ARIEL UNIVERSITY

Communication Networks Course

Assignment 4 – Network Layer

Assignment Notes and Instructions

This assignment is to be done in **pairs** only, with exceptions approved by emailing the course coordinator.
You must upload all the necessary files in a ZIP file named after both of the student's IDs. For example, $123456789\_987654321.zip$.

1) Please ensure that you submit your assignment on or before the deadline indicated in the submission box.

2) All assignment files, including code, Wireshark .pcap files, and a PDF with necessary screenshots and descriptions, must be submitted in a ZIP file.

3) You may use any reference material available on the course Moodle or provided during exercises.

4) While you're free to refer to any online resources, it's important to avoid copying entire code blocks from websites, including those found on GitHub. If a student is caught doing so, they will fail the assignment with a score of 0. Additionally, please make sure to document all websites used to complete the assignment in the PDF.

5) Your code should be well structured and designed. To ensure clarity, include helpful comments in the code and use meaningful variable names. Additionally, please remember to provide a *makefile* that compiles all of the programs.

6) **The assignment code must be written in C only.**

7) The assignment is individual and should not be assisted by anyone outside or inside the university. You can seek help during reception hours from the course staff or ask a question in the course forum. It is forbidden to share code sections, upload solutions, or parts of solutions on any Internet website or communication platform.

8) The assignment's code files must compile and run properly on the Ubuntu 22.04 LTS operating system, which will be used to test the submissions. You **mustn't** code it in Windows, as it uses different APIs for sockets. WSL won't work, as the task needs a tool that works only in a native Linux environment.

9) **If you use any AI, please add the prompts. AI should/may help but not solve**

Good Luck!

Part A – Ping Program – 50 points

In this part, we will create a program called "ping", which is used to check the communication between two endpoints – a local host ("our computer"); and a remote host (another host in a local/remote network).

In most operating systems, the syntax of this command is:

$$\$\ ping\ [options] < destination >$$

where $< destination >$ is the IP address of the machine we want to test communication with. See $man\ 8\ ping$ for more information.

In this part of the assignment, you will implement your own version of "ping". Write a program named $ping.c$, which will be slightly more advanced than the well-known ping.

Your implementation should have the following features:

- Support options via flags (see $getopt(3)$):
  - Flag `$-a$` (address) will receive the desired IP address to ping.
    This flag is necessary.
  - Flag `$-t$` (type) will specify the communication type. The value of this flag should be either '4' (for IPv4), or '6' (for IPv6). This flag is necessary.
  - Flag `$-c$` (count) will send a specified number of pings .
    This flag is optional.
  - Flag `$-f$` (flood) will send the ping packets "back to back", namely, without any delay between them. This flag is optional.
  - You **MUST** implement all the flags (a, c, f, t). By saying "This flag is optional" we mean that it is optional when running the tool.

- Support both IPv4 (via ICMP) and IPv6 (via ICMPv6) (see Appendix A).

- Show statistics after the program finishes running; it's recommended to use the function $signal(2)$ - see Appendix B.

- Support a timeout – If a reply is not received within 10 seconds of sending a ping, the program will terminate; it's recommended to use function $poll(2)$ - see appendix B.

- The .pcap files you submit should include at least the following:
  - 1 scenario that uses IPv4.
  - 1 scenario that uses IPv6.
  - 1 scenario that uses the -c flag.
  - 1 scenario that uses the -f flag.
    Note that the same scenario may address several demands. E.g., you may submit a single scenario that uses IPv4 with -c, and another scenario that uses IPv6 with -f.

Example output:

```
foo@bar:~$ sudo ./ping -t 4 -c 4 -a 8.8.8.8
Pinging 8.8.8.8 with 64 bytes of data:
64 bytes from 8.8.8.8: icmp_seq=1 ttl=117 time=5.980ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=117 time=6.830ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=117 time=6.970ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=117 time=8.450ms

--- 8.8.8.8 ping statistics ---
4 packets transmitted, 4 received, time 3028.23ms
rtt min/avg/max/mdev = 5.980/7.05/8.450/0.888ms
```

Figure 1 – Usage for example for Ping program.

Guidance:

1) The program will send an ICMP or ICMPv6 ECHO-Request packet.

2) The program will wait for a reply (ICMP or ICMPv6 ECHO-Reply packet).
   a. If no reply is received within 10 seconds, proceed to step 5 and end the program.
   b. If a reply is received, measure the time it took (RTT) and save it, and continue to step 3.

3) Print packet statistics: received packet size, IP address of the sender, the sequence number of the packet, and the RTT that you have measured.
   The "sequence number" should be the packet number in the current sequence, i.e. start with counter=0, and increment the counter each time a packet is sent.

4) The program will pause for one second before sending the next packet, unless the user adds the `$-f$` flag.

5) When the program ends, print the session's statistics: number of packets transmitted, number of packets received, run time of the whole program, andmin, max, avg of the collected RTTs.

Usage:

- Minimal usage (specifying only the necessary flags):

$$\$\, sudo\ ./ping\ -a\ <address>\ -t\ <4\mid 6>$$

Your goal in this part is to emulate the original "ping" tool as similarly as possible.

Simple schematics of the program:

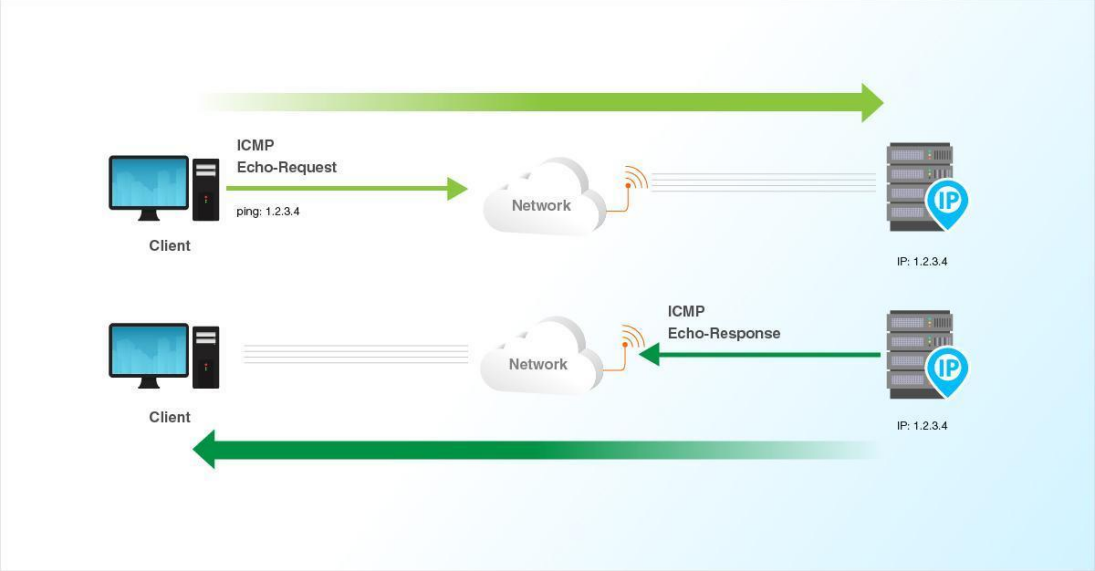

Figure 2 – Schematics of a Ping program.
Image Credits:
https://maliki.id/konsep-icmp-ip-dan-arp/

Part B – Trace Program – 50 points

$traceroute$ and $tracert$ are computer network diagnostic commands for displaying routes (paths) and measuring transit delays of packets across a network. In particular, these functions generalize Ping, which only measures the RTT from the destination point.

In this assignment, you will create your own version of the "traceroute" tool.

Write a program named $traceroute.c$:

Print the routing path of a packet from our local machine to a remote machine.

Usage:

$$\$\ sudo\ ./traceroute\ -a < destination >$$

Where:

- $< destination >$ – destination IP address.

Example output:

```
foo@bar:~$ sudo ./traceroute -a 8.8.8.8
traceroute to 8.8.8.8, 30 hops max
  1   172.18.192.1  0.565ms  0.455ms  0.274ms
  2   10.100.102.1  1.568ms  1.301ms  1.192ms
  3   212.143.208.119  6.484ms  4.190ms  4.856ms
  4   207.232.57.152  5.871ms  6.293ms  4.321ms
  5   207.232.57.143  4.940ms  3.952ms  4.665ms
  6   72.14.197.132  6.343ms  6.581ms  6.656ms
  7   *  *  *
  8   8.8.8.8  6.904ms  5.958ms  7.652ms
```

Figure 3 – Usage for example for Traceroute program.

Guidance:

- For this task, you have to build your own IP header from scratch (See Appendix A).
- The program will send ICMP ECHO-Request packets with TTL value starting from TTL=1, and incremented up to TTL=64 or until the destination IP address is reached – the earliest of the two. When an ICMP packet is received, check if the source IP address matches the destination IP address. If yes, end the program. else, continue with the hops.
- For each "hop" (round), the program sends 3 ICMP ECHO-Request packets, to ensure that the packets were reached and weren't dropped. If a reply doesn't arrive in 1 second, mark it with asterisk (*), which marks failure. **Note that 3 failed packets don't necessarily mean that the host doesn't exist, as some routers are configured to not forward ICMP packets due to security reasons.**
- The maximum allowed number of hops is 30. If the program doesn't finish, mark it as a failure (destination unreachable) and end the program.

Simple schematics of the program:

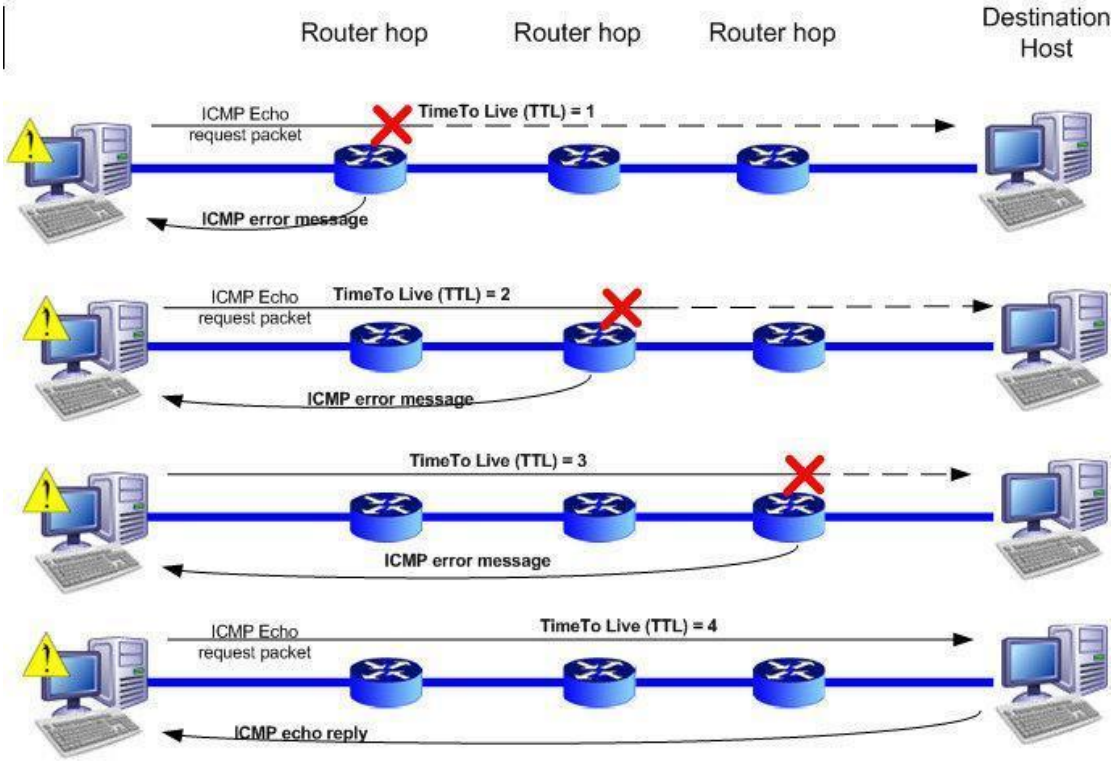

Figure 4 – Schematics of a Traceroute program.
Image Credits:
https://labtekajeclub.wordpress.com/2019/03/16/ttl-time-to-live-dan-scope-target-scope-pada-mikrotik/

Part C – Network Scanner – bonus 5 points

In this part, your task is to create a network scanner, which is a useful tool for exploring and discovering devices on a network. Your program should receive as arguments an IP address and a subnet mask; systematically scan all the IP addresses within the specified range; and print them.

Write a program named $discovery.c$, which will do the following (Guidance):

1) Receive as arguments $(-a)$ IPv4 address and $(-c)$ a subnet-mask.

2) Run over the range of IP addresses.

3) To each IP in the range – if active, print that address, if not, continue to the next address.

Usage:

- $\$\ sudo\ ./discovery\ -a\ <IP>\ -c\ <subnetm>$

Example output:

```
foo@bar:~$ sudo ./discovery -a 10.0.0.0 -c 24
scanning 10.0.0.0/24:
10.0.0.1
10.0.0.2
10.0.0.8
10.0.0.13
10.0.0.138
Scan Complete!
```

Figure 6 – Usage for example for Discovery program.

Part D – ICMP Tunneling – Bonus 5 points

Now that we have learned about the network layer and basic routing concepts in previous sections, we will proceed with some manipulations. We will try to take advantage of a vulnerability that can be found while using the ICMP protocol.

Many firewalls are configured to allow ICMP traffic without any restrictions on packet dropping due to the assumption that ICMP is well-defined and can only be used for standard purposes.

In this section, we will develop a tool for ICMP tunneling, which is intended to extract confidential information from the target's computer without their awareness.

Write a program named $tunnle.c$, which leaks the contents of a file from the target's system as follows:

1) Read the file located on the target's computer.
2) Send its content as a payload via ICMP ECHO-REQUEST packets to 1.2.3.4.
3) There is no need to receive those packets (just perform the tunneling via ICMP).
4) The program should not output anything (the attack should be in the "background").

Usage:

- $\$ sudo ./tunnel$

Example of ICMP Tunneling attack detection:

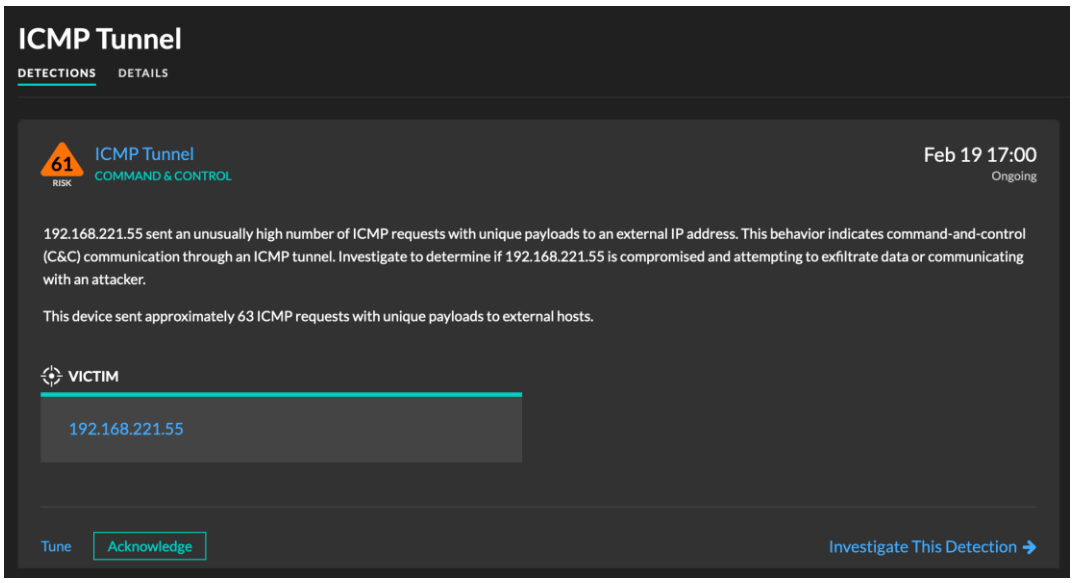

Figure 7 – Network monitoring tool detected ICMP tunneling.
Image Credits:
https://www.extrahop.com/company/blog/2021/detect-and-stop-icmp-tunneling/

Appendix A – Network Layer Protocols Headers

Depicted below are the headers of IPv4, IPv6, ICMP, and ICMPv6.

Internet Protocol version 4 Header																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Version				IHL				DSCP						ECN		Total Length (2 Bytes)															
Identification (2 Bytes)																Flags (3 Bits)			Fragment Offset (13 Bits)											
Time To Live (1 Byte)								Protocol (1 Byte)								Header Checksum (2 Bytes)															
Source IP Address (4 Bytes)																															
Destination IP Address (4 Bytes)																															
Options (Variable)																															
Payload																															

Internet Protocol version 6 Header																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Version				Traffic Class (1 Byte)								Flow Label (20 Bits)																			
Payload Length (2 Bytes)																Next Header (1 Byte)								Hop Limit (1 Byte)					
Source address (16 Bytes)																															
Destination address (16 Bytes)																															
Payload																															

Internet Control Message Protocol and Internet Control Message Protocol version 6 Header																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Type								Code								Checksum															
Payload																															

ICMP Echo Request Header																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Type (8 for IPv4, 128 for IPv6)								Code (Always 0)								Checksum (2 Bytes)															
Identifier (2 Bytes)																Sequence Number (2 Bytes)															
Payload																															

ICMP Echo Reply Header																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Type (0 for IPv4, 129 for IPv6)								Code (Always 0)								Checksum (2 Bytes)															
Identifier (2 Bytes)																Sequence Number (2 Bytes)															
Payload																															

Recommendations:

- Packet crafting:
  It's recommended (but not necessary) to use these structures:
  1. For crafting IPv4 header – use `` `struct iphdr` ``.
  2. For crafting IPv6 header – use `` `struct ip6_hdr` ``.
  3. For crafting ICMP header – use `` `struct icmphdr` ``.
  4. For crafting ICMPv6 header – use `` `struct icmp6_hdr` ``.

Appendix B – Help functions

It's recommended (but not necessary) to use these functions:

- ***Signaling* is a primitive scheme of inter-process communication. A process can send another process a signal. The user may write a *signal handler* function that the process that received the signal will execute. In particular, we recommend (but do not obligate) using signals in your program as follows.**

  The $signal(2)$ function allows the user to handle asynchronous events (for example, keyboard interruption) during the program execution. Therefore, we recommend using this function to handle termination signals (e.g., SIGINT) for displaying ping statistics like the number of pings sent, replies received, any missed replies and etc.

```
#include <signal.h>

…
…

// Function to display statistics on termination

void display_statistics(int signum);

…
…

// Set up signal handler for termination

signal(SIGINT, display_statistics);

…
…
```

- The $poll(2)$ system function allows the process to wait for an event to occur and to wake up the process when the event occurs. It includes a timeout parameter to specify how long the process should wait for an event. Therefore, we recommend using this function to wait for a reply with a timeout.

```
#include <poll.h>
…
…
int result = 0;
struct pollfd pfd;
int timeout = 10000; // Timeout of 10 seconds in milliseconds
…
…
sock = socket(AF_INET, …);
…
…
// Monitor a socket descriptor for input availability
pfd.fd = sock; // socket descriptor
pfd.events = POLLIN; // Watch for data to read
…
…
// Use poll to wait for reply or timeout
int result = poll(&pfd, 1, timeout);
```

```
if (result < 0) {
        // Error occurred
      perror("poll(2) failed");
      …
      exit(errno);
   } else if(result == 0)  {
      // Timeout occurred
        printf("Timeout occurred! No reply received.\n");
        …
   } else {
        if (pfd.revents & POLLIN) {
                printf("Reply received!\n");
      …
      …
      }
   }
…
…
```

Appendix C – Checksum function

A checksum is a small-sized block of data derived from another block of digital data for the purpose of detecting errors that may have been introduced during its transmission or storage. By themselves, checksums are often used to verify data integrity but are not relied upon to verify data authenticity. Below is a simple yet efficient implementation of the checksum function. You can either use this given code, or code your own checksum function.

```c
/*
 * @brief    A checksum function that returns 16 bit checksum for data.
 * @param data    The data to do the checksum for.
 * @param bytes    The length of the data in bytes.
 * @return    The checksum itself as 16 bit unsigned number.
 * @note    This function is taken from RFC1071, can be found here:
 * @note    https://tools.ietf.org/html/rfc1071
 * @note    It is the simplest way to calculate a checksum and is not very strong.
 *        However, it is good enough for this assignment.
 * @note    You are free to use any other checksum function as well.
 *        You can also use this function as such without any change.
*/
unsigned short int calculate_checksum(void *data, unsigned int bytes) {
    unsigned short int *data_pointer = (unsigned short int *)data;
    unsigned int total_sum = 0;
    // Main summing loop
    while (bytes > 1) {
        total_sum += *data_pointer++;
        bytes -= 2;
    }

    // Add left-over byte, if any
    if (bytes > 0)
        total_sum += *((unsigned char *)data_pointer);

    // Fold 32-bit sum to 16 bits
    while (total_sum >> 16)
        total_sum = (total_sum & 0xFFFF) + (total_sum >> 16);

    return (~((unsigned short int)total_sum));
}
```