PKU-CompNet (H) Fall'22
Lab Assignment (Premium): Protocol Stack
Report

Jiaheng Li (2000013054)

October 12, 2022

1 Lab 1: Link-layer

1.1 Writing Task 1 (WT1)

The third frame in the 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 result. (`Dislayed: 827`)
2. The destination address of the Ethernet frame is `ff:ff:ff:ff:ff:ff`, which is the broadcast address.
3. The 71st byte is `0x15`.

1.2 Programming Task 1 (PT1)

(For the convenience of later tasks, the code is reconstructed when implementing the network layer. Therefore, the description here may be different from the version submitted before in Lab 1.)

We design the interface a bit differently from the reference in the instruction, and so do the tasks below.

For this task, we design class `NetBase` and `Ethernet` (in `src/include/{NetBase, Ethernet}.h` and implementation in `src/lib/{NetBase, Ethernet}.c`).

The class `NetBase` is for managing `pcap` "devices" and flows, while class `Ethernet` built on the former is for managing Ethernet devices (with Ethernet address...).

Methods for the required functions are:

```
Ethernet::Device *Ethernet::addDeviceByName(const char *name);
Ethernet::Device *Ethernet::findDeviceByName(const char *name);
```

while more documents are provided in the Doxygen-style comment in the header files.

The handler (descriptor) for devices is `Ethernet::Device *`.

The method finds the device by its name through `pcap_findalldevs()` and gets its Ethernet address. Then it opens the `pcap` flow, and creates the `Device` object for management.

1.3 Programming Task 2 (PT2)

The class `Ethernet` also supports sending and receiving Ethernet frames.

Methods for the required functions are:

```
int Ethernet::Device::sendFrame(const void *data, int dataLen, const Addr &dst,
                                int etherType);

class RecvCallback {
public:
```

```
  int etherType; // The matching etherType, -1 for any.
  virtual int handle(const void *data, int dataLen, const Info &info) = 0;
};
void Ethernet::addRecvCallback(RecvCallback *callback);
```

The `sendFrame()` method simply adds the Ethernet header and send it through `pcap_sendpacket()`.

The `addRecvCallback()` method simply adds the callback to a list.

To receive packets, we should call `NetBase::setup()`, `Ethernet::setup()`, and finally `NetBase::loop()`. `NetBase::loop()` will loop forever (if not interrupted), checking for each `pcap` devices for new receiving frames using the "nonblocking mode" and `pcap_dispatch()`, and then invokes the callback from `Ethernet` (or other registered) for each frame. It also provides an interface for callback in each loop iteration, allowing users to handle their work.

All the routines in the network stack library run on a single thread (and are required to do so). For application, they may create a separate thread for the network stack library, and work on another user thread, using something like a "task dispatcher" to synchronize with the loop callbacks.

We implemented a CLI program (in `src/tools/cli/`, and will be built to `build/tools/cli`), allowing users to interactively test the library. It uses a task dispatcher (in `src/include/LoopDispatcher.h`) to send tasks about the network stack to the network thread, and uses mutexes to wait for completion in the user thread.

The CLI program uses the GNU Readline Library (`libreadline-dev`) (only) for easier interaction.

1.4 Checkpoint 1 (CP1)

The typescript record is at `checkpoints/CP1/{typescript, timing.log}`.

In the script, several calls to `add-device` and `find-device` in a row shows that they can correctly add (only) the valid Ethernet devices on the host, get their Ethernet addresses, and correctly find the added devices.

1.5 Checkpoint 2 (CP2)

The typescript record is at `checkpoints/CP2/{typescript, timing.log}`.

We implemented the `eth-test` command in the CLI, which exchanges frames through all configured devices. It sends the first control message using the broadcast address and gets the real destination address through the response. Then the hosts exchange data simultaneously (using threads and the dispatcher), and finally print the sent and received checksum.

In the script, the `eth_test` tool above is simultaneously run on each of the 5 hosts in the example network provided in `vnetUtils`, using all of the 8 devices and 4 links among them. After the run, we can see that the sending and receiving checksum of each link matches, which shows the (basic) correctness of our sending and receiving implementation.

2 Lab 2: Network-layer

2.1 Writing Task 2 (WT2)

1. The "Target MAC address" field (bytes 32-37, indexed from 0) in the ARP Reply is the same as the Sender MAC address in the ARP Request.

2. There are 6 packets meeting the requirement. We can see this by filtering `ip.flags.df == 0` and checking the "Displayed" status.

3. The header length of IPv4 is 20 bytes, while for IPv6 it is 40 bytes.

2.2 Programming Task 3 (PT3)

We designed the class `IP` for handling basic IPv4 operations.

Methods for the required functions are:

```
int IP::sendPacket(const void *data, int dataLen, const Addr &src,
                   const Addr &dst, int protocol, SendOptions options = {});

class RecvCallback {
public:
  bool promiscuous; // If matches destination IP of other hosts.
  int protocol;     // The matching IP `protocol` field, -1 for any.
  virtual int handle(const void *data, int dataLen, const Info &info) = 0;
};
void IP::addRecvCallback(RecvCallback *callback);
```

It just works like the sending/receiving methods in the Ethernet layer, which simply adds the header and sends it to the layer below, or adds the callback to a list.

The class `IP` has registered a callback in `Ethernet`, to receive Ethernet frames of type IP. It then analyzes the header and distributes the received packets to its own callbacks.

For forwarding, we also implemented a class `IPForward` as a user to the `IP`. It registered a callback in `IP` and then use `IP` sending methods to forward the received packets out.

For routing, we first have the static routing in class `LpmRouting` (which derivates from the abstract `IP::Routing`), which implements the Longest Prefix Match in a static table manually added by the user. It then supports the `IP` class to send packets out. (ARP protocol also participates in this process, see the writing task below).

It has the interface:

```
int LpmRouting::match(const Addr &addr, HopInfo &res,
                      std::function<void()> waitingCallback) override;
int LpmRouting::setEntry(const Entry &entry);
```

The `waitingCallback` is for handling the waiting for ARP replies, as will be described in the tasks below.

We also have a dynamic routing algorithm, the (non-standard) `RIP` with similar interfaces, and cooperates with other "routers" in the network, which will also be described in the tasks below.

2.3 Writing Task 3 (WT3)

To send an IP packet, we first need the routing algorithm (or routing table) to find the IP (gateway) address of the next hop, or find out that they are directly connected.

Then, we get the IP address of the exactly next hop. We can check the ARP table, or send an ARP Request (on misses) to get the MAC address of the next hop.

However, sending the ARP Request brings the problem of the non-immediate sending of our packets. It is impossible to wait for the ARP Response in our network thread. Therefore, the sending method will return an error. For convenience, we also implement the automatic resend (with callback) option in `IP::SendOptions` and maybe in the upper layers, by registering a resending callback in the ARP Response receiving procedure. In the user thread, we can also use a mutex to wait for the completion of resending. In this way, this problem will not bring too much complexity to coding.

We implement the ARP sending method and responder in the class `ARP`, which may be called by the routing classes to find exactly the MAC address of the next hop.

```
int ARP::match(NetworkLayer::Addr netAddr, LinkLayer::Addr &linkAddr,
               std::function<void()> waitingCallback)
int ARP::sendRequest(NetworkLayer::Addr target);
int ARP::LinkLayerHandler::handle(const void *packet, int packetCapLen,
                                  const Info &info);
```

2.4 Writing Task 4 (WT4)

We use a distance vector algorithm similar to the Routing Information Protocol (RIP). It is implemented in the class `RIP`.

For the routing in the same subnet, we assume (in default) that they can reach each other directedly (so we only need the ARP protocol to get its MAC address), so their gateway is set to `0.0.0.0`, and metrics are set to 0.

Then, for each subnet, we maintain an element in the distance vector about the minimum hops to reach that subnet. We may also need the next hop (gateway) to reach that subnet, as well as the expiring time for updating.

Every 30 seconds (or other custom value), each router will broadcast its distance vector to its neighbors. And when a router receives an RIP "Response", it could update its own distance vector (and updates the next hop or other information at the same time):

$$d_i \leftarrow \min\{d_i, recv_i + 1\},$$

while we use 16 as infinity, every distance larger than 16 will be ignored.

In this way, with sufficient broadcasts, we can find the path of minimal number of hops to each subnet, and then the router in the subnet will directly forward it to the host.

To handle the changes or failures, every entry will be expired in 180 seconds (by setting its metric to 16), and then after 120 seconds, it will be deleted from the routing table.

Also, when a host adds to a network, it will immediately send an RIP Request, to get the information from its neighbors, to set up quickly.

To optimize the handling of changes, we also use triggered update, which means to broadcast the RIP response as soon as any metric changes in our vector. Also, if we receive the new distance from our best choice before, we will take it even if it is worse than our metric before. These methods can significantly improve performance in changes or failures, especially for the problem of "counting to infinity" slowly.

The RIP protocol runs on the UDP port 520 (with UDP broadcasts), therefore we also implement the class `UDP` over our `IP`.

2.5 Testing

We have tested the implementation in several ways. For example, in the example virtual network, our network stack could correctly forwards ICMP messages from the Linux `ping` on hosts connected through several "routers".

We also implemented a simple tool in the CLI like `nc -u`, just called `nc-u` in our CLI, to send messages through UDP. They cooperates correctly in the network.

To work with the Linux `nc` program, we need to disable the TCP/UDP checksum overload in the system using:

```
ethtool -K <device> rx off tx off sg off tso off
```

After that, our program can correctly cooperate with the Linux `nc`. It can also correctly forward the traffic of `iperf`.

For testing, we also implemented (part of) the ICMP protocol (sender and automatic responder), and our `ping` and `traceroute` command in the CLI. They cooperate correctly with each other, or with the Linux system responders/programs (if we do not disable the kernel network stack).

2.6 Checkpoint 3 (CP3)

The typescript record is at `checkpoints/CP3/{typescript, timing.log}`.

The hexdump of the packet is:

```
08:36:33.694000 IP localhost > localhost: ICMP echo request, id 45298, seq 1, length 32
0x0000:  9af1 db8d 59ba 4247 83b2 2ecc 0800 4500  ....Y.BG......E.
0x0010:  0034 0000 4000 4001 21ff 0a64 0101 0a64  .4..@.@.!..d...d
0x0020:  0302 0800 3b3f b0f2 0001 4c61 622d 4e65  ....;?....Lab-Ne
0x0030:  7453 7461 636b 2050 696e 6720 5465 7374  tStack.Ping.Test
0x0040:  0a00                                     ..
```

The first 14 bytes are the Ethernet header:

- `0x9af1db8d59ba`: Destination Address (6)
- `0x424783b22ecc` Source Address (6);
- `0x8000`: EtherType (2), IPv4.

The next 20 bytes are the IPv4 header, each:

- `0x4`: IPv4 version, `0x5`: IHL;
- `0x00`: Type of Service.
- `0x0034`: Total Length.
- `0x0000`: Identification for fragments.
- `0b010`: Flags (`1` for Don't fragment). The other part of `0x4000` for fragment offset.
- `0x40`: Time to Live.
- `0x01`: Protocol (ICMP).
- `0x21ff`: Header checksum.
- `0x0a640101`: Source Address: `10.100.1.1`;
- `0x0a640302`: Destination Address: `10.100.3.2`;

The other 32 bytes are the ICMP Echo (ping) Request message with data.

- `0x8`: Op code, ICMP Echo (ping) Request.
- `0x0`: Code, 0 for Echo Request.
- `0x3b3f`: Checksum
- `0xb0f2`: Identifier
- `0x0001`: Sequence number
- Others are the data: `"Lab-NetStack Ping Test"`...

2.7 Checkpoint 4 (CP4)

The typescript record is at `checkpoints/CP4/{typescript, timing.log}`.

We use the automatic configuring command in our CLI, which configures the IP, netmask as the `vnetUtils` set to the Linux system, and turn on the RIP dynamic routing as we pass the `-r` option.

We can see that, when we run the routing on all the hosts, ns1 can successfully ping to ns4.

After we kill the process in ns2, n1 cannot reach other hosts anymore. We can see that the routing table finally (after expiring and cleaning time) contains only its own subnet, as ns2 is on the only way from ns1 to other hosts.

After we run the routing in ns2 again, ns1 can reach ns4 again.

2.8 Checkpoint 5 (CP5)

The typescript record is at `checkpoints/CP5/{typescript, timing.log}`.

The output file is at `checkpoints/CP5/outs/ns*.out`.

We use the `traceroute` command in our CLI to measure the distance, we can see that it's exactly one TTL decrease for a hop.

The distances before disconnecting ns5 are in the left table (the number on row x, column y indicates the distance from x to y), while the distances after disconnecting are in the right one.

	ns1	ns2	ns3	ns4	ns5	ns6
ns1	-	1	2	3	2	3
ns2	1	-	1	2	1	2
ns3	2	1	-	1	2	1
ns4	3	2	1	-	3	2
ns5	2	1	2	3	-	1
ns6	3	2	1	2	1	-

	ns1	ns2	ns3	ns4	ns6
ns1	-	1	2	3	3
ns2	1	-	1	2	2
ns3	2	1	-	1	1
ns4	3	2	1	-	2
ns6	3	2	1	2	-

We can see that the minimum distances do not change even if we disconnect ns5, and our routing implementation can find still the shorted path after disconnection.

2.9 Checkpoint 6 (CP6)

The typescript record is at `checkpoints/CP6/{typescript, timing.log}`.

If we use the topology as CP5 (configuration file at `checkpoints/CP6/vnet.txt` showing the IPs), we could manually config the routing table of ns5 as follows:

```
route-add 10.100.3.1 255.255.255.255 veth2-3 10.100.2.2
route-add 10.100.3.0 255.255.255.0 veth2-5 10.100.4.2
route-add 10.100.0.0 255.255.0.0 veth2-3 10.100.2.2
route-add 10.100.1.0 255.255.255.0 veth2-1 10.100.1.1
```

Then we run other hosts as routers automatic, and try some `traceroute`, we will find:

```
ns2 - 10.100.3.1 (ns3)
ns2 - 10.100.4.2 (ns5) - 10.100.5.2 (ns6) - 10.100.6.1 (ns3) - 10.100.3.2 (ns4)
ns2 - 10.100.2.2 (ns3) - 10.100.6.2 (ns6) - 10.100.4.2 (ns5)
ns2 - 10.100.2.2 (ns3) - 10.100.5.2 (ns6)
ns2 - 10.100.2.2 (ns3) - 10.100.6.2 (ns6)
```

It just behaves like we apply the ”longest prefix matching” rule on the destination.