Report For CompNet Lab2

> 1900013234 Haihong Tang

Part A

(For better demonstration, I updated implementation for this part.)

To check my program, I use `./build/checkCP1CP2` , whose source code is `./check/checkCP1CP2.c`.

PT1

codelist: `inc/device.h`, `src/device.c`, `inc/inc.h`, `src/inc.c`

PT2

codelist: `inc/packetio.h`, `src/packetio.c`

CP1

From the beginning, I used vnet tools to construct a simple veth-pair as below:

```
✘ rgnoh@rgnoh-VirtualBox  ~/CompNet/vnetUtils/helper  sudo ./addNS n1
[sudo] rgnoh 的密码：
 rgnoh@rgnoh-VirtualBox  ~/CompNet/vnetUtils/helper  sudo ./addNS n2
 rgnoh@rgnoh-VirtualBox  ~/CompNet/vnetUtils/helper  sudo ./connectNS n1 n2 v1
 v2 10.100.1
```

Then I run `checkCP1CP2` on `n1` and `n2` respectively. To check whether my program can detect network interfaces on the host, I use these codes below:

```
    if(addDevice(dev1) != -1){
        printf("Device founded: %s\n", dev1);
    }
    if(addDevice(dev2) != -1){
        printf("Device founded: %s\n", dev2);
    }
```

The result proved me correct.

```
root@rgnoh-VirtualBox:~/CompNet/lab2/build# sudo ./checkCP1CP2
Device founded: v1
Couldn't find device v2
```

```
root@rgnoh-VirtualBox:~/CompNet/lab2/build# sudo ./checkCP1CP2
Couldn't find device v1
Device founded: v2
```

CP2

In my implementation, `checkCP1CP2` uses a thread to receive packets per device. When a packet arrives, it calls `printInfoCallback` to display the packet. `n1` and `n2` use broadcasting to send packets.

```c
    setFrameReceiveCallback(printInfoCallBack);
    char buf[128] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    for(int i = 0; i < 10; i++){
        sendFrame(buf, strlen(buf), ETH_P_IP, BroadcastMac, 0);
        sleep(1);
    }
```

```
root@rgnoh-VirtualBox:~/CompNet/lab2/build# sudo ./checkCP1CP2
Device founded: v1
Couldn't find device v2
get a packet!
length: 50      device ID: 0    device name:v1
ff ff ff ff ff ff
ba 46 a3 fa 3d 1a
08 00
41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50
51 52 53 54 55 56 57 58
59 5a 30 31 32 33 34 35
36 37 38 39

get a packet!
length: 50      device ID: 0    device name:v1
ff ff ff ff ff ff
ba 46 a3 fa 3d 1a
08 00
41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50
51 52 53 54 55 56 57 58
```

```
root@rgnoh-VirtualBox:~/CompNet/lab2/build# sudo ./checkCP1CP2
Couldn't find device v1
Device founded: v2
get a packet!
length: 50      device ID: 0    device name:v2
ff ff ff ff ff ff
02 9c f8 7e b9 5d
08 00
41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50
51 52 53 54 55 56 57 58
59 5a 30 31 32 33 34 35
36 37 38 39

get a packet!
length: 50      device ID: 0    device name:v2
ff ff ff ff ff ff
02 9c f8 7e b9 5d
08 00
41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50
51 52 53 54 55 56 57 58
```

Part B

PT3

codelist: `inc/ip.h`, `src/ip.c`, `inc/arp.h`, `src/arp.c` (also modified `inc/packetio.h` and `src/packetio.c`)

WT1

> `sendFrame()` requires the caller to provide the destination MAC address when sending IP packets, but users of IP layer will not provide the address to you. Explain how you addressed this problem when implementing IP protocol.

In my implementation, when sending packets for routing, since a host's routing table should be sent to all of its neighbors, we can just set the MAC address `ff:ff:ff:ff:ff:ff` for broadcasting.

As for normal IP packets, I implement a simple ARP protocol in `src/arp.c`. When sending packets to `nextHop` decided by matching the routing table, we first look up ARP table for MAC address. If not found, the host will broadcast an ARP request, and the targeted host will send back an ARP reply with its IP address and MAC address.

WT2

> Describe your routing algorithm

I implemented Distance Vector Algorithm. From the beginning, every host has an initial routing table. To evaluate the distance, I used the number of hops.

For each host, create a main thread to process packets received and send routing packets. For each device, create a sub-thread to receive packets and store them in queue. Every `0.75` seconds (approximately), the main thread send routing packets via its devices to their neighbors. It also processes packets received to update routing table.

The routing packets use an unassigned IP protocol id `200` to identify itself. For the content, the packet contains the number of elements in routing table and routing table itself.

corner cases:

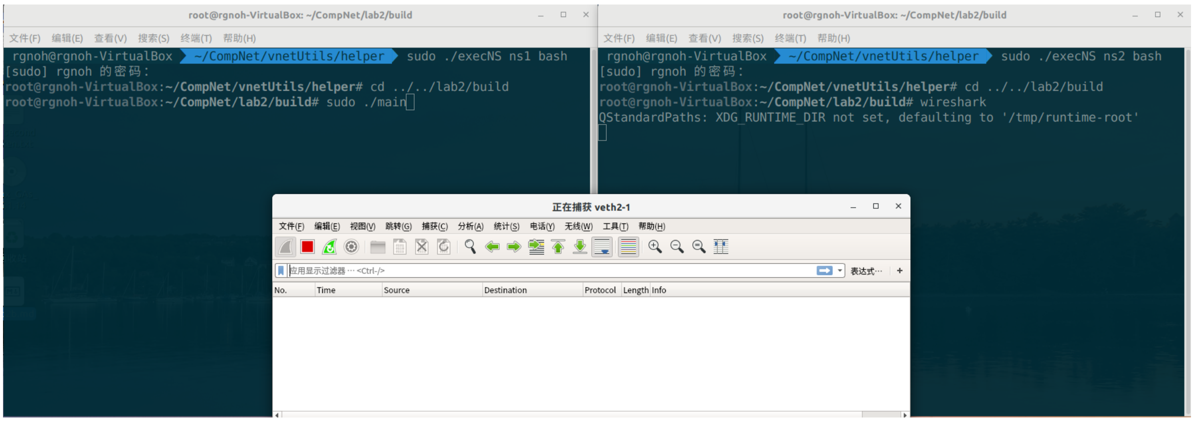
If we use naive Distance Vector Algorithm, then we will definitely encounter count-to-infinity case. To solve this, I attach a timestamp on every routing table entry and set a threshold. If the entry hasn't updated for a long time, then it is invalid. Also, I regard every table element with distance over `IP_TTL_THRESHOLD` unreachable. So the count-to-infinity case will finally converge, although the speed is not fast enough.

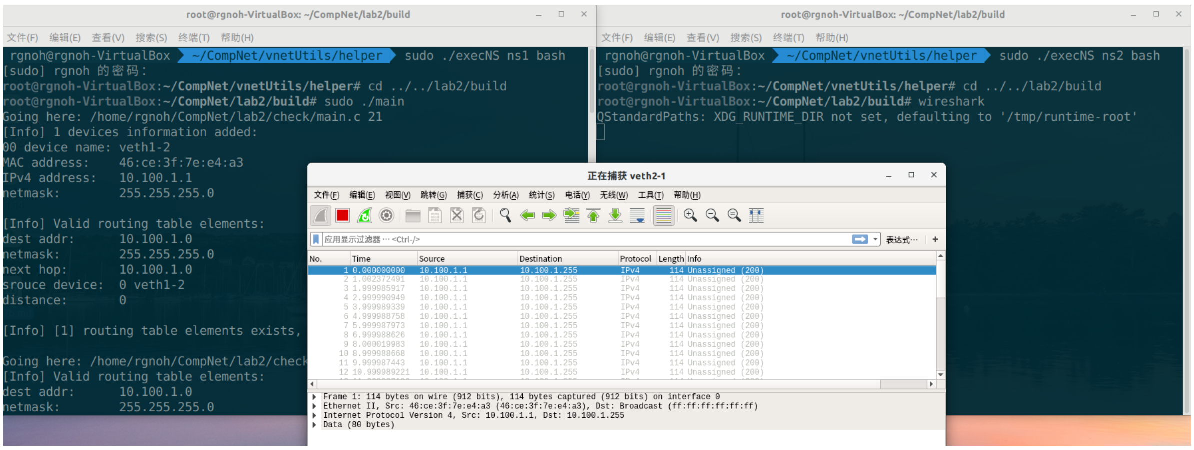
CP3

> Use tcpdump / wireshark to capture the IP packets generated by your implementation. Hexdump the content of any one packet here, and show meanings for each byte.

In this checkpoint, I use the virtual network described in `CP4`. To create it, use `example/makeVNet` with input `checkpoints/CP3_4.txt` .

Then I run `build/main` on `ns1` and run `wireshark` on `ns2`.





Here is information of the first packet, which is printed by wireshark. Its pdf file is `checkpoints/CP3result.pdf`.

```
No.     Time           Source                Destination           Protocol Length Info
      1 0.000000000    10.100.1.1            10.100.1.255          IPv4     114    Unassigned (200)
Frame 1: 114 bytes on wire (912 bits), 114 bytes captured (912 bits) on interface 0
Ethernet II, Src: 46:ce:3f:7e:e4:a3 (46:ce:3f:7e:e4:a3), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
    Destination: Broadcast (ff:ff:ff:ff:ff:ff)
    Source: 46:ce:3f:7e:e4:a3 (46:ce:3f:7e:e4:a3)
    Type: IPv4 (0x0800)
Internet Protocol Version 4, Src: 10.100.1.1, Dst: 10.100.1.255
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 100
    Identification: 0x0000 (0)
    Flags: 0x0000
    Time to live: 6
    Protocol: Unassigned (200)
    Header checksum: 0x1f88 [validation disabled]
    [Header checksum status: Unverified]
    Source: 10.100.1.1
    Destination: 10.100.1.255
Data (80 bytes)
0000  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0020  00 00 00 00 00 00 00 00 0a 64 01 00 ff ff ff 00   .........d......
0030  0a 64 01 00 00 00 00 00 00 00 00 00 00 00 00 00   .d..............
0040  80 04 88 61 00 00 00 00 6f 85 09 00 00 00 00 00   ...a....o.......
```

```
0000  ff ff ff ff ff ff 46 ce  3f 7e e4 a3 08 00 45 00   ······F· ?~····E·
0010  00 64 00 00 00 00 06 c8  1f 88 0a 64 01 01 0a 64   ·d······ ···d···d
0020  01 ff 01 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0040  00 00 00 00 00 00 00 00  00 00 0a 64 01 00 ff ff   ········ ···d····
0050  ff 00 0a 64 01 00 00 00  00 00 00 00 00 00 00 00   ···d···· ········
0060  00 00 80 04 88 61 00 00  00 00 6f 85 09 00 00 00   ·····a·· ··o·····
0070  00 00                                              ··
```

This IP packet is sent for routing, which has a protocol `200` assigned by myself to identify its type. The meaning of each byte in header is shown above. The payload is `ns1`'s routing table.

CP4

> Use vnetUtils or other tools to create a virtual network with the following topology and show that:
> (1) ns1 can discover ns4;
> (2) after we disconnect ns2 from the network, ns1 cannot discover ns4;
> (3) after we connect ns2 to the network again, ns1 can discover ns4.
>
> ns1 --- ns2 --- ns3 --- ns4

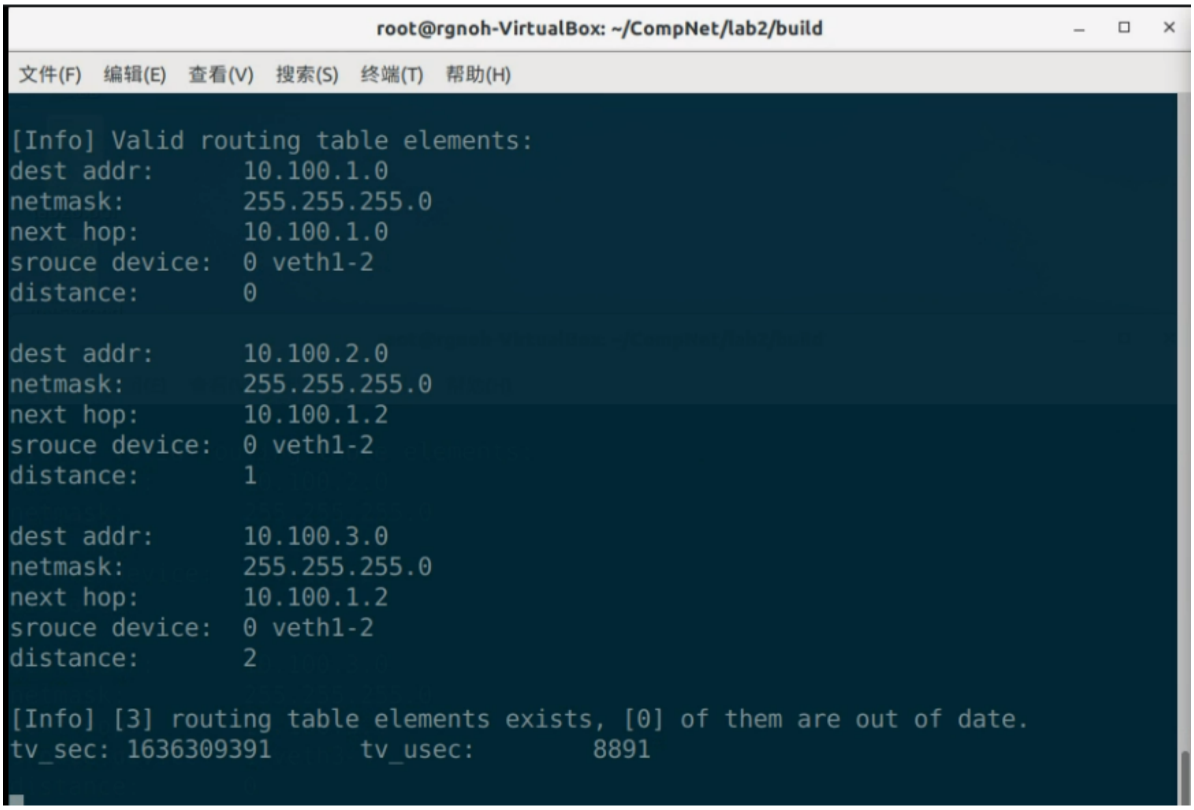
In this checkpoint, I printed the routing table every time after sending routing packets. `printRoutingTable()` in `src/ip.c` could print all valid table elements.

```c
printf("[Info] Valid routing table elements:\n");
for(i = 0; i < RoutingTableID; i++){
    if(tv.tv_sec - RoutingTable[i].ts.tv_sec <= IP_TIME_ENTRY_THRESHOLD &&
        RoutingTable[i].dis <= IP_TTL_THRESHOLD){
        temp = RoutingTable[i].dest.s_addr;
        cptr = &temp;
        printf("dest addr:\t%u.%u.%u.%u\n", cptr[0], cptr[1], cptr[2], cptr[3]);

        temp = RoutingTable[i].mask.s_addr;
        cptr = &temp;
        printf("netmask:\t%u.%u.%u.%u\n", cptr[0], cptr[1], cptr[2], cptr[3]);

        temp = RoutingTable[i].nextHop.s_addr;
        cptr = &temp;
        printf("next hop:\t%u.%u.%u.%u\n", cptr[0], cptr[1], cptr[2], cptr[3]);

        printf("srouce device:\t%d %s\n", RoutingTable[i].srcdev, rev_devs[RoutingTable[i].srcdev].name);

        printf("distance:\t%d\n", RoutingTable[i].dis);

        putchar('\n');
    }
    else cnt++;
}
printf("[Info] [%d] routing table elements exists, [%d] of them are out of date.\n", RoutingTableID, cnt);
printf("tv_sec:\t%d\ttv_usec:\t%d\n\n", tv.tv_sec, tv.tv_usec);
```

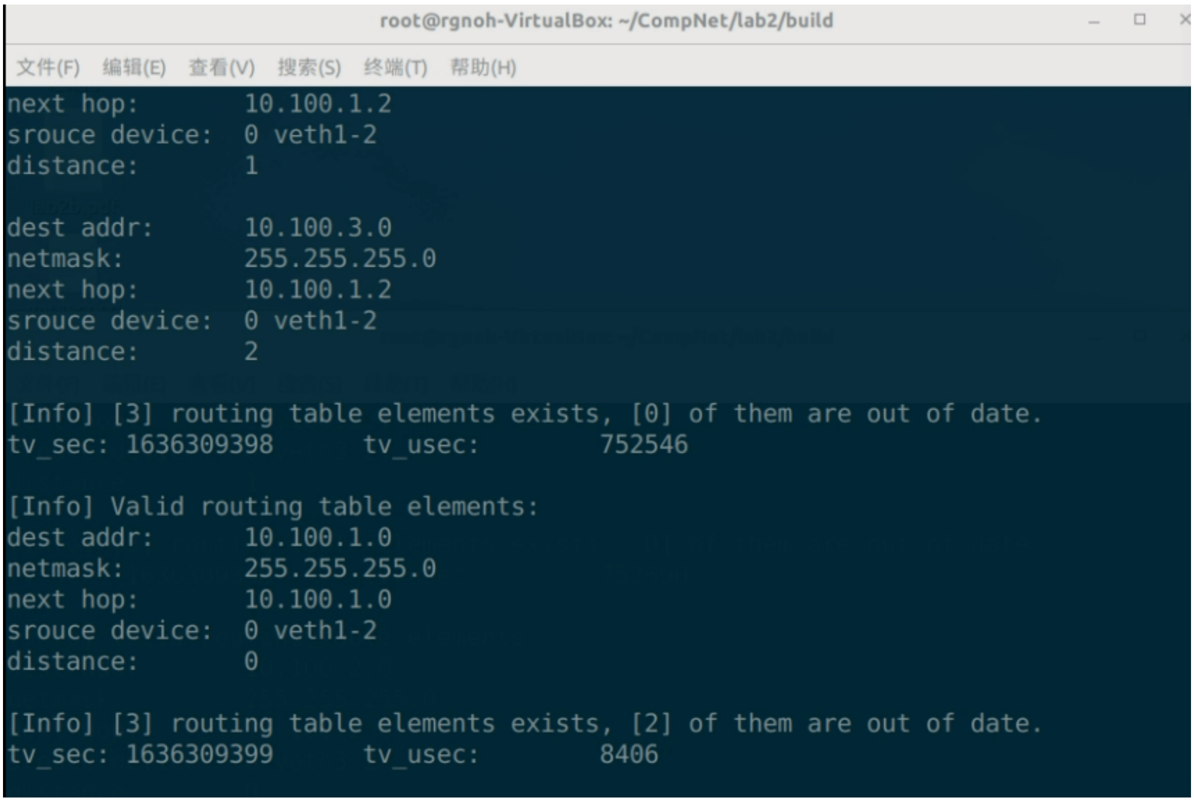
I record a video `checkpoints/lab2partBCP4.mp4` to show it more clearly.

At first, I run `build/main` on `ns1, ns2, ns3, ns4` . Initially, every host only have information about their own NIC in their routing table.

At `0:07` , `ns1` 's routing table already contained `ns4` 's information (the last one) , which proved that `ns1` had discovered `ns4` .

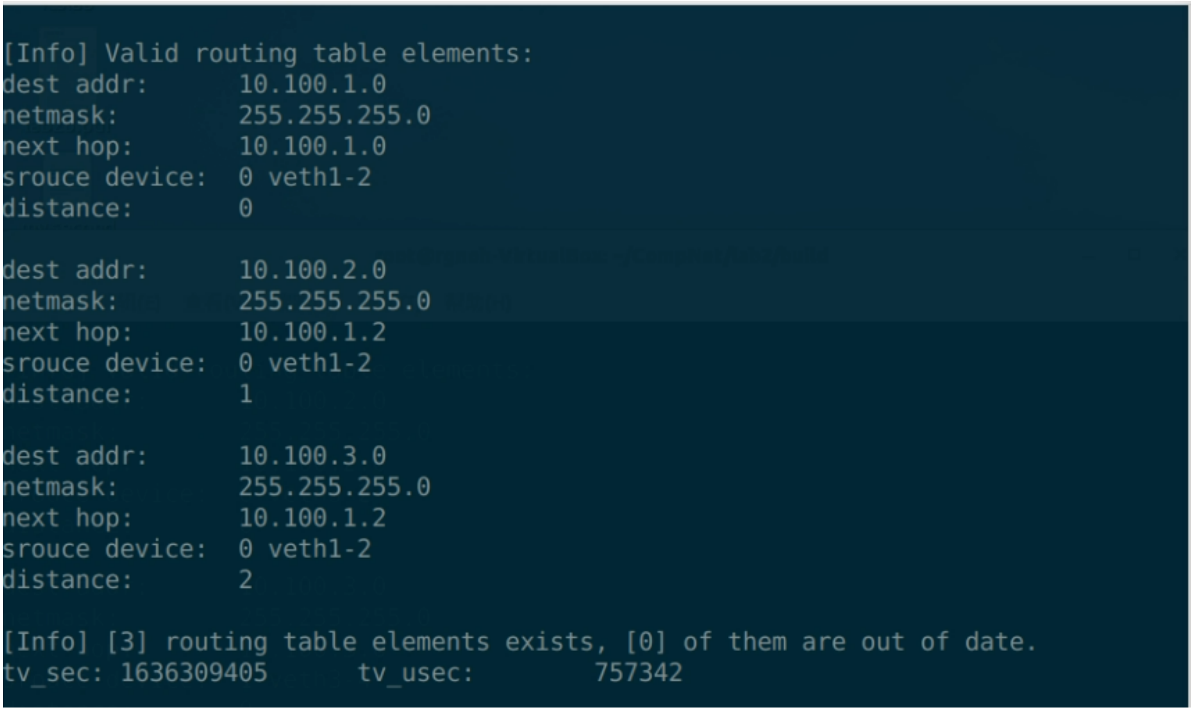


At `0:12` , I manually disconnect `ns2` . At `0:15` , `ns1` didn't have a valid entry to `ns4` .



```
root@rgnoh-VirtualBox: ~/CompNet/lab2/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
next hop:       10.100.1.2
srouce device:  0 veth1-2
distance:       1

dest addr:      10.100.3.0
netmask:        255.255.255.0
next hop:       10.100.1.2
srouce device:  0 veth1-2
distance:       2

[Info] [3] routing table elements exists, [0] of them are out of date.
tv_sec: 1636309398      tv_usec:        752546

[Info] Valid routing table elements:
dest addr:      10.100.1.0
netmask:        255.255.255.0
next hop:       10.100.1.0
srouce device:  0 veth1-2
distance:       0

[Info] [3] routing table elements exists, [2] of them are out of date.
tv_sec: 1636309399      tv_usec:        8406
```

At `0:19`, I manually connect `ns2` again. At `0:22`, `ns1`'s routing table contained `ns4`'s information again.



```
[Info] Valid routing table elements:
dest addr:      10.100.1.0
netmask:        255.255.255.0
next hop:       10.100.1.0
srouce device:  0 veth1-2
distance:       0

dest addr:      10.100.2.0
netmask:        255.255.255.0
next hop:       10.100.1.2
srouce device:  0 veth1-2
distance:       1

dest addr:      10.100.3.0
netmask:        255.255.255.0
next hop:       10.100.1.2
srouce device:  0 veth1-2
distance:       2

[Info] [3] routing table elements exists, [0] of them are out of date.
tv_sec: 1636309405      tv_usec:        757342
```

CP5

> Create a virtual network with the following topology and show the distances between each pair of hosts. The distance depends on your routing algorithm.
> After that, disconnect ns5 from the network and show the distances again.
>
> ```
> ns1 --- ns2 --- ns3 --- ns4
>          |       |
>         ns5 --- ns6
> ```

I evaluate the distance by hops and router. If two host are in the same subnetwork, then the distance will be 0.

I recorded 2 videos, `checkpoints/lab2partBCP5-1.mp4` and `checkpoints/lab2partBCP5-2.mp4`, which show the distance before disconnecting `ns5` and after disconnecting `ns5` respectively. The distance is just correct. Check the videos for details.

CP6

> Show the “longest prefix matching” rule applies in your implementation.

In this checkpoint, I construct this virtual network below:

> ns1 --- ns2 --- ns3
> |
> ns4

And here are the settings. The detail can be found in `checkpoints/CP6.txt` .

> 1 2 10.100.1.1/24 10.100.1.2/24
> 2 3 10.100.2.1/24 10.100.2.2/24
> 2 4 10.100.2.3/26 10.100.2.4/26

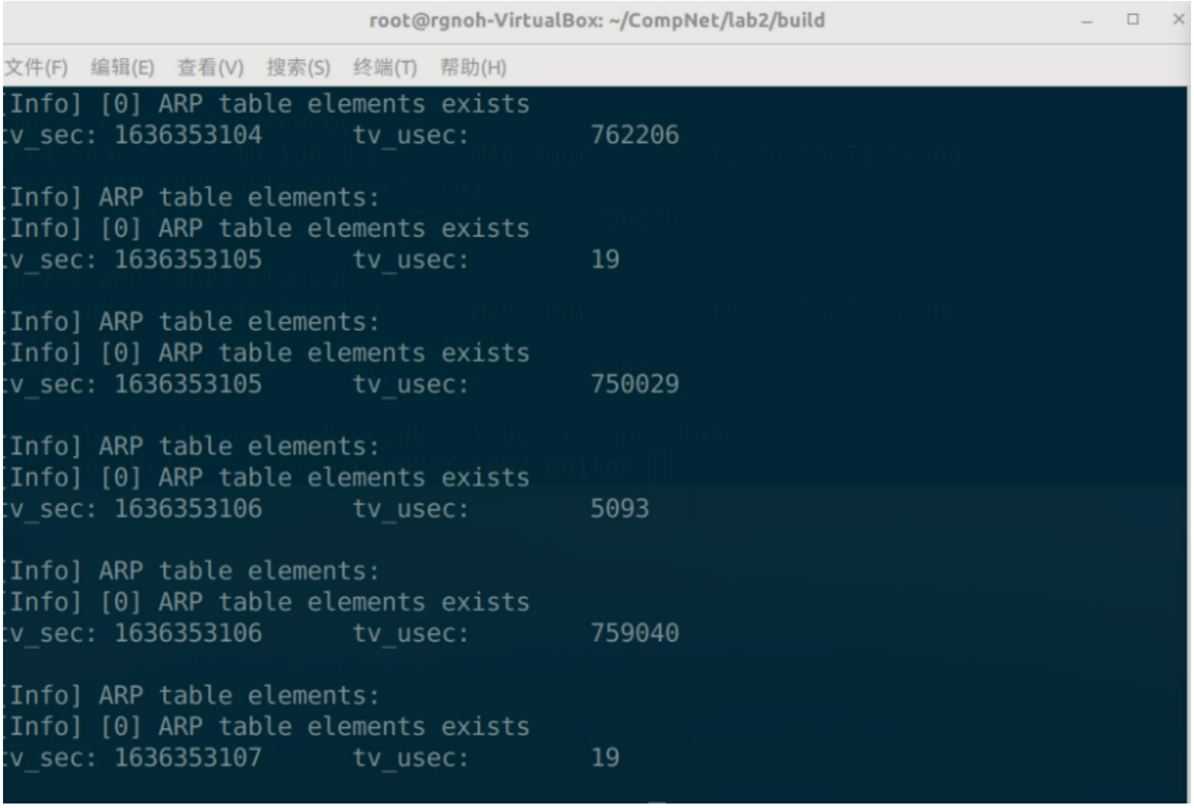
To check the "longest prefix matching" rule, I send an IP packet from `ns1` to `ns4`, with source address `10.100.1.1` and destination address `10.100.2.4` . If the longest prefix matching works, the packet will be sent to `ns4` at `ns2`, and this packet **will not arrive at** `ns3`.

I modified some functions in `check/main.c` and `packetio.c` to support sending packets from `ns1` to `ns4` . I also record a video `checkpoints/lab2partBCP6` for this checkpoint.

At `0:25`, `ns4` received the packet sent by `ns1`. (I use an unassigned protocol `201` to identify this packet.)

```
get an IP packet!
length: 58      protocol: 201
source addr:    10.100.1.1
destination addr:       10.100.2.4

payload:
T H I S
  I S
A   T E
S T   P
A C K E
T   F R
O M   N
S 1   T
O   N S
4 !

[Info] ARP table elements:
IPv4 addr:      10.100.2.3      MAC addr:       16:18:56:fe:a0:dc
IPv4 addr:      10.100.2.4      MAC addr:       46:8b:c6:ab:15:ac
[Info] [2] ARP table elements exists
tv_sec: 1636353096      tv_usec:        14974
```

And the ARP table of `ns3` is always empty, which implies that the packet never reaches `ns3`.



Part C

In my Part B implementation, when sending an IP packet and there isn't a corresponding MAC address, I just throw this IP packet and send an ARP request. Therefore, it will cause much packet loss in layer 3.

For better performance, I modified code in `ip.c`. Now when an IP packet cannot be sent due to the lack of routing table elements or ARP table elements, it will be saved in a queue for 10 seconds. It will be sent if corresponding routing table elements or ARP table elements are found.

PT4

codelist: `inc/socket.h`, `src/socket.c`, `inc/tcp.h`, `src/tcp.c` (also modified `inc/packetio.h` and `src/packetio.c`)

WT3

> Describe how you correctly handled TCP state changes.

Follow the graph in `RFC 793` and use functions in `src/socket.h` and `src/packetio.c: processTCPPacket()`. In general, they handle sending and receiving, or active change and passive change respectively.

`processTCPPacket` is the callback function of TCP packets. It uses information in TCP header to find the socket, then do operations according to the socket's current state and flags in TCP header.

Some functions in `socket.h` can change the state. They check the socket's state at the beginning of calling procedure and may change the state before it returns.

For example, in 3-way handshake, `listen()` changes the state to `LISTEN` and makes the socket be ready for `SYN` packets. `connect()` sends a `SYN` and changes the state to `SYN_SENT` in order to wait for a `SYN+ACK`. After the peer socket with state `LISTEN` receives the `SYN` packet, `processTCPPacket` push it in listen queue. `accept()` pop the listen queue (if not empty) , changes the state to `SYN_RECV` and send a `SYN+ACK` . After `SYN+ACK` arrives, `processTCPPacket` changes the state from `SYN_SENT` to `ESTABLISHED` and sends an `ACK` (meanwhile, `connect()` detects the state change and return). After this `ACK` arrives, `processTCPPacket` changes the state from `SYN_RECV` to `ESTABLISHED` . Meanwhile, `accept()` detects the state change and return.

CP7

> Use tcpdump / wireshark to capture the TCP packets generated by your implementation. Hexdump the content of any one packet here, and show meanings for each byte in the TCP header.

Since this checkpoint only cares about the content of TCP packet, I just add a TCP header as IP payload and use `sendIPPacket` to send, instead of using APIs in `socket.h`.

In this checkpoint, I construct this virtual network below:

> ns1 --- ns2

I send a packet from `ns1` (10.100.1.1) to `ns2` (10.100.1.2) with source port `1234` and destination port `4321`. Details can be found in `src/packetio.c`.

```
2054 0.797015798   a2:cb:7e:8e:1f:aa   96:46:93:55:ed:12   ARP    42 10.100.1.2 is at a2:cb:7e:8e:1f:aa
2055 0.797097355   10.100.1.1          10.100.1.2          TCP   112 1234 → 4321 [ACK] Seq=1 Ack=1 Win=32768 Len=58
2056 0.809494141   a2:cb:7e:8e:1f:aa   96:46:93:55:ed:12   ARP    42 10.100.1.2 is at a2:cb:7e:8e:1f:aa
```

Here is information of this packet, which is printed by wireshark. The pdf file is `checkpoints/CP7result.pdf`.

```
No.     Time           Source                Destination           Protocol Length Info
   2055 0.797097355    10.100.1.1            10.100.1.2            TCP      112    1234 → 4321 [ACK] Seq=1 Ack=1 Win=32768
Len=58
Frame 2055: 112 bytes on wire (896 bits), 112 bytes captured (896 bits) on interface 0
Ethernet II, Src: 96:46:93:55:ed:12 (96:46:93:55:ed:12), Dst: a2:cb:7e:8e:1f:aa (a2:cb:7e:8e:1f:aa)
Internet Protocol Version 4, Src: 10.100.1.1, Dst: 10.100.1.2
Transmission Control Protocol, Src Port: 1234, Dst Port: 4321, Seq: 1, Ack: 1, Len: 58
    Source Port: 1234
    Destination Port: 4321
    [Stream index: 0]
    [TCP Segment Len: 58]
    Sequence number: 1    (relative sequence number)
    [Next sequence number: 59    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    0101 .... = Header Length: 20 bytes (5)
    Flags: 0x010 (ACK)
    Window size value: 32768
    [Calculated window size: 32768]
    [Window size scaling factor: -1 (unknown)]
    Checksum: 0xffff [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    [SEQ/ACK analysis]
    [Timestamps]
    TCP payload (58 bytes)
Data (58 bytes)
0000  54 48 49 53 20 49 53 20 41 20 54 45 53 54 20 50   THIS IS A TEST P
0010  41 43 4b 45 54 21 31 32 33 34 35 36 37 38 39 30   ACKET!1234567890
0020  61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70   abcdefghijklmnop
0030  71 72 73 74 75 76 77 78 79 7a                     qrstuvwxyz
```

The meaning of each byte in header is shown above.

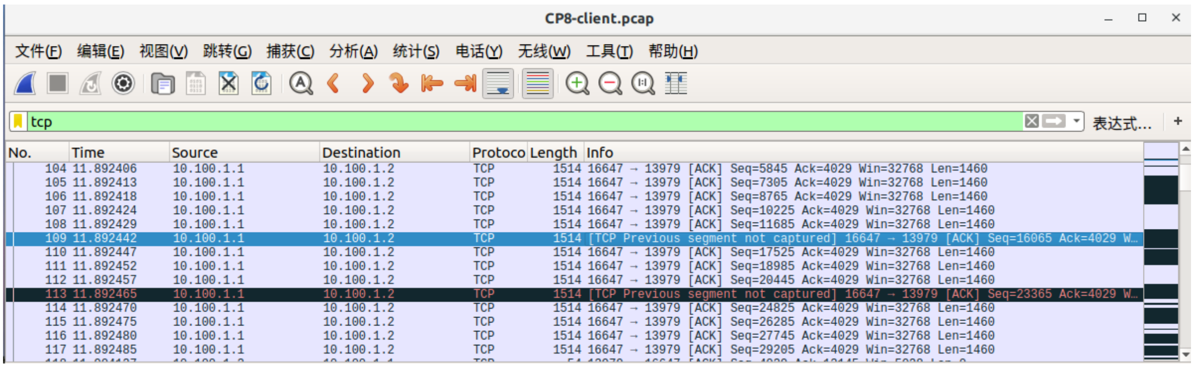
CP8

> Show your implementaion provides reliable delivery (i.e., it can detect packet loss and retransmit the lost packets). You are encouraged to attach a screenshot of the wireshark packet trace here. Check section 4.2 to see how to emulate a lossy link.

In this checkpoint, I constructed the same virtual network in CP7. To emulate a lossy link, I typed the command `tc qdisc add dev veth1-2 root netem loss 20%` at `ns1` . I ran `build/clientCP8` on `ns1` to send `20 * 1460` bytes to `build/serverCP8` on `ns2`. There source codes are `check/client.c` and `check/serverCP8.c` respectively.

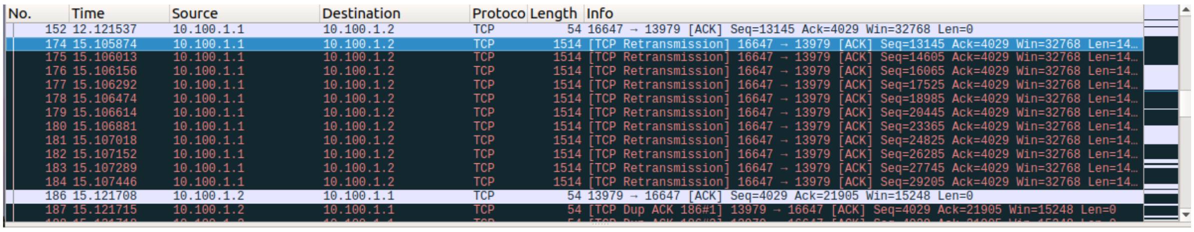
Since this checkpoint only cares about loss detection and retransmission , for simplicity, I didn't implement `close()` here.

Traces can be found in `checkpoints/CP8-client.pcap` and `checkpoints/CP8-server.pcap`. Here are screenshots from `checkpoints/CP8-client.pcap`.



No.	Time	Source	Destination	Protoco	Length	Info
104	11.892406	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=5845 Ack=4029 Win=32768 Len=1460
105	11.892413	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=7305 Ack=4029 Win=32768 Len=1460
106	11.892418	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=8765 Ack=4029 Win=32768 Len=1460
107	11.892424	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=10225 Ack=4029 Win=32768 Len=1460
108	11.892429	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=11685 Ack=4029 Win=32768 Len=1460
109	11.892442	10.100.1.1	10.100.1.2	TCP	1514	[TCP Previous segment not captured] 16647 → 13979 [ACK] Seq=16065 Ack=4029 W…
110	11.892447	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=17525 Ack=4029 Win=32768 Len=1460
111	11.892452	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=18985 Ack=4029 Win=32768 Len=1460
112	11.892457	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=20445 Ack=4029 Win=32768 Len=1460
113	11.892465	10.100.1.1	10.100.1.2	TCP	1514	[TCP Previous segment not captured] 16647 → 13979 [ACK] Seq=23365 Ack=4029 W…
114	11.892470	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=24825 Ack=4029 Win=32768 Len=1460
115	11.892475	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=26285 Ack=4029 Win=32768 Len=1460
116	11.892480	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=27745 Ack=4029 Win=32768 Len=1460
117	11.892485	10.100.1.1	10.100.1.2	TCP	1514	16647 → 13979 [ACK] Seq=29205 Ack=4029 Win=32768 Len=1460

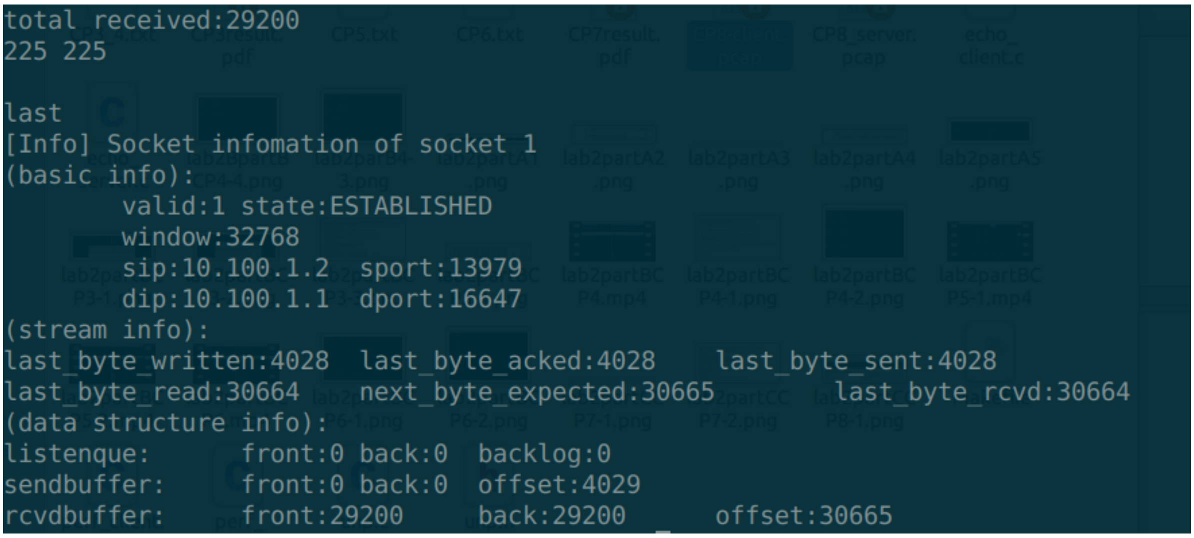
It can be found bytes between `13145` and `16064` were lost at `No.109`.



No.	Time	Source	Destination	Protoco	Length	Info
152	12.121537	10.100.1.1	10.100.1.2	TCP	54	16647 → 13979 [ACK] Seq=13145 Ack=4029 Win=32768 Len=0
174	15.105874	10.100.1.1	10.100.1.2	TCP	1514	[TCP Retransmission] 16647 → 13979 [ACK] Seq=13145 Ack=4029 Win=32768 Len=14…
175	15.106013	10.100.1.1	10.100.1.2	TCP	1514	[TCP Retransmission] 16647 → 13979 [ACK] Seq=14605 Ack=4029 Win=32768 Len=14…
176	15.106156	10.100.1.1	10.100.1.2	TCP	1514	[TCP Retransmission] 16647 → 13979 [ACK] Seq=16065 Ack=4029 Win=32768 Len=14…
177	15.106292	10.100.1.1	10.100.1.2	TCP	1514	[TCP Retransmission] 16647 → 13979 [ACK] Seq=17525 Ack=4029 Win=32768 Len=14…
178	15.106474	10.100.1.1	10.100.1.2	TCP	1514	[TCP Retransmission] 16647 → 13979 [ACK] Seq=18985 Ack=4029 Win=32768 Len=14…
179	15.106614	10.100.1.1	10.100.1.2	TCP	1514	[TCP Retransmission] 16647 → 13979 [ACK] Seq=20445 Ack=4029 Win=32768 Len=14…
180	15.106881	10.100.1.1	10.100.1.2	TCP	1514	[TCP Retransmission] 16647 → 13979 [ACK] Seq=23365 Ack=4029 Win=32768 Len=14…
181	15.107018	10.100.1.1	10.100.1.2	TCP	1514	[TCP Retransmission] 16647 → 13979 [ACK] Seq=24825 Ack=4029 Win=32768 Len=14…
182	15.107152	10.100.1.1	10.100.1.2	TCP	1514	[TCP Retransmission] 16647 → 13979 [ACK] Seq=26285 Ack=4029 Win=32768 Len=14…
183	15.107289	10.100.1.1	10.100.1.2	TCP	1514	[TCP Retransmission] 16647 → 13979 [ACK] Seq=27745 Ack=4029 Win=32768 Len=14…
184	15.107446	10.100.1.1	10.100.1.2	TCP	1514	[TCP Retransmission] 16647 → 13979 [ACK] Seq=29205 Ack=4029 Win=32768 Len=14…
186	15.121708	10.100.1.2	10.100.1.1	TCP	54	13979 → 16647 [ACK] Seq=4029 Ack=21905 Win=15248 Len=0
187	15.121715	10.100.1.2	10.100.1.1	TCP	54	[TCP Dup ACK 186#1] 13979 → 16647 [ACK] Seq=4029 Ack=21905 Win=15248 Len=0

The protocol stack detected the loss at `No.174` and did some retransmission.

In my implementation then, the protocol stack will check EVERY unacked packet's timestamp and retransmit, so there are many retransmissions in the picture above.



```
total received:29200
225 225

last
[Info] Socket infomation of socket 1
(basic info):
        valid:1 state:ESTABLISHED
        window:32768
        sip:10.100.1.2  sport:13979
        dip:10.100.1.1  dport:16647
(stream info):
last_byte_written:4028  last_byte_acked:4028    last_byte_sent:4028
last_byte_read:30664    next_byte_expected:30665        last_byte_rcvd:30664
(data structure info):
listenque:      front:0 back:0  backlog:0
sendbuffer:     front:0 back:0  offset:4029
rcvdbuffer:     front:29200     back:29200      offset:30665
```

Shown in the output of `build/serverCP8`, all `20*1460 = 29200` bytes were received and read by `build/serverCP8` .

CP9

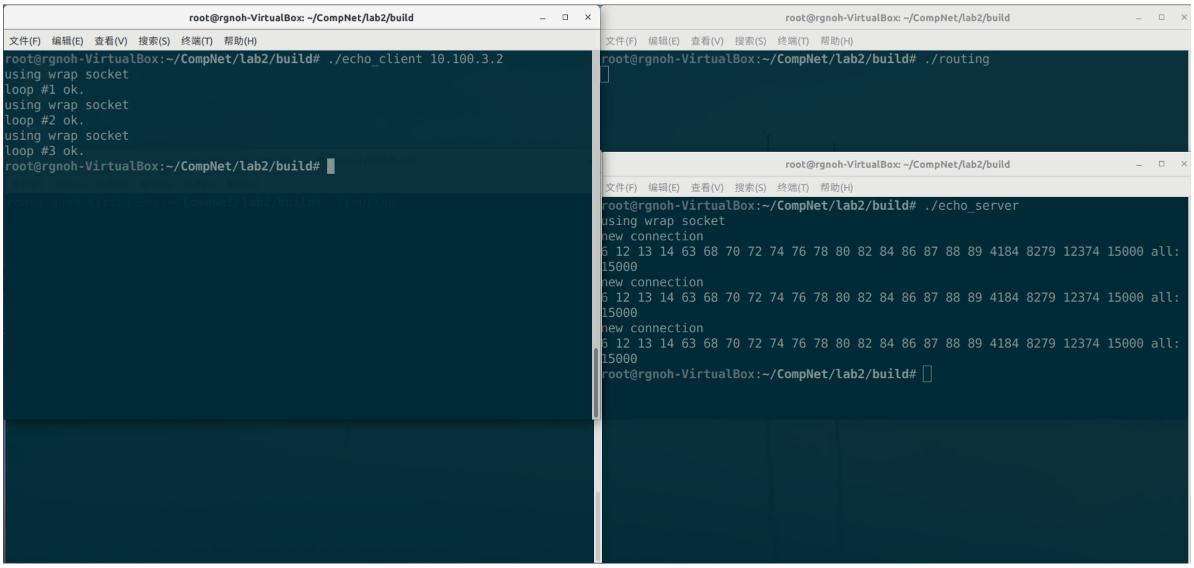
> Create a virtual network with the following topology and run echo_server at ns4 and echo_client at ns1. The source code is under the folder called "checkpoints". Paste the output of them here. Note that you are not allowed to make changes to the source code (i.e., the *.h and *.c files). Check out section 4 to see how to hijack the library functions such as listen().

> ns1 --- ns2 --- ns3 --- ns4

Here are the settings of the virtual network. It can be found in `checkpoints/CP9.txt`.

> 1 2 10.100.1.1/24 10.100.1.2/24
> 2 3 10.100.2.1/24 10.100.2.2/24
> 3 4 10.100.3.1/24 10.100.3.2/24

Run `echo_server` at `ns4` and `echo_client` at `ns1`, and run `build/routing` on `ns2, ns3` for routing. The results are shown below.



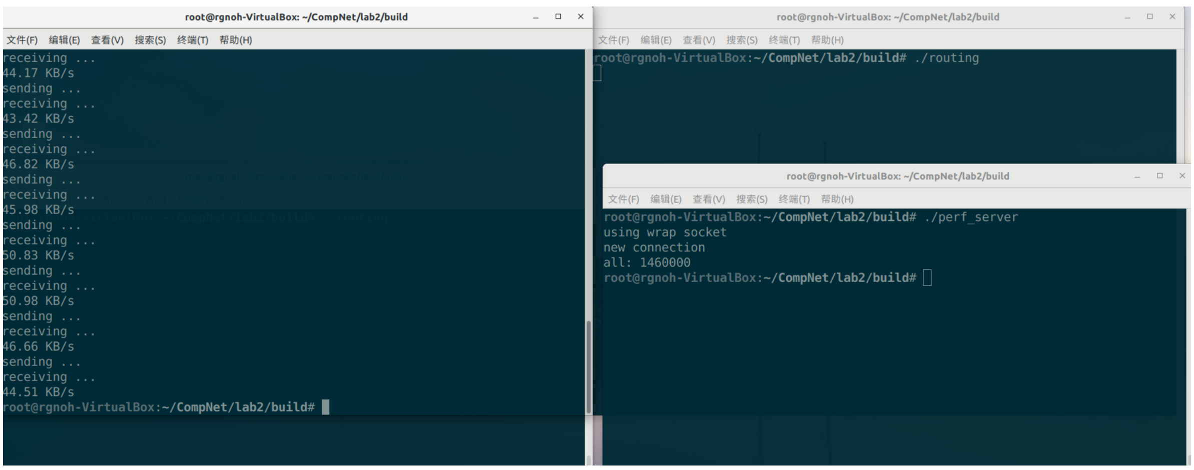
In `CP9` and `CP10`, I hijacked lib functions (see `CMakeLists.txt`). To demonstrate that, in my implementation of `__wrap_socket()`, it will print `using wrap socket` at the beginning.

CP10

> Create a virtual network with the following topology and run perf_server at ns4 and perf_client at ns1. Paste the output of them here. Again, you are not allowed to make changes to the source code.
>
> ns1 --- ns2 --- ns3 --- ns4

The virtual network is the same as CP9's.

Run `perf_server` at `ns4` and `perf_client` at `ns1`, and run `build/routing` on `ns2, ns3` for routing. The results are shown below.



Note that when `perf_client` return, it doesn't call `close()`. So `perf_server` should close at `FIN_WAIT2` after a timeout.

(It's poorly slow because of my poor implementation of synchronization)