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 ..
```

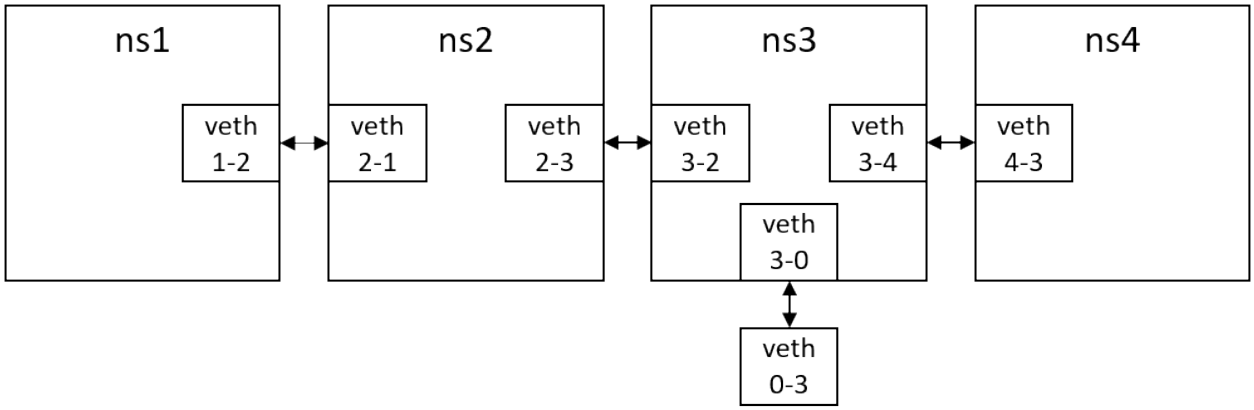
Code Lists

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
```

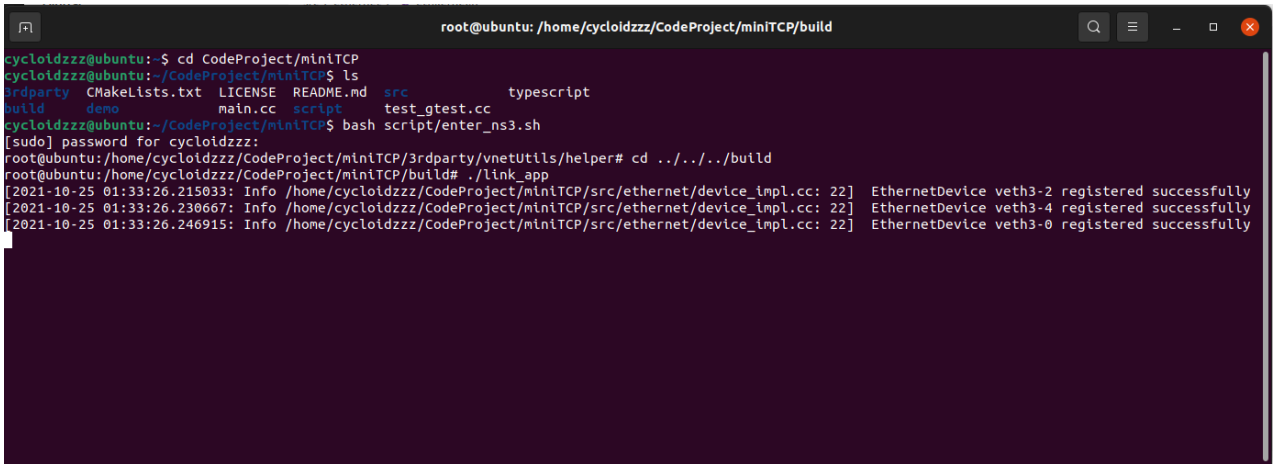
To go back to the **build directory in miniTCP** after executing the bash script , please run the following command in the terminal.

```
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.



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

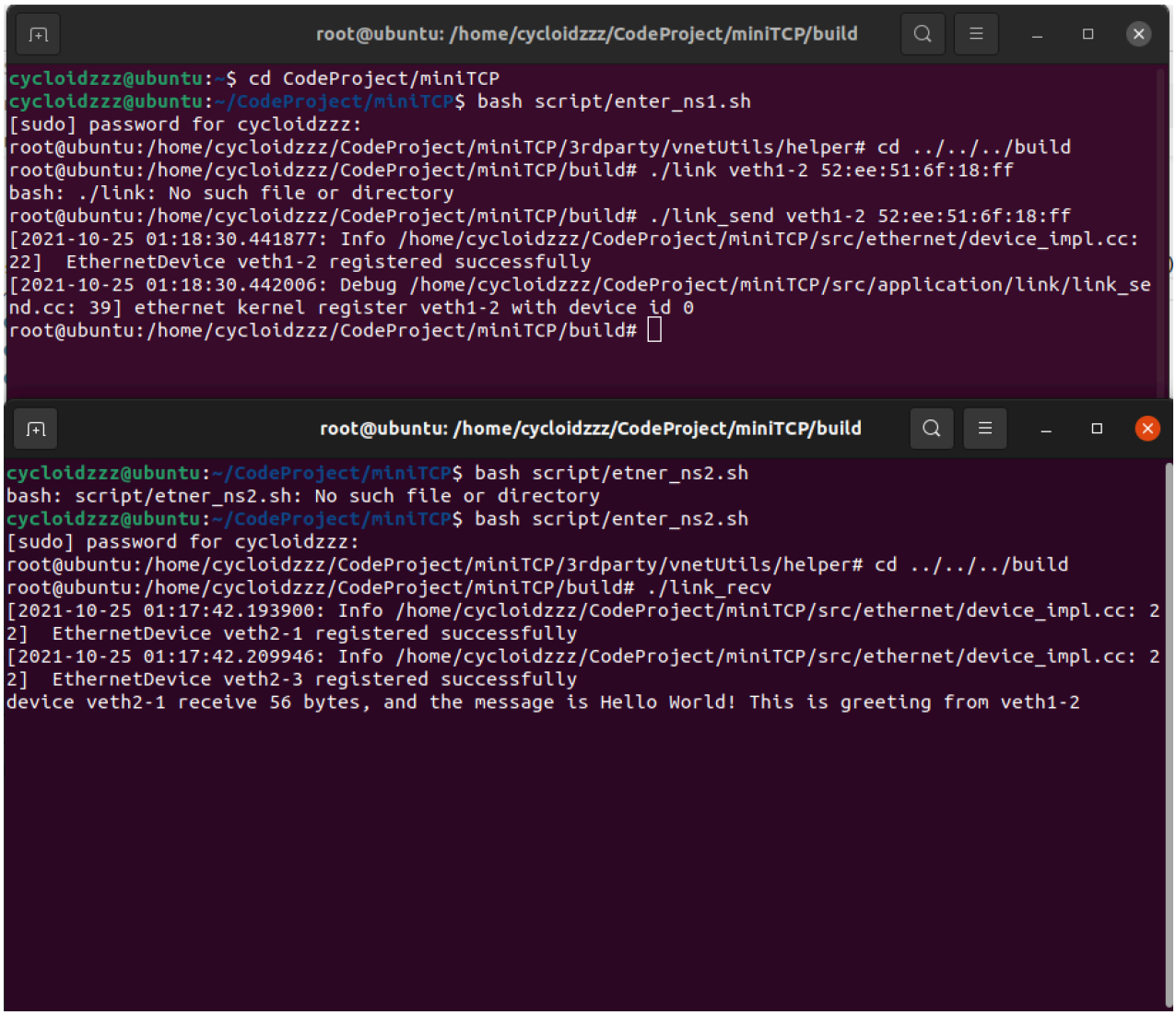
```
./link_recv
```

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.



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 onw IP address, if it matches or the receiver IP is an 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 mark as inreachable by setting it as 16 (the inreachable threshold) and the status of the entry will be marked as GARBAGE, a garbage collection will set set for this entry. If this entry is still not updated by the corresponding neighbor within 12 seconds, this entry will be remove 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 an serialized object. It is consisted 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 deserielize 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 covergence 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 inreachable.

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.

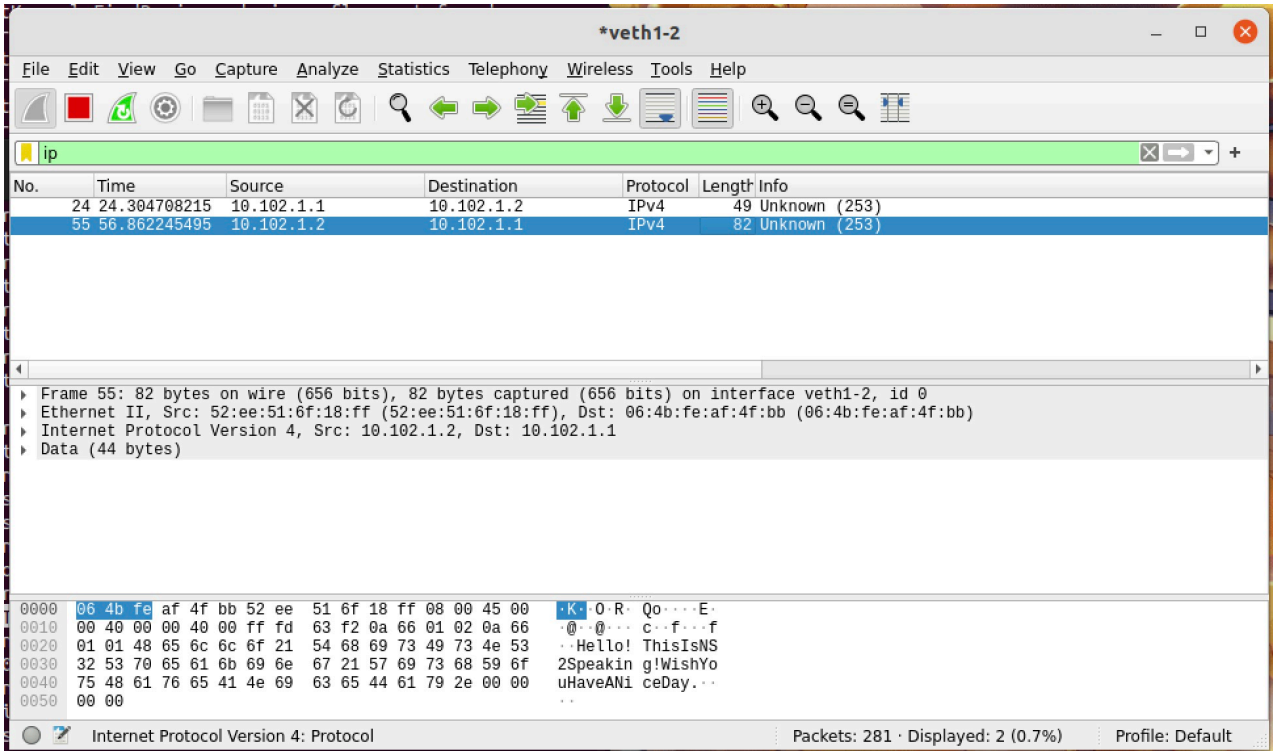
```
send 10.102.1.1 10.102.1.2 HelloWorld!
```

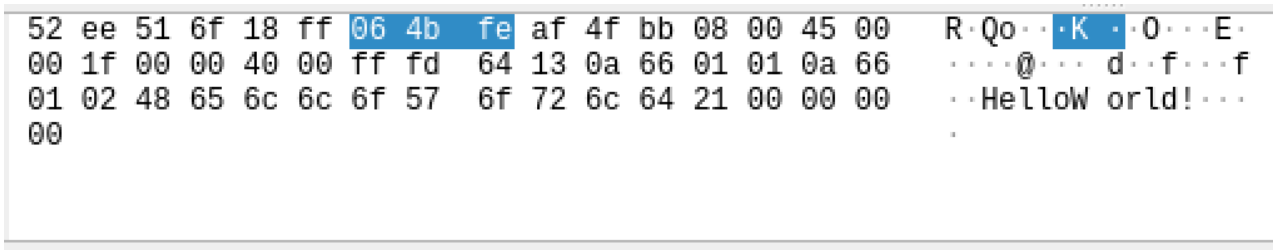
The terminal of ns1 will receive the message and print out the message.

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.





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.

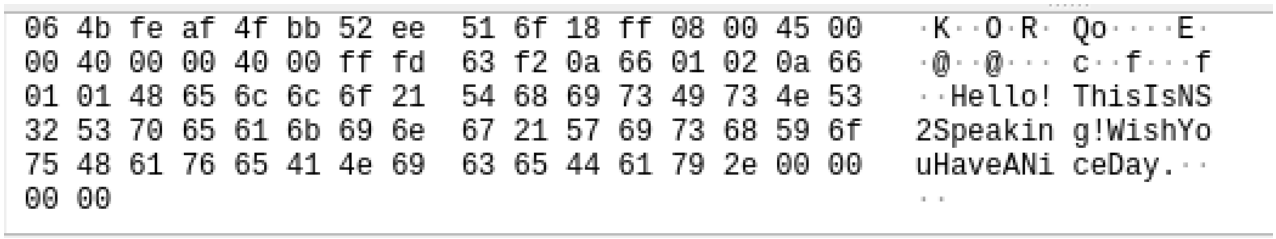
The tenth byte is 0xfd, which means the protocol running on the ip is 253 (an arbitrary number I use to send IP packets).

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!"



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.

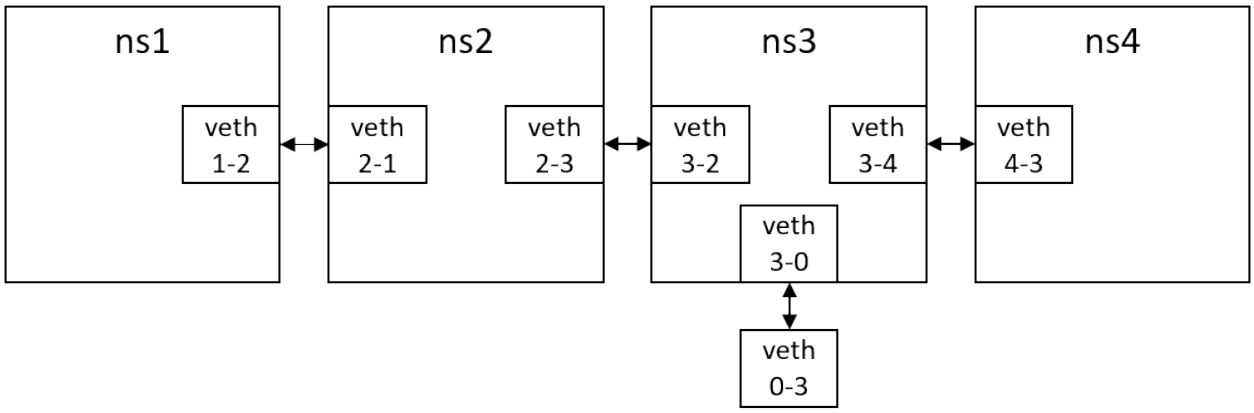
The seventeenth to the twentieth bytes are 0x0a660101, which means the destination IP address is 10.102.1.1.

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 to get the sudo privilege.

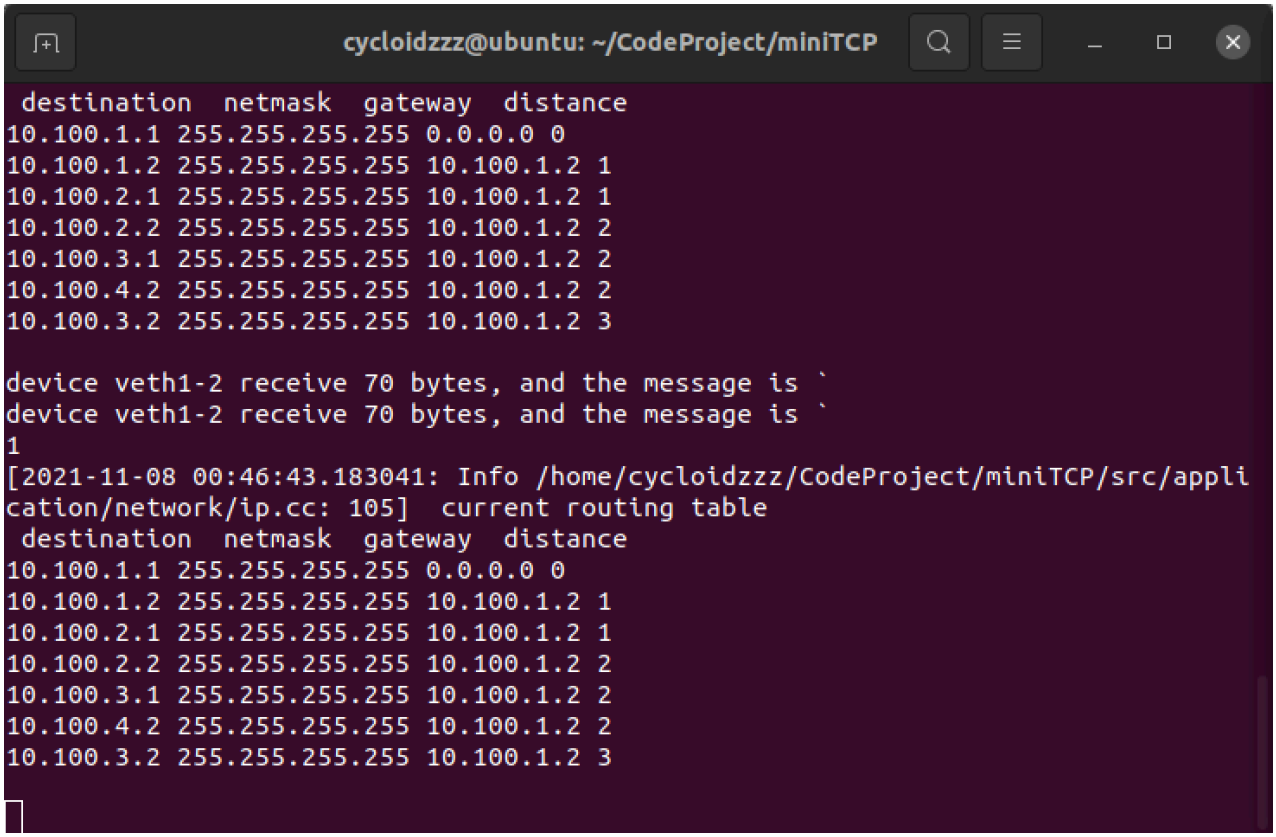
After executing the command, you may need to wait for several seconds to wait for the convergence of the routing algorithm.

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

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

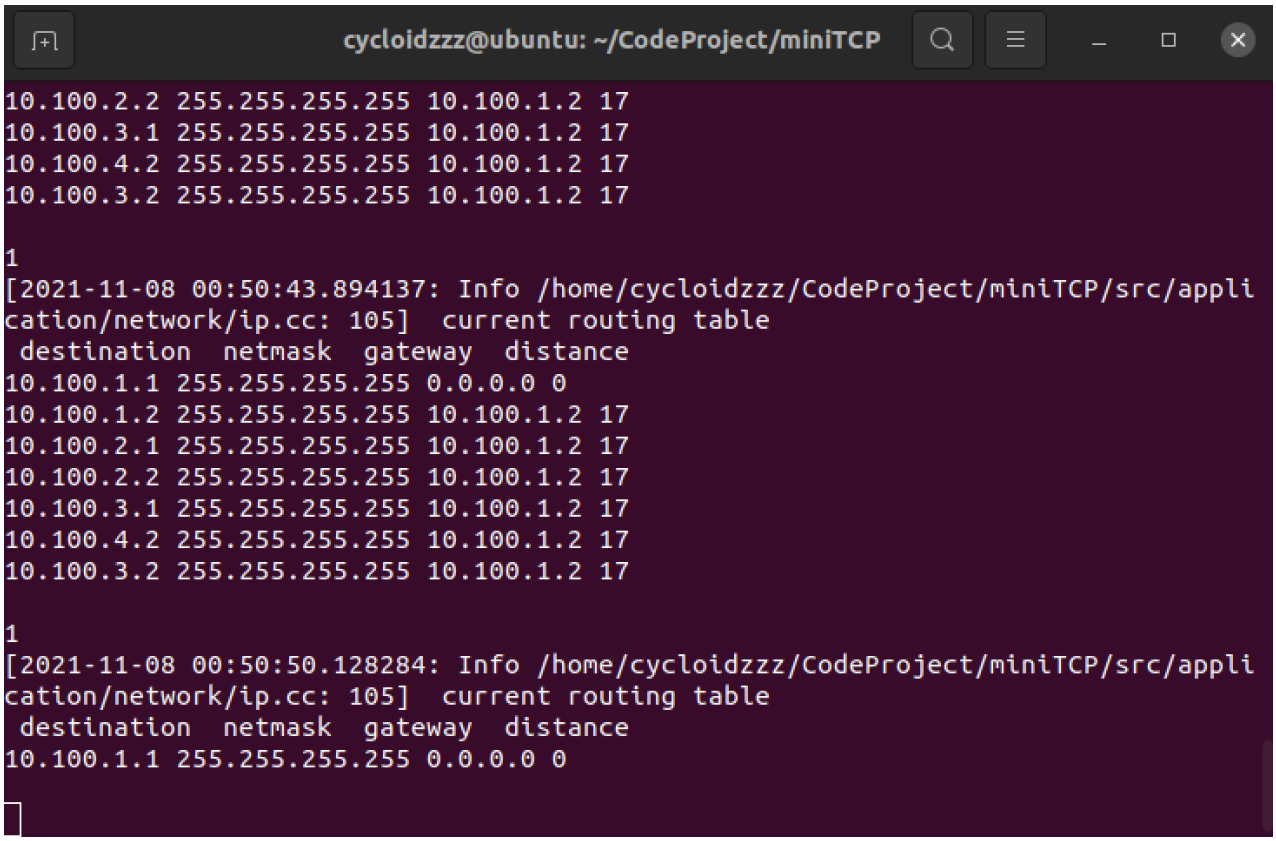
3. Result

(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.



(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.

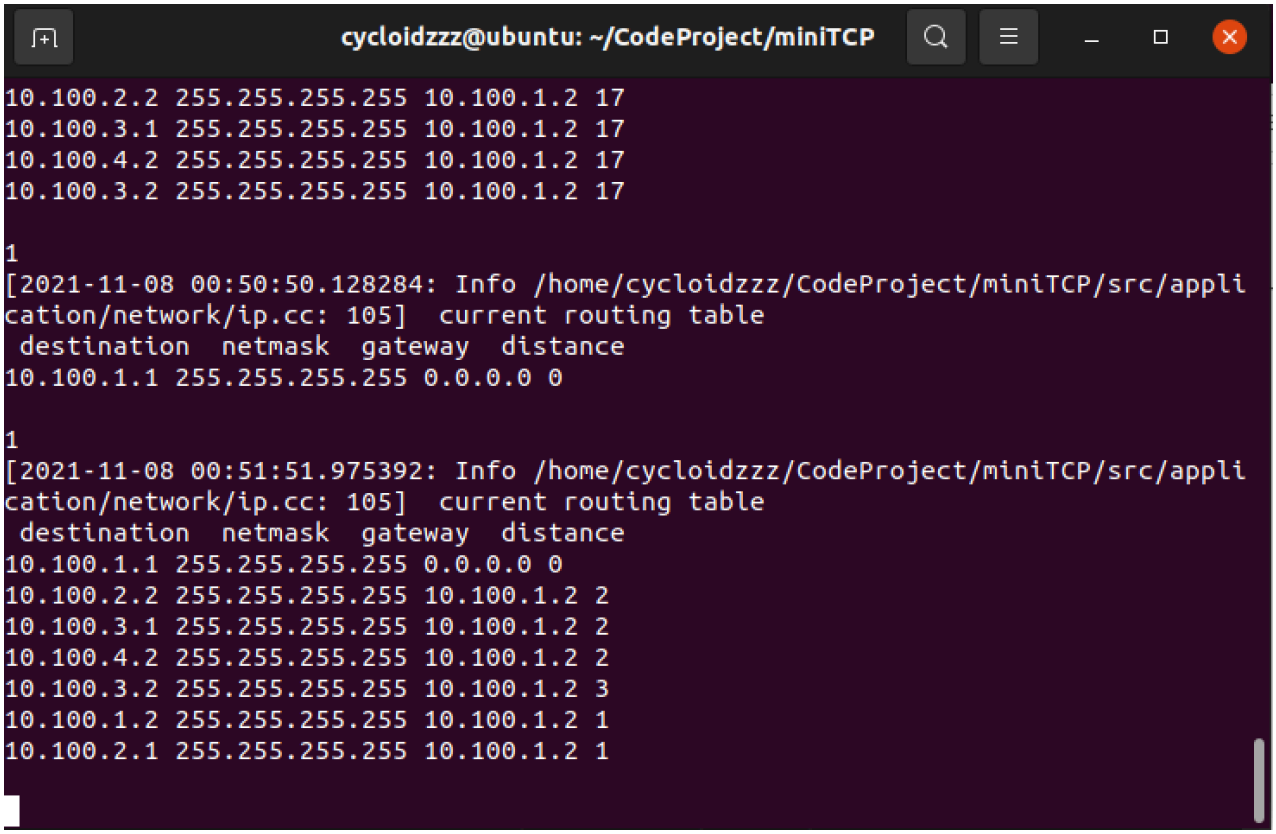
The routing table of NS1 after removing NS2. The ip addresses belongs to NS2, NS3, NS4 are all marked as inreachable (distance equals 17) and being deleted in the next round.



(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.



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
```

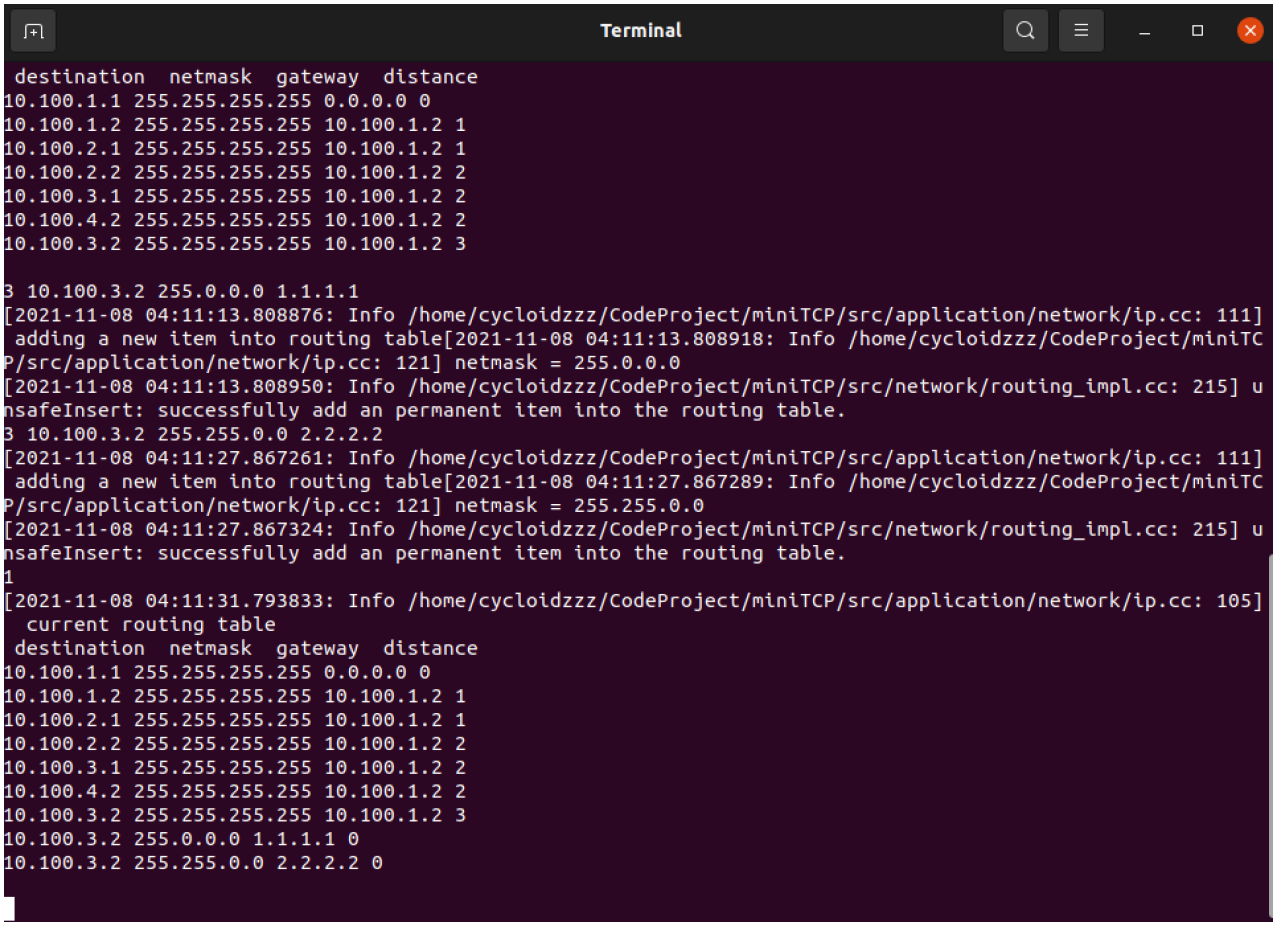
3. Result

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

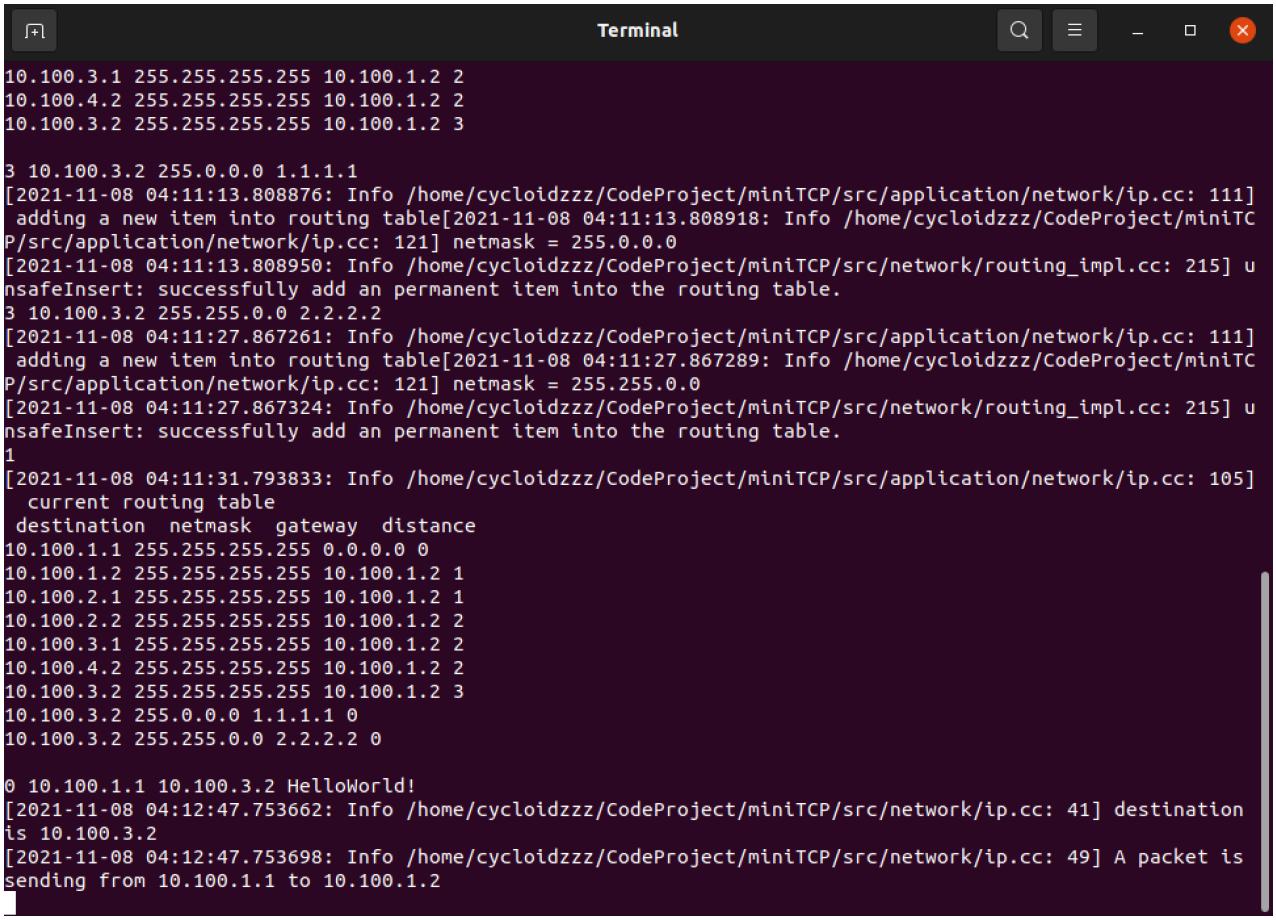
```
add 10.100.3.2 255.255.0.0 2.2.2.2
```



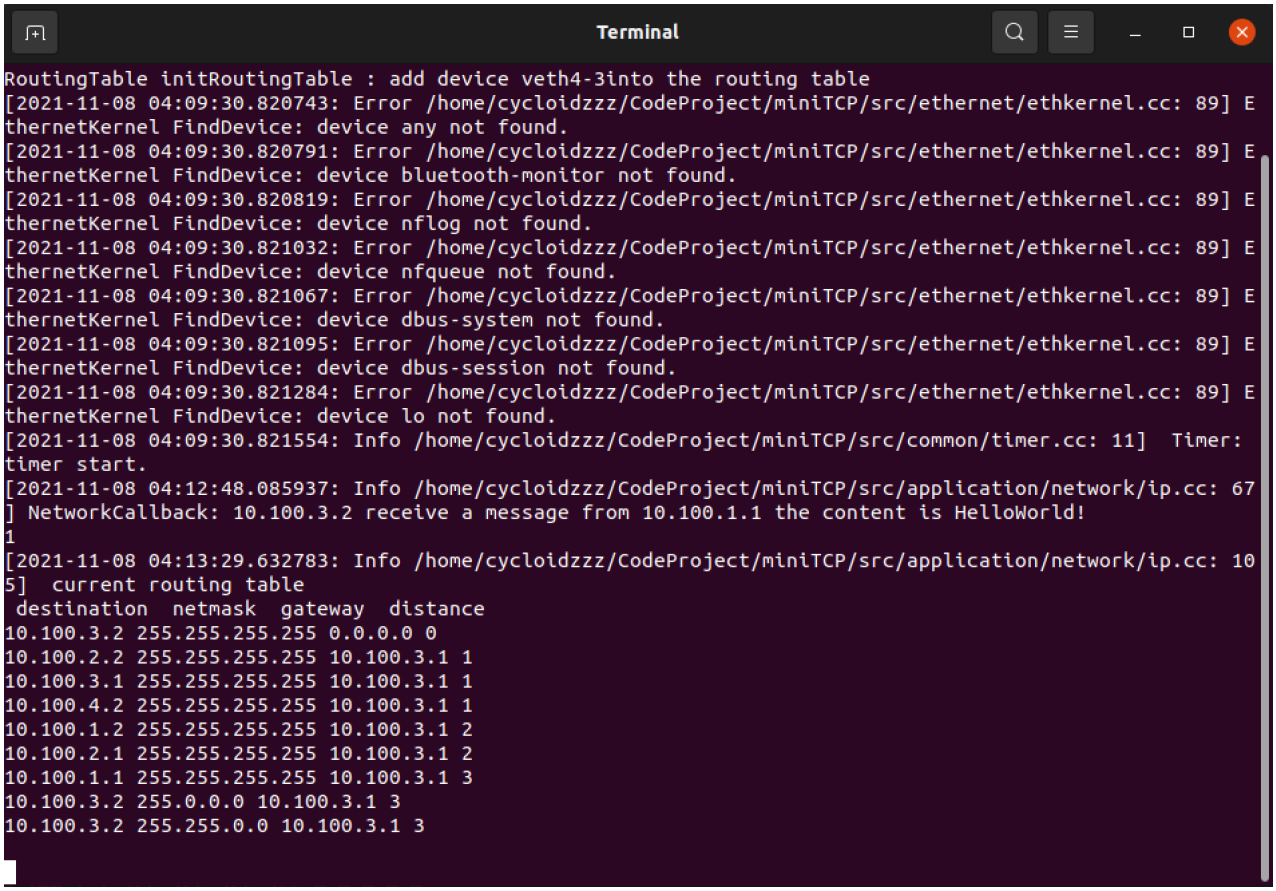
If the routing table fails to apply the longest prefix matching rule, then the device with IP address 10.100.3.2, which corresponds to NS4, would not receive the message from NS1.

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!
```



The only device (veth4-3) in NS4 can successfully receive the message from NS1as the following picture shows, which proves that our routing table faithfully follows the longest prefix matching rule.



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 nitify 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 find 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 direectly 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
```

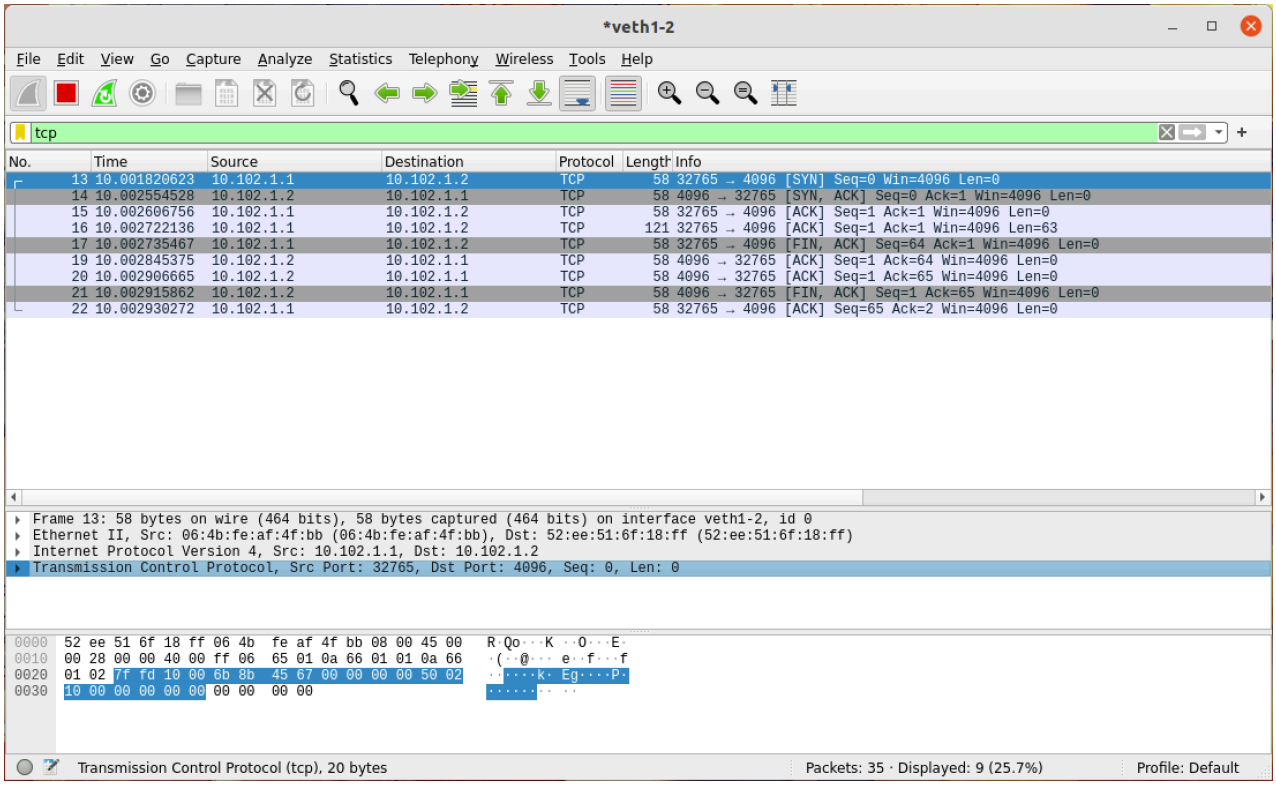
In order to capture packets from the network with network number NS (1 or 2), please execute the following command in the miniTCP directory **instead of the build directory**.

```
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.

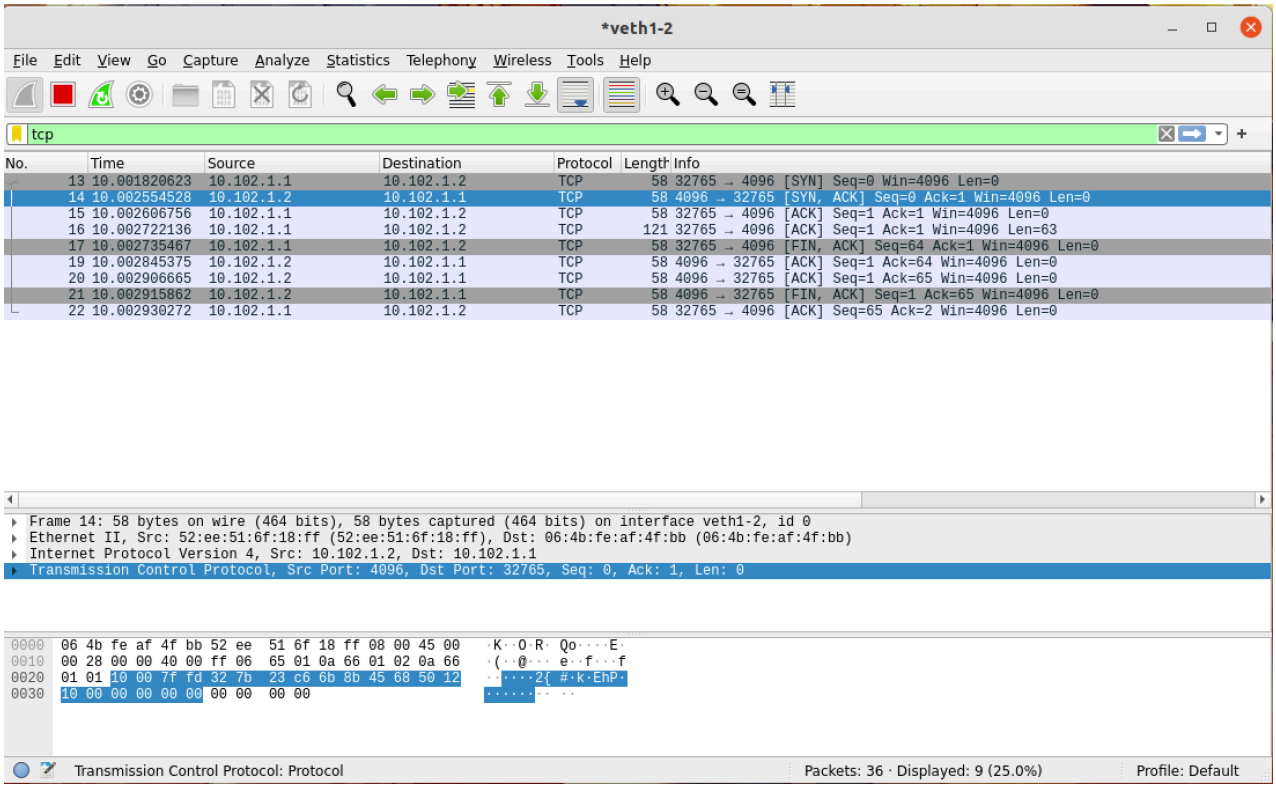
The next 4 bytes are 00 00 00 00, which stands for the ack number (in big endian) is 0x0.

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).

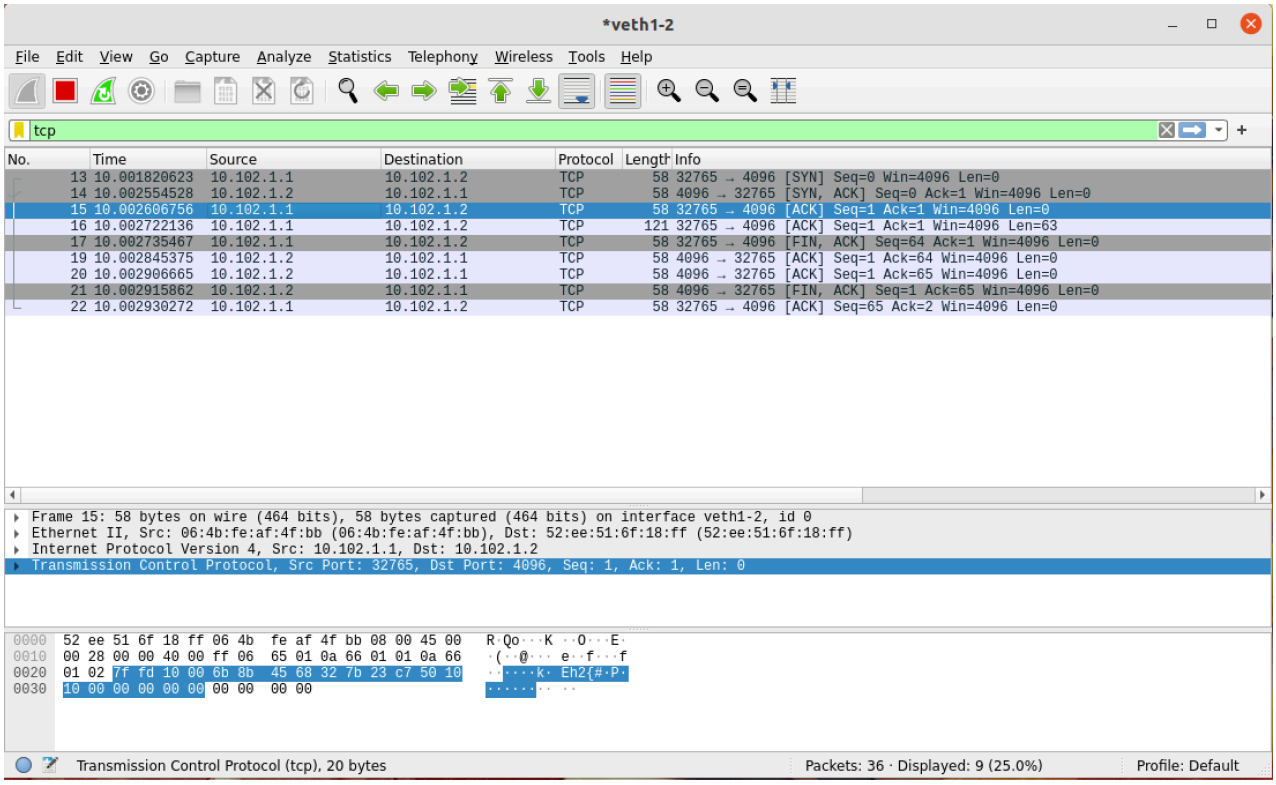
The next 4 bytes are 23 c6 6b 8b, which stands for the ack number of this TCP packet (in big endian).

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).

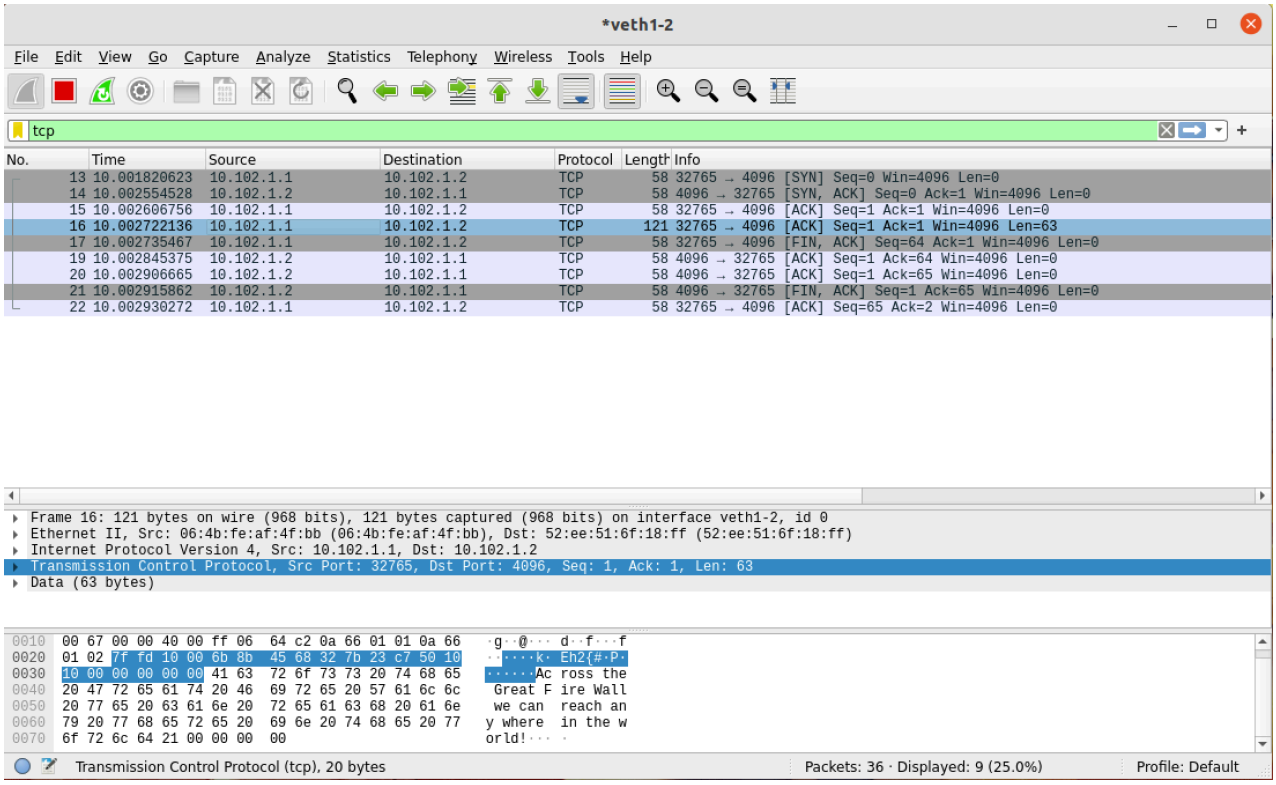
The next 4 bytes are 32, 7b, 23, c7, which stands for the ack number of this TCP packet (in big endian).

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).

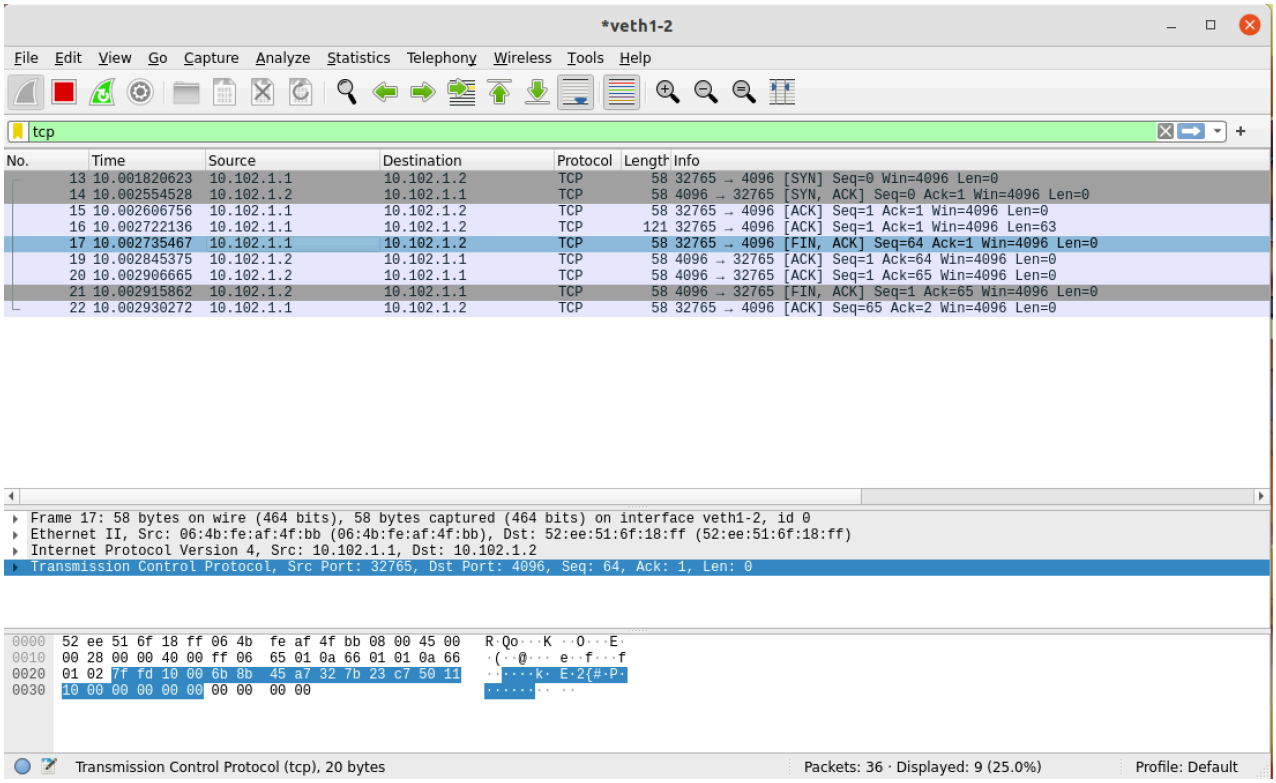
The next 4 bytes are 32, 7b, 23, c7, which stands for the ack number of this TCP packet (in big endian).

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).

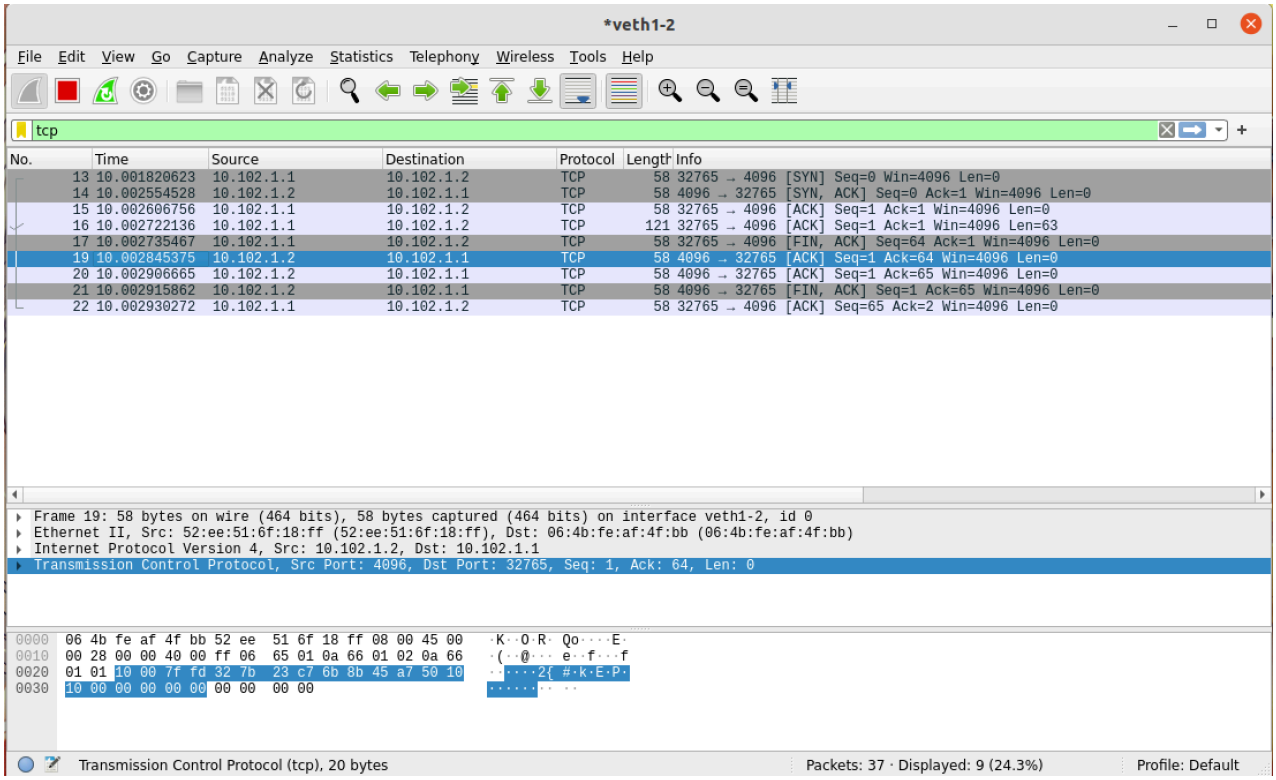
The next 4 bytes are 32, 7b, 23, c7, which stands for the ack number of this TCP packet (in big endian).

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).

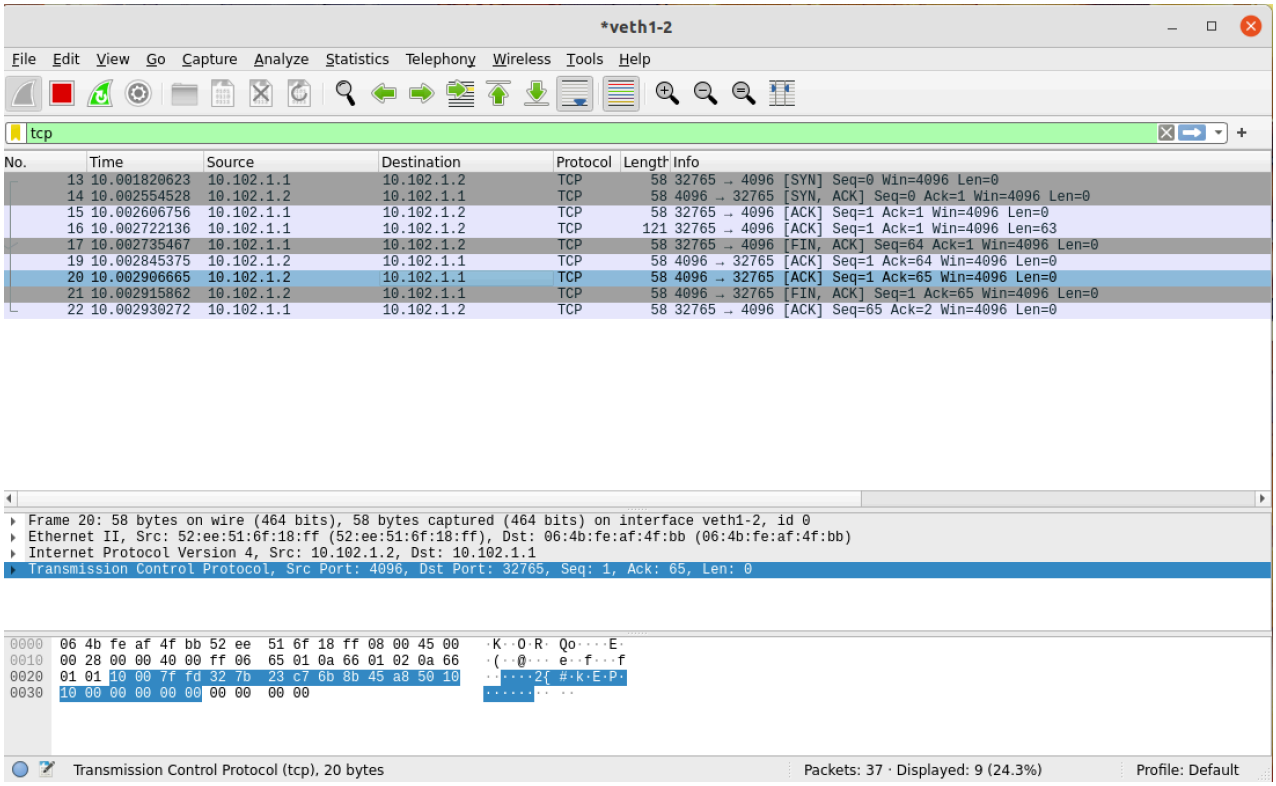
The next 4 bytes are 6b, 8b, 45, a7, which stands for the ack number of this TCP packet (in big endian).

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).

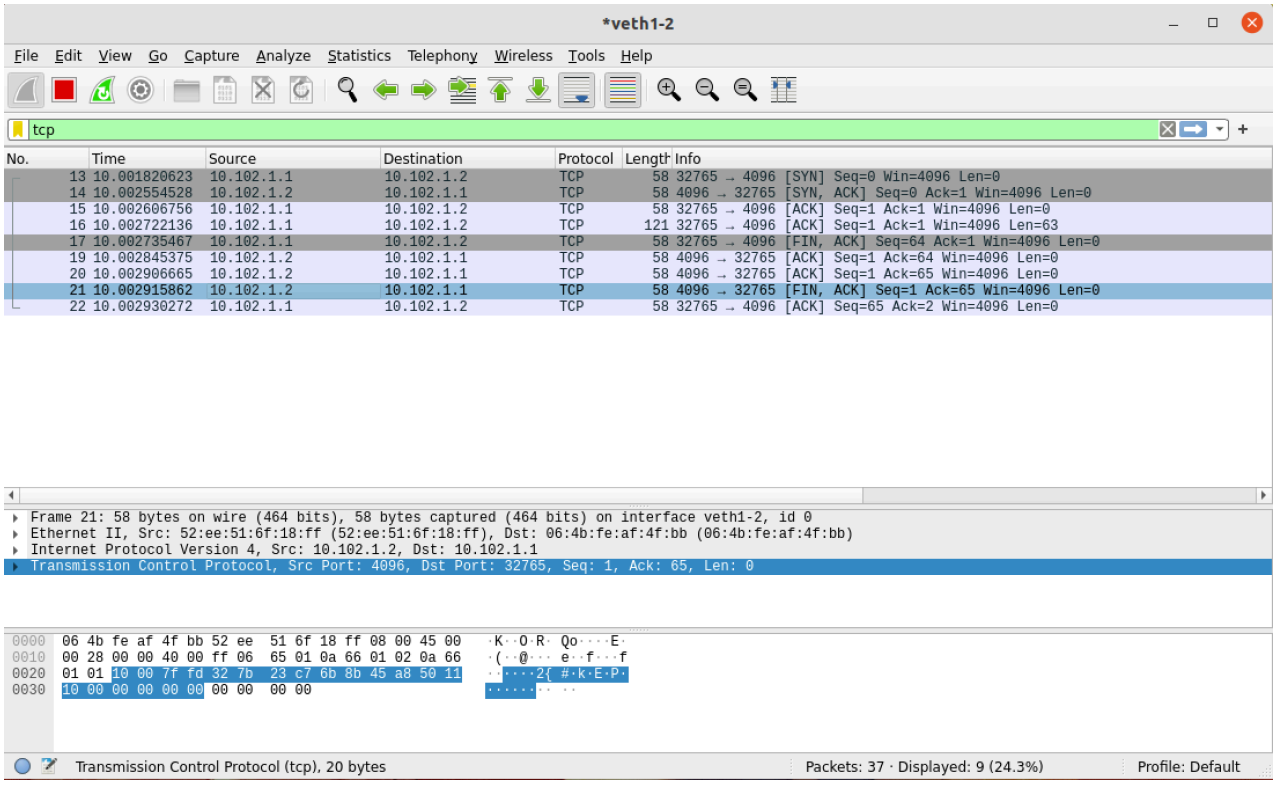
The next 4 bytes are 6b, 8b, 45,a8 which stands for the ack number of this TCP packet (in big endian).

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).

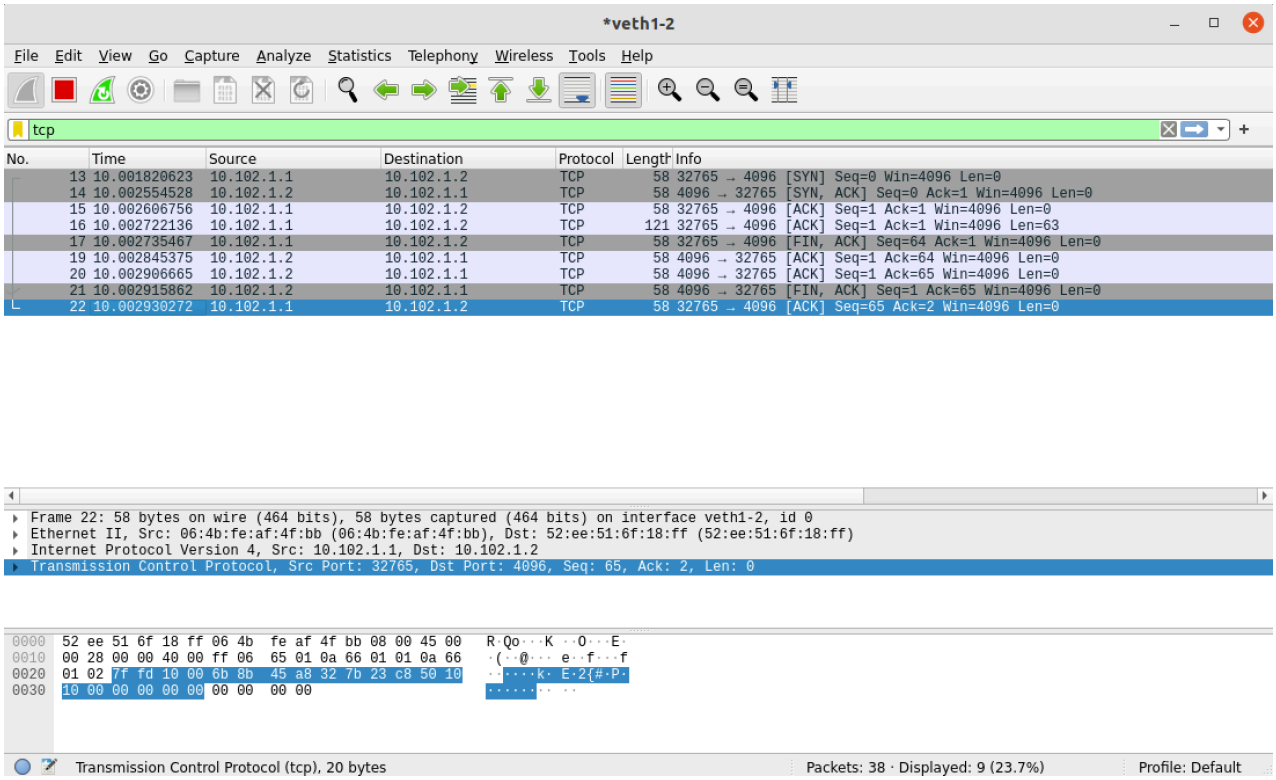
The next 4 bytes are 23, c7, 6b, 8b, which stands for the ack number of this TCP packet (in big endian).

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).



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
```

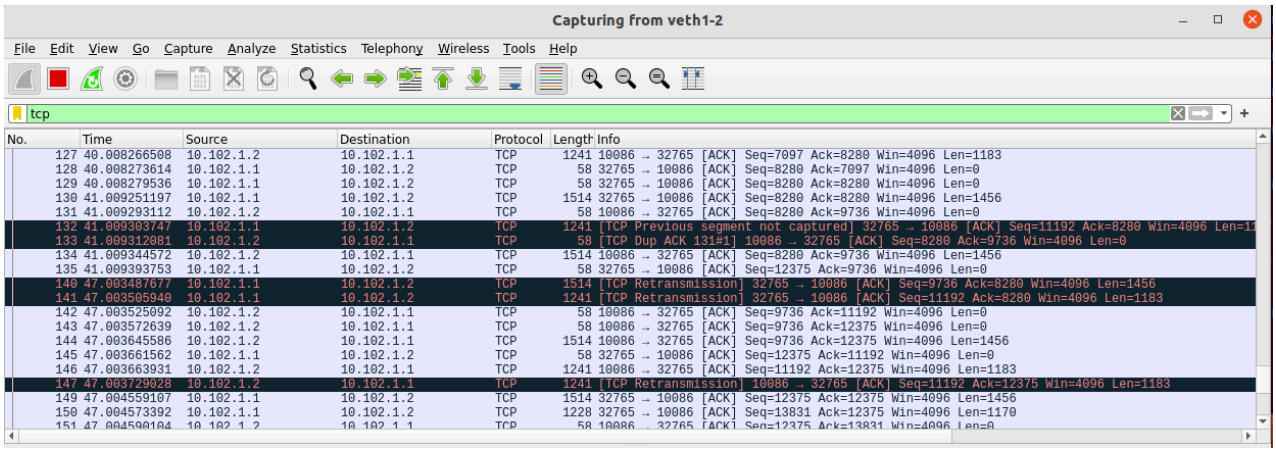
After checking out the checkpoint9, please execute the following command to delete the configuration generated by vnetUtils.

```
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.

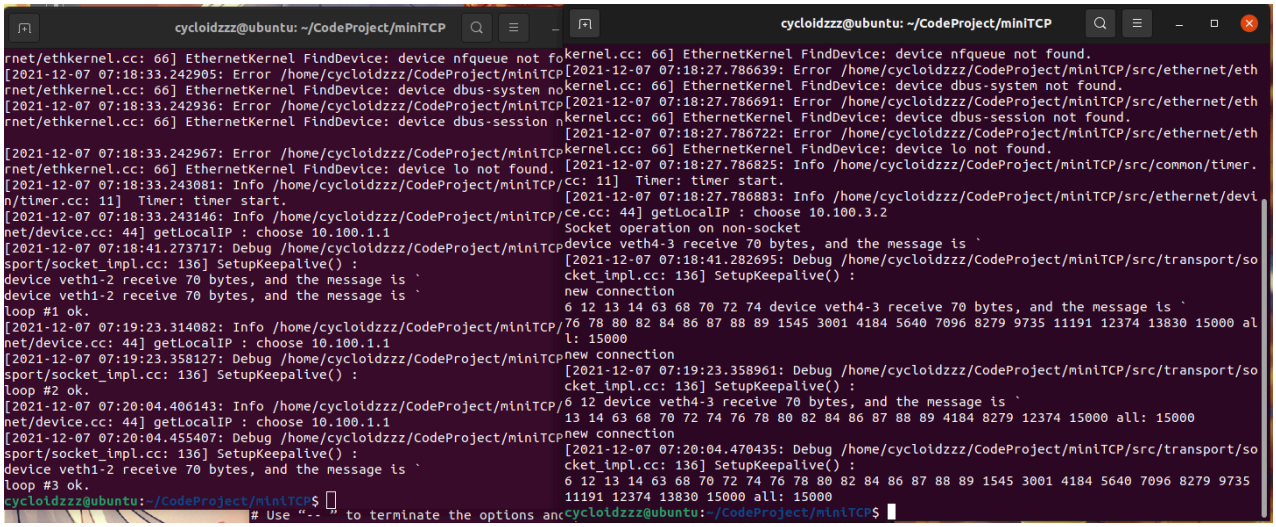
```
python3 script/check9.py -install
```

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.



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.