# Ping using UDP

The goal of this assignment is to understand the various fields in the UDP and ICMP packets. In this assignment, you are to convert a UDP packet to an ICMP echo request in the send path and convert the ICMP echo reply to a UDP packet in the receive path of the E1000 device driver.

## 1 Background

### 1.1 UDP

User Datagram Protocol (UDP) enables an application to talk to another application running on a different host. Once a UDP packet arrives at the transport layer from the network layer, its job is to deliver the packet to the application, which is trying to receive a packet on a port number that matches the destination port number in the packet. During the send path, the transport layer adds the source and destination port numbers to the packet for application-to-application delivery.

A UDP packet header looks like the following:

```
struct udphdr {
  __be16 source; // source port
  __be16 dest;   // destination port
  __be16 len;    // length of UDP header + data
  __sum16 check; // Checksum
};
```

The len field contains the sum of the length of the payload and the length of the UDP header (which is always eight), in Big-endian format. Here, `__be16` indicates a 2-byte value in Big-endian format. The size of the UDP header is 8 bytes. A pseudo IP header is prepended to the UDP packet for checksum computation. The checksum is performed on the buffer that stores the pseudo IP header, followed by the UDP header and payload. However, the pseudo IP header is not included in the actual packet; it's used just for the checksum computation. If the payload size is not a multiple of 16 bits, an additional 8-bit padding with zero value is appended to the buffer during the checksum computation.

The pseudo IP header has the following fields.

```
struct pseudo_header {
  __be32 saddr; // source IP address
  __be32 daddr; // destination IP address
  __u8 zero;  // always zero
  __u8 protocol;  // 17 for UDP
  __be16 total_len; // same as the length field in the UDP header
};
```

Here, `__be32` is a 4-byte Big-endian value. The value of `total_len` is the same as the `len` field in the UDP header. The `saddr` and `daddr` are the source and destination IP addresses, which are also present in the IP header `struct iphdr` as shown below:

```
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 ICMP=1, UDP=17
  __sum16 check;  // header checksum
  __be32 saddr; // source IP address
  __be32 daddr; // destination IP address
  char options[]; // variable length
};
```

Notice that the length of an IP header could be variable. 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 `check` field contains the checksum of the header. The checksum algorithm is discussed in class. If you make any changes to the IP header, you must update the `check` field to reflect the new checksum. The `protocol` field in the `struct iphdr` is 17 for UDP and one for ICMP.

## 1.2 ICMP

The Internet Control Message Protocol (ICMP) is used for end-hosts to communicate network-layer information to each other. The ICMP message is neither a UDP message nor a TCP message. It's considered part of the network layer, but it's stored similarly to a UDP packet, i.e., IP header followed by ICMP header followed by ICMP payload, as shown in Figure 1.

An ICMP header has the following fields:

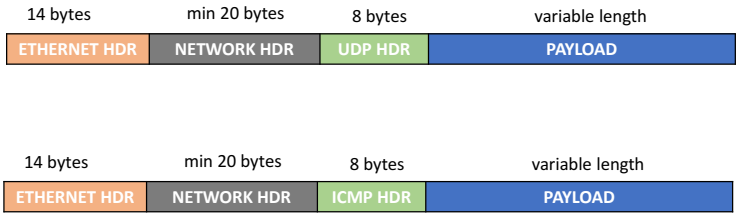


Figure 1: A UDP packet (top), an ICMP packet (bottom)


Figure 2: A packet with headers

```
struct icmphdr {
  __u8    type;
  __u8    code;
  __sum16 checksum;
  __be16  id;
  __be16  sequence;
};
```

ICMP packets can be of different types, and `id` and `sequence` fields can be interpreted differently depending on the type (the Linux kernel uses a union for other interpretations of these fields). In this assignment, we are interested in ICMP echo requests and ICMP echo replies. The `type` field for ICMP echo request is eight, and the `type` field for ICMP echo reply is zero. The `code` field is zero for both the ICMP echo request and reply. During the echo request, the application set the `id` to a value that can be used by the kernel to identify it, e.g., the process id. When an ICMP echo reply is received, the OS looks at the `id` field and delivers the message to the application that previously sent the echo request with this `id`. A process can send multiple echo requests. The sequence field is used to correlate a request with a reply. The ICMP payload can be of variable length. The destination host simply ignores the value of the payload. The payload is copied as it is in an ICMP reply. The `checksum` field contains the checksum of the ICMP header and the payload. Notice that before the checksum computation, the value of the `checksum` field is set to zero in all kinds of packet headers, e.g., ICMP, UDP, or IP packet headers.

## 1.3 Packet structure

A packet (Figure 2) 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, respectively. The Linux kernel also provides `eth_hdr`, `ip_hdr`, `udp_hdr`, `tcp_hdr` routines 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.

### 1.4 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 assignment, the relevant routines are `e1000_xmit_frame` and `e1000_receive_skb`.

`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.

The E1000 driver uses `e1000_receive_skb` to handle the packet to the network layer after receiving a packet. At this point, the various fields of the struct `sk_buff` might not have been initialized properly. However, `skb->data` points to the IP header (i.e., struct iphdr*). Using the value of the header length, total length, and protocol in the IP header, you can compute the address of other headers and the payload.

## 2 User Tools

You are provided an application: `udpping`. The `udpping` application takes an IP address, say `172.23.65.98`, as input. For convenience, we will use `172.23.65.98` as a placeholder for the user-supplied IP address. After receiving the inputs, `updping` instructs the network driver to send an ICMP echo request destined to `172.23.65.98`. `udpping` creates and sends a UDP message of eight bytes 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. In the rest of the four bytes, a magic number 0xDECAF is stored (in the Little-endian format). In the rest of the document, whenever we mention the magic number, it's always 0xDECAF.

# 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, you need to fetch the destination address from the payload (say X), convert the UDP packet into ICMP echo request with the destination address X, and send it to X. You are allowed to change the first four bytes of the payload, but not the magic number because it's needed to identify the ICMP reply in the receive path (i.e., e1000_receive_skb). In the receive path, check for all ICMP echo replies. If the payload contains the magic number, you need to convert it back into the UDP packet before handing it to the network layer.

Notice that the transport layer needs to know the destination port number in order to deliver it to the correct application. After sending a packet udpping expects a response on the same port number used for sending. Therefore, to find the correct destination port, you can save the source port somewhere in the first four bytes while converting the UDP packet into an ICMP packet in the send path. In the send path, you need to initialize the fields correctly in the ICMP header and save the source port number in the payload. Alternatively, you can store the source port number in the `id` field of the ICMP header. In this case, you don't need to change the payload. You also need to change the destination address and protocol in the IP header. Finally, you need to compute and store the correct checksum in both the ICMP and IP headers.

In the receive path, after you have identified the correct ICMP reply, you need to convert it into a UDP packet. You need to properly initialize all the fields in the UDP header, including the destination port number, which is stored in the payload (or in the `id` field of the ICMP header). You need to change the `protocol` field in the IP header. Finally, you need to properly compute and store the checksum in both UDP and IP headers.

# 4 Other details

The Intel NIC supports fast checksum computation. Therefore, instead of computing the checksum itself while sending a packet, the Linux kernel prefers configuring the NIC device to compute it before sending. The NIC takes the starting address of the buffer, the length of the buffer, and the offset at which the checksum needs to be stored as input. Using this information, the NIC computes the checksum of the buffer and stores it at the correct offset. As the checksum offset and the data used for the checksum are different for UDP and ICMP, the automatic computation can cause inconsistencies. The better strategy would be to compute the checksum of the IP header and the ICMP packet yourself. There is a field called `ip_summed` in struct `sk_buff`. If this field is set to `CHECKSUM_PARTIAL`, E1000 configures the NIC to compute the checksum. To stop checksum computation in the NIC, you can set this field to `CHECKSUM_COMPLETE` after computing and storing the correct checksums.

# 5 Environment

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

To build the tools, run make in the `udpping` folder. It will create two applications `udpping` and `cping`. `cping` is a tool that can be used to create and send an ICMP packet to a given destination. It takes the IP address of the destination as input. You can use it for debugging. `cping` will need sudo access to send the ICMP packets.

To debug whether an ICMP packet was received on a different host, I suggest you work on two machines, run Wireshark on the destination machine with a display filter for ICMP packets. If you use the ping or cping application, you should be able to see ICMP packets in Wireshark at the destination host. After you have successfully converted a UDP packet to an ICMP packet, you can see your packet in Wireshark, too! 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 `udpping`.

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.

# 6 Experiments

You need to do the following experiments.

Ping Google using: `ping www.google.com`
Ping Google's IP address using: `./udpping 142.251.43.100`.
Replace 142.251.43.100 with Google's actual IP address in your experiment. The ping to Google should work. If it works fine, you should see the following message:

```
UDP echo: 142.251.43.100
Congrats: test passed
```

Try pinging Google again using `ping www.google.com` to test that the normal ping stack is unaffected by your driver changes. It should work correctly.

# 7 How to submit.

Create a design document 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. Are you getting the expected output after pinging Google using `udpping`?
3. Is ping to Google using the ping application work correctly after a successful ping to Google using udpping?
4. Write the names and roll numbers of all group members.

Submit `e1000_main.c` and your design documentation. Use the naming convention for the assignments and homework. Your assignment will not be evaluated if you don't follow the submission guidelines.