Simulating packet drop using the E1000 network driver

This assignment aims to simulate the packet loss/corruption on the physical link. In this assignment, you will drop packets inside the E1000 network driver for a target IP with a given rate.

1 Background

1.1 Big and Little-endian

Big-endian: Store the most significant byte first in the memory.
Little-endian: Store the least significant byte first in the memory.

For example, let's say we want to store a 4-byte word 0xab12cd87 at location 0x1000.

In Big-endian:
0xab will be stored at 0x1000.
0x12 will be stored at 0x1001.
0xcd will be stored at 0x1002.
0x87 will be stored at 0x1003.

In Little-endian:
0x87 will be stored at 0x1000.
0xcd will be stored at 0x1001.
0x12 will be stored at 0x1002.
0xab will be stored at 0x1003.

On an Intel CPU, the data is stored in the Little-endian format. In network headers, the data is stored in Big-endian format, often called network byte order. For example, an IP address 7.8.9.3 (decimal format) is converted to a 32-bit hexadecimal value 0x07080903. Here, 0x07 is the most-significant byte. However, if we are to store it in a network header, 0x07 will be stored first, followed by 0x08, 0x09, and 0x03, respectively. If you read the IP address in the network header on the CPU, you will see 0x03090807, because it was stored in Big-endian format; however, the CPU will interpret it in Little-endian format.

14-bytes minimum 20-bytes 8-bytes for UDP minimum 20-bytes for TCP variable length

Figure 1: A packet with all the headers

1.2 Packet structure

A packet (Figure figure1) at the data-link layer has several headers. The first 14 bytes correspond to the data-link layer (or Ethernet) header. The next is the network layer (or IP) header. The minimum size of the IP header is 20 bytes. However, it can be more than 20 bytes if certain features are requested. After that, there is a transport layer header. If the packet is a UDP packet, the header size is eight bytes. For a TCP packet, the header size is at least 20 bytes (it can be more than 20 bytes if certain features are enabled).

struct sk_buff facilitates easy access to different headers in a packet. struct sk_buff provides various fields such as mac_header, network_header, and transport_header that point to the link-layer, network-layer, and transport-layer headers, respec tively. The Linux kernel also provides eth_hdr, ip_hdr, udp_hdr, tcp_hdr rou tines to fetch the different headers from struct sk_buff. These routines return a pointer of type struct ethhdr, struct iphdr, struct udphdr, and struct tcphdr. These structs allow users to conveniently read/write to different fields in the corresponding headers. Below are the definitions of structs corresponding to the various types of packet headers. Here, _be16 and _be32 indicate 2-byte and 4-byte values, respectively, in the Big-endian format.

1.3 Data-link layer header (14 bytes)

```
struct ethhdr {
   unsigned char h_dest[6]; // destination mac address
   unsigned char h_source[6]; // source mac address
   __be16 h_proto; // 2-byte protocol
};
```

1.4 Network layer header (minimum 20 bytes)

The actual length of the network header is the value of the ihl field multiplied by four. The total size of the packet, i.e., length of IP header + length of Transport header + payload size is stored in the tot len field. The protocol field stores the type of transport-layer protocol. It’s six for TCP and 17 for UDP.

```
struct iphdr {
   __u8 ihl:4; // length of header = ihl * 4
   __u8 version:4; // 4-bit version
   __u8 tos;
```

```
    __be16 tot_len; // Entire packet size, header + data
    __be16 id;
    __be16 frag_off;
    __u8 ttl; // Hop limit, decremented at each hop
    __u8 protocol; // protocol TCP=6, UDP=17
    __sun16 check; // header checksum
    __be32 saddr; // source IP address
    __be32 daddr; // destination IP address
    char options[]; // variable length
};
```

1.5 UDP header (8 bytes)

```
struct udphdr {
    __be16 source; // source port
    __be16 dest; // destination port
    __be16 len; // length of UDP header + data
    __sum16 check; // checksum of UDP header, payload, part of IP header };
```

1.6 TCP header (minimum 20 bytes)

The actual length of the TCP header is the value of the doff field multiplied by four.

```
struct tcphdr {
    __be16 source; // source port
    __be16 dest; // destination port
    __be32 seq; // sequence number
    __be32 ack_seq; // acknowledgement number
    __u16 ae:1;
    __u16 res1:1;
    __u16 doff:4; // Length of TCP header = doff * 4
    __u16 fin:1;
    __u16 syn:1; // start connection
    __u16 rst:1;
    __u16 psh:1;
    __u16 ack:1; // acknowledgment valid
    __u16 urg:1;
    __u16 ece:1;
    __u16 cwr:1;
    __be16 window; // window size
    __sum16 check; // checksum of TCP header, data, part of IP header __be16
    urg_ptr;
    char options[]; // variable length
};
```

1.7 E1000 device driver

The main job of the E1000 device driver is to facilitate the sending and receiving of packets using transmit and receive rings, as discussed in the lecture. In this as signment, the relevant routines are e1000 xmit frame and e1000 clean tx irq.

e1000 xmit frame takes an argument skb of type struct sk buff as input. skb contains pointers to the various headers of the actual packet, which is ready for transmission. This routine updates an available descriptor in the transmit ring with the physical address of the packet and updates the TDT register, which allows the NIC to transmit the packet.

e1000 clean tx irq is called as a part of the interrupt handling for sending packets. Once the packet has been sent, the NIC notifies the device driver by sending an interrupt. This routine notes that the transmit descriptors can now be available for future transmits. It may also free the buffers used for the sent packets.

2 User Tools

You are provided three applications: server, client, and block. server is a TCP server that binds itself to all available network interfaces on a host. client is a TCP client that takes the IP address of the host machine running the server as input, and sends 100000 64-byte packets to the server. The server terminates after receiving all the packets from the client. The block application takes an IP address, say 172.23.65.98, and a drop rate, say 15, as input. For convenience, we will use 172.23.65.98 and 15 as placeholders for the user supplied IP address and the drop rate, respectively. After receiving the inputs, block instructs the network driver to drop packets destined to 172.23.65.98 at a rate of 15%. Because the E1000 driver is not a standalone module in the Linux Kernel, it’s hard to pass direct arguments from a user application. Therefore, block creates and sends a UDP message of five bytes (containing the arguments) to a fixed IP address 100.100.100.100. The first four bytes of the UDP message contain 172, 23. 65, 98, i.e., the bytes corresponding to the user-supplied IP address. The fifth byte is the drop rate 15.

3 Implementation

In this assignment, you need to monitor all ongoing packets in the E1000 driver, and if a UDP packet is destined to 100.100.100.100, record the IP address, say x, and drop rate, say r, in the message to drop the future packets to x with the rate r. You need to implement everything in the e1000 main.c file. You can interpret all ready-to-send packets in e1000 xmit frame. Notice that after dropping a packet, you may need to release the buffer, as done in the interrupt handler. The TCP protocol at the transport layer should automatically recover from the packet loss.

If your implementation is correct, you can test it by running the server application on the host machine and the client application in the virtual ma chine. On the virtual machine, before invoking client try dropping packets to your host machine by running ./block 172.23.65.98 5, where 172.23.65.98 is the IP address of your host machine. If everything works perfectly, it will print ‘‘100001 packets received in ... milliseconds’’. You can experiment with different drop rates and monitor the total time.

In your implementation, you can drop packets for only one IP address at a given point. For example, after executing ./block 172.23.65.98 5, if you execute ./block 192.3.23.11 10, the driver will stop dropping the packet for 172.23.65.98; instead, it will now drop packets for 192.3.23.11 at a rate of 10%. You can also change the drop rate by executing ./block 192.3.23.11 15. From this point onward, the driver will drop packets to 192.3.23.11 at a rate of 15%.

4 Environment

For this assignment, you need to clone the project repository. To clone, run:
git clone https://github.com/Systems-IIITD/usertools.

To build the tools, run make in the usertools folder. It will create three applications server, client, and block. You have to make all the changes in the e1000 main.c file. Compile and use the E1000 driver in the same way as you did in the homework. You are not supposed to make any changes to any other files in the Linux kernel or the applications in usertools, except for experimenting with the receive buffer (discussed in the next section).

To recompile the e1000 module after making your changes, run: make M=drivers/net/ethernet/intel/e1000 from the linux-6.16 directory. To reuse your new implementation,
copy linux-6.16/drivers/net/ethernet/intel/e1000/e1000.ko to your VM, and reload the driver by running:
sudo modprobe -r e1000 followed by sudo insmod e1000.ko.

You are encouraged to write scripts to make copying and reloading easier inside the VM. You can also refer to the “PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer’s Manual” (available on Google Classroom) for more details about the software interface of the NIC driver.

5 Experiments

You need to do the following experiments. We will assume the IP address of your host is 172.23.65.98 in this discussion. Replace 172.23.65.98 with the IP address of your host in the actual experiment.

5

Run server using .server on the host. Drop 1% packets to the host by run ning ./block 172.23.65.98 1 inside the VM. Record the number of retrans mitted segments before sending packets using netstat -s |grep retransmitted. Run client inside the VM using ./client 172.23.65.98. Record the num ber of retransmitted segments after sending packets using netstat -s |grep retransmitted. Report the transfer time reported by the server, the number of retransmitted segments before and after transmitting the packets using client. Repeat the same experiments with 0%, 5%, 10%, and 15% drop rates.

Ping Google using ping www.google.com. Block Google's IP address using ./block 142.251.43.100 100. Replace 142.251.43.100 with Google's actual IP address in your experiment. The ping to Google should not work.

Change the buf size in server.c at Line-21 to 128, i.e., replace buf[64] with buf[128]. Recompile the server by running make. Is the number of packets received by the server with and without this change the same?

6 How to submit.

Rename e1000 main.c to groupNumber studentRollNumber studentName e1000 main.c. Replace groupNumber with your group's actual ID, studentRollNumber with your roll number, and studentName with your first name. Create a design doc ument in the PDF format and answer the following questions in your design documentation.

1. Briefly explain your design and the key changes in the e1000 main.c.
2. What is the transfer time without dropping any packets?
3. What was the number of retransmitted segments before sending packets in the above experiment?
4. What was the number of retransmitted segments after sending packets in the above experiment?
5. What is the transfer time with a 1% packet drop?
6. What was the number of retransmitted segments before sending packets in the above experiment?
7. What was the number of retransmitted segments after sending packets in the above experiment?
8. What is the transfer time with a 5% packet drop?
9. What was the number of retransmitted segments before sending packets in the above experiment?
10. What was the number of retransmitted segments after sending packets

in the above experiment?

11. What is the transfer time with a 10% packet drop?
12. What was the number of retransmitted segments before sending packets in the above experiment?
13. What was the number of retransmitted segments after sending packets in the above experiment?
14. What is the transfer time with a 15% packet drop?
15. What was the number of retransmitted segments before sending packets in the above experiment?
16. What was the number of retransmitted segments after sending packets in the above experiment?
17. Does ping to Google work after dropping 100% packets to it?
18. Did you notice any change in the number of received packets by the server after changing the buffer size to 128? If yes, what is the reason?
19. Write the names and roll numbers of all group members.

Rename the design documentation as groupNumber studentRollNumber studentName.pdf. Replace groupNumber with your group's actual ID, studentRollNumber with your roll number, and studentName with your first name. Submit both the files. All members of the group are required to submit the files. Your assign ment will not be evaluated if you don't follow the submission guidelines.