard input. For example, when given an input file, this program reads the file and treats each line as a command for device management operations. It calls the functions implemented in PT1 and outputs the operation results. The operation commands include:
   - `addDevice <dev_name>`
   - `findDevice <dev_name>`
   - `findAllAddedDevice`
   - `findAllValidDevice`
   - `exit`
2. `cp1.sh`: In simple terms, this script uses `vnetUtils/examples` to build a network, calls `test_device_manager` within ns3, and uses `cp1.sh` as the input file (which contains a series of device management operations). It outputs the results to `cp1.log`. The `typescript` file in this directory records the process of running this script.

1.5 Checkpoint 2(CP2).

For Checkpoint 2 (CP2), the implementation code is located in `checkpoints/CP2/cp2.sh`, `src/tests/ethernet/{set_sender and set_receiver}.cpp`. The main idea is to establish a network based on `vnetUtils/examples` and enable multiple rounds of communication between connected devices, while recording their output.

1. `set_sender <dev> <dst_mac> <send_num>`: This sets up the sender. Each parameter represents the device name for sending, the destination MAC address, and the number of messages to send. After running this program, the device `dev` will send `send_num` rounds of frames to `dst_mac`, where each round of sending has a different payload. The payload looks like "Hello! This is message 10 from device veth0-3 with mac 7a:85:90:d4:1b:cc." for easy calibration, identification, and verification. It also outputs relevant information during the sending process, such as:

   ```
   [INFO] Device veth3-0 send frame 1:
   src_mac: 66:a4:fd:02:9b:30
   dst_mac: 7a:85:90:d4:1b:cc
   pay_load: Hello! This is message 1 from device veth3-0 with mac
   66:a4:fd:02:9b:30.
   pay_load_len: 72
   ethtype: 0x8888
   ```

2. `set_receiver <dev> <recv_num>`: This sets up the receiver. Each parameter represents the device name for receiving and the number of messages to receive. After running this program,

whenever the `dev` device receives a message, it prints related information. This is used for cross-checking against the sent messages.

```
[INFO] Device veth0-3 receive frame 1.
src_mac: 66:a4:fd:02:9b:30
dst_mac: 7a:85:90:d4:1b:cc
pay_load: Hello! This is message 1 from device veth3-0 with mac
66:a4:fd:02:9b:30.
pay_load_len: 72
ethtype: 0x8888
```

3. `cp2.sh`: In simple terms, this script uses `vnetUtils/examples` to build a network and tests communication between each veth pair within the network. For example, it first sets `veth0-3` and `veth3-0` as receivers, and then it allows `veth0-3` and `veth3-0` to send 100 messages to each other, recording their outputs. The output logs for both receivers and senders are saved in the `checkpoints/CP2/log/` folder.
   - For the receiver, the log file, for example, `veth0-3_receive.log`, records the following information:

     ```
     [INFO] Device veth0-3 ready to receive 100 frames.
     [INFO] Device veth0-3 receive frame 1.
     src_mac: 66:a4:fd:02:9b:30
     dst_mac: 7a:85:90:d4:1b:cc
     pay_load: Hello! This is message 1 from device veth3-0 with mac
     66:a4:fd:02:9b:30.
     pay_load_len: 72
     ethtype: 0x8888
     [INFO] Device veth0-3 receive frame 2.
     src_mac: 66:a4:fd:02:9b:30
     dst_mac: 7a:85:90:d4:1b:cc
     pay_load: Hello! This is message 2 from device veth3-0 with mac
     66:a4:fd:02:9b:30.
     pay_load_len: 72
     ethtype: 0x8888
     [INFO] Device veth0-3 has received 2 frames, expect 2 frames.
     [INFO] Device veth0-3 stop receiving frames.
     ```

   - For the sender, the log file, for example, `veth3-0_send.log`, records similar information:

     ```
     [INFO] Device veth3-0 send frame 1:
     src_mac: 66:a4:fd:02:9b:30
     dst_mac: 7a:85:90:d4:1b:cc
     pay_load: Hello! This is message 1 from device veth3-0 with mac
     66:a4:fd:02:9b:30.
     pay_load_len: 72
     ethtype: 0x8888
     ```

```
[INFO] Device veth3-0 send frame 2:
src_mac: 66:a4:fd:02:9b:30
dst_mac: 7a:85:90:d4:1b:cc
pay_load: Hello! This is message 2 from device veth3-0 with mac
66:a4:fd:02:9b:30.
pay_load_len: 72
ethtype: 0x8888
```

Generally, you can compare the output logs of receivers and senders to validate the correctness of the code.

- `checkpoints/CP2/typescript` records the log information during the execution of `cp2.sh`.

1.5 Running Helper

To get everything running, you don't need to know all the details! You can simply run the following command, and it will prompt you the message:

```
> bash ./run.sh
Enter choice (CP1/CP2/HANDIN/MAKE/CLEAN):
```

Typically, you should start by entering "MAKE". This will build the executable files (located in the `build` directory).

After that, you can choose either "CP1" or "CP2," and it will execute the relevant programs for Checkpoint 1 or Checkpoint 2, respectively, and record the running logs using `script`.

If you enter "HANDIN", you will get the compressed package ready for submission.

If you enter "CLEAN", it will clear all logs, executable files, and compressed packages.

This script simplifies the process of compiling, running, and managing your project.