README for LAB #3

Yuyang Zhou
2000013061

1 Implemented

The features implemented are `PT1, PT2, PT3, PT4`. and this document include the answer of `WT5`, `CP7` ∼ `CP8`.

Due to some linking issues, I failed to hijack my socket into TA's test solution. So I could't give the answer for `CP9, CP10`.

2 WT5

(1), (2)

There are 2 sessions.

- Session 1, between `10.0.0.74:43120` and `112.27.207.221:80`, 12 segments in total.
- Session 2, between `10.0.0.74:43122` and `112.27.207.221:80`, 1652 segments in total.

(3)

The window size of Packet 86 is 43520.

The windows size depends on the number of remaining spaces in the buffer, which save the received, but not readed messages. It is cauculated using the size of receiving buffer, minus the length of received, but not readed messages.

3 Lower-level Code Arrangement

This part is copy from the `readme.pdf` from Lab2.

macro.h

define the macros used in the following programs.

- `swap(T &a, T &b)`

  * Swap two items

constant.h

define the constants used in the following programs.

type.h

define the specific types used in the following programs.

name2addr.h

find the Mac Address(actually hardware address) & IP Address of specific device

- `findMac(const char* device, uint8_t* mac_addr)`

  * find the Mac Address of specific device

  * @param device the name of the device request for Mac address

  * @param mac_addr the pointer to the memory used to save the Mac address

  * @return 0 on success, 1 on failure.

- `findIP(const char* device, struct in_addr &ip_addr)`

  * find the IP Address of specific device

  * @param device the name of the device request for Mac address

  * @param ip_addr the pointer to the memory used to save the IP address

  * @return 0 on success, -1 on failure.

device.h

Library supporting network device management.

- `checkDevice(const char* device)`

  * Check whether the device name exists in the network.

  * @param device Name of network device to check.

  * @return True on success , False on error.

- `addDevice(const char* device)`

  * Add a device to the library for sending / receiving packets .

  * @param device Name of network device to send / receive packet on .

  * @return A non - negative _device - ID_ on success , -1 on error .

- `findDevice(const char* device)`

  * Find a device added by 'addDevice'.

  * @param device Name of the network device .

  * @return A non - negative _device - ID_ on success , -1 if no such device was found .

In order to manage this, the code contains two structures `DeviceNode` and `DeviceManager`, and implements the following methods

- DeviceNode

    - `char* device_names` The name of the device
    - `pcap_t* receive_handler` The handler used to receive messages
    - `pcap_t* send_handler` The handler used to send messages
    - `frameReceiveCallback callback` The default link layer callback function
    - `IPPacketReceiveCallback callback` The default IP Layer callback function
    - `uint8_t mac_addr[8]` Mac address of the device
    - `struct in_addr ip_addr` IP address of the device
    - `int index` The index of the device
    - `bool isEqualDevice(const char* device)`

      * Check if the name of the device is equal to 'device'

      * @param device The name of device to check out

      * @return True on same, False on different
    - `void setCallback(frameReceiveCallback __callback__)`

      * Set up the link layer callback function of the device

      * @param ___callback___ the pointer of callback function to set up
    - `void setIPCallback(IPPacketReceiveCallback __callback__)`

      * Set up the IP layer callback function of the device

      * @param ___callback___ the pointer of callback function to set up
    - `int setDevice(const char* device)`

      * Initalize the deviceNode with name device

      * @param device The name of device used to set up

      * @return 0 on success, -1 on failure.

- DeviceManager

    - `DeviceNode** device_list` The pointer to the piece of memory, to save the pointer of DeviceNodes
    - `int device_count` The number of devices in the manager
    - `int device_bound` The maximum number of devices can be saved now
    - `DeviceNode* operator [](const int index)`

      * Return the pointer of the index-th DeviceNode

      * @param index The index of device to find
    - `int addDevice(const char* device)`

      * Add a device to the library for sending / receiving packets .

      * @param device Name of network device to send / receive packet on .

      * @return A non - negative _device - ID__ on success , -1 on error .

  - `findDevice(const char* device)`
    - * Find a device added by 'addDevice'.
    - * @param device Name of the network device .
    - * @return A non - negative _device - ID_ on success , -1 if no such device was found .
  - `count()`
    - * @return The number of devices in the manager.

packetio.h

Library supporting sending / receiving Ethernet II frames.

- `int sendFrame (const void * buf, int len, int ethtype, const void *destmac, int id)`

  * Encapsulate some data into an Ethernet II frame and send it .

  * @param buf Pointer to the payload .

  * @param len Length of the payload .

  * @param ethtype EtherType field value of this frame .

  * @param destmac MAC address of the destination .

  * @param id ID of the device (returned by ”addDevice”) to send on .

  * @return 0 on success , -1 on error .

- `int setFrameReceiveCallback(frameReceiveCallback callback, int id)`

  * Register a callback function to be called each time an Ethernet II frame was received .

  *@param callback the callback function.

  * @return 0 on success , -1 on error.

- `int LinkHandInPacket(struct pcap_pkthdr* pkt_header, const u_char* framebuf, int index)`

  * After receive a packet captureed on specific device, try to handle it using the default function, and print the raw message if the function is not found.

  * @param pkt_header the header of the packet captured.

  * @param framebuf the buffer of the packet captured.

  * @param index the index of the packet captured.

  * @return 0 on success , -1 on error.

- `int receiveAllFrame(int id, int frame_count)`

  * try to receive specific number of Ethernet frames from device ID id.

  * @param id The Index of device to receive the package.

* @param frame_count A number,-1 represents receiving until error occurs, 0-65535 represents the number of packet expected to receive.

* @return the number of packages received,

iptable.h

Library for a data structure which gives the mapping between IP address and the information about IP address.

We implemented following structure to maintain it.

- `IPTableNode`
  - `class T value` The value saved in the trie node.
  - `int child[4]` The index of the 4 childs of the node.
- `IPTable`
  - `IPTableNode<class T>* mem` the pointer to the memory, which save the nodes in the trie
  - `int node_released` the number of nodes in the trie
  - `int node_count` the maximum number of nodes mem can save.
  - `find(const uint32_t &addr)` * find if the information about IP address

    * @param addr the IP address to be checked

    * @return 1 on information exist, 0 on not found
  - `class T& operator [](const uint32_t &addr)`

    * find if the information about IP address, and set the piece of memory if the information is not found

    * @param addr the IP address to be checked

    * @return the information

routing.h

Library for a data structure which gives the mapping between IP address and the information about IP address.

We implemented following structure to maintain it.

- `RoutingTableNode`
  - `bool rule` if the node contains a routing rule
  - `std::pair<macAddress, int> value` the routing rule in (next hop address, next hop device index) format
  - `int child[2]` the index of child node.

- `RoutingTable`
  - `IPTableNode* mem` the pointer to the memory, which save the nodes in the trie
  - `int node_released` the number of nodes in the trie
  - `int node_count` the maximum number of nodes mem can save.
  - `void setNextHopMac(uint32_t dst, struct in_addr mask, std::pair<macAddress,int> value)`

    * set the given routing rule

    * @param dst the destination

    * @param mask the mask of the destination

    * @param value the (next hop address, next hop device) pair
  - `int queryNextHopMac(uint32_t dst, std::pair<macAddress,int> *value)`

    * @brief find the (next hop address, next hop device) pair according to the rule set above

    * @param dst the destination v

    * @param value the pointer of (next hop address, next hop device) pair

    * @return 0 on at least one matching rules, -1 on no matching tules.

mytime.h

Library for microsecond timer.

- `gettime()`

  * get the current time in microsecond(us)

  * @return the current time in microsecond(us)

arp.h

Library for my ARP-Like Routing algorithm.

- `int ARPCallback(const void* __buffer, const void* __mac_addr, int len, int index)`

  The callback function for my ARP-Like routing algorithm. The function is called if the device receive a ARP-Like packet.

  * @param ___buffer the buffer of the ARP-Like Header

  * @param ___mac_addr the device which send the ARP-Header

  * @param len the length of the buffer

  * @param index the index of the device which receive the ARP-Header.

  * @return 0 on success, 1 on failure.

- `int getNextHopMac(struct in_addr dst_ipaddr, void* nextHopMac, int &index)`

  * find the Mac address for the next hop

  * @param dst_ipaddr the ip address of the device which packet will be send.

  * @param nextHopMac the piece of memory to save the Mac address of the nextHop

  * @param index the index of device to retransmit the packet.

  * @return 0 on success, -1 on failure.

callback.h

Library for callback functions.

- `egLinkCallback(const void* __buffer, const void* __mac_addr, int len, int index))`

  * Example LinkLayer Callback function.

  * @param __buffer the message from the packet.

  * @param __mac_addr the mac address of the source of the packet.

  * @param len the length of __buffer

  * @param the index of device which receive the packet.

  * @return 0 on success, 1 on failure.

- `egLinkCallback(const void* __buffer, const void* __mac_addr, int len, int index, uint16_t proto))`

  * Link Layer Callback function used in 5-layer netstack model

  * @param __buffer the message from the packet.

  * @param __mac_addr the mac address of the source of the packet.

  * @param len the length of __buffer

  * @param the index of device which receive the packet.

  * @return 0 on success, 1 on failure.

  * @param proto the protocol used in the packet

- `egIPCallback(const void* __buffer, struct IPHeader header, int len, int index))`

  * Example IP Layer Callback function.

  * @param __buffer the message from the packet.

  * @param header the header of the IP Packet

  * @param len the length of __buffer

  * @param the index of device which receive the packet.

  * @return 0 on success, 1 on failure.

ip.h

Library supporting sending / receiving IP packets encapsulated in an Ethernet II frame .

- `int sendIPPacket(const struct in_addr src ,const struct in_addr dest ,int proto , const void * buf , int len)`

  * Send an IP packet to specified host .

  * @param src Source IP address .

  * @param dest Destination IP address .

  * @param proto Value of ‘protocol ‘field in IP header .

  * @param buf pointer to IP payload

  * @param len Length of IP payload

  * @return 0 on success , -1 on error .

- `void setIPPacketReceiveCallback(IPPacketReceiveCallback callback, int index)`

  * Register a callback function to be called each time an IP packet was received .

  * @param callback The callback function .

  * @return 0 on success , -1 on error .

- `int IPHandInPacket(const void* __buffer, int len)`

  * Handle the IPPacket, retransmit it if the destination of the packet is not the device, and call the callback function if it is the destination

  * @param ___buffer raw IPPacket including information and header

  * @param len the length of ___buffer .

  * @return 0 on success , -1 on error .

- `int setRoutingTable(const struct in_addr dest, const struct in_addr mask, const void* nextHopMac, const char* device)`

  * Manully add an item to routing table . Useful when talking with real Linux machines . * * @param dest The destination IP prefix .

  * @param mask The subnet mask of the destination IP prefix .

  * @param nextHopMAC MAC address of the next hop .

  * @param device Name of device to send packets on .

  * @return 0 on success , -1 on error

4 Higher-level code arrangement

I'll just give a brief for each file, because this piece of code is even longer the the former two labs.

- `packetio.h` I change the packet receiveing rule in it. In Lab2 I'll accept the routing packets send by me, thus gives duplicated routing packets. and In this time, I'll ignore them. This helps to reduce the number of packets transmitted, and prevent the dead-lock in the network system.
- `IOBuffer.h` This gives a quick implementation of buffer area for each socket to use. Each socket will have a waiting buffer for messages preparing to send; a send buffer save the message which have been sent but not receive acknowledgement; and a receive buffer save the message received, but not read by user.

  I use a ring-like queue to save the message, and the size of the queue grows up is the message length gets longer. It can support up to $2^{21} - 1$ byte of messages, and gives the support of retransmission.
- `moniter.h` This is the moniter, which gives the backend monitering of the whole TCP/IP stack, and the way to start the backend moniter. The backend will be lauched if any interface in `getaddrinfo` is called, and after being launched, It will try to receive possible packets from each device in `5ms` intervals, and retransmit un-acknowledged messages in `50ms` intervals. Also, when the moniter is launched, it will give the support of `Ctrl+C` for a force quit.
- `portmanager.h` This gives the map between (ip address, port) pair and the socket id. Also when using `connect()`, it can gives an empty port index assigned to the socket, or report that it doesn't exist.
- `retrans.h` This gives `retransQ`, which provide retransmit management. It saved the length of messages sent in every `50ms` intervals. If the message doesn't get the acknowledgement after `50ms`, it will report the moniter and retransmit it. And if retransmit fails for `62`times (3100ms, respectively), it will be dropped, and warn the user.
- `portal.h` This implements the socket using `socketNode`. And we also implement `socketManager`, which manage all the socket we have already setup, and gives the mapping between socket index and socket node. The socket index is shifted by a constant(`0x12345 = 74565`), avoid collusion with other frequent-used fildes.

  In the `soceketNode`, we save the state of the socket(`binded, connected, hang, state`); the source address and the dest address of the connection(`addr, dst_addr`); the packages for the connections which is waiting for accept(`trap_ip_frame,trap_tcp_frame,trap_count,trap_bound`); several buffers for waiting, sending, receiving buffers(`fwait, fsend, freceive`); the informations about sequence numbers and acknowledgements(`syn, ack`); and the informations retransmission queue(`retransQ`);

  For `read, write, close` operations, `rfc` gives the full implementing details about how to handle it, so we omit the details. For `socket`, actually it doesn't play an important role init, so we can just save the values, and ignore it in further implemention; For `bind, listen, connect`, we can found the implemention in `OPEN`, it discussed passive and active connections, which can be regarded as `listen` and `connect`. so we can also omit this.

For `accept`, we can found it in the documentation, but we found that when dealing with `SEGMENT ARRIVES` in `LISTEN` state, after the segment is accepted, it actually return a segment which means that the device accept the connection. So we can move the part from `packetHandle()` to `accept()`, and only save the informations in `trap_ip_frame,trap_tcp_frame,trap_count` when it arrives.

For `SEGMENT ARRIVE`, we can found it in the documentation. Except `urgent bit, secutiry check`,we can ignore it because it's a simplified version, but all others should be work. So just copy it is ok. And we should also remove the message in fsend if we receive an acknowledgement.

At last, for `retransmit()`, we use a ring-like queue `retransQ`. we remove the message that waiting for too-long, ignore the message if it ha been acknowledged, but not pushed into the queue yet.

- `socket.h` This gives all the required interfaces(but I failed to hijack them). Except `getaddrinfo, freeaddrinfo`, every other interface will interact with `socket_manager`.
- `callback.h` I modified the IP Layer callback functions. If the packet is send to the server, then it will be handle to the `socketNode`. Otherwise the packet is not transmited to the device, and we'll try to find the path to the destinations, then retransmit it.
- `tcp.h` This provide the service for sending TCP packet to a specific desinations. It will find the pseudo TCP header, calculate the checksum in TCP protocol, filling the blanks in the header, and hand it into IP layer.
- `lab2.cpp` This is the old `main.cpp` in Lab2, gives 4 different test function for IP layer.
- `router.cpp` This is the router in Lab3, which can gives message transfer, but could not build any TCP connection with it.
- `server.cpp` This is the server in Lab3, which can gives message transfer, and can also build TCP connection with another server.In order to make the output results clearer and easier to understand, I implement a simple terminal, which provide interaction with users and the backend. The usage can be found using `'help'`.
- `multi-threading` We have 2 different threading running at the same time, after the backend is lauched. The one is the frontend, which gives the interaction with the user. The other one is the backend, which moniter the whole stack, provide routing, message handling, message sending and so on. We use semaphores to prevent data preemption: only one thread is allowed to change the data in socket at the same time.

5 Usage

The executable file is generated using `cmake`.

```
mkdir build && cd build
cmake ..
make
```

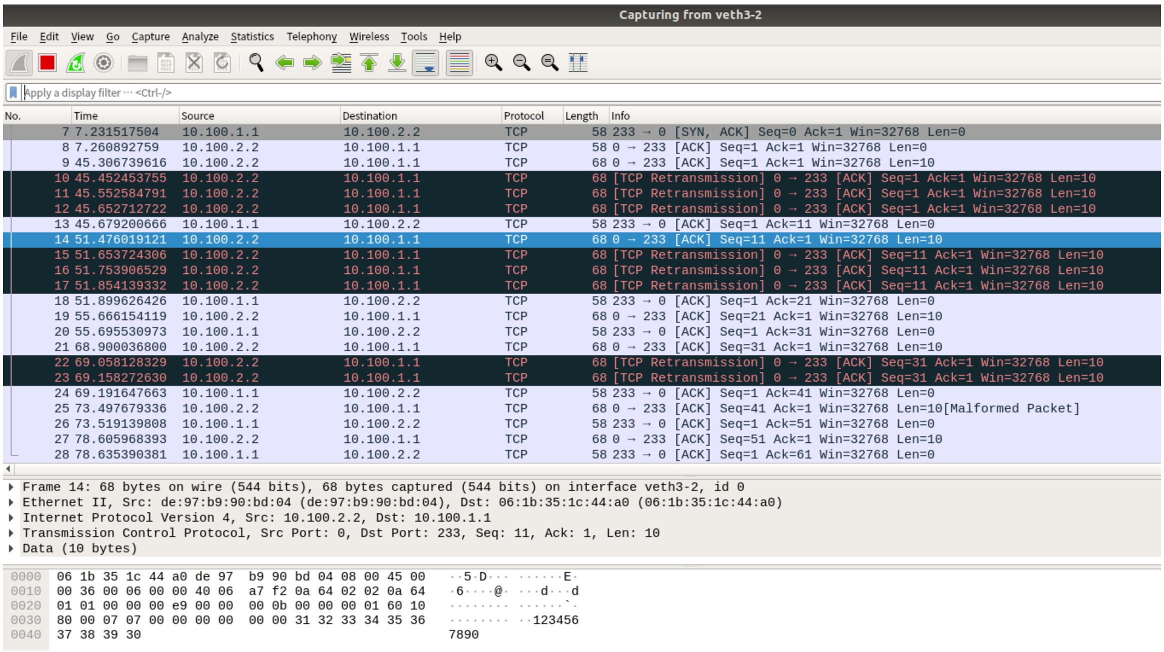
Then executable file is saved in `/build/src/router` and `/build/src/server`, and you can use the following command to run it.

```
sudo ./src/router
sudo ./src/server
```

6 CheckPoint

CP7

The following image shows the 14th frame we transmit in `CP8-3`. the packet trace is provided in `checkpoints`


Capturing from veth3-2
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.     Time          Source        Destination   Protocol  Length  Info
      7 7.231517504   10.100.1.1    10.100.2.2    TCP       58 233 → 0 [SYN, ACK] Seq=0 Ack=1 Win=32768 Len=0
      8 7.260892759   10.100.2.2    10.100.1.1    TCP       58 0 → 233 [ACK] Seq=1 Ack=1 Win=32768 Len=0
      9 45.306739616  10.100.2.2    10.100.1.1    TCP       68 0 → 233 [ACK] Seq=1 Ack=1 Win=32768 Len=10
     10 45.452453755  10.100.2.2    10.100.1.1    TCP       68 [TCP Retransmission] 0 → 233 [ACK] Seq=1 Ack=1 Win=32768 Len=10
     11 45.552584791  10.100.2.2    10.100.1.1    TCP       68 [TCP Retransmission] 0 → 233 [ACK] Seq=1 Ack=1 Win=32768 Len=10
     12 45.652712722  10.100.2.2    10.100.1.1    TCP       68 [TCP Retransmission] 0 → 233 [ACK] Seq=1 Ack=1 Win=32768 Len=10
     13 45.679200666  10.100.1.1    10.100.2.2    TCP       58 233 → 0 [ACK] Seq=1 Ack=11 Win=32768 Len=0
     14 51.476019121  10.100.2.2    10.100.1.1    TCP       68 0 → 233 [ACK] Seq=11 Ack=1 Win=32768 Len=10
     15 51.653724306  10.100.2.2    10.100.1.1    TCP       68 [TCP Retransmission] 0 → 233 [ACK] Seq=11 Ack=1 Win=32768 Len=10
     16 51.753906529  10.100.2.2    10.100.1.1    TCP       68 [TCP Retransmission] 0 → 233 [ACK] Seq=11 Ack=1 Win=32768 Len=10
     17 51.854139332  10.100.2.2    10.100.1.1    TCP       68 [TCP Retransmission] 0 → 233 [ACK] Seq=11 Ack=1 Win=32768 Len=10
     18 51.899626426  10.100.1.1    10.100.2.2    TCP       58 233 → 0 [ACK] Seq=1 Ack=21 Win=32768 Len=0
     19 55.666154119  10.100.2.2    10.100.1.1    TCP       68 0 → 233 [ACK] Seq=21 Ack=1 Win=32768 Len=10
     20 55.695530973  10.100.1.1    10.100.2.2    TCP       58 233 → 0 [ACK] Seq=1 Ack=31 Win=32768 Len=0
     21 68.900036800  10.100.2.2    10.100.1.1    TCP       68 0 → 233 [ACK] Seq=31 Ack=1 Win=32768 Len=10
     22 69.058128329  10.100.2.2    10.100.1.1    TCP       68 [TCP Retransmission] 0 → 233 [ACK] Seq=31 Ack=1 Win=32768 Len=10
     23 69.158272630  10.100.2.2    10.100.1.1    TCP       68 [TCP Retransmission] 0 → 233 [ACK] Seq=31 Ack=1 Win=32768 Len=10
     24 69.191647663  10.100.1.1    10.100.2.2    TCP       58 233 → 0 [ACK] Seq=1 Ack=41 Win=32768 Len=0
     25 73.497679336  10.100.2.2    10.100.1.1    TCP       68 0 → 233 [ACK] Seq=41 Ack=1 Win=32768 Len=10[Malformed Packet]
     26 73.519139808  10.100.1.1    10.100.2.2    TCP       58 233 → 0 [ACK] Seq=1 Ack=51 Win=32768 Len=0
     27 78.605968393  10.100.2.2    10.100.1.1    TCP       68 0 → 233 [ACK] Seq=51 Ack=1 Win=32768 Len=10
     28 78.635390381  10.100.1.1    10.100.2.2    TCP       58 233 → 0 [ACK] Seq=1 Ack=61 Win=32768 Len=0
```

```
▸ Frame 14: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface veth3-2, id 0
▸ Ethernet II, Src: de:97:b9:90:bd:04 (de:97:b9:90:bd:04), Dst: 06:1b:35:1c:44:a0 (06:1b:35:1c:44:a0)
▸ Internet Protocol Version 4, Src: 10.100.2.2, Dst: 10.100.1.1
▸ Transmission Control Protocol, Src Port: 0, Dst Port: 233, Seq: 11, Ack: 1, Len: 10
▸ Data (10 bytes)
```

```
0000  06 1b 35 1c 44 a0 de 97  b9 90 bd 04 08 00 45 00   ··5·D··· ······E·
0010  00 36 00 06 00 00 40 06  a7 f2 0a 64 02 02 0a 64   ·6····@· ···d···d
0020  01 01 00 00 00 e9 00 00  00 0b 00 00 00 01 60 10   ········ ······`·
0030  80 00 07 07 00 00 00 00  00 00 31 32 33 34 35 36   ········ ··123456
0040  37 38 39 30                                        7890
```

Figure 1: CP-7: The frame we send on device `ns3`

The TCP Header locates in bit `0x0022` ∼ `0x39`. and the meaning is shown in the following table.

- `0x0022` ∼ `0x0023`: the source port of the connection.
- `0x0024` ∼ `0x0025`: the destination port of the connection.
- `0x0026` ∼ `0x0029`: sequence number of the segment

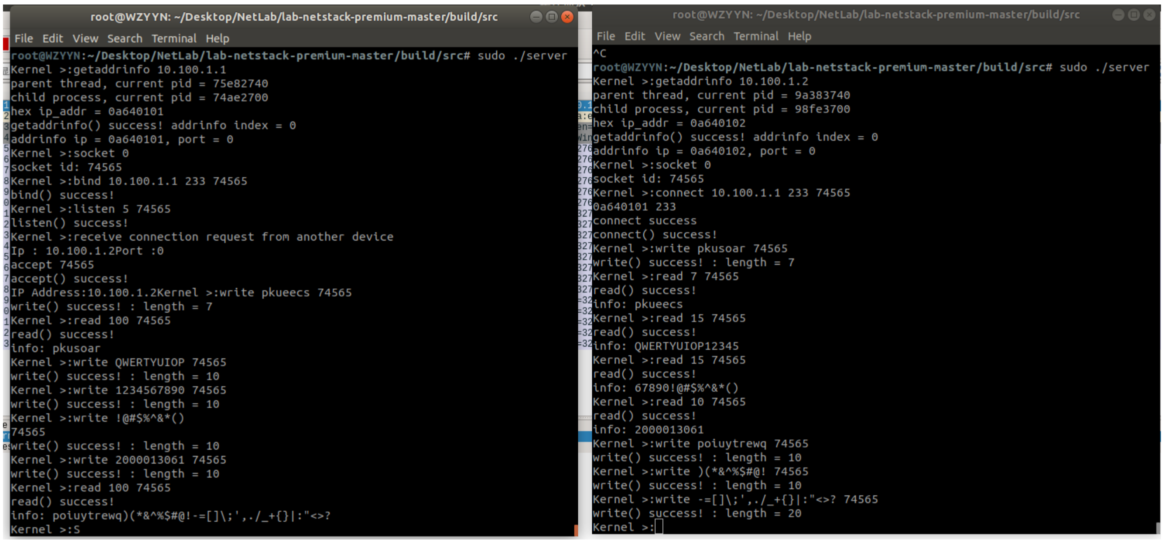
- `0x002a` ∼ `0x002d`: the acknowlegde number that the source device want to ack. used if `ACK` bit is on.
- higher 4 `0x002e`: the number of long(32 bit) contained in TCP header, 6 in this packet
- lower 6 bits `0x002f`: the control bits, which are `URG, ACK, PSH, RST, SYN, FIN` from high to low respectively.
- `0x0030` ∼ `0x31`: the window size of the prefered receiving packet size if the receiver send message to sender.
- `0x0032` ∼ `0x0033`: the checksum of the header, including length and ip address.
- `0x0034` ∼ `0x0035`: the urgent pointer, point to the data which is required to process as quick as possible. Used only `URG` is set.
- `0x0036` ∼ `0x0039`: the options in TCP header.

CP8

Set UP the virtual network saved in `network1.txt`. and then launch 2 bashs, which located in `ns1,ns2`. In each bash, launch the server and you can interact with the server.

In the below image is my testing trace with server. First I build the connection between two servers, which have the port address `10.100.1.1:233` and `10.100.1.2:0`. The I send messages in both direction, and it shows that it can support a Half Duplex communications between servers.

The packet captured in `veth1-2` at `ns1` while we testing the communications is provided in `checkpoints/test1-connection.txt`, which can be load using wireshark. The screen shot of the interact trace can also be found in `checkpoints`.

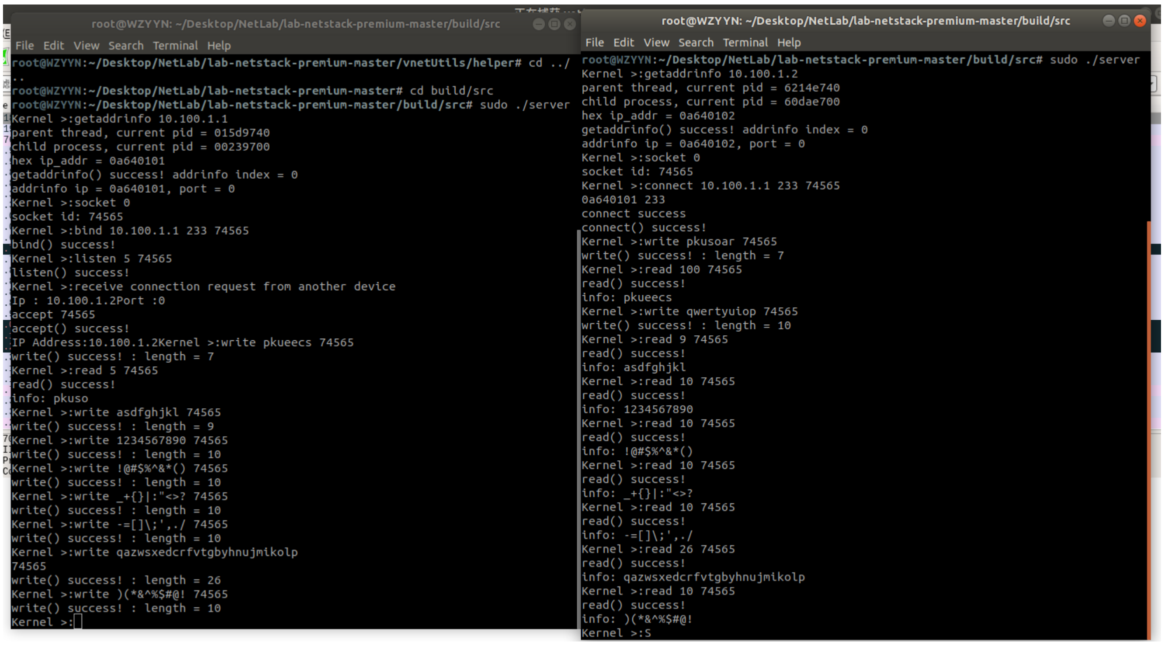

root@WZYYN: ~/Desktop/NetLab/lab-netstack-premium-master/build/src
File Edit View Search Terminal Help
root@WZYYN: ~/Desktop/NetLab/lab-netstack-premium-master/build/src
File Edit View Search Terminal Help

```
root@WZYYN:~/Desktop/NetLab/lab-netstack-premium-master/build/src# sudo ./server
Kernel >:getaddrinfo 10.100.1.1
parent thread, current pid = 75e82740
child process, current pid = 74ae2700
hex ip_addr = 0a640101
getaddrinfo() success! addrinfo index = 0
addrinfo ip = 0a640101, port = 0
Kernel >:socket 0
socket id: 74565
Kernel >:bind 10.100.1.1 233 74565
bind() success!
Kernel >:listen 5 74565
listen() success!
Kernel >:receive connection request from another device
Ip : 10.100.1.2Port :0
accept 74565
accept() success!
IP Address:10.100.1.2Kernel >:write pkueecs 74565
write() success! : length = 7
Kernel >:read 100 74565
read() success!
info: pkusoar
Kernel >:write QWERTYUIOP 74565
write() success! : length = 10
Kernel >:write 1234567890 74565
write() success! : length = 10
Kernel >:write !@#$%^&*()
74565
write() success! : length = 10
Kernel >:write 2000013061 74565
write() success! : length = 10
Kernel >:read 100 74565
read() success!
info: poiuytrewq)(*&^%$#@!-=[]\;',./_+{}|:"<>?
Kernel >:S
```

```
^C
root@WZYYN:~/Desktop/NetLab/lab-netstack-premium-master/build/src# sudo ./server
Kernel >:getaddrinfo 10.100.1.2
parent thread, current pid = 9a383740
child process, current pid = 98fe3700
hex ip_addr = 0a640102
getaddrinfo() success! addrinfo index = 0
addrinfo ip = 0a640102, port = 0
Kernel >:socket 0
socket id: 74565
Kernel >:connect 10.100.1.1 233 74565
0a640101 233
connect success
connect() success!
Kernel >:write pkusoar 74565
write() success! : length = 7
Kernel >:read 7 74565
read() success!
info: pkueecs
Kernel >:read 15 74565
read() success!
info: QWERTYUIOP12345
Kernel >:read 15 74565
read() success!
info: 67890!@#$%^&*()
Kernel >:read 10 74565
read() success!
info: 2000013061
Kernel >:write poiuytrewq 74565
write() success! : length = 10
Kernel >:write )(*&^%$#@! 74565
write() success! : length = 10
Kernel >:write -=[]\;',./_+{}|:"<>? 74565
write() success! : length = 20
Kernel >:
```

Figure 2: CP8-1: the connection is currectly set up.

If we set packet drop using `tc qdisc add dev veth2-1 root netem loss 50%` in `ns2` after the connection is set up, Then the server can correctly detect the packet loss if the message is not acknowledged. After the packet loss is set up, I send 7 different messages from `ns1` to `ns2`; one of them have exactly `1` retransmission, and the other one have `3` retransmission. The screen shot of the interact is shown below and the packet captured in `veth2-1` is provided in `checkpoints/test2-retransmission.txt`

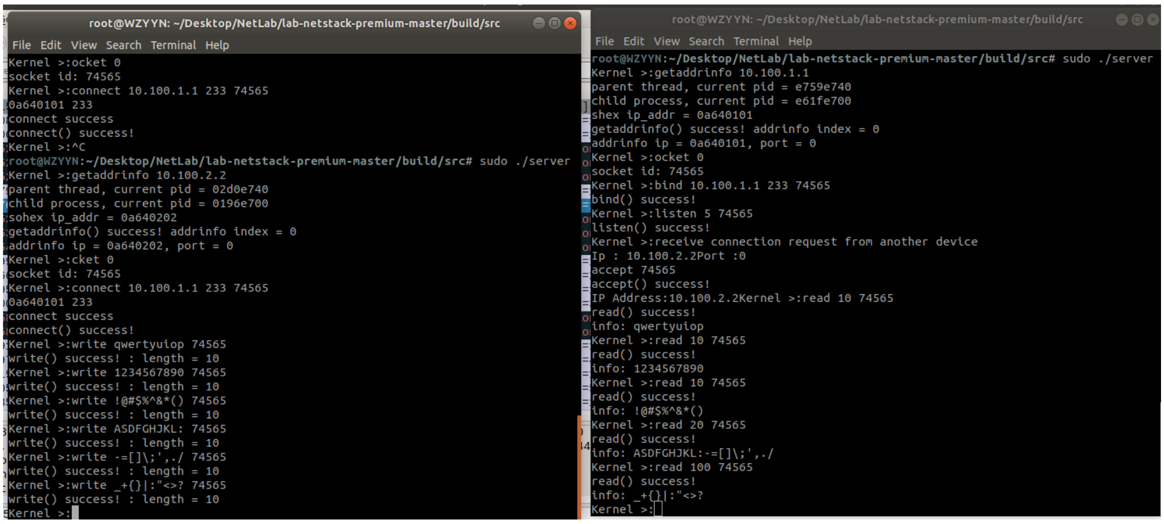

root@WZYYN: ~/Desktop/NetLab/lab-netstack-premium-master/build/src
File Edit View Search Terminal Help
root@WZYYN: ~/Desktop/NetLab/lab-netstack-premium-master/build/src
File Edit View Search Terminal Help

```
root@WZYYN:~/Desktop/NetLab/lab-netstack-premium-master/vnetUtils/helper# cd ../
..
root@WZYYN:~/Desktop/NetLab/lab-netstack-premium-master# cd build/src
root@WZYYN:~/Desktop/NetLab/lab-netstack-premium-master/build/src# sudo ./server
Kernel >:getaddrinfo 10.100.1.1
parent thread, current pid = 015d9740
child process, current pid = 00239700
hex ip_addr = 0a640101
getaddrinfo() success! addrinfo index = 0
addrinfo ip = 0a640101, port = 0
Kernel >:socket 0
socket id: 74565
Kernel >:bind 10.100.1.1 233 74565
bind() success!
Kernel >:listen 5 74565
listen() success!
Kernel >:receive connection request from another device
Ip : 10.100.1.2Port :0
accept 74565
accept() success!
IP Address:10.100.1.2Kernel >:write pkueecs 74565
write() success! : length = 7
Kernel >:read 5 74565
read() success!
info: pkuso
Kernel >:write asdfghjkl 74565
write() success! : length = 9
Kernel >:write 1234567890 74565
write() success! : length = 10
Kernel >:write !@#$%^&*() 74565
write() success! : length = 10
Kernel >:write _+{}|:"<>? 74565
write() success! : length = 10
Kernel >:write -=[]\;',./ 74565
write() success! : length = 10
Kernel >:write qazwsxedcrfvtgbyhnujmikolp
74565
write() success! : length = 26
Kernel >:write )(*&^%$#@! 74565
write() success! : length = 10
Kernel >:
```

```
root@WZYYN:~/Desktop/NetLab/lab-netstack-premium-master/build/src# sudo ./server
Kernel >:getaddrinfo 10.100.1.2
parent thread, current pid = 6214e740
child process, current pid = 60dae700
hex ip_addr = 0a640102
getaddrinfo() success! addrinfo index = 0
addrinfo ip = 0a640102, port = 0
Kernel >:socket 0
socket id: 74565
Kernel >:connect 10.100.1.1 233 74565
0a640101 233
connect success
connect() success!
Kernel >:write pkusoar 74565
write() success! : length = 7
Kernel >:read 100 74565
read() success!
info: pkueecs
Kernel >:write qwertyuiop 74565
write() success! : length = 10
Kernel >:read 9 74565
read() success!
info: asdfghjkl
Kernel >:read 10 74565
read() success!
info: 1234567890
Kernel >:read 10 74565
read() success!
info: !@#$%^&*()
Kernel >:read 10 74565
read() success!
info: _+{}|:"<>?
Kernel >:read 10 74565
read() success!
info: -=[]\;',./
Kernel >:read 26 74565
read() success!
info: qazwsxedcrfvtgbyhnujmikolp
Kernel >:read 10 74565
read() success!
info: )(*&^%$#@!
Kernel >:S
```

Figure 3: CP8-2: retransmit the message if ACK is blocked.

Also I test is the message is blocked. I run `server` in `ns1`,`ns3`, and `router` in `ns2`. After connection is set up, I use `tc qdisc add dev veth2-1 root netem loss 50%` to set packet drop, and send 6 different message to `ns1`. Almost every one have retransmission happened, and they are received by `ns1` in correct order. The screen shot of the interact is shown below and the packet captured in `veth3-2` is provided in `checkpoints/test3-retransmission.txt`


root@WZYYN: ~/Desktop/NetLab/lab-netstack-premium-master/build/src
File Edit View Search Terminal Help
root@WZYYN: ~/Desktop/NetLab/lab-netstack-premium-master/build/src
File Edit View Search Terminal Help

```
Kernel >:ocket 0
socket id: 74565
Kernel >:connect 10.100.1.1 233 74565
0a640101 233
connect success
connect() success!
Kernel >:^C
root@WZYYN:~/Desktop/NetLab/lab-netstack-premium-master/build/src# sudo ./server
Kernel >:getaddrinfo 10.100.2.2
parent thread, current pid = 02d0e740
child process, current pid = 0196e700
sohex ip_addr = 0a640202
getaddrinfo() success! addrinfo index = 0
addrinfo ip = 0a640202, port = 0
Kernel >:cket 0
socket id: 74565
Kernel >:connect 10.100.1.1 233 74565
0a640101 233
connect success
connect() success!
Kernel >:write qwertyuiop 74565
write() success! : length = 10
Kernel >:write 1234567890 74565
write() success! : length = 10
Kernel >:write !@#$%^&*() 74565
write() success! : length = 10
Kernel >:write ASDFGHJKL: 74565
write() success! : length = 10
Kernel >:write -=[]\;',./ 74565
write() success! : length = 10
Kernel >:write _+{}|:"<>? 74565
write() success! : length = 10
Kernel >:
```

```
root@WZYYN:~/Desktop/NetLab/lab-netstack-premium-master/build/src# sudo ./server
Kernel >:getaddrinfo 10.100.1.1
parent thread, current pid = e759e740
child process, current pid = e61fe700
shex ip_addr = 0a640101
getaddrinfo() success! addrinfo index = 0
addrinfo ip = 0a640101, port = 0
Kernel >:ocket 0
socket id: 74565
Kernel >:bind 10.100.1.1 233 74565
bind() success!
Kernel >:listen 5 74565
listen() success!
Kernel >:receive connection request from another device
Ip : 10.100.2.2Port :0
accept 74565
accept() success!
IP Address:10.100.2.2Kernel >:read 10 74565
read() success!
info: qwertyuiop
Kernel >:read 10 74565
read() success!
info: 1234567890
Kernel >:read 10 74565
read() success!
info: !@#$%^&*()
Kernel >:read 20 74565
read() success!
info: ASDFGHJKL:-=[]\;',./
Kernel >:read 100 74565
read() success!
info: _+{}|:"<>?
Kernel >:
```

Figure 4: CP8-3: retransmit the message if message is blocked.