

CompNet Lab2

1900013039 邓朝萌

Build

Build Libraries:

```
1 | $ cd src  
2 | $ make clean  
3 | $ make
```

Build Libraries for debug:

```
1 | $ cd src  
2 | $ make clean  
3 | $ make debug
```

Build testfiles(after build libraries):

```
1 | $ cd src/test  
2 | $ make clean  
3 | $ make
```

3.2 Link-layer: Packet I/O on Ethernet

Programming Task 1 (PT1).

Finished.

Programming Task 2 (PT2).

Finished.

Checkpoint 1 (CP1).

I used `examples/example.txt` to build the vNet. I ran `./eth_reciever` on `ns2` and ran `./eth_sender` on `ns1`.

Here is the screenshots. They show that my implementation can detect network interfaces on the host.

You should use command `make debug` to build the libraries to show the device name in `initDevice()`, and modify the mac address in `eth_sender.c` to the MAC address of `veth2-1`.

```

root@ubuntu:/home/ava/Lab2/src/test# ./eth_sender
Device name: veth1-2
Device name: any
Device name: bluetooth-monitor
Device name: nflog
Device name: nfqueue
Device name: lo
Receive from device id 0, length 28
Source address: 4a:11:3e:c7:40:3f
Data:
Avava AvA!
Receive from device id 0, length 28
Source address: 4a:11:3e:c7:40:3f
Data:
Avava AvA!
Receive from device id 0, length 28
Source address: 4a:11:3e:c7:40:3f
Data:
Avava AvA!
^C
root@ubuntu:/home/ava/Lab2/src/test# 

root@ubuntu:/home/ava/Lab2/src/test# ./eth_receiver
Hello, I'm Diana!
^C
root@ubuntu:/home/ava/Lab2/src/test# ./eth_receiver
Device name: veth2-1
Device name: veth2-3
Device name: any
Device name: bluetooth-monitor
Device name: nflog
Device name: nfqueue
Device name: lo
Receive from device id 0, length 35
Source address: fe:37:98:dc:83:ef
Data:
Hello, I'm Diana!
Receive from device id 0, length 35
Source address: fe:37:98:dc:83:ef
Data:
Hello, I'm Diana!
Receive from device id 0, length 35
Source address: fe:37:98:dc:83:ef
Data:
Hello, I'm Diana!
^C
root@ubuntu:/home/ava/Lab2/src/test# 

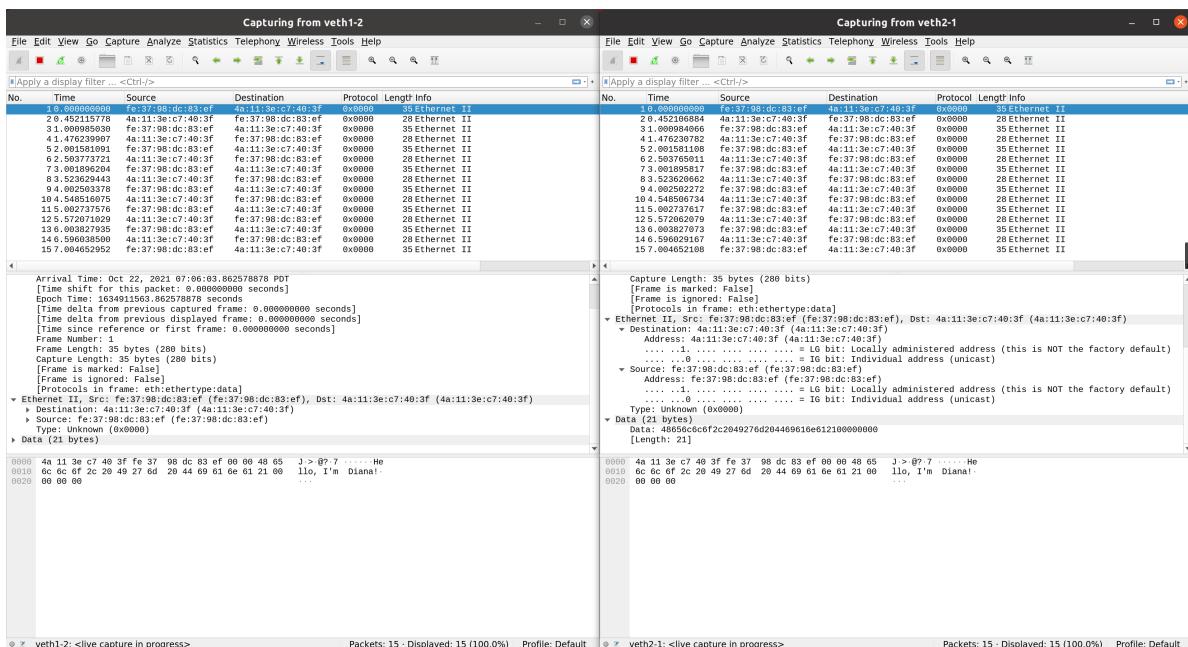
```

Checkpoint 2 (CP2).

I used `examples/example.txt` to build the vNet. I ran `./eth_reciever` on `ns2` and ran `./eth_sender` on `ns1`.

Here is the screenshots.

Traces are dumped in `checkpoints/CP02`.



3.3 Network-layer: IP Protocol

Programming Task 3 (PT3).

Finished.

Writing Task 1 (WT1).

I implemented the ARP protocol supports broadcasting and request the MAC address. The ARP service broadcast its IP address and MAC address every 5 seconds. Other device in the same veth can receive the data and update the ARP table. When a device sends a IP packet to the device in same veth, it first query the ARP table to find out the MAC address corresponding to the IP address. If query failed, it will just broadcast the packet and broadcast a ARP request. Otherwise, the next hop MAC address is in the routing table and the device can just send to it.

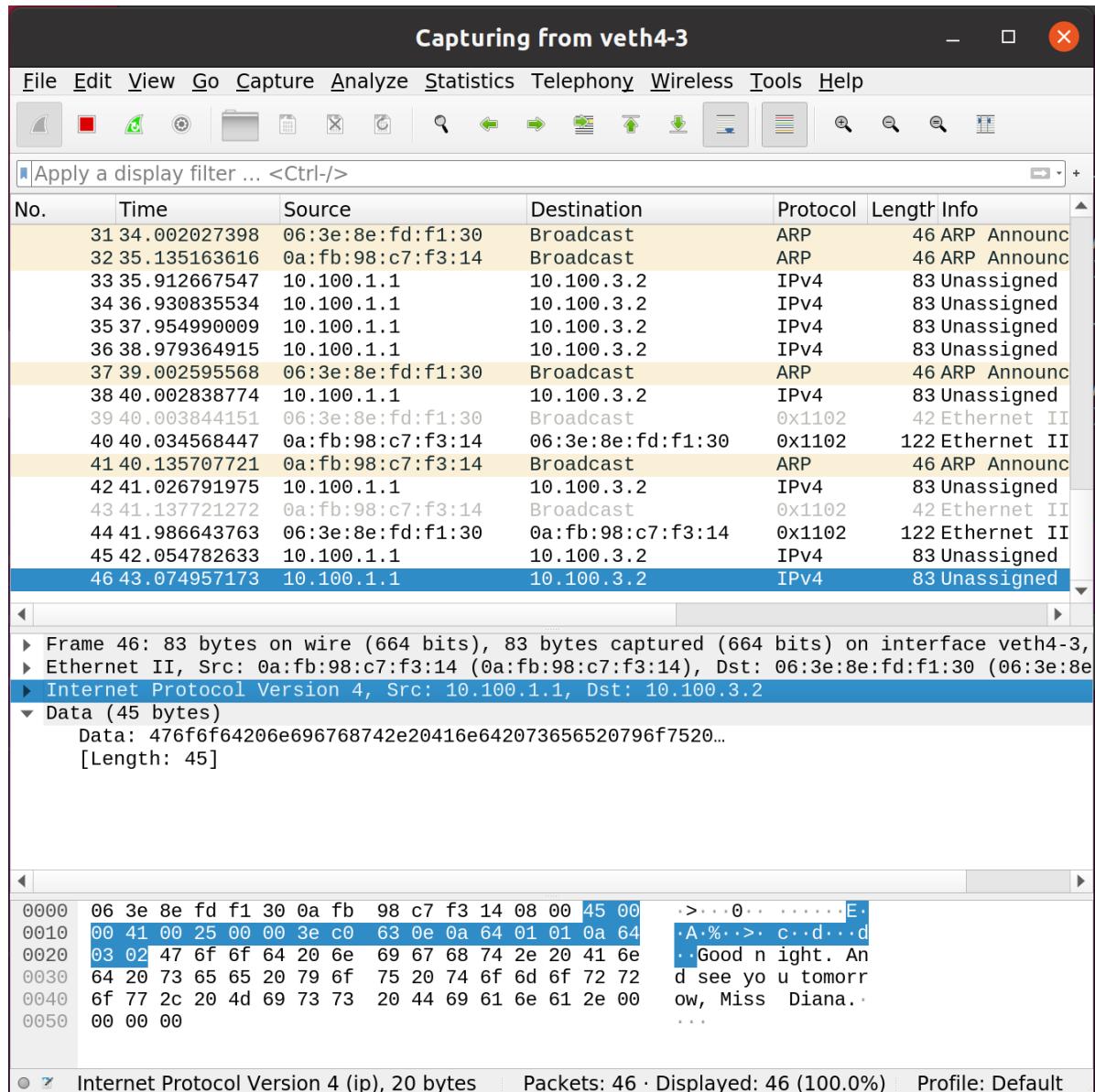
Writing Task 2 (WT2).

I used the Distance Vector routing algorithm. I defined the distance between two adjacent routers is 1 (like Routing Information Protocol). Routers send vectors every 10 seconds to adjacent routers. I use the "Split horizon, poison reverse" strategy at routing table updating to avoid the problems like "Count to infinity".

Checkpoint 3 (CP3).

I used `examples/example.txt` to build the vNet. I ran `./router` on `ns2`, `ns3`, `ns4` and ran `./ip_sender` on `ns1`.

After some routing updating routine, `ns4` can receive the IP packets sent by `ns1`:



One captured IP packet is dumped here:

```
1 45 00 00 41 00 25 00 00 3e c0 63 0e 0a 64 01 01  
2 0a 64 03 02 47 6f 6f 64 20 6e 69 67 68 74 2e 20  
3 41 6e 64 20 73 65 65 20 79 6f 75 20 74 6f 6d 72  
4 72 72 6f 77 2c 20 4d 69 73 73 20 44 69 61 6e 61  
5 2e
```

The first byte is `0x45`, the most significant 4 bits of the first byte is `0100`, it is the version field. For IPv4, this is always equal to 4, the least significant 4 bits of the first byte is `0101`, it is the Internet Header Length, means the length of the IP header is $(5 \ll 2) = 20$ bytes.

The 2nd byte is `0x00`, means the type of service (ToS) is `0x00`.

The 3rd-4th bytes is `0x00 0x41`, means the total length of the IP packet is `0x0041`.

The 5th-6th bytes is `0x00 0x25`, means the identification field of the IP packet is `0x0025`.

The 7th-8th bytes is `0x00 0x00`, the most significant 3 bits of the 7th byte is `000`. The first bit is reversed to `0`. The second bit is `0`, means no Don't Fragment. The second bit is `0`, means no More Fragments. The rest bits is all `0`, means fragment offset is `0`.

The 9th byte is `0x3e`, means the Time to live (TTL) of the packet is `0x3e=62`. If this packet is forwarded 62 more hops, it will be dropped.

The 10th byte is `0xc0`, means the Protocol of the packet content is `0xc0`. It doesn't corresponds to any existing protocol.

The 11th-12th bytes is `0x63 0x0e`, means the header checksum of this packet is `0x630e`.

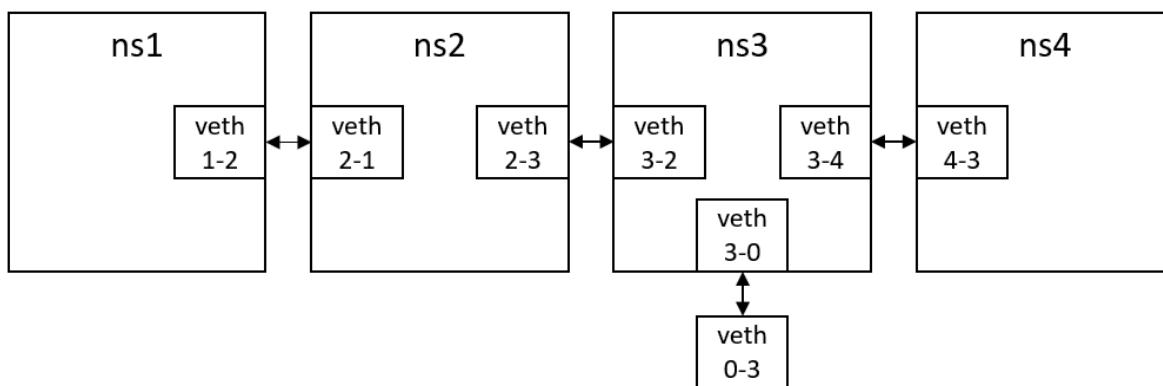
The 13th-16th bytes is `0xa0 0x64 0x01 0x01`, means the IPv4 address of the sender of the packet is `10.100.1.1`.

The 17th-20th bytes is `0xa0 0x64 0x03 0x02`, means the IPv4 address of the receiver of the packet is `10.100.3.2`.

The rest bytes is the IP packet contents. These bytes means a string `"Good night. And see you tomorrow, Miss Diana."`.

Checkpoint 4 (CP4).

I used `examples/example.txt` to build the vNet. The network has the following topology:



`veth1-2, veth2-1` has ip address `10.100.1.0/24`, `veth2-3, veth3-2` has ip address `10.100.2.0/24`, `veth3-4, veth4-3` has ip address `10.100.3.0/24`, `veth3-0` has ip address `10.100.4.0/24`,

I ran `./router` on `ns1, ns2, ns3, ns4`, after some routing updating routine, the routing table in `ns1` is as following:

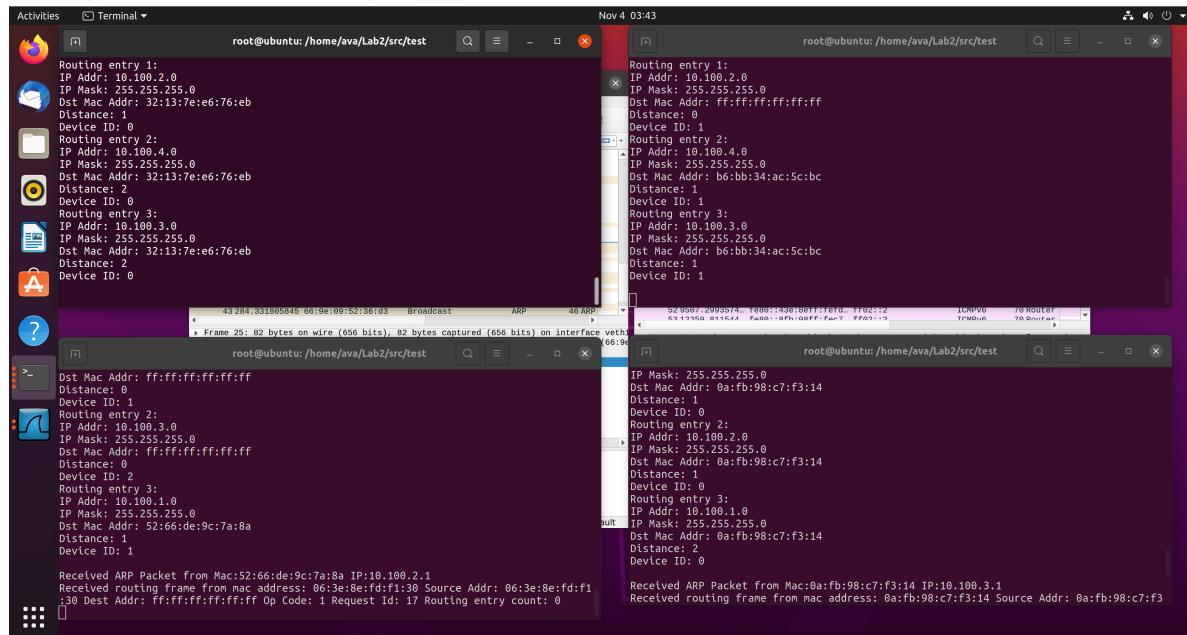
```
1 | Routing table length: 4
2 | Routing entry 0:
3 | IP Addr: 10.100.1.0
4 | IP Mask: 255.255.255.0
5 | Dst Mac Addr: ff:ff:ff:ff:ff:ff
```

```

6 Distance: 0
7 Device ID: 0
8 Routing entry 1:
9 IP Addr: 10.100.2.0
10 IP Mask: 255.255.255.0
11 Dst Mac Addr: 32:13:7e:e6:76:eb
12 Distance: 1
13 Device ID: 0
14 Routing entry 2:
15 IP Addr: 10.100.4.0
16 IP Mask: 255.255.255.0
17 Dst Mac Addr: 32:13:7e:e6:76:eb
18 Distance: 2
19 Device ID: 0
20 Routing entry 3:
21 IP Addr: 10.100.3.0
22 IP Mask: 255.255.255.0
23 Dst Mac Addr: 32:13:7e:e6:76:eb
24 Distance: 2
25 Device ID: 0

```

The topleft terminal is running in `ns1`.



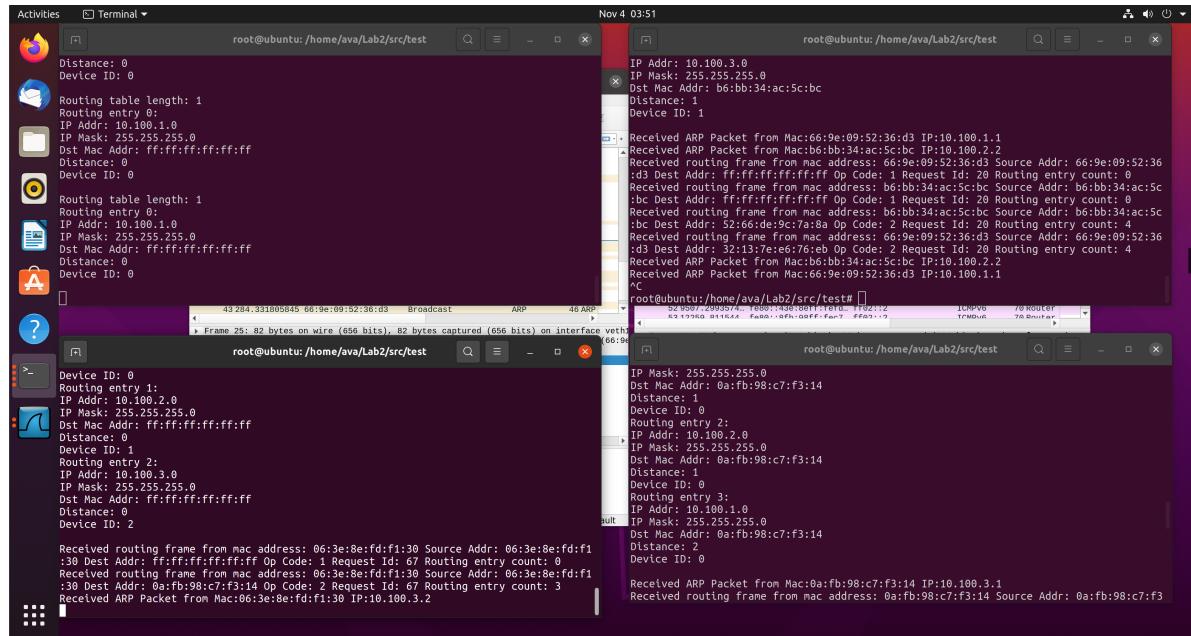
Then stop the `./router` in `ns2`, after some routing updating routine, the routing table in `ns1` is as following:

```

1 Routing table length: 1
2 Routing entry 0:
3 IP Addr: 10.100.1.0
4 IP Mask: 255.255.255.0
5 Dst Mac Addr: ff:ff:ff:ff:ff:ff
6 Distance: 0
7 Device ID: 0

```

The top right terminal is running in ns2.



Then run `./router` in ns2, after some routing updating routine, the routing table in ns1 is as following:

```
1 Routing table length: 4
2 Routing entry 0:
3 IP Addr: 10.100.1.0
4 IP Mask: 255.255.255.0
5 Dst Mac Addr: ff:ff:ff:ff:ff:ff
6 Distance: 0
7 Device ID: 0
8 Routing entry 1:
9 IP Addr: 10.100.2.0
10 IP Mask: 255.255.255.0
11 Dst Mac Addr: 32:13:7e:e6:76:eb
12 Distance: 1
13 Device ID: 0
14 Routing entry 2:
15 IP Addr: 10.100.4.0
16 IP Mask: 255.255.255.0
17 Dst Mac Addr: 32:13:7e:e6:76:eb
18 Distance: 2
19 Device ID: 0
20 Routing entry 3:
21 IP Addr: 10.100.3.0
22 IP Mask: 255.255.255.0
23 Dst Mac Addr: 32:13:7e:e6:76:eb
24 Distance: 2
25 Device ID: 0
```

Checkpoint 5 (CP5).

I used `checkpoint/CP05/vnet.txt` to build the vNet. The network has the following topology:

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

veth1-2, veth2-1 has ip address 10.100.1.0/24, veth2-3, veth3-2 has ip address 10.100.2.0/24, veth3-4, veth4-3 has ip address 10.100.3.0/24, veth2-5, veth5-2 has ip address 10.100.4.0/24, veth5-6, veth6-5 has ip address 10.100.5.0/24, veth6-3, veth3-6 has ip address 10.100.6.0/24.

I ran ./router on ns1, ns2, ns3, ns4, ns5, ns6, after some routing updating routine, the routing table is as following:

```
1 # ns1
2 Routing table length: 6
3 Routing entry 0:
4 IP Addr: 10.100.1.0
5 IP Mask: 255.255.255.0
6 Dst Mac Addr: ff:ff:ff:ff:ff:ff
7 Distance: 0
8 Device ID: 0
9 Routing entry 1:
10 IP Addr: 10.100.2.0
11 IP Mask: 255.255.255.0
12 Dst Mac Addr: ee:a6:29:c7:19:37
13 Distance: 1
14 Device ID: 0
15 Routing entry 2:
16 IP Addr: 10.100.4.0
17 IP Mask: 255.255.255.0
18 Dst Mac Addr: ee:a6:29:c7:19:37
19 Distance: 1
20 Device ID: 0
21 Routing entry 3:
22 IP Addr: 10.100.3.0
23 IP Mask: 255.255.255.0
24 Dst Mac Addr: ee:a6:29:c7:19:37
25 Distance: 2
26 Device ID: 0
27 Routing entry 4:
28 IP Addr: 10.100.6.0
29 IP Mask: 255.255.255.0
30 Dst Mac Addr: ee:a6:29:c7:19:37
31 Distance: 2
32 Device ID: 0
33 Routing entry 5:
34 IP Addr: 10.100.5.0
35 IP Mask: 255.255.255.0
36 Dst Mac Addr: ee:a6:29:c7:19:37
37 Distance: 2
38 Device ID: 0
39
40 # ns2
41
42 Routing table length: 6
43 Routing entry 0:
44 IP Addr: 10.100.1.0
45 IP Mask: 255.255.255.0
46 Dst Mac Addr: ff:ff:ff:ff:ff:ff
47 Distance: 0
48 Device ID: 0
49 Routing entry 1:
```

```
50 IP Addr: 10.100.2.0
51 IP Mask: 255.255.255.0
52 Dst Mac Addr: ff:ff:ff:ff:ff:ff
53 Distance: 0
54 Device ID: 1
55 Routing entry 2:
56 IP Addr: 10.100.4.0
57 IP Mask: 255.255.255.0
58 Dst Mac Addr: ff:ff:ff:ff:ff:ff
59 Distance: 0
60 Device ID: 2
61 Routing entry 3:
62 IP Addr: 10.100.3.0
63 IP Mask: 255.255.255.0
64 Dst Mac Addr: 6e:af:70:0a:5b:b8
65 Distance: 1
66 Device ID: 1
67 Routing entry 4:
68 IP Addr: 10.100.6.0
69 IP Mask: 255.255.255.0
70 Dst Mac Addr: 6e:af:70:0a:5b:b8
71 Distance: 1
72 Device ID: 1
73 Routing entry 5:
74 IP Addr: 10.100.5.0
75 IP Mask: 255.255.255.0
76 Dst Mac Addr: 8e:4b:75:73:75:fb
77 Distance: 1
78 Device ID: 2
79
80 # ns3
81
82 Routing table length: 6
83 Routing entry 0:
84 IP Addr: 10.100.2.0
85 IP Mask: 255.255.255.0
86 Dst Mac Addr: ff:ff:ff:ff:ff:ff
87 Distance: 0
88 Device ID: 0
89 Routing entry 1:
90 IP Addr: 10.100.3.0
91 IP Mask: 255.255.255.0
92 Dst Mac Addr: ff:ff:ff:ff:ff:ff
93 Distance: 0
94 Device ID: 1
95 Routing entry 2:
96 IP Addr: 10.100.6.0
97 IP Mask: 255.255.255.0
98 Dst Mac Addr: ff:ff:ff:ff:ff:ff
99 Distance: 0
100 Device ID: 2
101 Routing entry 3:
102 IP Addr: 10.100.1.0
103 IP Mask: 255.255.255.0
104 Dst Mac Addr: ae:b6:eb:c1:d1:44
105 Distance: 1
106 Device ID: 0
107 Routing entry 4:
```

```
108 IP Addr: 10.100.4.0
109 IP Mask: 255.255.255.0
110 Dst Mac Addr: ae:b6:eb:c1:d1:44
111 Distance: 1
112 Device ID: 0
113 Routing entry 5:
114 IP Addr: 10.100.5.0
115 IP Mask: 255.255.255.0
116 Dst Mac Addr: ee:8c:26:5f:bb:bc
117 Distance: 1
118 Device ID: 2
119
120 # ns4
121
122 Routing table length: 6
123 Routing entry 0:
124 IP Addr: 10.100.3.0
125 IP Mask: 255.255.255.0
126 Dst Mac Addr: ff:ff:ff:ff:ff:ff
127 Distance: 0
128 Device ID: 0
129 Routing entry 1:
130 IP Addr: 10.100.2.0
131 IP Mask: 255.255.255.0
132 Dst Mac Addr: 8a:f7:2e:48:e8:aa
133 Distance: 1
134 Device ID: 0
135 Routing entry 2:
136 IP Addr: 10.100.6.0
137 IP Mask: 255.255.255.0
138 Dst Mac Addr: 8a:f7:2e:48:e8:aa
139 Distance: 1
140 Device ID: 0
141 Routing entry 3:
142 IP Addr: 10.100.1.0
143 IP Mask: 255.255.255.0
144 Dst Mac Addr: 8a:f7:2e:48:e8:aa
145 Distance: 2
146 Device ID: 0
147 Routing entry 4:
148 IP Addr: 10.100.4.0
149 IP Mask: 255.255.255.0
150 Dst Mac Addr: 8a:f7:2e:48:e8:aa
151 Distance: 2
152 Device ID: 0
153 Routing entry 5:
154 IP Addr: 10.100.5.0
155 IP Mask: 255.255.255.0
156 Dst Mac Addr: 8a:f7:2e:48:e8:aa
157 Distance: 2
158 Device ID: 0
159
160 # ns5
161
162 Routing table length: 6
163 Routing entry 0:
164 IP Addr: 10.100.4.0
165 IP Mask: 255.255.255.0
```

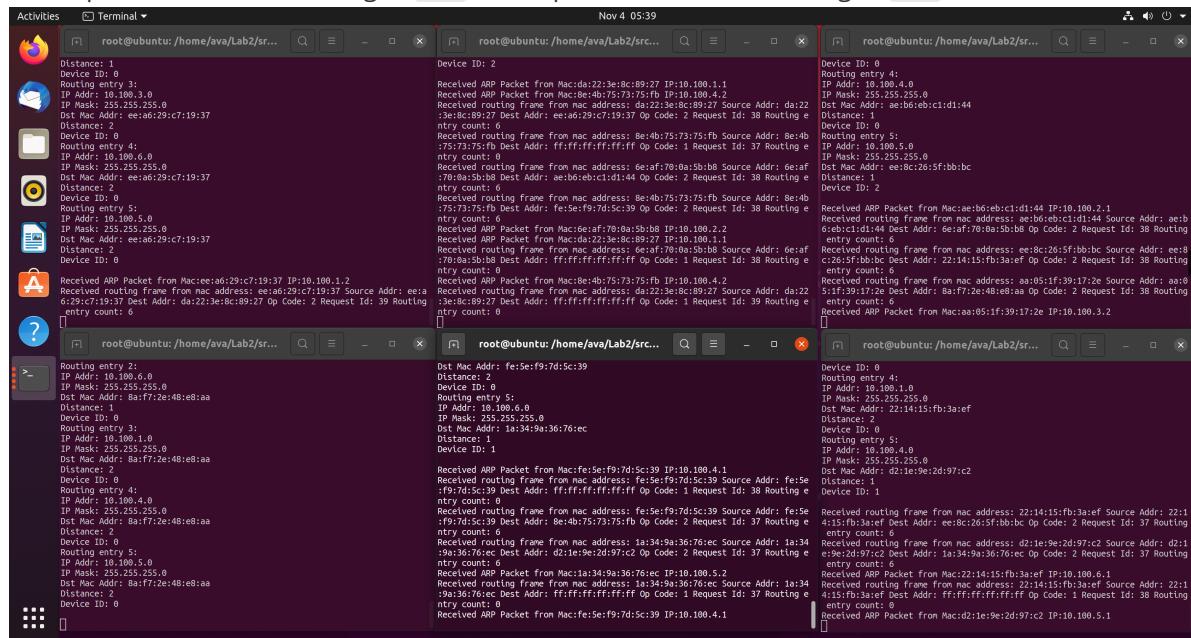
```
166 Dst Mac Addr: ff:ff:ff:ff:ff:ff
167 Distance: 0
168 Device ID: 0
169 Routing entry 1:
170 IP Addr: 10.100.5.0
171 IP Mask: 255.255.255.0
172 Dst Mac Addr: ff:ff:ff:ff:ff:ff
173 Distance: 0
174 Device ID: 1
175 Routing entry 2:
176 IP Addr: 10.100.1.0
177 IP Mask: 255.255.255.0
178 Dst Mac Addr: fe:5e:f9:7d:5c:39
179 Distance: 1
180 Device ID: 0
181 Routing entry 3:
182 IP Addr: 10.100.2.0
183 IP Mask: 255.255.255.0
184 Dst Mac Addr: fe:5e:f9:7d:5c:39
185 Distance: 1
186 Device ID: 0
187 Routing entry 4:
188 IP Addr: 10.100.3.0
189 IP Mask: 255.255.255.0
190 Dst Mac Addr: fe:5e:f9:7d:5c:39
191 Distance: 2
192 Device ID: 0
193 Routing entry 5:
194 IP Addr: 10.100.6.0
195 IP Mask: 255.255.255.0
196 Dst Mac Addr: 1a:34:9a:36:76:ec
197 Distance: 1
198 Device ID: 1
199
200 # ns6
201
202 Routing table length: 6
203 Routing entry 0:
204 IP Addr: 10.100.6.0
205 IP Mask: 255.255.255.0
206 Dst Mac Addr: ff:ff:ff:ff:ff:ff
207 Distance: 0
208 Device ID: 0
209 Routing entry 1:
210 IP Addr: 10.100.5.0
211 IP Mask: 255.255.255.0
212 Dst Mac Addr: ff:ff:ff:ff:ff:ff
213 Distance: 0
214 Device ID: 1
215 Routing entry 2:
216 IP Addr: 10.100.2.0
217 IP Mask: 255.255.255.0
218 Dst Mac Addr: 22:14:15:fb:3a:ef
219 Distance: 1
220 Device ID: 0
221 Routing entry 3:
222 IP Addr: 10.100.3.0
223 IP Mask: 255.255.255.0
```

```

224 Dst Mac Addr: 22:14:15:fb:3a:ef
225 Distance: 1
226 Device ID: 0
227 Routing entry 4:
228 IP Addr: 10.100.1.0
229 IP Mask: 255.255.255.0
230 Dst Mac Addr: 22:14:15:fb:3a:ef
231 Distance: 2
232 Device ID: 0
233 Routing entry 5:
234 IP Addr: 10.100.4.0
235 IP Mask: 255.255.255.0
236 Dst Mac Addr: d2:1e:9e:2d:97:c2
237 Distance: 1
238 Device ID: 1

```

The topleft terminal is running in `ns1`, the topmid terminal is running in `ns2`, etc.



After stop the `./router` in `ns5`, the routing table is as following:

```

1 # ns1
2
3 Routing table length: 6
4 Routing entry 0:
5 IP Addr: 10.100.1.0
6 IP Mask: 255.255.255.0
7 Dst Mac Addr: ff:ff:ff:ff:ff:ff
8 Distance: 0
9 Device ID: 0
10 Routing entry 1:
11 IP Addr: 10.100.2.0
12 IP Mask: 255.255.255.0
13 Dst Mac Addr: ee:a6:29:c7:19:37
14 Distance: 1
15 Device ID: 0
16 Routing entry 2:
17 IP Addr: 10.100.4.0
18 IP Mask: 255.255.255.0
19 Dst Mac Addr: ee:a6:29:c7:19:37
20 Distance: 1

```

```
21 Device ID: 0
22 Routing entry 3:
23 IP Addr: 10.100.3.0
24 IP Mask: 255.255.255.0
25 Dst Mac Addr: ee:a6:29:c7:19:37
26 Distance: 2
27 Device ID: 0
28 Routing entry 4:
29 IP Addr: 10.100.6.0
30 IP Mask: 255.255.255.0
31 Dst Mac Addr: ee:a6:29:c7:19:37
32 Distance: 2
33 Device ID: 0
34 Routing entry 5:
35 IP Addr: 10.100.5.0
36 IP Mask: 255.255.255.0
37 Dst Mac Addr: ee:a6:29:c7:19:37
38 Distance: 3
39 Device ID: 0
40
41 # ns2
42
43 Routing table length: 6
44 Routing entry 0:
45 IP Addr: 10.100.1.0
46 IP Mask: 255.255.255.0
47 Dst Mac Addr: ff:ff:ff:ff:ff:ff
48 Distance: 0
49 Device ID: 0
50 Routing entry 1:
51 IP Addr: 10.100.2.0
52 IP Mask: 255.255.255.0
53 Dst Mac Addr: ff:ff:ff:ff:ff:ff
54 Distance: 0
55 Device ID: 1
56 Routing entry 2:
57 IP Addr: 10.100.4.0
58 IP Mask: 255.255.255.0
59 Dst Mac Addr: ff:ff:ff:ff:ff:ff
60 Distance: 0
61 Device ID: 2
62 Routing entry 3:
63 IP Addr: 10.100.3.0
64 IP Mask: 255.255.255.0
65 Dst Mac Addr: 6e:af:70:0a:5b:b8
66 Distance: 1
67 Device ID: 1
68 Routing entry 4:
69 IP Addr: 10.100.6.0
70 IP Mask: 255.255.255.0
71 Dst Mac Addr: 6e:af:70:0a:5b:b8
72 Distance: 1
73 Device ID: 1
74 Routing entry 5:
75 IP Addr: 10.100.5.0
76 IP Mask: 255.255.255.0
77 Dst Mac Addr: 6e:af:70:0a:5b:b8
78 Distance: 2
```

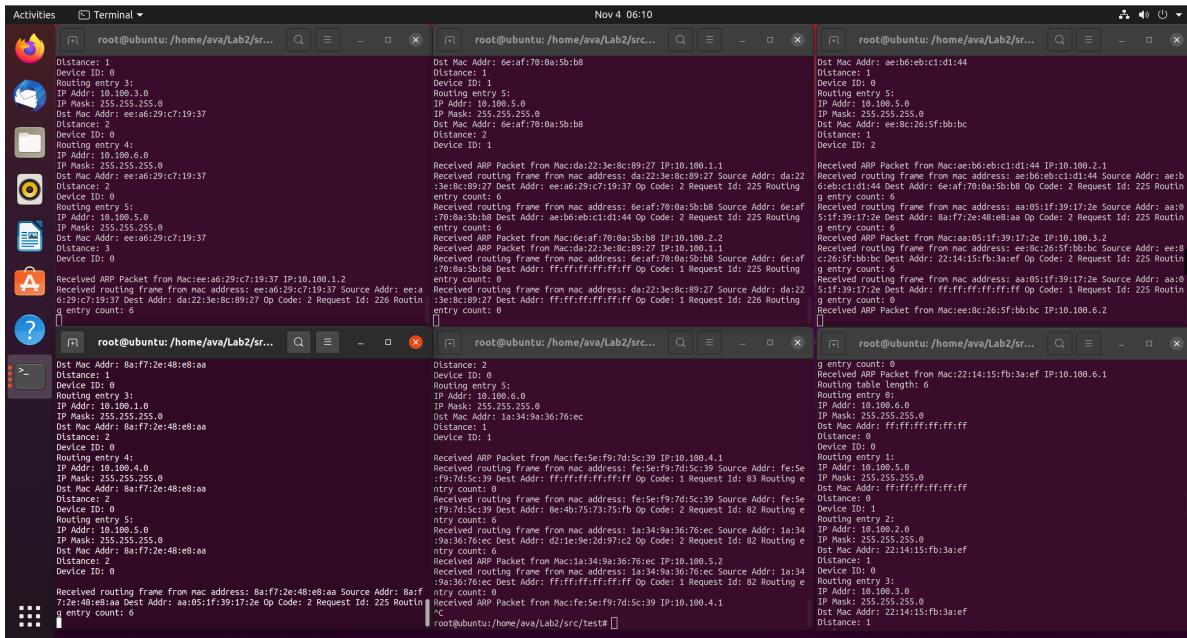
```
79 Device ID: 1
80
81 # ns3
82
83 Routing table length: 6
84 Routing entry 0:
85 IP Addr: 10.100.2.0
86 IP Mask: 255.255.255.0
87 Dst Mac Addr: ff:ff:ff:ff:ff:ff
88 Distance: 0
89 Device ID: 0
90 Routing entry 1:
91 IP Addr: 10.100.3.0
92 IP Mask: 255.255.255.0
93 Dst Mac Addr: ff:ff:ff:ff:ff:ff
94 Distance: 0
95 Device ID: 1
96 Routing entry 2:
97 IP Addr: 10.100.6.0
98 IP Mask: 255.255.255.0
99 Dst Mac Addr: ff:ff:ff:ff:ff:ff
100 Distance: 0
101 Device ID: 2
102 Routing entry 3:
103 IP Addr: 10.100.1.0
104 IP Mask: 255.255.255.0
105 Dst Mac Addr: ae:b6:eb:c1:d1:44
106 Distance: 1
107 Device ID: 0
108 Routing entry 4:
109 IP Addr: 10.100.4.0
110 IP Mask: 255.255.255.0
111 Dst Mac Addr: ae:b6:eb:c1:d1:44
112 Distance: 1
113 Device ID: 0
114 Routing entry 5:
115 IP Addr: 10.100.5.0
116 IP Mask: 255.255.255.0
117 Dst Mac Addr: ee:8c:26:5f:bb:bc
118 Distance: 1
119 Device ID: 2
120
121 # ns4
122
123 Routing table length: 6
124 Routing entry 0:
125 IP Addr: 10.100.3.0
126 IP Mask: 255.255.255.0
127 Dst Mac Addr: ff:ff:ff:ff:ff:ff
128 Distance: 0
129 Device ID: 0
130 Routing entry 1:
131 IP Addr: 10.100.2.0
132 IP Mask: 255.255.255.0
133 Dst Mac Addr: 8a:f7:2e:48:e8:aa
134 Distance: 1
135 Device ID: 0
136 Routing entry 2:
```

```
137 IP Addr: 10.100.6.0
138 IP Mask: 255.255.255.0
139 Dst Mac Addr: 8a:f7:2e:48:e8:aa
140 Distance: 1
141 Device ID: 0
142 Routing entry 3:
143 IP Addr: 10.100.1.0
144 IP Mask: 255.255.255.0
145 Dst Mac Addr: 8a:f7:2e:48:e8:aa
146 Distance: 2
147 Device ID: 0
148 Routing entry 4:
149 IP Addr: 10.100.4.0
150 IP Mask: 255.255.255.0
151 Dst Mac Addr: 8a:f7:2e:48:e8:aa
152 Distance: 2
153 Device ID: 0
154 Routing entry 5:
155 IP Addr: 10.100.5.0
156 IP Mask: 255.255.255.0
157 Dst Mac Addr: 8a:f7:2e:48:e8:aa
158 Distance: 2
159 Device ID: 0
160
161 # ns6
162
163 Routing table length: 6
164 Routing entry 0:
165 IP Addr: 10.100.6.0
166 IP Mask: 255.255.255.0
167 Dst Mac Addr: ff:ff:ff:ff:ff:ff
168 Distance: 0
169 Device ID: 0
170 Routing entry 1:
171 IP Addr: 10.100.5.0
172 IP Mask: 255.255.255.0
173 Dst Mac Addr: ff:ff:ff:ff:ff:ff
174 Distance: 0
175 Device ID: 1
176 Routing entry 2:
177 IP Addr: 10.100.2.0
178 IP Mask: 255.255.255.0
179 Dst Mac Addr: 22:14:15:fb:3a:ef
180 Distance: 1
181 Device ID: 0
182 Routing entry 3:
183 IP Addr: 10.100.3.0
184 IP Mask: 255.255.255.0
185 Dst Mac Addr: 22:14:15:fb:3a:ef
186 Distance: 1
187 Device ID: 0
188 Routing entry 4:
189 IP Addr: 10.100.1.0
190 IP Mask: 255.255.255.0
191 Dst Mac Addr: 22:14:15:fb:3a:ef
192 Distance: 2
193 Device ID: 0
194 Routing entry 5:
```

```

195 IP Addr: 10.100.4.0
196 IP Mask: 255.255.255.0
197 Dst Mac Addr: 22:14:15:fb:3a:ef
198 Distance: 2
199 Device ID: 0

```



Checkpoint 6 (CP6).

I used `checkpoint/CP06/vnet.txt` to build the vNet. The network has the following topology:

```

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

```

`veth1-2`, `veth2-1` has ip address `10.100.1.0/24`, `veth2-3`, `veth3-2` has ip address `10.100.2.0/24`, `veth3-4`, `veth4-3` has ip address `10.100.3.0/24`, `veth2-5`, `veth5-2` has ip address `10.100.4.0/24`.

I ran `./router` on `ns3`, `ns4`, `ns5`, ran `./router_2` on `ns2`, ran `./ip_sender` on `ns1`.
`./router_2` is a common `./router` with manually setting up the routing table that packet with "dest ip:0.0.0.0, mask:0.0.0.0" will be forwarded to `ns5`.

After some routing updating routine, the routing table of `ns2` is as following:

```

1 Routing table length: 5
2 Routing entry 0:
3   IP Addr: 10.100.1.0
4   IP Mask: 255.255.255.0
5   Dst Mac Addr: ff:ff:ff:ff:ff:ff
6   Distance: 0
7   Device ID: 0
8   Routing entry 1:
9     IP Addr: 10.100.2.0
10    IP Mask: 255.255.255.0
11    Dst Mac Addr: ff:ff:ff:ff:ff:ff
12    Distance: 0
13    Device ID: 1
14    Routing entry 2:

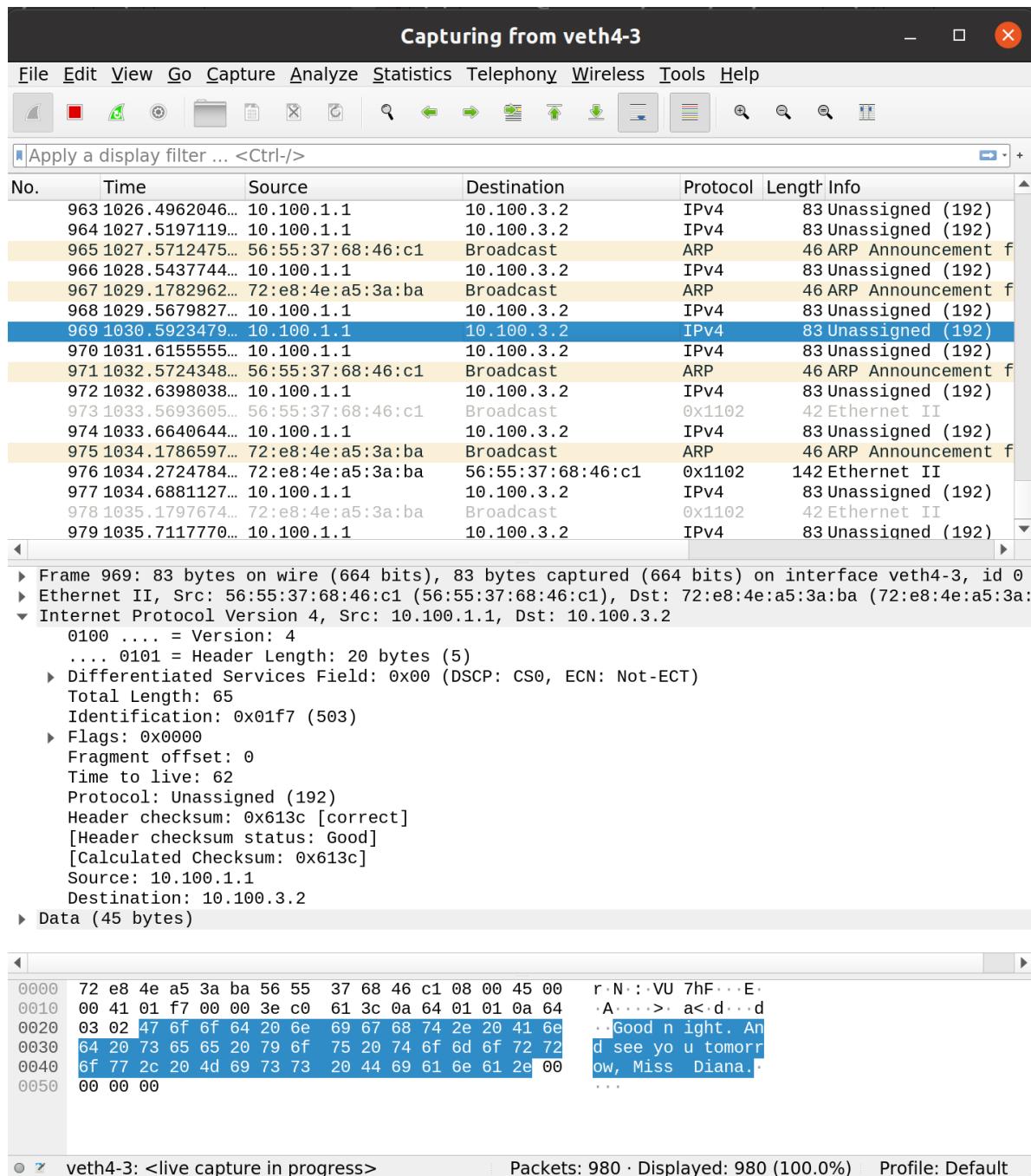
```

```
15 IP Addr: 10.100.4.0
16 IP Mask: 255.255.255.0
17 Dst Mac Addr: ff:ff:ff:ff:ff:ff
18 Distance: 0
19 Device ID: 2
20 Routing entry 3:
21 IP Addr: 0.0.0.0
22 IP Mask: 0.0.0.0
23 Dst Mac Addr: ff:ff:ff:ff:ff:ff
24 Distance: 0
25 Device ID: 2
26 Routing entry 4:
27 IP Addr: 10.100.3.0
28 IP Mask: 255.255.255.0
29 Dst Mac Addr: 8e:8c:a0:b7:8c:c9
30 Distance: 1
31 Device ID: 1
```

`./ip_sender` in `ns1` send packet to `10.100.3.2` at `ns4`. When the packet arrived at `ns2`, `ns2` print the message below which showed the `ns2` forwarded the ip packet applying the "longest prefix matching" rule:

```
1 | Forward IP Packet Form Src 10.100.1.1, Dst 10.100.3.2, TTL=63, Next Hop MAC:
  | 8e:8c:a0:b7:8c:c9 Len=65
```

And we can see `ns4` successfully received the packet with correct TTL:



3.4 Transport-layer: TCP Protocol

Programming Task 4 (PT4).

Finished. (`__wrap_getaddrinfo`) is implemented by call built-in `getaddrinfo` directly)

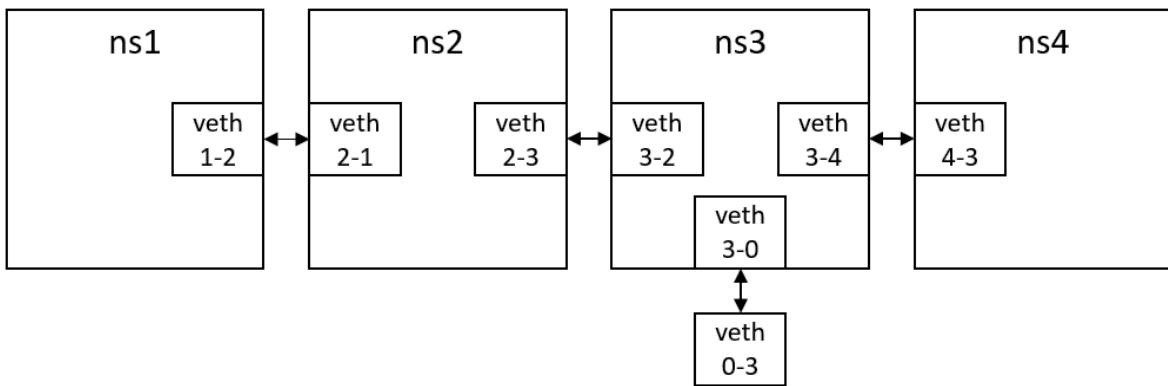
Writing Task 3 (WT3).

- I use a `struct socket_info_t` to save all the information about a socket. The `struct socket_info_t` records each socket's state, type, addr, port, etc.
- When user program call `socket`, the protocol stack alloc a new empty `CLOSE` socket. When user program call `bind`, the protocol stack set some information of the socket and change the socket state to `BINDED`.
- When user program call `listen`, the protocol stack set some information of the socket and change the socket state to `LISTEN`.
- When user program call `connect`, the protocol stack set some information of the socket, change the socket state to `SYN_SENT`, and send a `SYN` segment to the server.

- When the server receives a `SYN` segment, the corresponding `LISTEN` socket add the connection to a waiting list. - When user program call `accept`, if there is a connection in a waiting list, the protocol stack alloc a new data socket, change the socket state to `SYN_RECV`, and send a `SYNACK` segment to the client. the protocol stack will wait a new `SYN` segment of the accepting socket if the waiting list is empty.
- When the client receives a `SYNACK` segment of the connecting socket, the protocol stack change the socket state to `ESTABLISHED`, and send a `ACK` segment to the server.
- When the client receives a `ACK` segment of a `SYN_RECV` socket, the protocol stack change the socket state to `ESTABLISHED`.
- When user program call `read`, the protocol stack will wait for the input ring buffer and send as most data as it can read data from the input buffer to the user program.
- When user program call `write`, the protocol stack will wait for the freespace of the output ring buffer and read as most data as it can write data to the output buffer from the user program, then write the data into the output buffer.
- When user program call `close`, the protocol stack will set some information of the socket, change the socket state to `FIN_WAIT1`, send a `FIN` segment to the remote.
- When the client/server receives a `FIN` segment of a socket, the protocol stack change the socket state to `CLOSE_WAIT`, send a `FIN ACK` segment to the remote, to wait for all data to be sent.
- When the client/server sent all of the data of a `CLOSE_WAIT` socket, the protocol stack change the socket state to `LAST_ACK` and send a `FIN` segment to the remote.
- When the client/server receives a `ACK` segment of a `FIN_WAIT1` socket, the protocol stack change the socket state to `FIN_WAIT2`, and write data of the socket is no longer allowed.
- When the client/server receives a `FIN` segment of a `FIN_WAIT2` socket, the protocol stack change the socket state to `TIME_WAIT`, send a `FIN ACK` segment to the remote, wait a long time for all packets to be die out, the close the socket.
- When the client/server receives a `ACK` segment of a `LAST_ACK` socket, the protocol stack close the socket.

Checkpoint 7 (CP7).

I used `examples/example.txt` to build the vNet. The network has the following topology:



`veth1-2, veth2-1` has ip address `10.100.1.0/24`, `veth2-3, veth3-2` has ip address `10.100.2.0/24`, `veth3-4, veth4-3` has ip address `10.100.3.0/24`, `veth3-0` has ip address `10.100.4.0/24`,

I ran `./router` on `ns2`, `ns3` for IP layer routing/forwarding and `./tcp_protocol_stack` `ns1`, `ns4` to support socket API.

I ran `./tcp_server` on `ns1` and `./tcp_client` on `ns4`. The `./tcp_client` send a high resolution `diana.png` to the `./tcp_server` and `./tcp_server` download it.



The packet captured by wireshark in `ns1` is saved at `checkpoints/CP07/cp07.pcapng`

I choose the `#100` packet. The TCP header is here:

```
1 | 0000  c3 50 04 63 00 00 e6 a9 00 00 00 02 50 10 7f ff  
2 | 0010  85 56 00 00
```

- The first `2` byte is `c3 50`, means the source port is `50000`.
- The `3rd-4th` byte is `04 63`, means the destination port is `1123`.
- The `5-8th` byte is `00 00 e6 a9`, means the raw sequence number is `59049`.
- The `9-12th` byte is `00 00 00 02`, means the raw ack number is `2`.
- The `13-14th` byte is `50 10`. The first 4 bit is `0101`, means the TCP header length is `5 * 4 = 20`. The next 12 bit is flag bits, only the ACK bit is `1`. means it is an ACK segment.

```

1 000 ..... = Reserved: Not set
2 ....0 ..... =Nonce: Not set
3 ....0.... .... = Congestion Window Reduced (CWR): Not set
4 ....0.. .... = ECN-Echo: Not set
5 ....0. .... =Urgent: Not set
6 ....1 .... =Acknowledgment: Set
7 ....0.... =Push: Not set
8 ....0.. =Reset: Not set
9 ....0. .... =Syn: Not set
10 ....0.... =Fin: Not set

```

- The 15-16th byte is 7f ff. means the window size of sender is 32767.
- The 17-18th byte is 85 56. It's the TCP checksum.
- The 19-20th byte is 00 00. It's the urgent pointer(not implemented).

Checkpoint 8 (CP8).

I used the same setting as the CP7. The only difference is I set a 2% packet with loss rate at line 647-649 in `tcp_protocol_stack.c`. (I don't know how to use `netem` in `veth` generated by `vnetutils`, so I manually set the loss rate in my protocol stack). The TCP client send data slowly, but all data is correctly delivered. The packet captured by wireshark in `ns1` is saved at `checkpoints/CP08/cp08.pcapng`.

Checkpoint 9 (CP9).

I build the `echo_server` and the `echo_client` in `/src/checkpoint`. The source code is without any modification. You can just run `make clean & make` in `/src/checkpoint` to build the program. As before, I also need to run `./router` on `ns2, ns3` for IP layer routing/forwarding and `./tcp_protocol_stack ns1, ns4` to support socket API.

The output of `echo_server` is here:

```

1 new connection
2 6 12 13 14 63 68 70 72 74 76 78 80 82 84 86 87 88 89 1549 4184 5644 8279 9739
   12374 13834 15000 all: 15000
3 new connection
4 6 12 13 14 63 68 70 72 74 76 78 80 82 84 86 87 88 89 4184 5644 8279 9739
   12374 13834 15000 all: 15000
5 new connection
6 6 12 13 14 63 68 70 72 74 76 78 80 82 84 86 87 88 89 1549 3009 4184 5644 8279
   9739 12374 13834 15000 all: 15000

```

The output of `echo_client` is here:

```

1 loop #1 ok.
2 loop #2 ok.
3 loop #3 ok.

```

The packet captured by wireshark in `ns1` is saved at `checkpoints/CP09/cp09.pcapng`.

Checkpoint 10 (CP10).

I build the `perf_server` and the `perf_client` in `/src/checkpoint`. The source code is without any modification. You can just run `make clean & make` in `/src/checkpoint` to build the program. As before, I also need to run `./router` on `ns2`, `ns3` for IP layer routing/forwarding and `./tcp_protocol_stack ns1 ns4` to support socket API.

The output of `perf_server` is here:

```
1 new connection
2 a11: 1460000
3 str_echo: read error
```

The output of `perf_client` is here:

```
1 sending ...
2 receiving ...
3 665.39 KB/s
4 sending ...
5 receiving ...
6 756.34 KB/s
7 sending ...
8 receiving ...
9 686.72 KB/s
10 sending ...
11 receiving ...
12 748.31 KB/s
13 sending ...
14 receiving ...
15 680.09 KB/s
16 sending ...
17 receiving ...
18 700.07 KB/s
19 sending ...
20 receiving ...
21 689.46 KB/s
22 sending ...
23 receiving ...
24 675.08 KB/s
25 sending ...
26 receiving ...
27 708.57 KB/s
28 sending ...
29 receiving ...
30 684.69 KB/s
```

The packet captured by wireshark in `ns1` is saved at `checkpoints/CP10/cp10.pcapng`.

3.5 Bonus: Test/Evaluation

Not implemented.

3.6 Challenge

Challenge 1 (CL1).

I used the same setting as the CP7. I run 1 `./tcp_server` on `ns1` and then run two `./tcp_client` simultaneously. The `./tcp_server` will accept one connection first, when the `./tcp_server` finished the connection, it will accept the next connection.

The output of the `./tcp_server` is in `checkpoints/CL01/trace.txt`.

The packet captured by wireshark in `ns1` is saved at `checkpoints/CL01/c101.pcapng`.

Challenge 2 (CL2).

To test the fragmentation and reassembly functions, you can modify the `IP_FRAG_SIZE` in `src/include/network/ip.h` to a smaller number. I modified it to 800. After modify it, you should rebuild all the libraries and user programs.

The rest setting is same as I used in the CP7. You can see the protocol stack ran successfully.

The packet captured by wireshark in `ns1` is saved at `checkpoints/CL02/c102.pcapng`.

Challenge 2 (CL4).

To test the TCP flow control functions, I modified the `./tcp_server` to read at most 16 byte each time, the the speed of receiving became extremely slow.

The rest setting is same as I used in the CP7. The server used near 1 minute to receive the `1MB` photo. But the data delivered successfully.

The packet captured by wireshark in `ns1` is saved at `checkpoints/CL04/c104.pcapng`.