

Task 5

Part A:

First, this is my code for this part:

```
#include <stdio.h>

#if defined _WIN32
// See at https://msdn.microsoft.com/en-us/library/windows/desktop/ms740506\(v=vs.85\).aspx
// link with Ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")
#include <winsock2.h>
#include <ws2tcpip.h>

/*
 * This was a surprise to me... This stuff is not defined anywhere under
 * MSVC.
 * They were taken from the MSDN ping.c program and modified.
 */

#define ICMP_ECHO 8
#define ICMP_ECHOREPLY 0
#define IP_MAXPACKET 65535

#pragma pack(1)

struct ip
{
    UINT8 ip_hl : 4; // length of the header
    UINT8 ip_v : 4; // Version of IP
    UINT8 ip_tos; // Type of service
    UINT16 ip_len; // total length of the packet
    UINT16 ip_id; // unique identifier of the flow
    UINT16 ip_off; // fragmentation flags
    UINT8 ip_ttl; // Time to live
    UINT8 ip_p; // protocol (ICMP, TCP, UDP etc)
    UINT16 ip_sum; // IP checksum
    UINT32 ip_src;
    UINT32 ip_dst;
};

struct icmp
{
    UINT8 icmp_type;
    UINT8 icmp_code; // type sub code
```

```

    UINT16 icmp_cksum;
    UINT16 icmp_id;
    UINT16 icmp_seq;
    UINT32 icmp_data;      // time data
};

#pragma pack()

// MSVC defines this in winsock2.h
//typedef struct timeval {
//    long tv_sec;
//    long tv_usec;
//} timeval;

int gettimeofday(struct timeval * tp, struct timezone * tzp)
{
    // Note: some broken versions only have 8 trailing zero's, the
    // correct epoch has 9 trailing zero's
    static const uint64_t EPOCH = ((uint64_t) 11644473600000000ULL);

    SYSTEMTIME  system_time;
    FILETIME    file_time;
    uint64_t    time;

    GetSystemTime( &system_time );
    SystemTimeToFileTime( &system_time, &file_time );
    time = ((uint64_t)file_time.dwLowDateTime )
        ;
    time += ((uint64_t)file_time.dwHighDateTime) << 32;

    tp->tv_sec  = (long) ((time - EPOCH) / 100000000L);
    tp->tv_usec = (long) (system_time.wMilliseconds * 1000);
    return 0;
}

#else // linux

#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys/time.h> // gettimeofday()

```

```

#endif

// IPv4 header len without options
#define IP4_HDRLen 20

// ICMP header len for echo req
#define ICMP_HDRLen 8

// Checksum algo
unsigned short calculate_checksum(unsigned short * paddress, int len);

// 1. Change SOURCE_IP and DESTINATION_IP to the relevant
//     for your computer
// 2. Compile it using MSVC compiler or g++
// 3. Run it from the account with administrative permissions,
//     since opening of a raw-socket requires elevated privileges.
//
//     On Windows, right click the exe and select "Run as administrator"
//     On Linux, run it as a root or with sudo.
//
// 4. For debugging and development, run MS Visual Studio (MSVS) as
//     admin by
//     right-clicking at the icon of MSVS and selecting from the right-
//     click
//     menu "Run as administrator"
//
// Note. You can place another IP-source address that does not belong
// to your
// computer (IP-spoofing), i.e. just another IP from your subnet, and
// the ICMP
// still be sent, but do not expect to see ICMP_ECHO_REPLY in most
// such cases
// since anti-spoofing is wide-spread.

// #define SOURCE_IP "172.18.0.1"
// i.e the gateway or ping to google.com for their ip-address
#define DESTINATION_IP "172.217.169.78"

int main ()
{
    struct ip iphdr; // IPv4 header
    struct icmp icmphdr; // ICMP-header
    char data[IP_MAXPACKET] = "This is the ping.\n";

    int datalen = strlen(data) + 1;

    //=====

```

```

// IP header
//=====

// IP protocol version (4 bits)
iphdr.ip_v = 4;

// IP header length (4 bits): Number of 32-bit words in header = 5
iphdr.ip_hl = IP4_HDRLLEN / 4; // not the most correct

// Type of service (8 bits) - not using, zero it.
iphdr.ip_tos = 0;

// Total length of datagram (16 bits): IP header + ICMP header +
ICMP data
iphdr.ip_len = htons (IP4_HDRLLEN + ICMP_HDRLLEN + datalen);

// ID sequence number (16 bits): not in use since we do not allow
fragmentation
iphdr.ip_id = 0;

// Fragmentation bits - we are sending short packets below MTU-size
and without
// fragmentation
int ip_flags[4];

// Reserved bit
ip_flags[0] = 0;

// "Do not fragment" bit
ip_flags[1] = 0;

// "More fragments" bit
ip_flags[2] = 0;

// Fragmentation offset (13 bits)
ip_flags[3] = 0;

iphdr.ip_off = htons ((ip_flags[0] << 15) + (ip_flags[1] << 14)
                      + (ip_flags[2] << 13) + ip_flags[3]);

// TTL (8 bits): 128 - you can play with it: set to some reasonable
number
iphdr.ip_ttl = 128;

// Upper protocol (8 bits): ICMP is protocol number 1
iphdr.ip_p = IPPROTO_ICMP;

// Source IP

```

```

//      if (inet_pton (AF_INET, SOURCE_IP, &(iphdr.ip_src)) <= 0)
//      {
//          fprintf (stderr, "inet_pton() failed for source-ip with
error: %d"
// #if defined _WIN32
//             , WSAGetLastError()
// #else
//             , errno
// #endif
//             );
//          return -1;
//      }

// Destination IPv
if (inet_pton (AF_INET, DESTINATION_IP, &(iphdr.ip_dst)) <= 0)
{
    fprintf (stderr, "inet_pton() failed for destination-ip with
error: %d"
#if defined _WIN32
        , WSAGetLastError()
#else
        , errno
#endif
        );
    return -1;
}

// IPv4 header checksum (16 bits): set to 0 prior to calculating in
order not to include itself.
iphdr.ip_sum = 0;
iphdr.ip_sum = calculate_checksum((unsigned short *) &iphdr,
IP4_HDRLEN);

//=====
// ICMP header
//=====

// Message Type (8 bits): ICMP_ECHO_REQUEST
icmphdr.icmp_type = ICMP_ECHO;

// Message Code (8 bits): echo request
icmphdr.icmp_code = 0;

// Identifier (16 bits): some number to trace the response.
// It will be copied to the response packet and used to map
response to the request sent earlier.
// Thus, it serves as a Transaction-ID when we need to make "ping"

```

```

    icmphdr.icmp_id = 18; // hai

    // Sequence Number (16 bits): starts at 0
    icmphdr.icmp_seq = 0;

    // ICMP header checksum (16 bits): set to 0 not to include into
checksum calculation
    icmphdr.icmp_cksum = 0;

    // Combine the packet
    char packet[IP_MAXPACKET];

    // First, IP header.
    memcpy (packet, &iphdr, IP4_HDRLLEN);

    // Next, ICMP header
    memcpy ((packet + IP4_HDRLLEN), &icmphdr, ICMP_HDRLLEN);

    // After ICMP header, add the ICMP data.
    memcpy (packet + IP4_HDRLLEN + ICMP_HDRLLEN, data, datalen);

    // Calculate the ICMP header checksum
    icmphdr.icmp_cksum = calculate_checksum((unsigned short *) (packet
+ IP4_HDRLLEN), ICMP_HDRLLEN + datalen);
    memcpy ((packet + IP4_HDRLLEN), &icmphdr, ICMP_HDRLLEN);

    struct sockaddr_in dest_in;
    memset (&dest_in, 0, sizeof (struct sockaddr_in));
    dest_in.sin_family = AF_INET;

    // The port is irrelant for Networking and therefore was zeroed.
#ifdef _WIN32
    dest_in.sin_addr.s_addr = iphdr.ip_dst;
#else
    dest_in.sin_addr.s_addr = iphdr.ip_dst.s_addr;
#endif

#ifdef _WIN32
    WSADATA wsaData = { 0 };
    int iResult = 0;

    // Initialize Winsock
    iResult = WSASocket(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSASocket failed: %d\n", iResult);
        return 1;
    }
}

```

```

#endif

    // Create raw socket for IP-RAW (make IP-header by yourself)
    int sock = -1;
    if ((sock = socket (AF_INET, SOCK_RAW, IPPROTO_RAW)) == -1)
    {
        fprintf (stderr, "socket() failed with error: %d"
#ifdef _WIN32
            , WSAGetLastError()
#else
            , errno
#endif
        );
        fprintf (stderr, "To create a raw socket, the process needs to
be run by Admin/root user.\n\n");
        return -1;
    }

    struct timeval start, end;
    gettimeofday(&start, NULL);

    // This socket option IP_HDRINCL says that we are building IPv4
    header by ourselves, and
    // the networking in kernel is in charge only for Ethernet header.
    //
    const int flagOne = 1;
    if (setsockopt (sock, IPPROTO_IP, IP_HDRINCL,
#ifdef _WIN32
        (const char*)
#endif
        &flagOne, // The above casting is important for Windows.
        sizeof (flagOne)) == -1)
    {
        fprintf (stderr, "setsockopt() failed with error: %d"
#ifdef _WIN32
            , WSAGetLastError()
#else
            , errno
#endif
        );
        return -1;
    }

    // Send the packet using sendto() for sending datagrams.
    if (sendto (sock, packet, IP4_HDRLEN + ICMP_HDRLEN + datalen, 0,
(struct sockaddr *) &dest_in, sizeof (dest_in)) == -1)
    {
        fprintf (stderr, "sendto() failed with error: %d"

```

```

    #if defined _WIN32
        , WSAGetLastError()
    #else
        , errno
    #endif

    );
    return -1;
}

bzero(packet, IP_MAXPACKET);
socklen_t sizeOfDest = sizeof(dest_in);
printf("before recvfrom function:\n");
int recvMessageBytes = recvfrom(sock, packet, sizeof(packet), 0,
(struct sockaddr *) &dest_in, &sizeOfDest);
printf("Number of bytes that were Received %d from the packet: IP
header :%d + ICMP header:%d + data:%d\nPacket Data: %s\n",
recvMessageBytes, IP4_HDRLen, ICMP_HDRLen, datalen, packet +
ICMP_HDRLen + IP4_HDRLen);
gettimeofday(&end, NULL);
float time_in_millis = ((end.tv_sec - start.tv_sec) * 1000.0f) +
((end.tv_usec - start.tv_usec) / 1000.0f);
unsigned long time_in_micros =
(1000000.0f * (end.tv_sec - start.tv_sec)) + (end.tv_usec -
start.tv_usec);

printf("RTT in milliseconds: %f\nRTT in microseconds: %lu\n",
time_in_millis, time_in_micros);

// Close the raw socket descriptor.
#if defined _WIN32
    closesocket(sock);
    WSACleanup();
#else
    close(sock);
#endif

return 0;
}

// Compute checksum (RFC 1071).
unsigned short calculate_checksum(unsigned short * paddress, int len)
{
    int nleft = len;
    int sum = 0;
    unsigned short * w = paddress;
    unsigned short answer = 0;

    while (nleft > 1)

```



```

    {
        sum += *w++;
        nleft -= 2;
    }

    if (nleft == 1)
    {
        *((unsigned char *)&answer) = *((unsigned char *)w);
        sum += answer;
    }

    // add back carry outs from top 16 bits to low 16 bits
    sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
    sum += (sum >> 16);                // add carry
    answer = ~sum;                      // truncate to 16 bits

    return answer;
}

```

Generally, I change a little bit the in addition I took the code that was given I calculate the time for the package until the receive function as required (The RTT) in milliseconds and microseconds. The data message that was received is the ping message.

In milliseconds it took: 66.674004.

In microsecond it took: 66674.

We can see in the picture below with the details I printed to the ubuntu shell.

```

root@DESKTOP-SP3N41Q:/home/gabi/tikshoretEx5# cd "/home/gabi/tikshoretEx5" && gcc ICMP.cpp -o ICMP && "/home/gabi/tikshoretEx5"/ICMP
before recvfrom function:
Number of bytest that were Received 47 from the packet: IP header :20 + ICMP header:8 + data:19
Packet Data: This is the ping.

RTT in milliseconds: 66.674004
RTT in microseconds: 66674

```

This is how it looks like on Wireshark; we can see that the protocols are ICMP protocols, and the messages are ping request and ping reply.

No.	Time	Source	Destination	Protocol	Length	Info
6	2.054564	10.0.0.11	172.217.169.78	ICMP	61	Echo (ping) request id=0x03e8, seq=0/0, ttl=63 (reply in
7	2.120754	172.217.169.78	10.0.0.11	ICMP	61	Echo (ping) reply id=0x03e8, seq=0/0, ttl=57 (request i

In the picture below I am showing the request message, as we can see the data as the total length, protocols etc., is similar.

>	Frame 6: 61 bytes on wire (488 bits), 61 bytes captured (488 bits) on interface \Device\NPF_{F5A650D1-9212-4D6A-B51E-F9BE70C7C469}, id 0
>	Ethernet II, Src: CloudNet_26:6b:cb (90:0f:0c:26:6b:cb), Dst: Sagemcom_6a:85:07 (b0:bb:e5:6a:85:07)
▼	Internet Protocol Version 4, Src: 10.0.0.11, Dst: 172.217.169.78
	0100 = Version: 4
 0101 = Header Length: 20 bytes (5)
>	Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
	Total Length: 47
	Identification: 0xbeda (48858)
>	Flags: 0x40, Don't fragment
	...0 0000 0000 0000 = Fragment Offset: 0
	Time to Live: 63
	Protocol: ICMP (1)
	Header Checksum: 0x1cc1 [validation disabled]
	[Header checksum status: Unverified]
	Source Address: 10.0.0.11
	Destination Address: 172.217.169.78
▼	Internet Control Message Protocol
	Type: 8 (Echo (ping) request)
	Code: 0
0000	b0 bb e5 6a 85 07 90 0f 0c 26 6b cb 08 00 45 00 ...j....&k...E.
0010	00 2f be da 40 00 3f 01 1c c1 0a 00 00 0b ac d9 ../.@.?.....
0020	a9 4e 08 00 bc 4e 03 e8 00 00 54 68 69 73 20 69 .N...N...This i
0030	73 20 74 68 65 20 70 69 6e 67 2e 0a 00 s the pi ng...

Part B:

In this part for every requirement of the part I will show this part on my code with some explanations to be more understandable (of course I also included all the libraries), and I will show how the sniffer works generally while running it on the ubuntu shell.

So first, I opened a raw socket as I did in the first part of this task, as we can see in the picture below:

```

printf("\nSniffer is ready...\n");
char packet[IP_MAXPACKET];
struct packet_mreq mreq;

int sock = -1;
if ((sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) == -1) {
    perror("Error with socket()");
    return 1;
}

memset(&mreq, 0, sizeof(mreq));
mreq.mr_type = PACKET_MR_PROMISC;
setsockopt(sock, SOL_PACKET, PACKET_ADD_MEMBERSHIP, &mreq, sizeof(mreq));

for(;;) {
    bzero(packet, IP_MAXPACKET);
    int dataSize = recvfrom(sock, packet, ETH_FRAME_LEN, 0, NULL, NULL);
    if (dataSize < 0) {
        printf("Recvfrom error, failed to get packets\n");
        break;
    }
}

```

After that I built a function that is processing this packet as it required in this task, and presenting this requirement:

```

void processPacket(unsigned char *buffer, int size) {

    struct iphdr *ip_hdr = (struct iphdr *) (buffer + ETH_HLEN);

    if (ip_hdr->protocol != IPPROTO_ICMP) { return; }

    printf("\n\nserial number for the packet: %d\n", ++counter);
    unsigned short ip_hdr_len = ip_hdr->ihl * 4;
    struct icmp_hdr *icmp_hdr = (struct icmp_hdr *) (buffer + ip_hdr_len + ETH_HLEN);
    int header_size = ETH_HLEN + ip_hdr_len + sizeof(icmp_hdr);

    unsigned type = (unsigned int) (icmp_hdr->type);

    printf("\n **** ICMP Header ****\n\n");
    printf("TYPE : %d", type);
    if (type == 11) {
        printf(" (TTL Expired)\n");
        return;
    } else if (type < 11) {
        if (type == ICMP_ECHO) {
            printf(" (Echo ping request)\n");
        }
        if (type == ICMP_ECHOREPLY) {
            printf(" (Echo ping reply)\n");
        }
        printf("\nCode : %d\n", (unsigned int) (icmp_hdr->code));
        struct sockaddr_in source, dest;
        memset(&source, 0, sizeof(source));
        source.sin_addr.s_addr = ip_hdr->saddr;

        printf("\n **** IP Header ****\n\n");
        printf("IP_SEC: %s\n", inet_ntoa(source.sin_addr));
        memset(&dest, 0, sizeof(dest));
        dest.sin_addr.s_addr = ip_hdr->daddr;
        printf("IP_DST: %s\n", inet_ntoa(dest.sin_addr));
    }
}

```

As we can see I am just printing the src Ip, dst Ip I am just checking if it is a reply or request and printing the code and the type as required.

Generally, about the code, I just opened a raw socket that is receiving ICMP packets.

Now I will show how it works on ubuntu shell:

This picture shows that the sniffer is ready to sniff some packets (before sending the ICMP packets).

```

gabi@DESKTOP-SP3N41Q:~$ sudo su
[sudo] password for gabi:
root@DESKTOP-SP3N41Q:/home/gabi# cd "/home/gabi/tikshoreEx5" && gcc sniffer.c -o sniffer && "/home/gabi/tikshoreEx5"/sniffer
Sniffer is ready...

```

Now I will send an ICMP packets(as I did in part A) with a different shell and we will see the sniffer results.

```

gabi@DESKTOP-SP3N41Q:~$ sudo su
[sudo] password for gabi:
root@DESKTOP-SP3N41Q:/home/gabi# cd "/home/gabi/tikshoreEx5" && gcc sniffer.c -o sniffer && "/home/gabi/tikshoreEx5"/sniffer
Sniffer is ready...

serial number for the packet: 1

**** ICMP Header ****

TYPE : 8 (Echo ping request)

Code : 0

**** IP Header ****

IP_SEC: 172.18.9.132
IP_DST: 172.217.169.78

serial number for the packet: 2

**** ICMP Header ****

TYPE : 0 (Echo ping reply)

Code : 0

**** IP Header ****

IP_SEC: 172.217.169.78
IP_DST: 172.18.9.132

```

As we can see we got 2 packets:

1. The first is what I sent to google as we can see the SRC and the DEST, we can see that the type is a request.
2. The second is a response from google (SRC), we can see all the information that was required as we saw in the first packet the Type is reply.

Just to make sure I was showing all the code parts, this is the libraries that I used:

```

tikshoreEx5 > C sniffer.c
1  #include <netinet/in.h>
2  #include <arpa/inet.h>
3  #include <string.h>
4  #include <stdio.h>
5  #include <netinet/ip_icmp.h>
6  #include <net/ethernet.h>
7  #include <linux/if_packet.h>
8
9  int counter = 0;

```