

Project 3: Network Layer Analysis and Routing Simulator

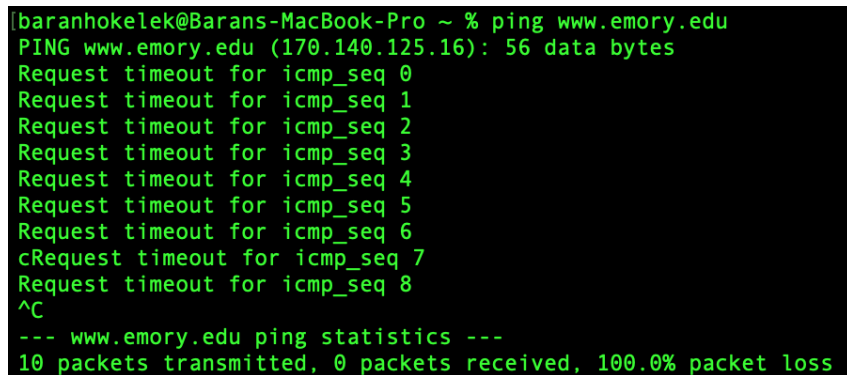
Baran Berkay Hökelek, 0060673

January 7, 2021

Description of the Project

This project is about the network layer. I am tasked to examine the network layer data, the principles behind network layer services, and routing. I practised with Wireshark, as well as a simple network routing simulator.

DISCLAIMER: The URL that was actually assigned to me was www.emory.edu, however, when I tried to ping it, I couldn't get any response - all the packets sent timed out. I contacted Ahnaf about this situation, he told me that the problem might be caused by a firewall on their end, and permitted me to use a different URL. So I chose www.stanford.edu, and everything worked fine.

A terminal window with a black background and green text. The text shows a user attempting to ping www.emory.edu from a MacBook-Pro. The command is 'ping www.emory.edu'. The output shows 'PING www.emory.edu (170.140.125.16): 56 data bytes' followed by ten 'Request timeout for icmp_seq' messages for sequences 0 through 9. The user then presses Ctrl-C (^C). The final output is '--- www.emory.edu ping statistics ---' and '10 packets transmitted, 0 packets received, 100.0% packet loss'.

```
[baranhokelek@Barans-MacBook-Pro ~ % ping www.emory.edu
PING www.emory.edu (170.140.125.16): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
Request timeout for icmp_seq 2
Request timeout for icmp_seq 3
Request timeout for icmp_seq 4
Request timeout for icmp_seq 5
Request timeout for icmp_seq 6
Request timeout for icmp_seq 7
Request timeout for icmp_seq 8
^C
--- www.emory.edu ping statistics ---
10 packets transmitted, 0 packets received, 100.0% packet loss
```

Figure 1: My attempt at pinging Emory's URL, and the timeout responses.

Part 1: ICMP Analysis

1.a: Ping Analysis

```
baranhokelek@Barans-MacBook-Pro ~ % ping -c 5 www.stanford.edu
PING 89wyd637cdel.wpeproxy.com (141.193.213.21): 56 data bytes
64 bytes from 141.193.213.21: icmp_seq=0 ttl=51 time=71.097 ms
64 bytes from 141.193.213.21: icmp_seq=1 ttl=51 time=62.102 ms
64 bytes from 141.193.213.21: icmp_seq=2 ttl=51 time=58.267 ms
64 bytes from 141.193.213.21: icmp_seq=3 ttl=51 time=61.216 ms
64 bytes from 141.193.213.21: icmp_seq=4 ttl=51 time=57.939 ms

--- 89wyd637cdel.wpeproxy.com ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 57.939/62.124/71.097/4.769 ms
```

Figure 2: Command-line prompt of the 5 pings and their associated statistics.

1. The 3 layers are Network, Link and Physical. ICMP is a supporting protocol to IP; and it operates on top of IP, but they are both part of the network layer.

```
> Frame 10: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface en0, id 0
> Ethernet II, Src: Apple_cc:00:27:c4:b3:01:cc:00:27, Dst: TaicangT_99:14:80 (b4:26:5d:99:14:80)
> Internet Protocol Version 4, Src: 192.168.1.15, Dst: 141.193.213.21
> Internet Control Message Protocol
```

Figure 3: Wireshark's depiction of the 3 main layers and the ICMP on top of IP.

2. TTL (Time To Live) is the number of remaining hops allowed for a package. It can be used to limit the lifespan of a data in a network. It resides on the network layer. Its size and content is the same in IPv4 and IPv6, just the positions vary.

```
Internet Protocol Version 4, Src: 192.168.1.15, Dst: 141.193.213.21
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 84
    Identification: 0xd594 (54676)
    Flags: 0x00
    Fragment Offset: 0
    Time to Live: 64
    Protocol: ICMP (1)
    Header Checksum: 0x0086 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 192.168.1.15
    Destination Address: 141.193.213.21
```

Figure 4: It can be seen that the network layer (IP) contains the TTL field.

3. ICMP is IP Protocol 1, meanwhile TCP is IP Protocol 6 and UDP is IP Protocol 17. They are all different protocols with different behaviour. On top of that, ICMP acts like a part of the IP whose only job is to signal between hosts in case of an error or an echo request. We don't need a server to be able to listen to multiple error message ports from a single host, so ICMP doesn't have port numbers.
4. The length of the data field on the ICMP part Type-8 is 48 bytes. It is the exact same data in all of the requests:
08090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f3031323334353637
5. The minimum TTL for the messages to reach to my URL destination turned out to be 13.

```

[baranhokelek@Barans-MacBook-Pro ~ % ping -c 1 -m 12 www.stanford.edu
PING 89wyd637cdel.wpeproxy.com (141.193.213.20): 56 data bytes
36 bytes from 212.187.202.54: Time to live exceeded
Vr HL TOS Len ID Flg off TTL Pro cks Src Dst
4 5 00 5400 008f 0 0000 01 01 948d 192.168.1.15 141.193.213.20

^C
--- 89wyd637cdel.wpeproxy.com ping statistics ---
1 packets transmitted, 0 packets received, 100.0% packet loss
[baranhokelek@Barans-MacBook-Pro ~ % ping -c 1 -m 13 www.stanford.edu
PING 89wyd637cdel.wpeproxy.com (141.193.213.20): 56 data bytes
64 bytes from 141.193.213.20: icmp_seq=0 ttl=52 time=55.864 ms

--- 89wyd637cdel.wpeproxy.com ping statistics ---
1 packets transmitted, 1 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 55.864/55.864/55.864/0.000 ms

```

Figure 5: The experiment to determine the minimum TTL. As seen, 13 is just fine but 12 is problematic.

6. The ID numbers stay the same across successive requests, meanwhile the seq numbers increase by 1(big endian) or 256(little endian) between each request-reply pair.

10	1.865456	192.168.1.15	141.193.213.21	ICMP	98	Echo (ping) request	id=0xc139, seq=0/0, ttl=64 (reply in 12)
12	1.936390	141.193.213.21	192.168.1.15	ICMP	98	Echo (ping) reply	id=0xc139, seq=0/0, ttl=51 (request in 10)
22	2.868658	192.168.1.15	141.193.213.21	ICMP	98	Echo (ping) request	id=0xc139, seq=1/256, ttl=64 (reply in 23)
23	2.930578	141.193.213.21	192.168.1.15	ICMP	98	Echo (ping) reply	id=0xc139, seq=1/256, ttl=51 (request in 22)
24	3.873152	192.168.1.15	141.193.213.21	ICMP	98	Echo (ping) request	id=0xc139, seq=2/512, ttl=64 (reply in 25)
25	3.932216	141.193.213.21	192.168.1.15	ICMP	98	Echo (ping) reply	id=0xc139, seq=2/512, ttl=51 (request in 24)
26	4.873835	192.168.1.15	141.193.213.21	ICMP	98	Echo (ping) request	id=0xc139, seq=3/768, ttl=64 (reply in 27)
27	4.934860	141.193.213.21	192.168.1.15	ICMP	98	Echo (ping) reply	id=0xc139, seq=3/768, ttl=51 (request in 26)
30	5.877809	192.168.1.15	141.193.213.21	ICMP	98	Echo (ping) request	id=0xc139, seq=4/1024, ttl=64 (reply in 31)
31	5.934814	141.193.213.21	192.168.1.15	ICMP	98	Echo (ping) reply	id=0xc139, seq=4/1024, ttl=51 (request in 30)

Figure 6: The information about the ping request & reply packets. The ID & Seq numbers can be seen on the right side.

1.b: Traceroute Analysis

```

[baranhokelek@Barans-MacBook-Pro ~ % traceroute -I www.stanford.edu
traceroute: Warning: www.stanford.edu has multiple addresses; using 141.193.213.20
traceroute to 89wyd637cdel.wpeproxy.com (141.193.213.20), 64 hops max, 72 byte packets
1 dsldevice.lan (192.168.1.1) 2.410 ms 1.349 ms 1.130 ms
2 195.87.128.37 (195.87.128.37) 3.990 ms 3.531 ms 3.730 ms
3 10.135.53.254 (10.135.53.254) 16.654 ms 16.499 ms 17.008 ms
4 * * *
5 10.190.52.161 (10.190.52.161) 39.945 ms 14.804 ms 15.495 ms
6 10.135.53.253 (10.135.53.253) 19.188 ms 17.654 ms 18.792 ms
7 10.135.52.169 (10.135.52.169) 16.531 ms 17.135 ms 15.628 ms
8 ae5-17-xcr1.izm.cw.net (195.2.31.133) 16.059 ms 16.006 ms 16.958 ms
9 ae28-xcr1.sof.cw.net (195.2.18.49) 35.550 ms 34.197 ms 30.993 ms
10 ae-7-bar2.sofia2.level3.net (4.68.70.13) 31.586 ms 31.070 ms 31.891 ms
11 ae-1-3103.edge1.budapest1.level3.net (4.69.143.118) 62.682 ms 63.939 ms 63.742 ms
12 212.187.202.54 (212.187.202.54) 57.186 ms 90.991 ms 57.028 ms
13 141.193.213.20 (141.193.213.20) 55.766 ms 54.676 ms 55.347 ms

```

Figure 7: The compnd-line prompt of the traceroute command and the path. Notice that no response came from the 4th router.

7. The ICMP header of a TTL Exceeded packet is 8 bytes long.

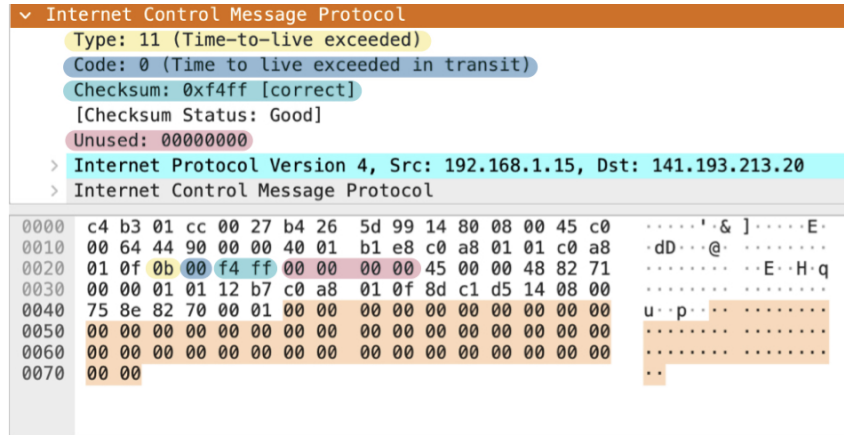


Figure 8: The ICMP Header is highlighted to show the different fields.

- Traceroute sends multiple packets with a range of TTLs. When one packet's TTL ends, the router that it dies in sends a "Time-to-live-exceeded" message, which enables my computer to see all the routers that the packets go through.



Figure 9: When the packet dies on a server, that server sends the TTL exceeded message

- Each router along the path is probed exactly 3 times.

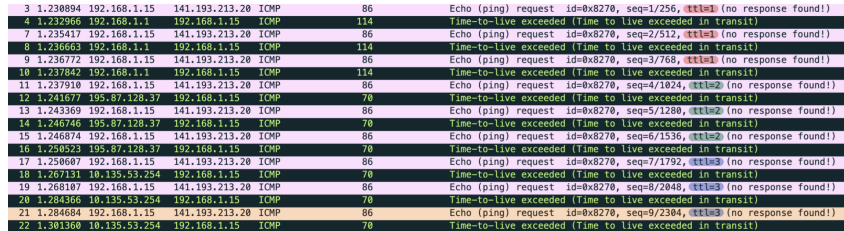


Figure 10: It can be seen that for each TTL value, 3 packets were sent.

- There has been one link with a significant amount of delay, which is the connection between IP addresses 10.135.53.254 and 10.190.52.161, at TTL=4. After each packet sent, traceroute waited for 5 seconds to get a response, but haven't received anything. However, when the packets with TTL=5 have been sent, a response came shortly after. This indicates that there might have been a problem or a delay on the route from the TTL=4 router to me.

23	1.301717	192.168.1.15	141.193.213.20	ICMP	86	Echo (ping) request	id=0x8278, seq=18/2568, ttl=4 (no response found!)	
28	6.307016	192.168.1.15	141.193.213.20	ICMP	86	Echo (ping) request	id=0x8278, seq=11/2816, ttl=4 (no response found!)	
31	11.3122	192.168.1.15	141.193.213.20	ICMP	86	Echo (ping) request	id=0x8278, seq=12/3072, ttl=4 (no response found!)	
34	16.3141	192.168.1.15	141.193.213.20	ICMP	86	Echo (ping) request	id=0x8278, seq=13/3328, ttl=4 (no response found!)	
39	16.3537	192.168.1.15	141.193.213.20	ICMP	71	Time-to-live exceeded (Time to live exceeded in transit)		No TTL Exceeded
36	16.3546	192.168.1.15	141.193.213.20	ICMP	86	Echo (ping) request	id=0x8278, seq=14/3584, ttl=5 (no response found!)	
37	16.3691	10.198.52.161	192.168.1.15	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)		Timeout Ping
39	16.3695	192.168.1.15	141.193.213.20	ICMP	65	Echo (ping) request	id=0x8278, seq=15/3840, ttl=5 (no response found!)	
39	16.3847	10.198.52.161	192.168.1.15	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)		

Figure 11: No response came after the TTL=4 packets were sent.

Part 2: Routing Implementation

Naive Flooding Algorithm

I did not write any extra code, as the code was already given for this algorithm.

- The total communication cost represents the sum of the total costs of all of the packets routed from all nodes. The path cost 9 is equal to the cost of the path with the least cost; however before discovering that path, our algorithm also routed 1 packets to node 2 (which costs 5), and from node 2 another packets to node 4 (which costs an additional 8), which brings the total cost to $9+5+8 = 22$.
- Here is a summary of the events that happen in Topology 2 when NaiveFlooding is used.
 - Node 1 routes packets to nodes 2 and 3.
 - Node 2 routes a packet to node 3. (Node 3 also routes a packet to node 2, but it is ignored for simplicity.)
 - Node 3, seeing that the previous node for that packet is node 2, routes that packet **back** to node 1.
 - Node 1, seeing that the previous node for that packet is node 3, routes that packet to node 2.
 - Node 2 **again** routes that packet to node 3.
 - Repeat.

As it can be clearly seen, the packet gets stuck in an infinite loop between nodes 1,2,3; never reaching to node 4 and causing the algorithm to terminate. The reason for this infinite loop is the NaiveFlooding algorithm's lack of bookkeeping. Had it recorded **all** the nodes it previously encountered (instead of just the previous one), this problem would've been avoided.

Flooding Algorithm

My job was simple here. I just created a variable called "flag", initialized it as 1, and switched it to 0 when "selectNeighbors" is called for the first time. Then I applied the exact same filtering process to the neighbors array as in NaiveFlooding algorithm, but only when the flag is 1.

```

int flag = 1;
@Override
public List<NeighborInfo> selectNeighbors(String origin, String destination,
String previousHop, List<NeighborInfo> neighbors) {
    if(flag == 1)
    {
        flag = 0;
        return neighbors.stream()
            .filter(n -> !n.address.equals(previousHop))
            .collect(Collectors.toList());
    }
    return null;
}

```

13. I have expected Flooding algorithm to give this path as a result because it is the shortest path possible and I didn't tamper with the cost-counting process. What I didn't expect was the total communication cost of 28 (I expected 24), because I forgot to consider the possibility of one packet pre-empting the other and causing unintended routing on some nodes. In this case, the packet being sent from node 3 to node 2 beats the one sent from node 1 to node 2, causing node 2 to route to nodes 1 and 4 instead of 3 and 4, thus increasing the total cost.

Naive Minimum Cost Algorithm

I sorted the "neighbors" array by comparing the cost of its elements, filtered out the previous node from the list, and returned the first element since it was bound to have the smallest cost.

```

@Override
public List<NeighborInfo> selectNeighbors(String origin, String destination,
String previousHop, List<NeighborInfo> neighbors) {
    return new ArrayList<NeighborInfo>() {
        {
            add(neighbors.stream()
                .filter(n -> !n.address.equals(previousHop))
                .sorted(Comparator.comparingInt(o -> o.cost))
                .collect(Collectors.toList()).get(0));
        }
    };
}

```

14. This algorithm suffers the same fate as NaiveFlooding in Topology 2. A packet travels from node 1 to 3, then from 3 to 2, and then back from 2 to 1, and so on.

Minimum Cost Algorithm

This algorithm creates a List called exclusionSet, then adds the previous node in it. After that, it checks all the neighbors and tries to find one that hasn't been excluded before. If it can find multiple elements like that, it returns a list containing the one with the smallest

cost. If all the neighbors have been excluded, it chooses one of the neighbors at random, and returns a list with that randomly chosen neighbor in it.

```
List<String> exclusionSet = new ArrayList<>();
NeighborInfo chosen = null;
@Override
public List<NeighborInfo> selectNeighbors(String origin, String destination,
String previousHop, List<NeighborInfo> neighbors) {

    // adds the previous node to the exclusion set if it isn't there already
    if(!exclusionSet.contains(previousHop) && previousHop != null)
    {
        exclusionSet.add(previousHop);
    }

    // finds neighbors that are not in the exclusion set
    List<NeighborInfo> nonExcludedNs = neighbors.stream()
        .filter(n -> !exclusionSet.contains(n.address)).collect(Collectors.toList());

    // If all the neighbors are excluded, picks one at random
    if(nonExcludedNs.size() == 0)
    {
        chosen = neighbors.get(rand.nextInt(neighbors.size()));
    }

    // else, picks the one with the smallest cost
    else {
        nonExcludedNs.sort(Comparator.comparingInt(o -> o.cost));
        chosen = nonExcludedNs.get(0);
    }

    // adds the selection to the exclusion set if it isn't there already
    if(!exclusionSet.contains(chosen.address))
    {
        exclusionSet.add(chosen.address);
    }

    // prints the exclusion set
    //System.out.println(Arrays.toString(exclusionSet.toArray()));

    // returns an array with the chosen node in it
    return new ArrayList<>() {{
        add(chosen);
    }};
}
```

15. Exclusion sets for each node at the end of simulation is as follows:

- Node 1: [Node 3, Node 2]
- Node 2: [Node 3, Node 1, Node 4]
- Node 3: [Node 1, Node 2]
- Node 4: No exclusion set