

# miniTCP

---

A mini network protocol stack built upon libpcap.

## Usage

---

To make the program, please first enter the directory of miniTCP and execute the following commands.

```
mkdir build  
cd build  
sudo cmake ..  
sudo make  
cd ..
```

## Code Lists

---

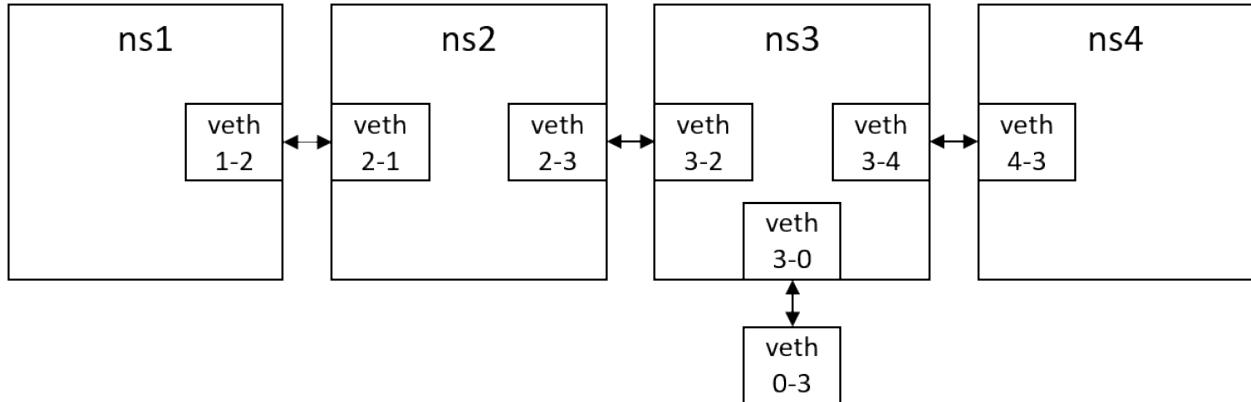
1. src/application : some application written for self evaluation and checkpoint.
2. src/common: some useful gadgets in implementing the network stack
3. src/ethernet: the code of ethernet layer
4. src/network: the code of network layer
5. src/transport : the code of transport layer

## Part A

---

### Note:

The network topology we use in checkpoint 1 and checkpoint 2 is exactly **the same as the example** given in the vnetUtils.



## Checkpoint 1: Show that your implementation can detect network interfaces on the host.

To run the demo, first use the following command to activate the NS environment.

```
bash script/install_ckpt1.sh
```

Run following command to enter ns#3.

```
bash script/enter_ns3.sh
```

To go back to the **build directory in miniTCP** after executing the bash script , please run the following command in the terminal.

```
cd ../../..../build
```

To see the demo, please enter the build director in the root of miniTCP and run the following command.

```
./link_app
```

You can see the result of devices like this, which implies that the ethernet kernel successfully find all devices.

```
cyclodzzz@ubuntu:~$ cd CodeProject/miniTCP
cyclodzzz@ubuntu:~/CodeProject/miniTCP$ ls
3rdparty CMakeLists.txt LICENSE README.md src          typescript
build   demo   main.cc  script  test_gtest.cc
cyclodzzz@ubuntu:~/CodeProject/miniTCP$ bash script/enter_ns3.sh
[sudo] password for cyclodzzz:
root@ubuntu:/home/cyclodzzz/CodeProject/miniTCP/3rdparty/vnetUtils/helper# cd ../../build
root@ubuntu:/home/cyclodzzz/CodeProject/miniTCP/build# ./link_app
[2021-10-25 01:33:26.215033: Info /home/cyclodzzz/CodeProject/miniTCP/src/ethernet/device_impl.cc: 22] EthernetDevice veth3-2 registered successfully
[2021-10-25 01:33:26.230667: Info /home/cyclodzzz/CodeProject/miniTCP/src/ethernet/device_impl.cc: 22] EthernetDevice veth3-4 registered successfully
[2021-10-25 01:33:26.246915: Info /home/cyclodzzz/CodeProject/miniTCP/src/ethernet/device_impl.cc: 22] EthernetDevice veth3-0 registered successfully
```

## Checkpoint 2: Show that your implementation can capture frames from a device and inject frames to a device using libpcap.

To run the demo, use the following command to activate the NS environment.

```
bash script/install_ckpt2.sh
```

Run following command to enter ns#1 in one terminal (namely, terminal#1).

```
bash script/enter_ns1.sh
```

Run following command on another terminal (namely, terminal#2) to enter ns#2.

```
bash script/etner_ns2.sh
```

To show that we can inject frames to a device as well as capture frames from a device, our demo is sending a greeting message from one terminal to another.

For example, If you want to send a greeting message from veth 1-2 (in terminal#1) to veth 2-1 (in terminal#2).

First start a listening application on terminal#2 by running

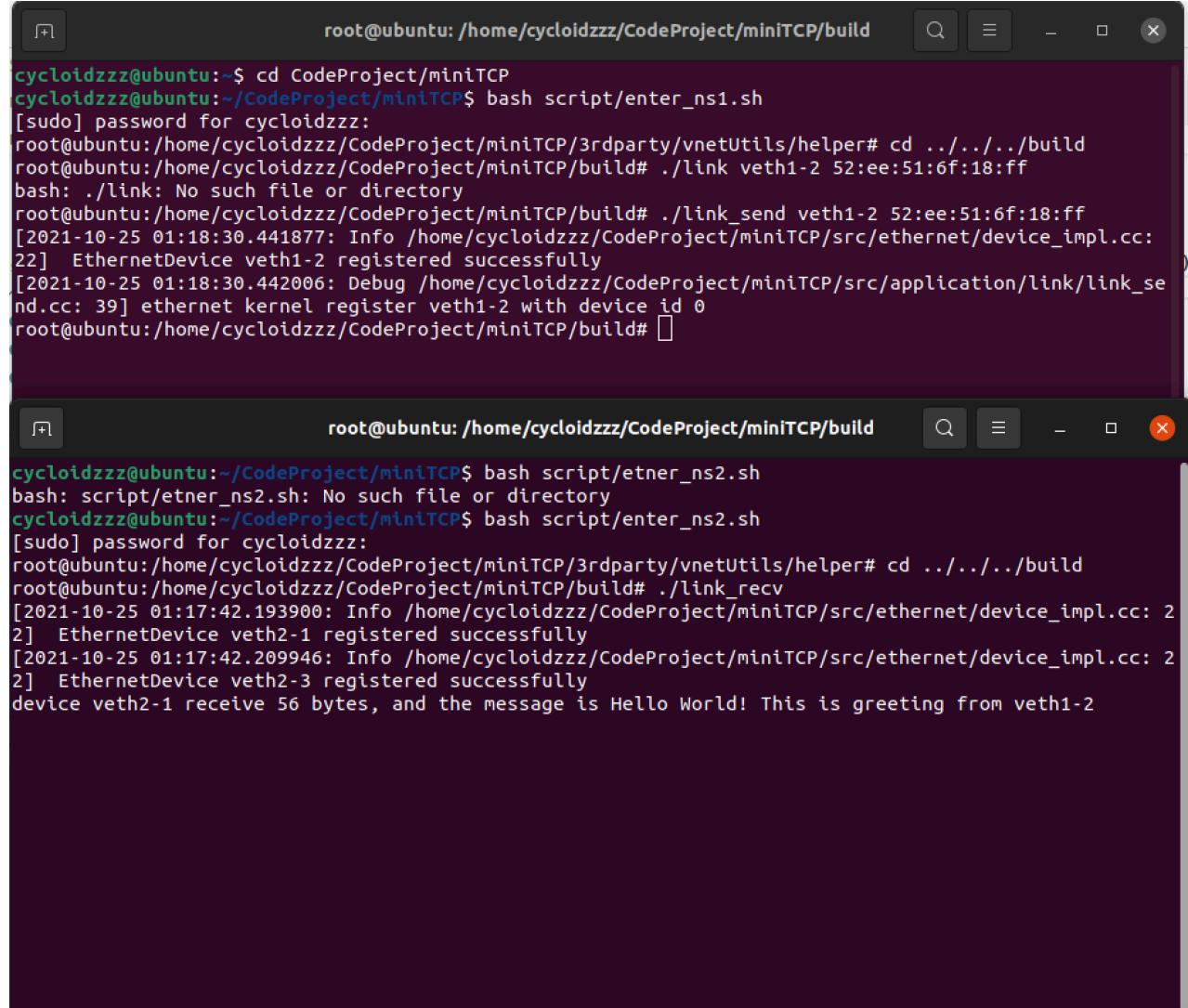
```
./link_recv
```

Suppose the mac address of destination (veth 2-1) is 52:ee:51:6f:18:ff, you can send the message by running on terminal#1.

```
./link_send veth1-2 52:ee:51:6f:18:ff
```

And the device veth 2-1 will receive this message and use a callback function to print out the message received.

The result of the checkpoint 2.



The image shows two terminal windows side-by-side. Both terminals are running on an Ubuntu system and are located in the /home/cycloidzzz/CodeProject/miniTCP/build directory. The top terminal window shows the command ./link\_send veth1-2 52:ee:51:6f:18:ff being run, followed by its output. The output includes log messages from the kernel and application code indicating the registration of the EthernetDevice and the receipt of a message from veth1-2. The bottom terminal window shows the command ./link\_recv being run, followed by its output. The output shows a log message indicating the receipt of a message from veth2-1, which contains the text "Hello World! This is greeting from veth1-2".

```
root@ubuntu:~/CodeProject/miniTCP$ bash script/enter_ns1.sh
[sudo] password for cycloidzzz:
root@ubuntu:/home/cycloidzzz/CodeProject/miniTCP/3rdparty/vnetUtils/helper# cd ../../..
root@ubuntu:/home/cycloidzzz/CodeProject/miniTCP/build# ./link_send veth1-2 52:ee:51:6f:18:ff
bash: ./link: No such file or directory
root@ubuntu:/home/cycloidzzz/CodeProject/miniTCP/build# ./link_send veth1-2 52:ee:51:6f:18:ff
[2021-10-25 01:18:30.441877: Info /home/cycloidzzz/CodeProject/miniTCP/src/ethernet/device_impl.cc: 22] EthernetDevice veth1-2 registered successfully
[2021-10-25 01:18:30.442006: Debug /home/cycloidzzz/CodeProject/miniTCP/src/application/link/link_send.cc: 39] ethernet kernel register veth1-2 with device id 0
root@ubuntu:/home/cycloidzzz/CodeProject/miniTCP/build# 
```

```
root@ubuntu:~/CodeProject/miniTCP$ bash script/etner_ns2.sh
bash: script/etner_ns2.sh: No such file or directory
root@ubuntu:~/CodeProject/miniTCP$ bash script/enter_ns2.sh
[sudo] password for cycloidzzz:
root@ubuntu:/home/cycloidzzz/CodeProject/miniTCP/3rdparty/vnetUtils/helper# cd ../../..
root@ubuntu:/home/cycloidzzz/CodeProject/miniTCP/build# ./link_recv
[2021-10-25 01:17:42.193900: Info /home/cycloidzzz/CodeProject/miniTCP/src/ethernet/device_impl.cc: 22] EthernetDevice veth2-1 registered successfully
[2021-10-25 01:17:42.209946: Info /home/cycloidzzz/CodeProject/miniTCP/src/ethernet/device_impl.cc: 22] EthernetDevice veth2-3 registered successfully
device veth2-1 receive 56 bytes, and the message is Hello World! This is greeting from veth1-2
```

## Part B

## Writing task1: Explain how you addressed this problem when implementing IP protocol.

I implement an arp protocol above the link layer [in the files ethernet/arp.h and ethernet/arp\_impl.h]. If a device needs to find out the corresponding MAC address of an IP address when trying to send the IP datagram, it can lookup the MAC address from the arp table.

To be more specific, the arp table on each host (terminals) is a singleton, the devices belong to this host would send out an ARP request as an announcement every 8 seconds. If a device receives an ARP request, it would first match whether the receiver IP matches its own IP address, if it matches or the receiver IP is a broadcast IP, it would send back an ARP reply. An ARP entry will be removed from the ARP table if it is not updated within 48 seconds.

## Writing task2: Describe your routing algorithm.

I implement RIP protocol above the link layer, which is a well-known distance vector algorithm.

To be more specific, every host would maintain a local routing table, recording the destination, netmask of the destination, the corresponding nexthop ip and the metric (also known as the distance). We use the **hop number** from this host to the destination as the metric.

A **timer will be maintained for each routing entry**. If a routing entry isn't updated by the host's neighbors within 18 seconds, the distance of the entry will be marked as unreachable by setting it as 16 (the unreachable threshold) and the status of the entry will be marked as GARBAGE, a garbage collection will be set for this entry. If this entry is still not updated by the corresponding neighbor within 12 seconds, this entry will be removed from the routing entry immediately. By using timeout, we can automatically adjust our routing table when the network condition changes.

The hosts in the network will **send its own distance vector to its neighbors every 3 seconds**. The distance vector is a serialized object. It consists of the sender ip (4 bytes), number of entries (4 bytes) and multiple distance vector entries.

A distance vector entry is defined as the following C struct. It mainly maintains the destination ip, the netmask and the distance from the sender host to the destination network.

```
/* src/network/routing_impl.h */
#pragma pack(4)
struct DVEntry {
    ip_t dest;
    ip_t netmask;
    int distance;
};
#pragma pack()
```

When the host receives a distance vector from its neighbour, the host first deserialize the distance vector, figure out the sender ip (the nexthop ip), the number of entries that the distance vector contains and the content of the distance vector. Then the host updates its own routing table according to the information it receives.

To accelerate the convergence of the algorithm, the **poison reversing** technique is also applied to the construction of the distance vector. Upon constructing the distance vector for a specific neighbor, we will mark the distance of the entries that learnt from this neighbor as unreachable.

Since I haven't finish the transport layer yet, the only layer I can relay on to send the distance vector would be the **link layer**, which means the current algorithm is **sending/receiving raw ethernet frame**, which seems to be quiet strange. I will use a formal RIP protocol packet to construct my RIP protocol when UDP is built.

## Checkpoint3

### 1. Topology and IP Configuration

We use two NSs connected with each other in checkpoint3, namely ns1 and ns2.

The IP configuration of each ns is shown in the following table.

NS	
ns1	veth1-2 (10.102.1.1)
ns2	veth2-1 (10.102.1.2)

### 2. Usage

To check out checkpoint3, please execute the following command in the directory of miniTCP in the terminal.

```
python3 script/check3.py -install
```

In the same time, please run the following command on a new terminal to capture IP packet on ns1.

```
bash script/capture_ns.sh 1
```

After checking out the checkpoint3, please execute the following command to delete the configuration generated by vnetUtils.

```
python3 script/check3.py -delete
```

### 3.Result

After several seconds, send a message from ns1 to ns2 by entering the following command in the terminal of ns1.

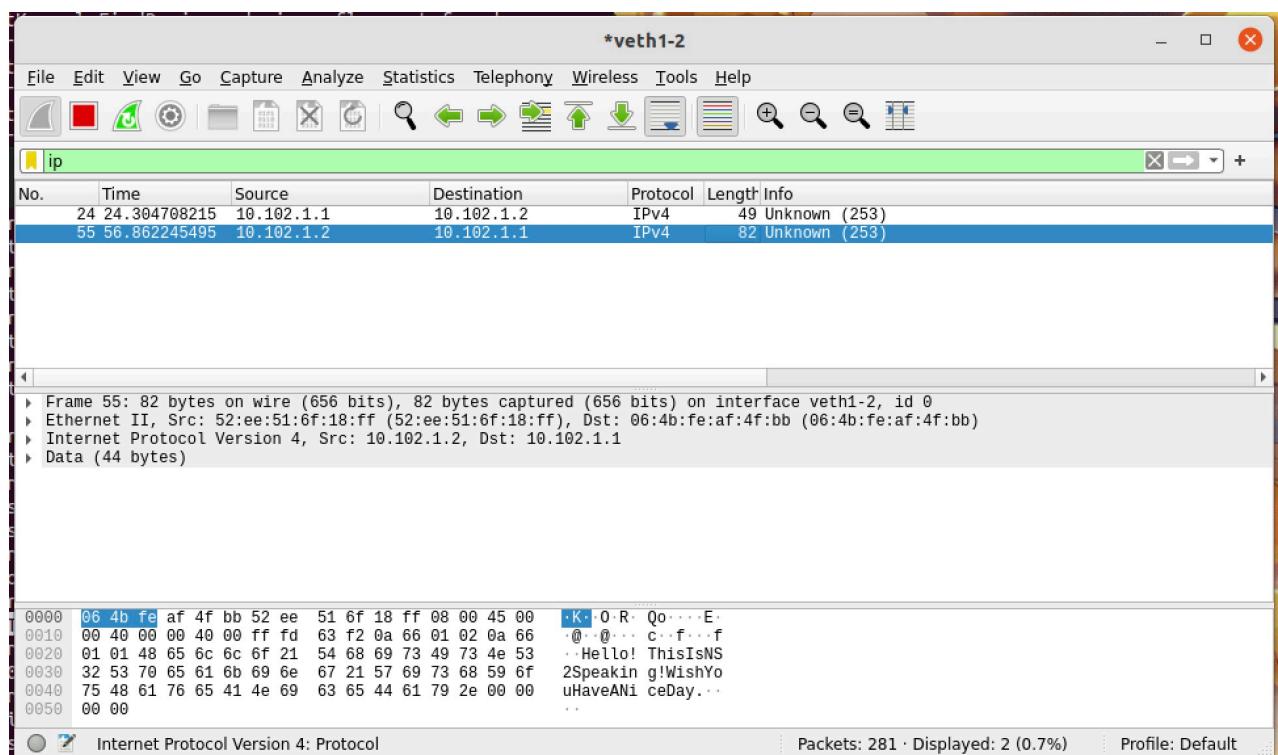
```
send 10.102.1.1 10.102.1.2 HelloWorld!
```

The terminal of ns1 will receive the message and print out the message.

Send a message from ns2 to ns1 by entering the following command in the terminal of ns2.

```
send 10.102.1.2 10.102.1.1 Hello!ThisIsNS2Speaking!WishYouHaveANiceDay.
```

Then we can capture 2 message by Wireshark.



```
52 ee 51 6f 18 ff 06 4b fe af 4f bb 08 00 45 00 R Qo...K...O...E...
00 1f 00 00 40 00 ff fd 64 13 0a 66 01 01 0a 66 ...@...d...f...f
01 02 48 65 6c 6c 6f 57 6f 72 6c 64 21 00 00 00 ...HelloWorld!...
00
```

To hexdump the first IP packet. The first 14 bytes are Ethernet II headers and we can omit them.

The first byte of the packet is 0x45, and the most significant 4 bits of the first byte is 0x0100, it means it is using ipv4, the remaining 4 bits of the first byte is 0x0101, it means the length of the ip header is  $5 * 4 = 20$ .

The next byte of the packet header is 0x00.

The third and the fourth bytes consists of the total length of the ip packet.

The fifth and the sixth bytes are 0x0000, which consists of the ip identification of the packet.

The seventh and the eighth bytes are 0x4000, the seventh bytes is 0x40, which means this packet is not a fragmented ip packet.

The ninth byte is 0xff, which means the time to live of the packet.

The tenth byte is 0xfd, which means the protocol running on the ip is 253 (an arbitrary number I use to send IP packets).

The eleventh and the twelfth bytes are 0x6413, which is the checksum of the IP header.

The thirteenth to the sixteenth bytes are 0x0a660101, which means the source IP address is 10.102.1.1.

The seventeenth to the twentieth bytes are 0x0a660102, which means the destination IP address is 10.102.1.2.

The rest bytes are the message, which means "HelloWorld!"

```
06 4b fe af 4f bb 52 ee 51 6f 18 ff 08 00 45 00 .K..O.R.Qo...E...
00 40 00 00 40 00 ff fd 63 f2 0a 66 01 02 0a 66 ...@...@...c...f...f
01 01 48 65 6c 6c 6f 21 54 68 69 73 49 73 4e 53 ...Hello! ThisIsNS
32 53 70 65 61 6b 69 6e 67 21 57 69 73 68 59 6f 2Speaking!WishYo
75 48 61 76 65 41 4e 69 63 65 44 61 79 2e 00 00 uHaveANi ceDay...
00 00 ...
```

To hexdump the first IP packet. The first 14 bytes are Ethernet II headers and we can omit them.

The first byte of the packet is 0x45, and the most significant 4 bits of the first byte is 0x0100, it means it is using ipv4, the remaining 4 bits of the first byte is 0x0101, it means the length of the ip header is  $5 * 4 = 20$ .

The next byte of the packet header is 0x00.

The third and the fourth bytes are 0x0040, which consists of the total length of the ip packet.

The fifth and the sixth bytes are 0x0000, which consists of the ip identification of the packet.

The seventh and the eighth bytes are 0x4000, the seventh bytes is 0x40, which means this packet is not a fragmented ip packet.

The ninth byte is 0xff, which means the time to live of the packet.

The tenth byte is 0xfd, which means the protocol running on the ip is 253 (an arbitrary number I use to send IP packets).

The eleventh and the twelfth bytes are 0x63f2, which is the checksum of the IP header.

The thirteenth to the sixteenth bytes are 0x0a660102, which means the source IP address is 10.102.1.2.

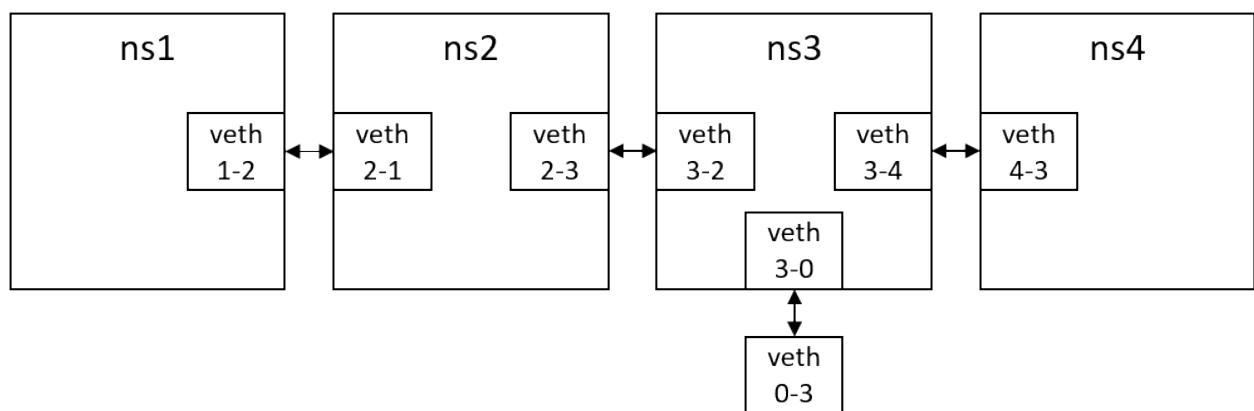
The seventeenth to the twentieth bytes are 0x0a660101, which means the destination IP address is 10.102.1.1.

The rest bytes are the message, which means "Hello!ThisIsNS2Speaking!WishYouHaveANiceDay."

## Checkpoint4

### 1. Topology and IP Configuration

The topology we use in this checkpoint is exactly the same as the example given in the vnetUils.



The IP address configuration is shown in the following table.

NS			
NS1	veth1-2 (10.100.1.1)		
NS2	veth2-1 (10.100.1.2)	veth2-3 (10.100.2.1)	
NS3	veth3-2 (10.100.2.2)	veth3-4 (10.100.3.1)	veth3-0 (10.100.4.2)
NS4	veth4-3 (10.100.3.2)		

## 2. Usage

To check out checkpoint4, please enter the miniTCP directory in the terminal and execute the following command

```
python3 script/check4.py -install
```

You may need to enter the password for several terminals to get the sudo privilege.

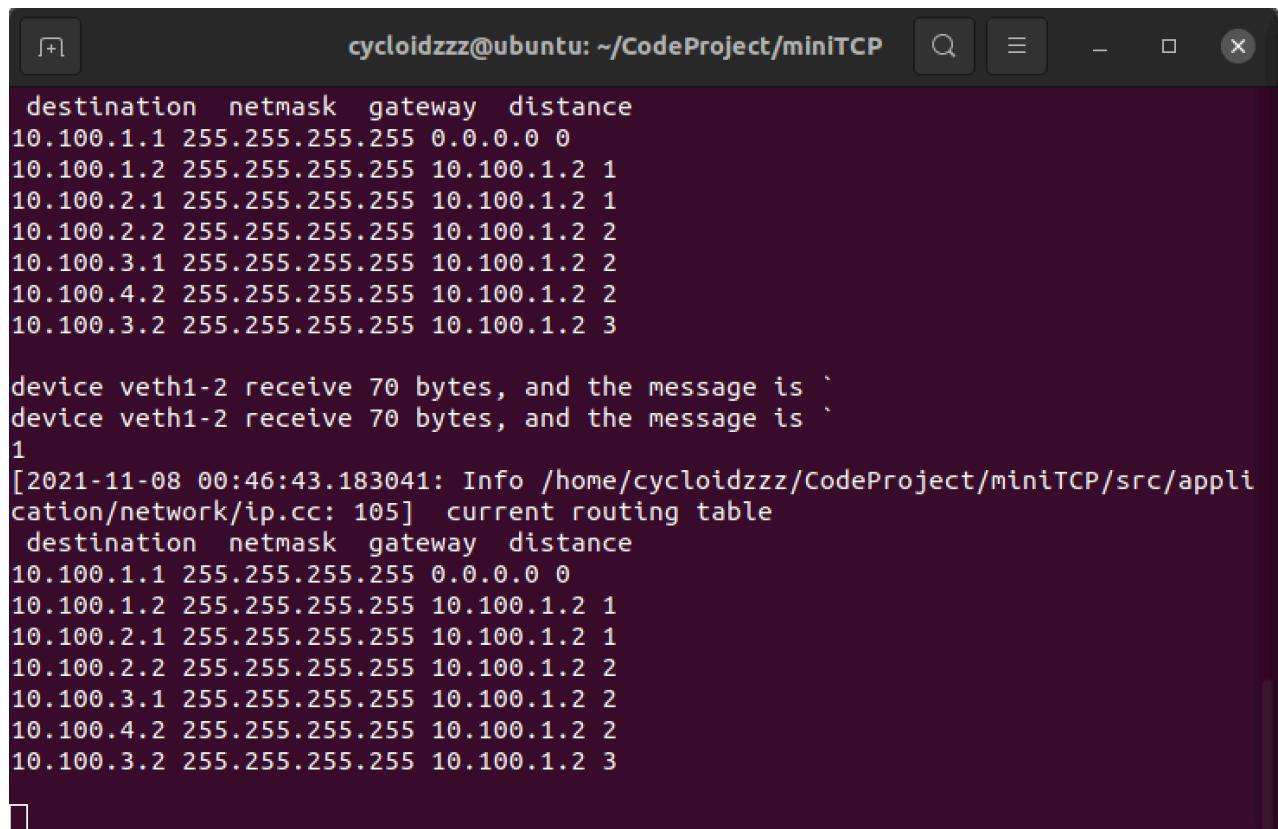
After executing the command, you may need to wait for several seconds to wait for the convergence of the routing algorithm.

After checking out the checkpoint4, please execute the following command to delete the configuration generated by vnetUtils.

```
python3 script/check4.py -delete
```

## 3. Result

(1) To show that NS1 finds NS4, enter 1 to let the terminal of NS1 to print out the routing table. We can find that the address 10.100.3.2 is in the routing table of NS1, which belongs to NS4.



The screenshot shows a terminal window titled "cycloidzzz@ubuntu: ~/CodeProject/miniTCP". The window contains the following text:

```
destination netmask gateway distance
10.100.1.1 255.255.255.255 0.0.0.0 0
10.100.1.2 255.255.255.255 10.100.1.2 1
10.100.2.1 255.255.255.255 10.100.1.2 1
10.100.2.2 255.255.255.255 10.100.1.2 2
10.100.3.1 255.255.255.255 10.100.1.2 2
10.100.4.2 255.255.255.255 10.100.1.2 2
10.100.3.2 255.255.255.255 10.100.1.2 3

device veth1-2 receive 70 bytes, and the message is ` 
device veth1-2 receive 70 bytes, and the message is ` 
1
[2021-11-08 00:46:43.183041: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 105] current routing table
destination netmask gateway distance
10.100.1.1 255.255.255.255 0.0.0.0 0
10.100.1.2 255.255.255.255 10.100.1.2 1
10.100.2.1 255.255.255.255 10.100.1.2 1
10.100.2.2 255.255.255.255 10.100.1.2 2
10.100.3.1 255.255.255.255 10.100.1.2 2
10.100.4.2 255.255.255.255 10.100.1.2 2
10.100.3.2 255.255.255.255 10.100.1.2 3
```

(2) To show that NS1 cannot find NS4 after disconnecting NS2, directly terminate the corresponding terminal of NS2. After several seconds, enter 2 to show the routing table of NS1 again, we can find out that the IP address corresponding to NS2, NS3 and NS4 disappear.

The routing table of NS1 after removing NS2. The ip addresses belongs to NS2, NS3, NS4 are all marked as unreachable (distance equals 17) and being deleted in the next round.

```
cyclodzzz@ubuntu: ~/CodeProject/miniTCP
```

```
10.100.2.2 255.255.255.255 10.100.1.2 17
10.100.3.1 255.255.255.255 10.100.1.2 17
10.100.4.2 255.255.255.255 10.100.1.2 17
10.100.3.2 255.255.255.255 10.100.1.2 17

1
[2021-11-08 00:50:43.894137: Info /home/cyclodzzz/CodeProject/miniTCP/src/application/network/ip.cc: 105] current routing table
destination netmask gateway distance
10.100.1.1 255.255.255.255 0.0.0.0 0
10.100.1.2 255.255.255.255 10.100.1.2 17
10.100.2.1 255.255.255.255 10.100.1.2 17
10.100.2.2 255.255.255.255 10.100.1.2 17
10.100.3.1 255.255.255.255 10.100.1.2 17
10.100.4.2 255.255.255.255 10.100.1.2 17
10.100.3.2 255.255.255.255 10.100.1.2 17

1
[2021-11-08 00:50:50.128284: Info /home/cyclodzzz/CodeProject/miniTCP/src/application/network/ip.cc: 105] current routing table
destination netmask gateway distance
10.100.1.1 255.255.255.255 0.0.0.0 0
```

(3) To show that NS1 can find NS4 again if NS2 reconnects to the network, please execute the following command in the directory of miniTCP in a new terminal.

```
bash script/enter_ns.sh 2
```

After several seconds, enter 2 in the terminal of NS1 to show the routing table, we can find out that the routing table of NS1 contains the IP addresses corresponding to NS2, NS3 and NS4 again.

The screenshot shows a terminal window with the title "cycloidzzz@ubuntu: ~/CodeProject/miniTCP". The window displays two log entries from the application "ip.cc" at different times. Both entries show the current routing table with destination, netmask, gateway, and distance fields.

```

10.100.2.2 255.255.255.255 10.100.1.2 17
10.100.3.1 255.255.255.255 10.100.1.2 17
10.100.4.2 255.255.255.255 10.100.1.2 17
10.100.3.2 255.255.255.255 10.100.1.2 17

1
[2021-11-08 00:50:50.128284: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 105] current routing table
destination netmask gateway distance
10.100.1.1 255.255.255.0 0.0.0.0 0

1
[2021-11-08 00:51:51.975392: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 105] current routing table
destination netmask gateway distance
10.100.1.1 255.255.255.0 0.0.0.0 0
10.100.2.2 255.255.255.255 10.100.1.2 2
10.100.3.1 255.255.255.255 10.100.1.2 2
10.100.4.2 255.255.255.255 10.100.1.2 2
10.100.3.2 255.255.255.255 10.100.1.2 3
10.100.1.2 255.255.255.255 10.100.1.2 1
10.100.2.1 255.255.255.255 10.100.1.2 1

```

## Checkpoint5

### 1. Topology and IP Configuration

The IP address configuration is shown in the following table.

NS			
1	veth1-2 (10.101.1.1)		
2	veth2-1 (10.101.1.2)	veth2-3 (10.101.2.1)	veth2-5 (10.101.4.1)
3	veth3-2 (10.101.2.2)	veth3-4 (10.101.3.1)	veth3-6 (10.101.6.1)
4	veth4-3 (10.101.3.2)		
5	veth5-2 (10.101.4.2)		veth5-6 (10.101.5.1)
6	veth6-3 (10.101.5.2)		veth6-5 (10.101.6.2)

## 2. Usage

To check out checkpoint5, please execute the following command in the directory of miniTCP in the terminal.

```
python3 script/check5.py -install
```

After checking out the checkpoint5, please execute the following command to delete the configuration generated by vnetUtils.

```
python3 script/check5.py -delete
```

## 3. Result

After several seconds, we can find out the distance from each NS to each IP address, which is shown in the following table.

Source/destination	NS1	NS2	NS3	NS4	NS5	NS6
NS1	0	1	2	3	2	3
NS2	1	0	1	2	1	2
NS3	2	1	0	1	2	1
NS4	3	2	1	0	3	2
NS5	2	1	2	3	0	1
NS6	3	2	1	2	1	0

To disconnect NS5, directly kill the terminal corresponding to NS5.

After several seconds, we can reprint the result of each terminal, the result is shown in the following table.

Source/Destination	NS1	NS2	NS3	NS4	NS6
NS1	0	1	2	3	3
NS2	1	0	1	2	2
NS3	2	1	0	1	1
NS4	3	2	1	0	2
NS6	3	2	1	2	0

## Checkpoint6

### 1. Topology and IP Configuration

We use the topology and the Ip configuration which are the same as the checkpoint4.

### 2. Usage

To check out checkpoint6, please execute the following command in the directory of miniTCP in the terminal.

```
python3 script/check4.py -install
```

After checking out the checkpoint6, please execute the following command to delete the configuration generated by vnetUtils.

```
python3 script/check4.py -delete
```

### 3. Result

We manully add two new routing entry into the routing table of NS1. By entering the following two lines into the terminal of NS1.

```
add 10.100.3.2 255.0.0.0 1.1.1.1
```

and

```
add 10.100.3.2 255.255.0.0 2.2.2.2
```

```
destination netmask gateway distance
10.100.1.1 255.255.255.255 0.0.0.0 0
10.100.1.2 255.255.255.255 10.100.1.2 1
10.100.2.1 255.255.255.255 10.100.1.2 1
10.100.2.2 255.255.255.255 10.100.1.2 2
10.100.3.1 255.255.255.255 10.100.1.2 2
10.100.4.2 255.255.255.255 10.100.1.2 2
10.100.3.2 255.255.255.255 10.100.1.2 3

3 10.100.3.2 255.0.0.0 1.1.1.1
[2021-11-08 04:11:13.808876: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 111]
adding a new item into routing table[2021-11-08 04:11:13.808918: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 121] netmask = 255.0.0.0
[2021-11-08 04:11:13.808950: Info /home/cycloidzzz/CodeProject/miniTCP/src/network/routing_impl.cc: 215] unsafeInsert: successfully add an permanent item into the routing table.
3 10.100.3.2 255.255.0.0 2.2.2.2
[2021-11-08 04:11:27.867261: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 111]
adding a new item into routing table[2021-11-08 04:11:27.867289: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 121] netmask = 255.255.0.0
[2021-11-08 04:11:27.867324: Info /home/cycloidzzz/CodeProject/miniTCP/src/network/routing_impl.cc: 215] unsafeInsert: successfully add an permanent item into the routing table.
1
[2021-11-08 04:11:31.793833: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 105]
current routing table
destination netmask gateway distance
10.100.1.1 255.255.255.255 0.0.0.0 0
10.100.1.2 255.255.255.255 10.100.1.2 1
10.100.2.1 255.255.255.255 10.100.1.2 1
10.100.2.2 255.255.255.255 10.100.1.2 2
10.100.3.1 255.255.255.255 10.100.1.2 2
10.100.4.2 255.255.255.255 10.100.1.2 2
10.100.3.2 255.255.255.255 10.100.1.2 3
10.100.3.2 255.0.0.0 1.1.1.1 0
10.100.3.2 255.255.0.0 2.2.2.2 0
```

If the routing table fails to apply the longest prefix matching rule, then the device with IP address 10.100.3.2, which corresponds to NS4, would not receive the message from NS1.

We send a message on NS1 by entering the following message on the terminal of NS1.

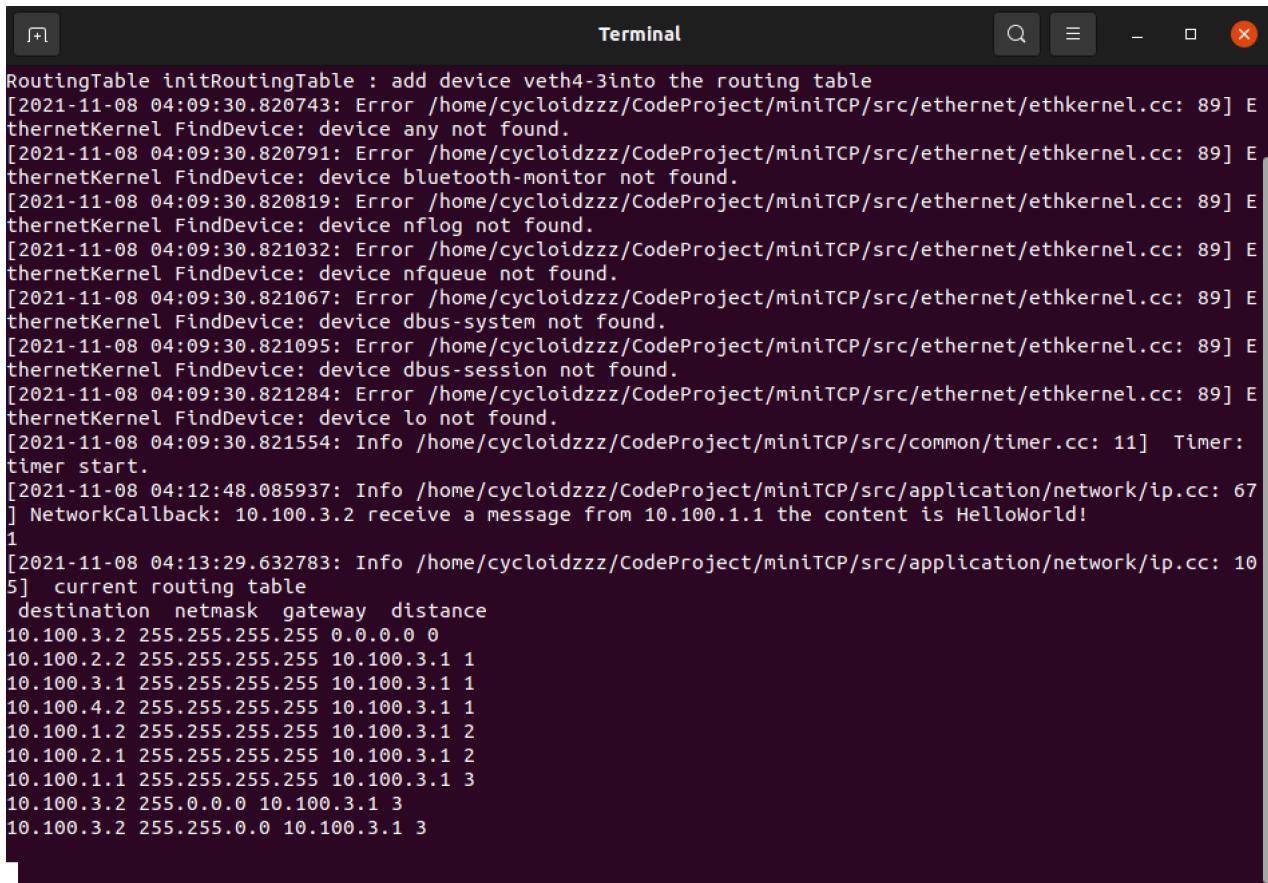
```
send 10.100.1.1 10.100.3.2 HelloWorld!
```

```
Terminal
10.100.3.1 255.255.255.255 10.100.1.2 2
10.100.4.2 255.255.255.255 10.100.1.2 2
10.100.3.2 255.255.255.255 10.100.1.2 3

3 10.100.3.2 255.0.0.0 1.1.1.1
[2021-11-08 04:11:13.808876: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 111]
    adding a new item into routing table[2021-11-08 04:11:13.808918: Info /home/cycloidzzz/CodeProject/miniT
P/src/application/network/ip.cc: 121] netmask = 255.0.0.0
[2021-11-08 04:11:13.808950: Info /home/cycloidzzz/CodeProject/miniTCP/src/network/routing_impl.cc: 215] u
nsafeInsert: successfully add an permanent item into the routing table.
3 10.100.3.2 255.255.0.0 2.2.2.2
[2021-11-08 04:11:27.867261: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 111]
    adding a new item into routing table[2021-11-08 04:11:27.867289: Info /home/cycloidzzz/CodeProject/miniT
P/src/application/network/ip.cc: 121] netmask = 255.255.0.0
[2021-11-08 04:11:27.867324: Info /home/cycloidzzz/CodeProject/miniTCP/src/network/routing_impl.cc: 215] u
nsafeInsert: successfully add an permanent item into the routing table.
1
[2021-11-08 04:11:31.793833: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 105]
    current routing table
    destination netmask gateway distance
10.100.1.1 255.255.255.255 0.0.0.0 0
10.100.1.2 255.255.255.255 10.100.1.2 1
10.100.2.1 255.255.255.255 10.100.1.2 1
10.100.2.2 255.255.255.255 10.100.1.2 2
10.100.3.1 255.255.255.255 10.100.1.2 2
10.100.4.2 255.255.255.255 10.100.1.2 2
10.100.3.2 255.255.255.255 10.100.1.2 3
10.100.3.2 255.0.0.0 1.1.1.1 0
10.100.3.2 255.255.0.0 2.2.2.2 0

0 10.100.1.1 10.100.3.2 HelloWorld!
[2021-11-08 04:12:47.753662: Info /home/cycloidzzz/CodeProject/miniTCP/src/network/ip.cc: 41] destination
is 10.100.3.2
[2021-11-08 04:12:47.753698: Info /home/cycloidzzz/CodeProject/miniTCP/src/network/ip.cc: 49] A packet is
sending from 10.100.1.1 to 10.100.1.2
```

The only device (veth4-3) in NS4 can successfully receive the message from NS1 as the following picture shows, which proves that our routing table faithfully follows the longest prefix matching rule.



```
RoutingTable initRoutingTable : add device veth4-3into the routing table
[2021-11-08 04:09:30.820743: Error /home/cycloidzzz/CodeProject/miniTCP/src/ethernet/ethkernel.cc: 89] E
thernetKernel FindDevice: device any not found.
[2021-11-08 04:09:30.820791: Error /home/cycloidzzz/CodeProject/miniTCP/src/ethernet/ethkernel.cc: 89] E
thernetKernel FindDevice: device bluetooth-monitor not found.
[2021-11-08 04:09:30.820819: Error /home/cycloidzzz/CodeProject/miniTCP/src/ethernet/ethkernel.cc: 89] E
thernetKernel FindDevice: device nflog not found.
[2021-11-08 04:09:30.821032: Error /home/cycloidzzz/CodeProject/miniTCP/src/ethernet/ethkernel.cc: 89] E
thernetKernel FindDevice: device nfqueue not found.
[2021-11-08 04:09:30.821067: Error /home/cycloidzzz/CodeProject/miniTCP/src/ethernet/ethkernel.cc: 89] E
thernetKernel FindDevice: device dbus-system not found.
[2021-11-08 04:09:30.821095: Error /home/cycloidzzz/CodeProject/miniTCP/src/ethernet/ethkernel.cc: 89] E
thernetKernel FindDevice: device dbus-session not found.
[2021-11-08 04:09:30.821284: Error /home/cycloidzzz/CodeProject/miniTCP/src/ethernet/ethkernel.cc: 89] E
thernetKernel FindDevice: device lo not found.
[2021-11-08 04:09:30.821554: Info /home/cycloidzzz/CodeProject/miniTCP/src/common/timer.cc: 11] Timer:
timer start.
[2021-11-08 04:12:48.085937: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 67
] NetworkCallback: 10.100.3.2 receive a message from 10.100.1.1 the content is HelloWorld!
1
[2021-11-08 04:13:29.632783: Info /home/cycloidzzz/CodeProject/miniTCP/src/application/network/ip.cc: 10
5] current routing table
destination netmask gateway distance
10.100.3.2 255.255.255.255 0.0.0.0 0
10.100.2.2 255.255.255.255 10.100.3.1 1
10.100.3.1 255.255.255.255 10.100.3.1 1
10.100.4.2 255.255.255.255 10.100.3.1 1
10.100.1.2 255.255.255.255 10.100.3.1 2
10.100.2.1 255.255.255.255 10.100.3.1 2
10.100.1.1 255.255.255.255 10.100.3.1 3
10.100.3.2 255.0.0.0 10.100.3.1 3
10.100.3.2 255.255.0.0 10.100.3.1 3
```

## Part C

### Usage

To run the program, please first run the following command in the main directory of miniTCP.

```
mkdir build
cd build
sudo cmake ..
sudo make
cd ..
```

## Writing Task 3 : Describe how you correctly handled TCP state changes.

I was planning to implement the TCP state changes by storing socket informations in three kinds of different sockets, namely non-synchronous socket(mainly **SYN Sent**, **SYN Received** and **Listening**) , established (**Established**) socket and time wait socket (mainly **FinWait1**, **FinWait2**, **TimeWait**, **Closing**, **LastAck**, **Closed**). However, I finally find out that even though splitting the states into 3 different sets of states would be more clear for programming, it will introduce much more programming work for me. Finally I choose to implement only **listen socket** and **established socket**, in which the established socket should maintain other TCP state transition except **Listening**.

To be more specific, when the user use the socket system call to initialize a socket, the only thing I need to do is to bind the socket with an arbitrary local ip and local port which is not occupied yet, and then I use a map to bind the socket with a system allocated file descriptor. The initial state of this first-created socket will be **Closed**. Whether a socket should be a **listen socket** or a **established socket** is postpone to the time when the user calls connect() or listen() with the corresponding file descriptor, the state of the socket will transfer to **SYN Sent** or **Listen** accordingly as well. These two different kinds of socket will be inserted into different hash tables.

When a TCP packet comes and the TCP worker wants to find out the corresponding socket, the worker will first search from the hash table of the established socket, if the worker can find the socket, the worker will process the packet with the handler of the socket. If it cannot find the socket from the established map, which implies that the TCP packet **must** be a SYN packet (the peer wants to connect with us for the first time), we can continue to find our local listen socket in the listen socket hash table.

What is **slightly different** from the standard implementation is that the valid state for the **listen socket** would be **Closed** and **Listening** only. The initial state for the listen socket will be **Closed**, when user call listen with the corresponding file descrptor, the state of the socket will be transfer to **Listening**. After processing and SYN packet from the peer, the listen socket will insert a **new socket** with **SYN Received** state into the established socket map. The listen socket only need to process the **SYN packets** and ignore others. This design is model after **Linux's** and can deal with multiple TCP connection requests by a single listener.

As for the established socket, I would like to devide the problem into two parts, namely the 3-way handshake transition and the 4-way handshake transition. The 3-way handshake transition need to deal with the state transition among **SYN sent**, **SYN received** and **Established**. Specifically, my implementation can handle two kinds of connection: (1) the client call connect() to connect the server (2) both sides call connect() to connect simultaneously.

When the user calls connect() with a socket descriptor, the state of the correspoding socket will be changed from **Closed** to **SYN sent** and the socket control block will be inserted into the established socket map. When the listen socket receives this SYN packet sent by connect(), the listen socket will create a new socket, change the state of the new socket to be **SYN received**, send back an SYN ACK packet and insert the new socket into established map. Then the next time when the ACK from the client comes, the TCP worker can directly find the stabled socket from the established map. The TCP worker then checks if the sequence number of the local socket number

and the ack number of the ACK packet are matched, if so, the worker will change its status from **SYN received** to **Established**. Otherwise, a RESET packet will be sent back to the client.

If a socket with state **SYN Sent** received a **SYN packet** from its peer, which implies that the peer opens the connection by using connect function simultaneously. The local socket will store the iss of the remote, send back an SYN ACK packet and switch to state **SYN Received**. If a socket with state **SYN Received** receives a SYN ACK packet, the local socket will send back a ACK packet or a RESET, depending on whether the ACK number of the SYN ACK packet matches the sequence number of the local socket or not.

What's more, since the SYN packet/ SYN ACK packet might get lost during transmission, we need to set up a **retransmission timer** for the SYN/SYN ACK packet retransmission. The initial timeout interval is set to be 1s, I use the exponential retrieve algorithm to better estimate the initial round trip time.

As for the 4-way handshake state transition, I only implement the simple case, in which only the client will call close() to close the socket. I need to implement following state transition in order to support the one-sided 4-way handshake.

- (1) When the user is trying to use the function close() to close the socket, the socket will change its state to **FinWait1** and send a FIN ACK message to the peer socket.
- (2) When a **Established** socket receives a FIN ACK message, the socket will sends back an ACK accordingly and change its own state to **CloseWait**.
- (3) When a socket with state **FinWait1** receives a ACK from remote and the ack number matches its local sequence number, this socket change to state **FinWait2**.
- (4) After transferring all the packet to the remote, a socket with state **CloseWait** will send a FIN ACK packet to the peer and turn to state **LastAck**.
- (5) When a socket with state **FinWait2** receives a FIN ACK and the sequence number matches, this socket turns to state **TimeWait**, sends back an ACK accordingly and wait for 2 maximum segment time (I choose the MSL=10 seconds in my own implementation, since a standard MSL might be too long for evaluation). If this receives duplicated FIN ACK, it will reply a ACK and set up the timewait timer again. If no duplicated FIN ACK is received within 2 MSL, the socket will then turn to state **Closed**.
- (6) When a socket with state **LastAck** receives a ACK which is matched, the socket will immediately turn to state **Closed**.

One thing you need to notice is that, since it is likely that the client may exit without using close() function to notify the server, the keepalive timer is required in my implementation (you may find it useful in checkpoint10). The keepalive will be called every 5 seconds, if it finds out the corresponding socket hasn't received any data from the remote socket, it will continue to probe in the next 5 rounds, if there is still no reply from the remote, the timer will directly turn the state of the socket to **Closed**.

# Checkpoint 7

## 1. Topology

In checkpoint7, two Linux nss, namely ns1 and ns2, will connect with each other, the nic in ns1 (veth1-2) is configured with 10.102.1.1 and the nic in ns2 (veth2-1) is configured with 10.102.1.2.

## 2. Usage

To check out checkpoint7, please execute the following command in the directory of miniTCP in the terminal.

```
python3 script/check7.py -install
```

After checking out the checkpoint9, please execute the following command to delete the configuration generated by vnetUtils.

```
python3 script/check7.py -delete
```

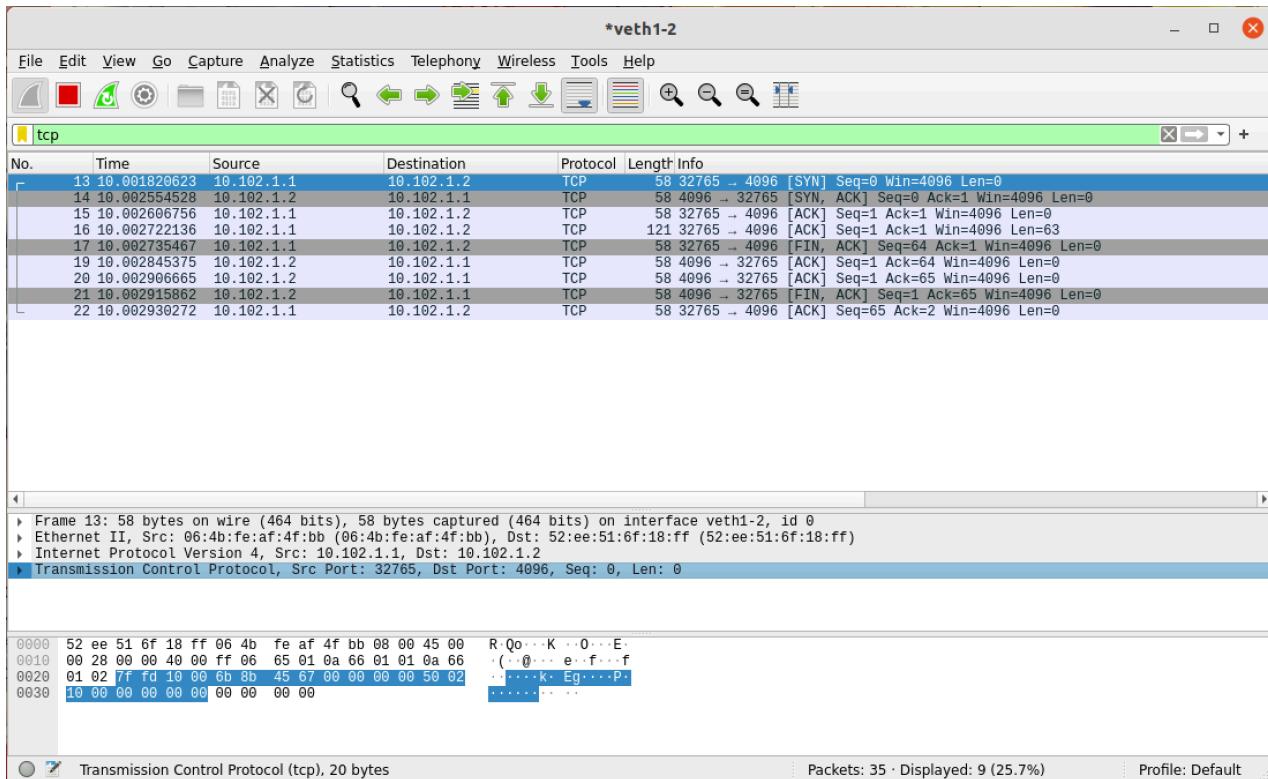
In order to capture packets from the network with network number NS (1 or 2), please execute the following command in the miniTCP directory **instead of the build directory**.

```
bash script/capture_ns.sh NS
```

## 3. Result

In this process, the client in ns1 will open the socket and send a piece message to the server in ns1 and close.

9 TCP packets are captured in the whole process.



The first 2 bytes are 7f fd, which means the TCP source port number (in big endian) is 32765.

The next 2 bytes are 10, 00, which stands for the TCP destination port number (in big endian) is 4096.

The next 4 bytes are 6b 8b 45 67, which stands for the sequence number of this TCP packet (in big endian) is 0x67458b6b.

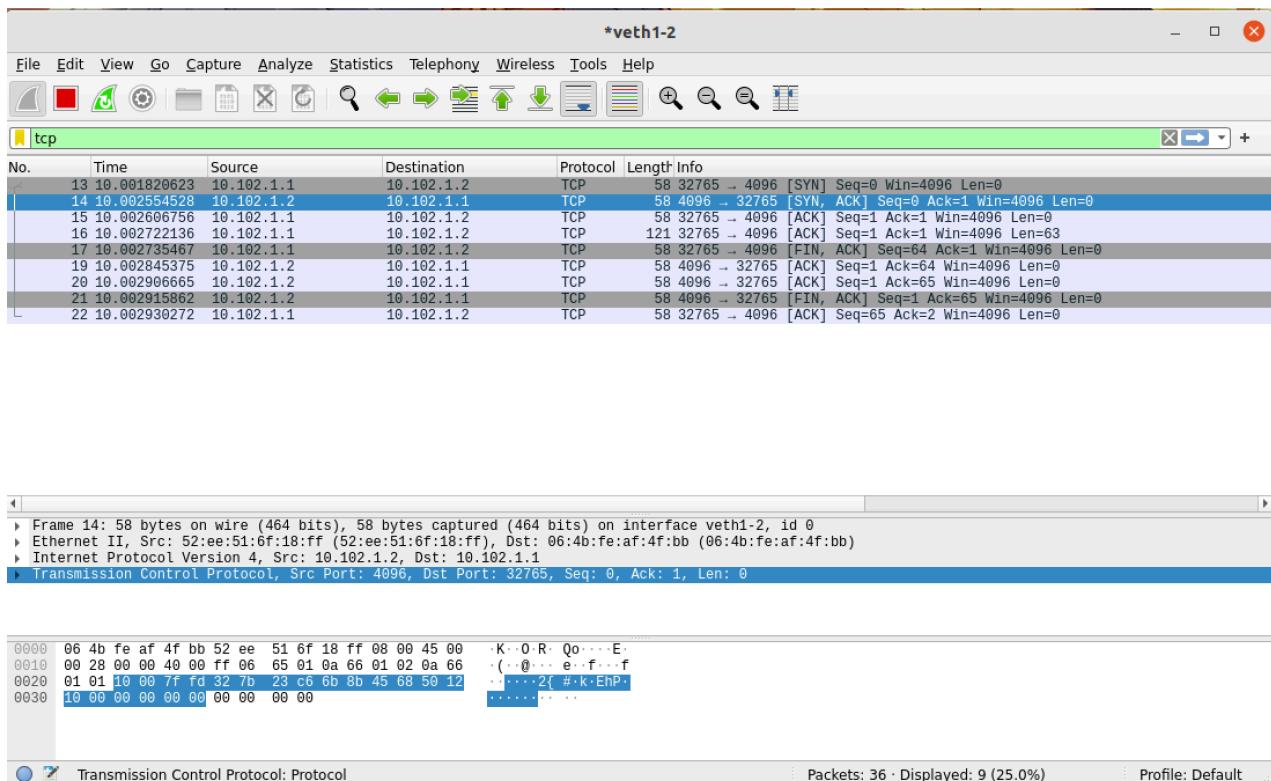
The next 4 bytes are 00 00 00 00, which stands for the ack number (in big endian) is 0x0.

The next 1 byte is 50, which stands for the header length is 20 bytes.

The next 1 byte is 0x02, which stands for the TCP control flags, and this is a SYN packet.

The next 2 bytes are 10, 00, which stands for the sender receiving window size (in big endian) is 4096.

The next 4 bytes are 00, 00, 00, 00, which stands for the TCP packet checksum (in big endian).



The first 2 bytes are 10, 00, which means the TCP source port number (in big endian) is 4096.

The next 2 bytes are 7f, fd, which stands for the TCP destination port number (in big endian) is 32765.

The next 4 bytes are 32, 7b, 23, c6, which stands for the sequence number of this TCP packet (in big endian).

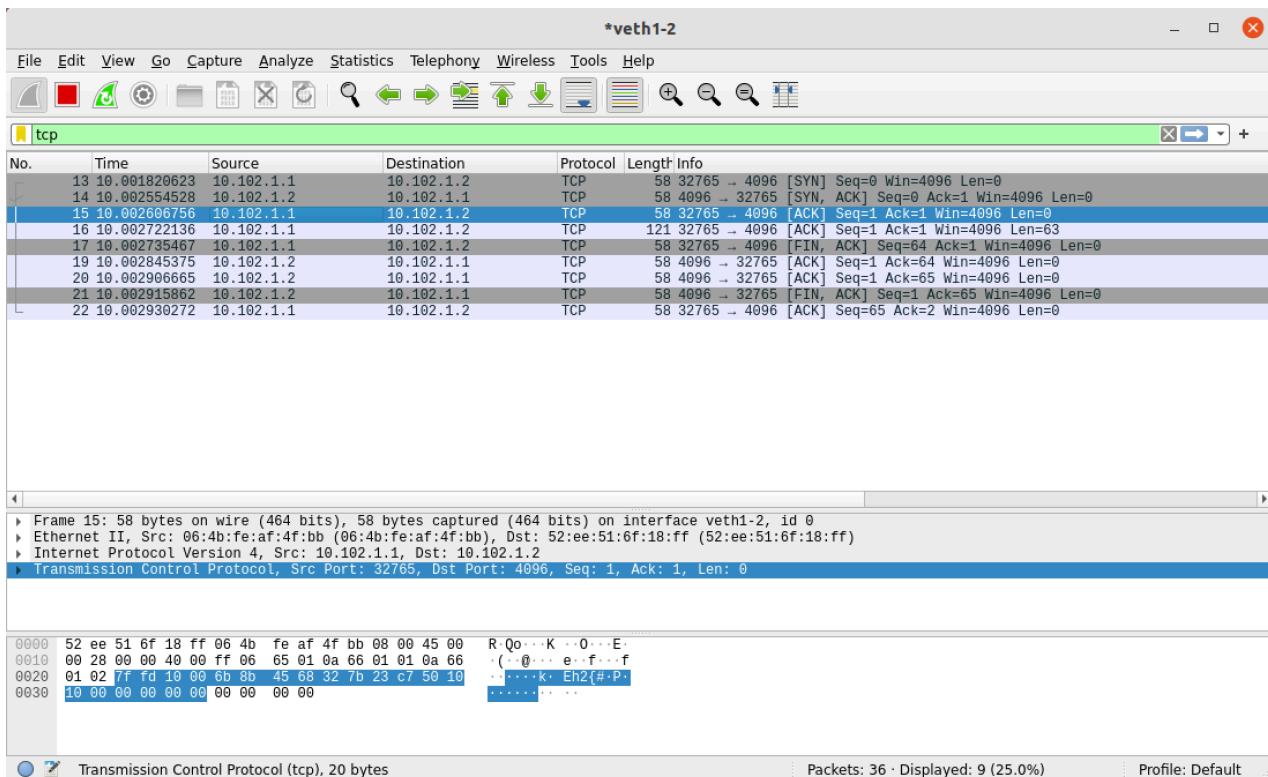
The next 4 bytes are 23 c6 6b 8b, which stands for the ack number of this TCP packet (in big endian).

The next 1 byte is 50, which stands for the header length is 20 bytes.

The next 1 byte is 0x12, which stands for the TCP control flags, and this is a SYN ACK packet.

The next 2 bytes are 10, 00, which stands for the sender receiving window size (in big endian) is 4096.

The next 4 bytes are 00, 00, 00, 00, which stands for the TCP packet checksum (in big endian).



The first 2 bytes are 7f, fd, which means the TCP source port number (in big endian).

The next 2 bytes are 10, 00 which stands for the TCP destination port number (in big endian).

The next 4 bytes are 6b, 8b, 45, 68, which stands for the sequence number of this TCP packet (in big endian).

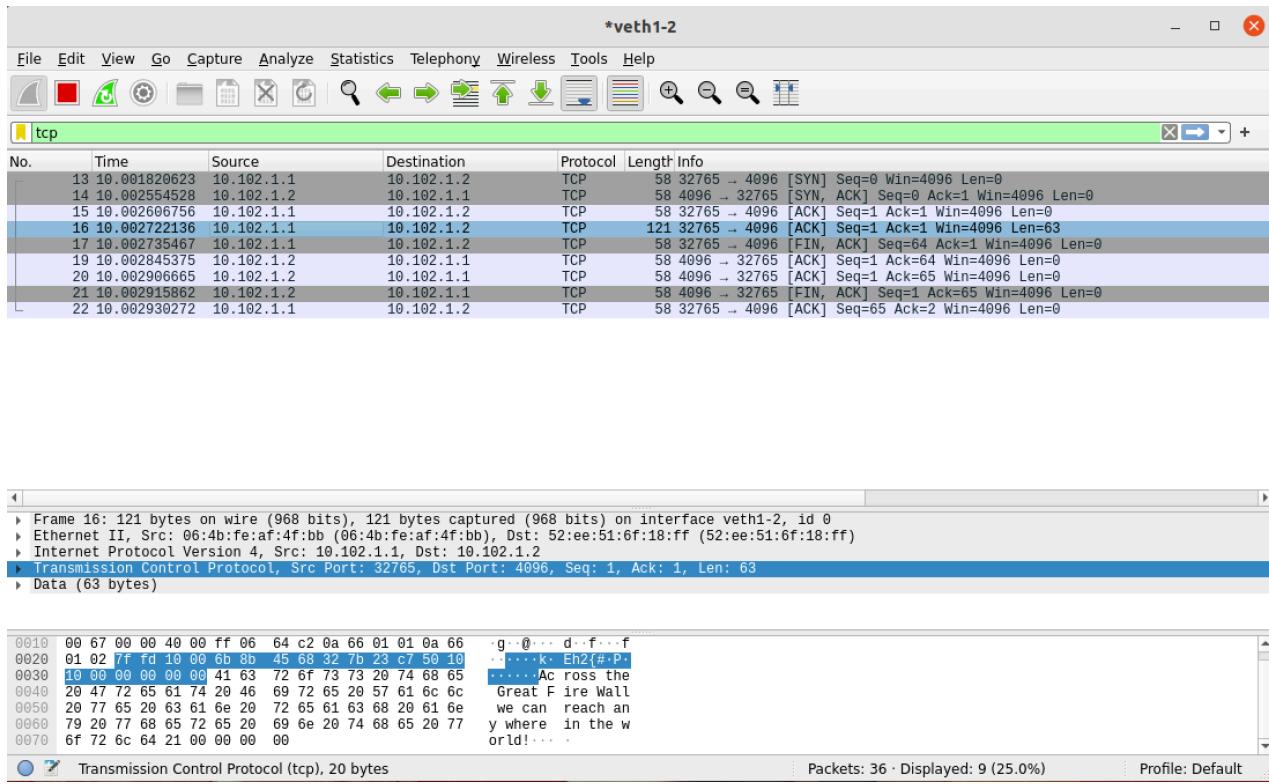
The next 4 bytes are 32, 7b, 23, c7, which stands for the ack number of this TCP packet (in big endian).

The next 1 byte is 50, which stands for the header length is 20 bytes.

The next 1 byte is 0x10, which stands for the TCP control flags, and this is a ACK packet.

The next 2 bytes are 10, 00, which stands for the sender receiving window size (in big endian) is 4096.

The next 4 bytes are 00, 00, 00, 00, which stands for the TCP packet checksum (in big endian).



The first 2 bytes are 7f, fd, which means the TCP source port number (in big endian).

The next 2 bytes are 10, 00 which stands for the TCP destination port number (in big endian).

The next 4 bytes are 6b, 8b, 45, 68, which stands for the sequence number of this TCP packet (in big endian).

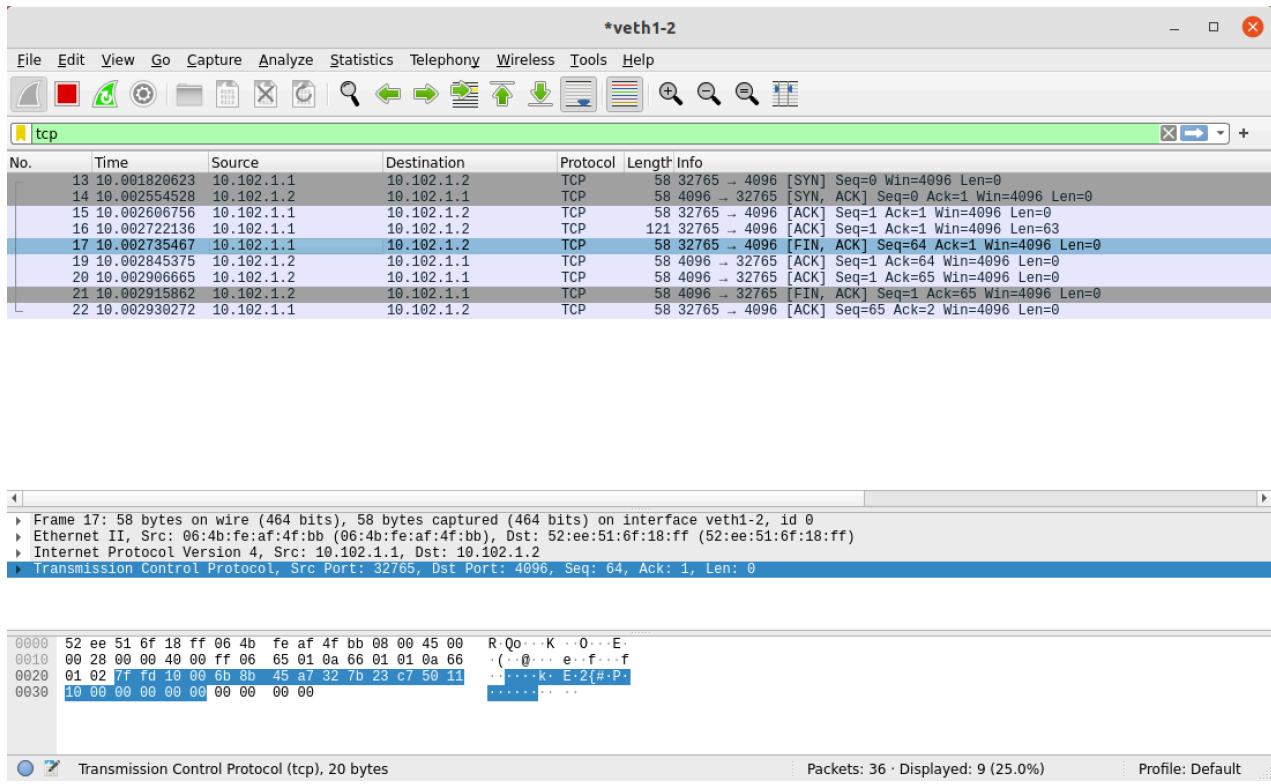
The next 4 bytes are 32, 7b, 23, c7, which stands for the ack number of this TCP packet (in big endian).

The next 1 byte is 50, which stands for the header length is 20 bytes.

The next 1 byte is 0x10, which stands for the TCP control flags, and this is a ACK packet.

The next 2 bytes are 10, 00, which stands for the sender receiving window size (in big endian) is 4096.

The next 4 bytes are 00, 00, 00, 00, which stands for the TCP packet checksum (in big endian).



The first 2 bytes are 7f, fd, which means the TCP source port number (in big endian).

The next 2 bytes are 10, 00 which stands for the TCP destination port number (in big endian).

The next 4 bytes are 6b, 8b, 45, a7, which stands for the sequence number of this TCP packet (in big endian).

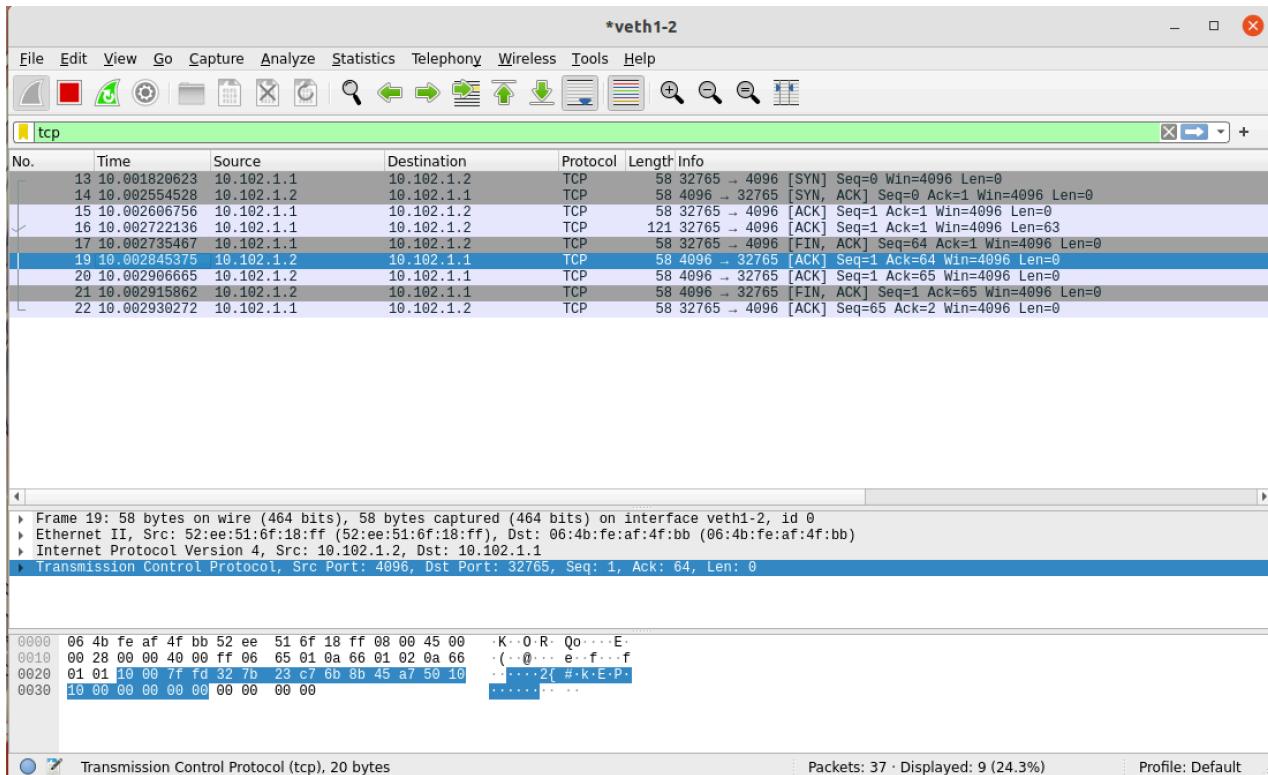
The next 4 bytes are 32, 7b, 23, c7, which stands for the ack number of this TCP packet (in big endian).

The next 1 byte is 50, which stands for the header length is 20 bytes.

The next 1 byte is 0x11, which stands for the TCP control flags, and this is a FIN ACK packet.

The next 2 bytes are 10, 00, which stands for the sender receiving window size (in big endian) is 4096.

The next 4 bytes are 00, 00, 00, 00, which stands for the TCP packet checksum (in big endian).



The first 2 bytes are 10, 00, which means the TCP source port number (in big endian).

The next 2 bytes are 7f, fd which stands for the TCP destination port number (in big endian).

The next 4 bytes are 32, 7b, 23, c7, which stands for the sequence number of this TCP packet (in big endian).

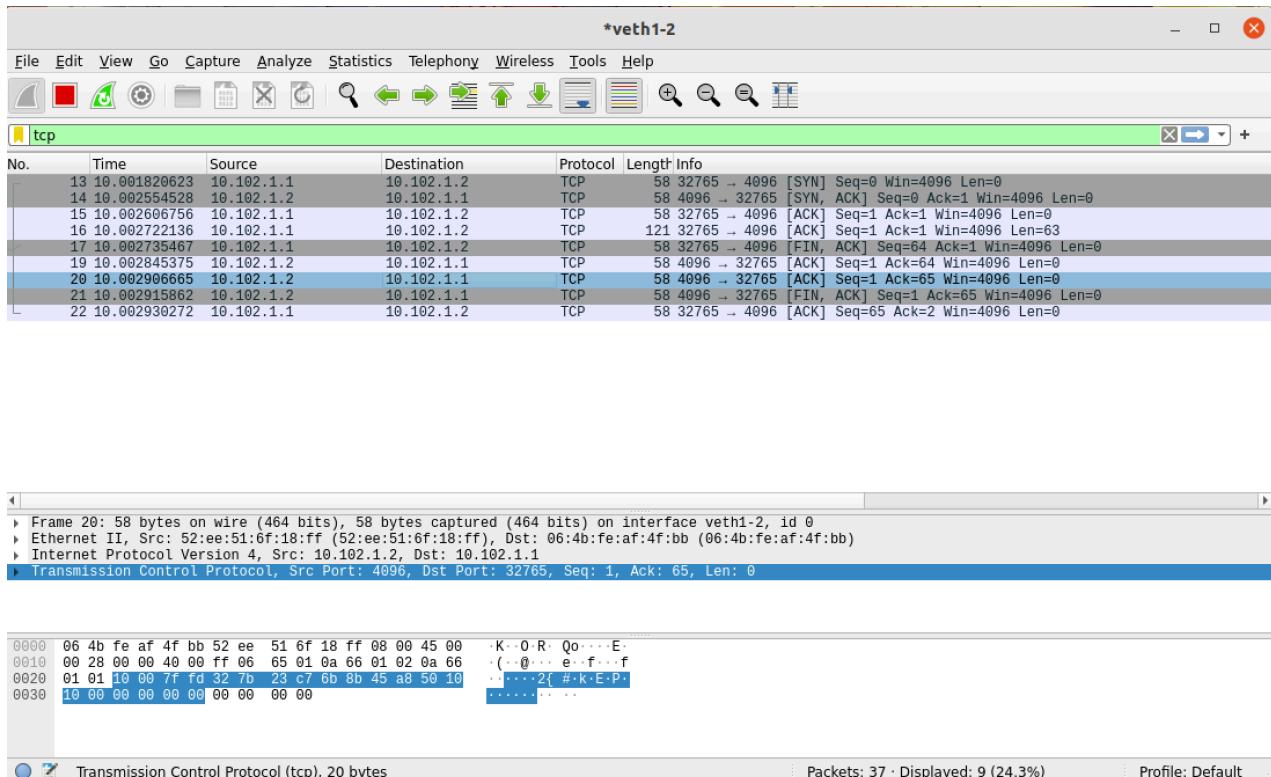
The next 4 bytes are 6b, 8b, 45, a7, which stands for the ack number of this TCP packet (in big endian).

The next 1 byte is 50, which stands for the header length is 20 bytes.

The next 1 byte is 0x10, which stands for the TCP control flags, and this is a ACK packet.

The next 2 bytes are 10, 00, which stands for the sender receiving window size (in big endian) is 4096.

The next 4 bytes are 00, 00, 00, 00, which stands for the TCP packet checksum (in big endian).



The first 2 bytes are 10, 00, which means the TCP source port number (in big endian).

The next 2 bytes are 7f, fd which stands for the TCP destination port number (in big endian).

The next 4 bytes are 32, 7b, 23, c7 which stands for the sequence number of this TCP packet (in big endian).

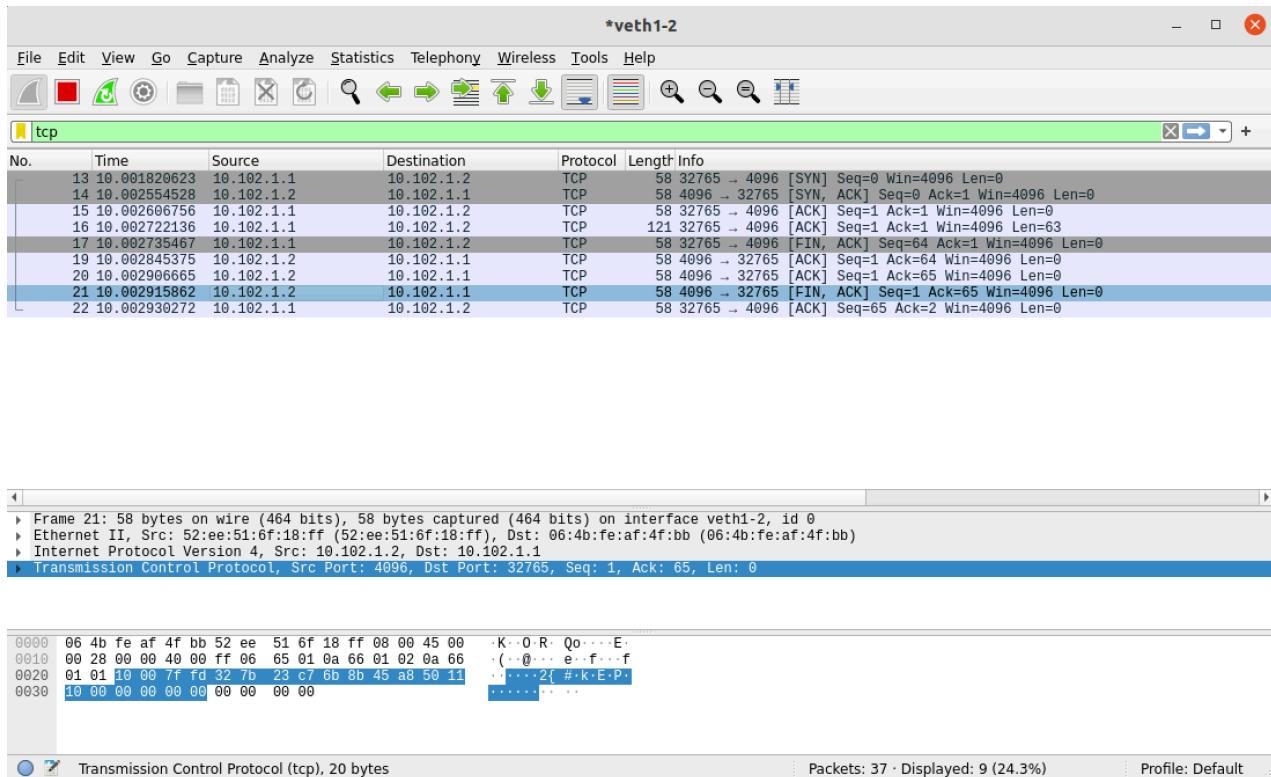
The next 4 bytes are 6b, 8b, 45,a8 which stands for the ack number of this TCP packet (in big endian).

The next 1 byte is 50, which stands for the header length is 20 bytes.

The next 1 byte is 0x10, which stands for the TCP control flags, and this is a ACK packet.

The next 2 bytes are 10, 00, which stands for the sender receiving window size (in big endian) is 4096.

The next 4 bytes are 00, 00, 00, 00, which stands for the TCP packet checksum (in big endian).



The first 2 bytes are 10, 00, which means the TCP source port number (in big endian).

The next 2 bytes are 7f, fd which stands for the TCP destination port number (in big endian).

The next 4 bytes are 32, 7b, 23, c7, which stands for the sequence number of this TCP packet (in big endian).

The next 4 bytes are 23, c7, 6b, 8b, which stands for the ack number of this TCP packet (in big endian).

The next 1 byte is 50, which stands for the header length is 20 bytes.

The next 1 byte is 0x11, which stands for the TCP control flags, and this is a FIN ACK packet.

The next 2 bytes are 10, 00, which stands for the sender receiving window size (in big endian) is 4096.

The next 4 bytes are 00, 00, 00, 00, which stands for the TCP packet checksum (in big endian).

*veth1-2						
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help						
tcp						
No.	Time	Source	Destination	Protocol	Length	Info
13	10.001820623	10.102.1.1	10.102.1.2	TCP	58	32765 → 4096 [SYN] Seq=0 Win=4096 Len=0
14	10.002554528	10.102.1.2	10.102.1.1	TCP	58	4096 → 32765 [SYN, ACK] Seq=0 Ack=1 Win=4096 Len=0
15	10.002606756	10.102.1.1	10.102.1.2	TCP	58	32765 → 4096 [ACK] Seq=1 Ack=1 Win=4096 Len=0
16	10.002722136	10.102.1.1	10.102.1.2	TCP	121	32765 → 4096 [ACK] Seq=1 Ack=1 Win=4096 Len=63
17	10.002735467	10.102.1.1	10.102.1.2	TCP	58	32765 → 4096 [FIN, ACK] Seq=64 Ack=1 Win=4096 Len=0
18	10.002845375	10.102.1.2	10.102.1.1	TCP	58	4096 → 32765 [ACK] Seq=1 Ack=64 Win=4096 Len=0
19	10.002906655	10.102.1.2	10.102.1.1	TCP	58	4096 → 32765 [ACK] Seq=1 Ack=65 Win=4096 Len=0
20	10.002915862	10.102.1.2	10.102.1.1	TCP	58	4096 → 32765 [FIN, ACK] Seq=1 Ack=65 Win=4096 Len=0
21	10.002930272	10.102.1.1	10.102.1.2	TCP	58	32765 → 4096 [ACK] Seq=65 Ack=2 Win=4096 Len=0
L						
22	10.002930272	10.102.1.1	10.102.1.2	TCP	58	32765 → 4096 [ACK] Seq=65 Ack=2 Win=4096 Len=0

```
Frame 22: 58 bytes on wire (464 bits), 58 bytes captured (464 bits) on interface veth1-2, id 0
Ethernet II, Src: 06:4b:fe:af:4f:bb (06:4b:fe:af:4f:bb), Dst: 52:ee:51:6f:18:ff (52:ee:51:6f:18:ff)
Internet Protocol Version 4, Src: 10.102.1.1, Dst: 10.102.1.2
Transmission Control Protocol, Src Port: 32765, Dst Port: 4096, Seq: 65, Ack: 2, Len: 0
```

0000	52 ee 51 6f 18 ff 06 4b fe af 4f bb 08 00 45 00	R Qo...K...O...E...
0010	00 28 00 00 40 00 ff 06 65 01 0a 66 01 01 0a 66	(...@...e...f...f
0020	01 02 7f fd 10 00 6b 8d 45 a8 32 7b 23 c8 50 10	.1...k. E-2{#-P-
0030	10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

Transmission Control Protocol (tcp), 20 bytes | Packets: 38 · Displayed: 9 (23.7%) | Profile: Default

The first 2 bytes are 7f, fd, which means the TCP source port number (in big endian).

The next 2 bytes are 10, 00 which stands for the TCP destination port number (in big endian).

The next 4 bytes are 6b, 8b, 45, a8, which stands for the sequence number of this TCP packet (in big endian).

The next 4 bytes are 32, 7b, 23, c8, which stands for the ack number of this TCP packet (in big endian).

The next 1 byte is 50, which stands for the header length is 20 bytes.

The next 1 byte is 0x10, which stands for the TCP control flags, and this is a ACK packet.

The next 2 bytes are 10, 00, which stands for the sender receiving window size (in big endian) is 4096.

The next 4 bytes are 00, 00, 00, 00, which stands for the TCP packet checksum (in big endian).

## Checkpoint 8

### 1. Topology

The topology we use is the same as the checkpoint7. Please checkout checkpoint7 for further details.

## 2. Usage

To check out checkpoint8, please execute the following command in the directory of miniTCP in the terminal.

```
python3 script/check8.py -install
```

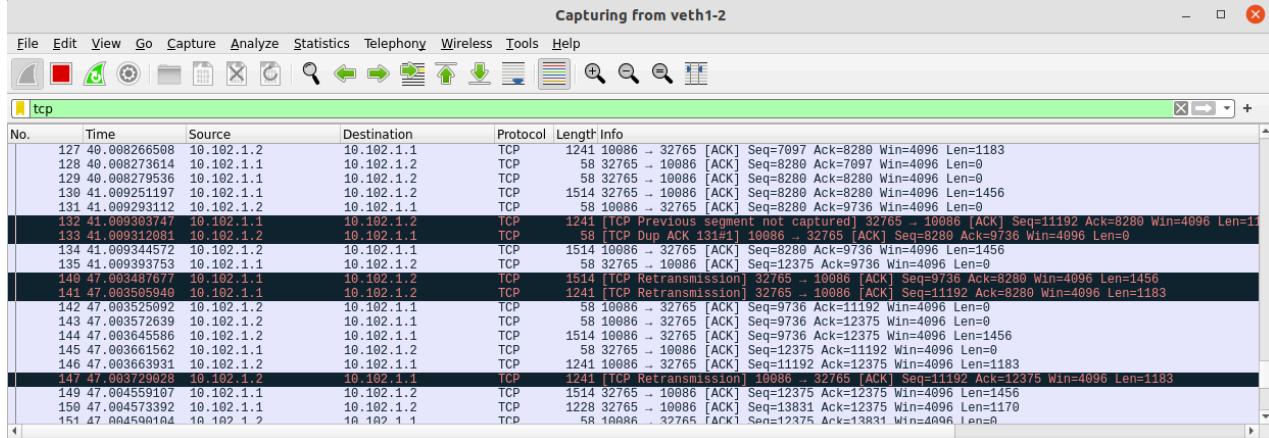
After checking out the checkpoint9, please execute the following command to delete the configuration generated by vnetUtils.

```
python3 script/check8.py -delete
```

In order to capture packets from the network with network number NS (1 or 2), please execute the following command in the miniTCP directory **instead of the build directory**.

```
bash script/capture_ns.sh NS
```

## 3. Result



The result might not be the same, since netem may drop packet randomly.

As the trace in wireshark shows, the packet transfer from 10.102.1.1 to 10.102.1.2 with sequence number 9736 is dropped by veth1-2. When the retransmission timer is timeout, it retransmits the packet with sequence number 9736 and 11192 (the frame #140 and #141). Finally these two packets are acknowledged by the frame #142 and #143.

# Checkpoint 9

## 1. Topology

The topology and the IP configuration is the same as checkpoint4, please see checkpoint4 above.

## 2. Usage

To check out checkpoint9, please execute the following command in the directory of miniTCP in the terminal.

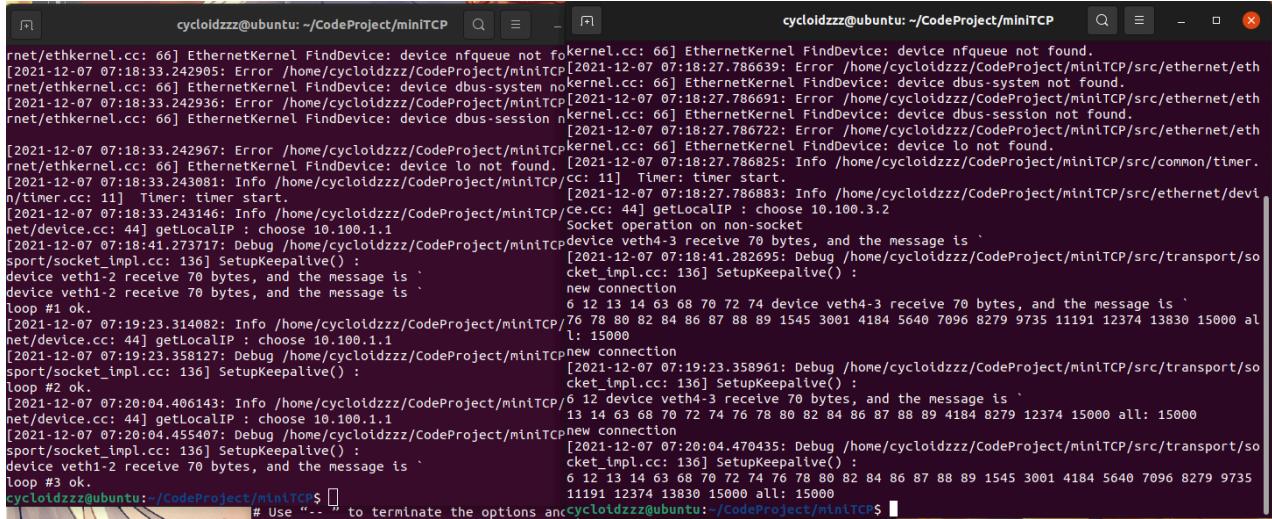
```
python3 script/check9.py -install
```

After checking out the checkpoint9, please execute the following command to delete the configuration generated by vnetUtils.

```
python3 script/check9.py -delete
```

## 3. Result

Only the result of the server and the client would be shown, here is the result of echo\_server and echo\_client.



```
cycliodzzz@ubuntu: ~/CodeProject/miniTCP$ python3 script/check9.py -install
[2021-12-07 07:18:33.242905: Error /home/cycliodzzz/CodeProject/miniTCP/kernel.cc: 66] EthernetKernel FindDevice: device nfqueue not found.
[2021-12-07 07:18:33.242905: Error /home/cycliodzzz/CodeProject/miniTCP/src/ethernet/eth_rnet/ethkernel.cc: 66] EthernetKernel FindDevice: device dbus-system not found.
[2021-12-07 07:18:33.242936: Error /home/cycliodzzz/CodeProject/miniTCP/kernel.cc: 66] EthernetKernel FindDevice: device dbus-session not found.
[2021-12-07 07:18:33.242936: Error /home/cycliodzzz/CodeProject/miniTCP/src/ethernet/eth_rnet/ethkernel.cc: 66] EthernetKernel FindDevice: device dbus-session not found.
[2021-12-07 07:18:33.242967: Error /home/cycliodzzz/CodeProject/miniTCP/kernel.cc: 66] EthernetKernel FindDevice: device lo not found.
[2021-12-07 07:18:33.243081: Info /home/cycliodzzz/CodeProject/miniTCP/cc: 11] Timer: timer start.
[2021-12-07 07:18:33.243146: Info /home/cycliodzzz/CodeProject/miniTCP/ce.cc: 44] getLocalIP : choose 10.100.1.1
[2021-12-07 07:18:41.273717: Debug /home/cycliodzzz/CodeProject/miniTCP/device/veth4-3 receive 70 bytes, and the message is `6 12 13 14 63 68 70 72 74 device veth4-3 receive 70 bytes, and the message is `6 12 13 14 63 68 70 72 74 76 78 80 82 84 86 87 88 89 1545 3001 4184 5640 7096 8279 9735 11191 12374 13830 15000 all: 15000
[2021-12-07 07:19:23.314082: Info /home/cycliodzzz/CodeProject/miniTCP/net/device.cc: 44] getLocalIP : choose 10.100.1.1
[2021-12-07 07:19:23.358127: Debug /home/cycliodzzz/CodeProject/miniTCP/sport/socket_impl.cc: 136] SetupKeepalive() : new connection
[2021-12-07 07:19:23.358961: Debug /home/cycliodzzz/CodeProject/miniTCP/src/transport/socket_impl.cc: 136] SetupKeepalive() : new connection
[2021-12-07 07:20:04.406143: Info /home/cycliodzzz/CodeProject/miniTCP/net/device.cc: 44] getLocalIP : choose 10.100.1.1
[2021-12-07 07:20:04.455407: Debug /home/cycliodzzz/CodeProject/miniTCP/sport/socket_impl.cc: 136] SetupKeepalive() : new connection
[2021-12-07 07:20:04.470435: Debug /home/cycliodzzz/CodeProject/miniTCP/src/transport/socket_impl.cc: 136] SetupKeepalive() : new connection
cycliodzzz@ubuntu: ~/CodeProject/miniTCP$ # Use "--" to terminate the options and cycliodzzz@ubuntu: ~/CodeProject/miniTCP$
```

# Checkpoint 10

## 1. Topology

The topology and the IP configuration is the same as checkpoint4, please see checkpoint4 above.

## 2. Usage

To check out checkpoint9, please execute the following command in the directory of miniTCP in the terminal.

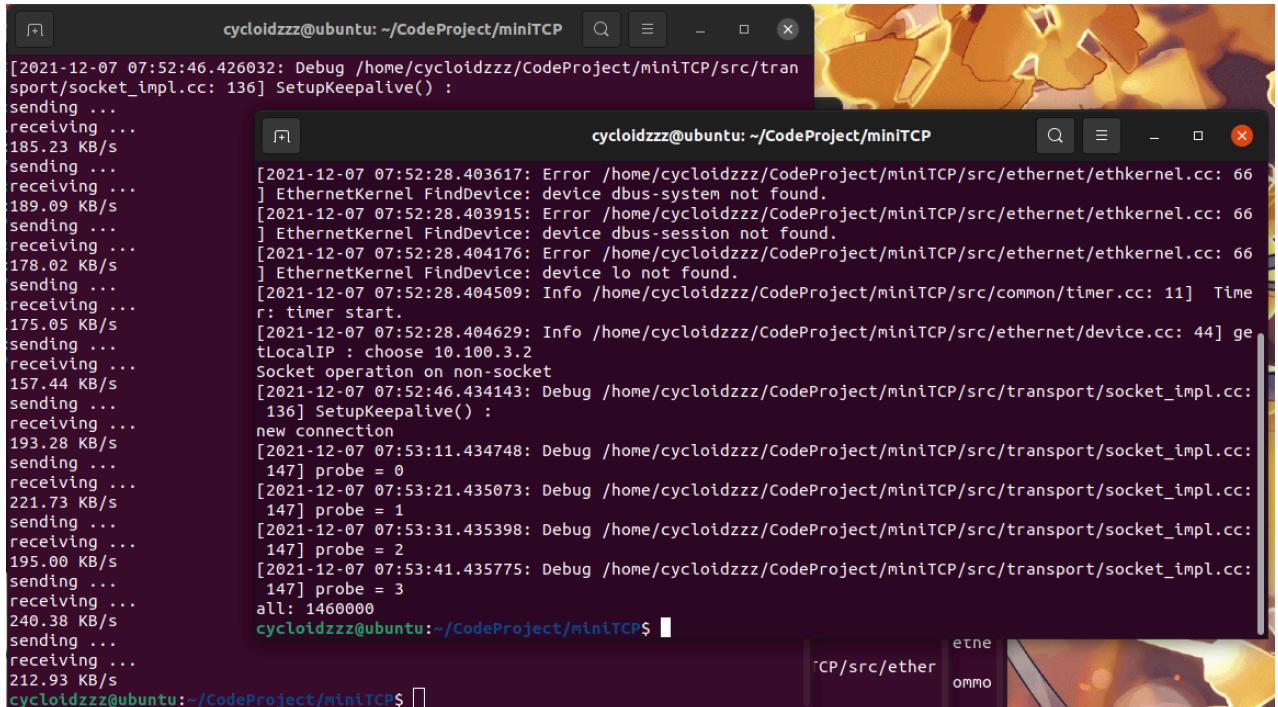
```
python3 script/check10.py -install
```

After checking out the checkpoint9, please execute the following command to delete the configuration generated by vnetUtils.

```
python3 script/check10.py -delete
```

## 3. Result

Only the result of the server and the client would be shown, here is the result of perf\_server and perf\_client.



The image shows two terminal windows side-by-side. Both windows have a purple header bar with the text "cycloidzzz@ubuntu: ~/CodeProject/miniTCP". The left terminal window displays the output of the perf\_server process, which includes various log messages about sending and receiving data at different rates (e.g., 185.23 KB/s, 189.09 KB/s, 178.02 KB/s, 175.05 KB/s, 157.44 KB/s, 193.28 KB/s, 221.73 KB/s, 195.00 KB/s, 240.38 KB/s, 212.93 KB/s) and setup keepalive operations. The right terminal window displays the output of the perf\_client process, which includes error messages from the ethkernel module indicating device dbus-system and dbus-session not found, and log messages from the common/timer.cc and device.cc files. Both terminals show the prompt "cycloidzzz@ubuntu:~/CodeProject/miniTCP\$".

