Lab 2 Traceroute

OS Installation

This project has been tested to work on Linux and Mac. If you're on Windows, you need to install Windows Subsystem for Linux (WSL). Please refer to the document Running Linux on Your Laptop.

Note on File Location (Windows Users)

After downloading the starter code (see below), do not run Linux commands on files located in the Windows filesystem (those under `/mnt/c`). This can be slow because you are running Linux commands on the Windows disc. Instead,

1. Create a project directory in your WSL home using `mkdir -p ~/projects`.
2. Copy the extracted starter code into `~/projects` using `cp -r /mnt/c/Users/<username>/Downloads/cs168-sp26-proj1-traceroute ~/projects/`. You can also drag the folder into the Explorer sidebar tab if you are using VSCode.
3. Navigate to the folder in your terminal using `cd ~/projects/cs168-sp26-proj1-traceroute` and work from there.

Python Installation

This project has been tested to work on Python 3.11 or later. You can run `python3 --version` or `python --version` in your terminal to check your Python version.

Starter Code

Download the starter code here.

In your terminal, use `cd` to navigate to the `Lab2-traceroute` directory. All of the Python commands should be run from this directory. To check that your setup works, in your terminal, run:

```
sudo python3 traceroute.py cmu.edu
```

If you see something like this, everything should be set up correctly:

```
traceroute to cmu.edu (128.2.42.10)
 1: *
 2: *
 3: *
(some lines omitted...)
```

```
26: *
27: *
28: *
29: *
30: *
```

You should only edit `traceroute.py`. There are comments marked “TODO” clearly indicating the places where you should fill in code.

Guidelines:

- Don’t modify any other files.
- Don’t add any new files.
- Don’t add any imports.
- Don’t edit any code outside the sections indicated by the comments.
- Don’t add any hard-coded global variables.
- Adding helper methods is fine (and encouraged).

Assignment Overview

You will be implementing the `traceroute` function in `traceroute.py`. Some useful helper functions for sending and receiving packets are implemented in `util.py` (don’t modify it).

Your Task: Traceroute

Your goal is to implement the `traceroute` function so that it reveals all the routers between your computer and the specified destination. Traceroute uses the Time-to-Live (TTL) field in an IP packet (a probe) to trigger ICMP "Time Exceeded" messages from routers.

Traceroute takes in three arguments:

- `ip` (string) is the specified destination.
- `sendsock` is an object you can use to send packets (more details below).
- `recvsock` is an object you can use to receive packets (more details below).

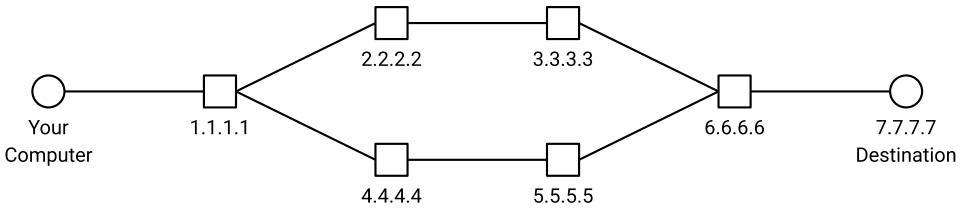
Traceroute returns a list of lists, with the following properties:

- The `i` th sublist contains the IP addresses of all the routers you found that are distance `i+1` away.
- For example, the 0th sublist contains all the routers that are 1 hop away.
- The routers inside each sublist can be in any order.
- If no routers were found for a certain distance, the `i` th sublist can be empty.
- If `ip` is discovered, the final sublist should just be `[ip]`.

Traceroute should also call `util.print_result` (more details below) on each sublist to display the routers at each distance. We won't grade the printed output, but it's useful for debugging (also, it looks cool).

Example of Calling Traceroute

Example: Suppose the network topology looks like this.



Assuming no errors occur, and our probes hit every router, then calling `traceroute` with `ip=7.7.7.7` should return something like this:

```
[["1.1.1.1"],
 ["2.2.2.2", "4.4.4.4"],
 ["3.3.3.3", "5.5.5.5"],
 ["6.6.6.6"],
 ["7.7.7.7"]]
```

Also, `traceroute` should print output that looks something like this:

```
traceroute to 7.7.7.7
 1: 1.1.1.1
 2: 2.2.2.2
    4.4.4.4
 3: 3.3.3.3
    5.5.5.5
 4: 6.6.6.6
 5: 7.7.7.7
```

Sending Packets

`sendsock` is an object you can use to send outgoing packets. It has the following useful methods:

- `sendsock.set_ttl(ttl)` sets the TTL to that number for all subsequent outgoing packets.
  - `ttl` (integer) is the TTL to set.
- `sendsock.sendto(msg.encode(), (ip, port))` sends an outgoing packet.
  - `msg` (string) is the UDP payload of the packet. We call `encode` to convert the string to raw bytes.
  - `ip` (string) is the destination IP address.
  - `port` (integer) is the destination port.

Example usage:

```
# Send a packet "Hello" to 4.4.4.4, port 33434, with TTL 12.
sendsock.set_ttl(12)
sendsock.sendto("Hello".encode(), ("4.4.4.4", 33434))

# Send a packet "Potato" to 5.5.5.5, port 33464, with TTL 20.
sendsock.set_ttl(20)
sendsock.sendto("Potato".encode(), ("5.5.5.5", 33464))
```

Receiving Packets

`recvsock` is an object you can use to receive incoming packets. It has the following useful methods:

- `recvsock.recv_select()` checks if there are any incoming packets available to be received.
    - If there is at least one packet to be received, it returns True.
    - Otherwise, the function waits until a packet is available, or a timeout expires.
    - If the timeout expires, it returns False.
- `buf, address = recvsock.recvfrom()` receives a single incoming packet.
    - It returns `buf`, the raw bytes of the packet (e.g. IPv4 header, followed by a UDP or ICMP header, followed by payload). To print out the bytes, you can call `.hex()` on the raw bytes.
    - It also returns `address`, a tuple containing the IP address and port that sent the packet.
    - If there is no packet to be received, the function throws an exception. Therefore, you should always call `recvsock.recv_select()` before calling `recvsock.recvfrom()`.

Example usage:

```
if recvsock.recv_select():  # Check if there's a packet to process.
    buf, address = recvsock.recvfrom()  # Receive the packet.

    # Print out the packet for debugging.
    print(f"Packet bytes: {buf.hex()}")
    print(f"Packet is from IP: {address[0]}")
    print(f"Packet is from port: {address[1]}")
```

Print Result

`print_result(routers: list[str], ttl: int)` can be used to print nicely-formatted output.

It takes in a TTL and a list of IP addresses (all the routers found by probing with that TTL).

Your implementation should call this function once for every TTL you probe.

Example usage:

```
util.print_result(["128.2.255.210", "128.2.42.10"], 7)
```

This will cause the TTL, the IP addresses of both machines, and their corresponding names, to be printed out nicely, like this:

```
 7: POD-D-DCNS-CORE2.GW.CMU.NET (128.2.255.210)
    CMU-VIP.ANDREW.CMU.EDU (128.2.42.10)
```

Stage 1: Run Traceroute Manually

Now that we know how to send and receive packets, let's try running traceroute manually. For this stage, you can temporarily comment out these lines of starter code:

```
# for ttl in range(1, TRACEROUTE_MAX_TTL+1):
#     util.print_result([], ttl)
# return []
```

1. In `traceroute`, write code that sends a packet to `ip` with a TTL of 1 and prints out the reply packet (as raw hex bytes). The payload can be any short message you want, e.g. "Potato." The starter code has destination port numbers you can use.
2. Then, try running that code:

```
3.  sudo python3 traceroute.py cmu.edu
```

4. Copy-paste the bytes of the reply packet into an online packet decoder like this one. You might have to ask the decoder to parse the packet as an IPv4 packet (since we've already stripped away the Layer 2 header).
5. Use the packet decoder and your knowledge of headers to read this packet. Some things to investigate:
    - Who sent the reply packet? Was it cmu.edu, or some intermediate router?
    - Did you discover an intermediate router? If so, what is its IP address? How did you learn that IP address from the headers?
    - What is the intermediate router trying to tell you? Is it trying to report an error? If so, how do you know what it's trying to say?
    - What are all the different headers in this packet, and why are they here?
6. Back in `traceroute`, try changing the TTL to 2, 3, 4, or 5, and re-running the code to send out the packet with your new TTL:

```
7.  sudo python3 traceroute.py cmu.edu
```

   You might have to try a few different TTLs here, since some routers along your path to cmu.edu might not reply to you.

8. Copy-paste the bytes of the new reply packet into the packet decoder, and investigate the packet.
    - Did you discover a new intermediate router?

    - What is the packet trying to tell you?
9. Back in `traceroute`, try changing the TTL to 30. This should be enough hops to reach cmu.edu. Re-run the code to send out the packet with your new TTL:

```
sudo python3 traceroute.py cmu.edu
```

11. Copy-paste the bytes of the new reply packet into the packet decoder, and investigate the packet.
    - Who sent the reply packet? Was it cmu.edu, or some intermediate router?
    - What is the packet trying to tell you? How do you know?

Stage 2: Parsing Packets

When you ran traceroute manually, each time you received a packet, you probably had to interpret the bytes of the various headers. Your traceroute implementation will need to do this parsing in code.

Your Task

To help you parse headers in code, we've provided three helper classes, corresponding to the three relevant protocols in the packets you'll receive: `IPv4`, `ICMP`, and `UDP`.

These classes are missing constructors. Your job is to fill in the constructors (the `__init__` functions).

The constructors take in `buffer`, which are the raw bytes of the packet header. For example, the constructor in `ICMP` takes in the bytes of the ICMP packet header.

Your constructor code should initialize all of the instance variables by parsing the bytes of the given packet.

Hints

`buffer` is a raw byte array, but you'll need to extract individual bits. This line of code converts `buffer` to a bitstring, `b`, that you can perform string operations on:

```
b = ''.join(format(byte, '08b') for byte in [*buffer])
```

This line of code converts a bitstring into an integer:

```
bitstring = '10010'          # A string of 1s and 0s.
number = int(bitstring, 2)   # 2, because it's a base-2 number.
print(number)                # Prints 18.
```

Testing and Debugging

From Stage 1, you have code that sends a packet and receives a reply packet.

To check your parsing code, you can call the constructor on the raw bytes you receive (slicing if needed to extract specific headers), and print the output. The classes already have `__str__` methods implemented for you to help the output look nice.

Then, you can compare the printed output with the online packet decoder to see if you're parsing packets correctly.

Stage 3: Basic Traceroute

Your Task

Fill in `traceroute` to discover all routers between you and the destination `ip`.

Some reminders:

- The payload can be any short message you want, e.g., "Potato."
- For each TTL, you should send `PROBE_ATTEMPT_COUNT` packets with that TTL.
- An IP should only be listed once per sublist (no duplicates in a sublist).
- When you see a response from the destination (`ip`), you should stop and return the routers you discovered along the way.
- The constructors you just wrote will be helpful for parsing packets.

Testing and Debugging

You can run your implementation to see if your output looks similar to the output of a real-world traceroute:

```
traceroute cmu.edu                  # A real traceroute.
sudo python3 traceroute.py cmu.edu  # Your traceroute.
```

Your output probably won't be exactly the same, because packets could travel along different paths each time you run traceroute.

Submission and Grading

Submit the `traceroute.py` file on Canvas, and a report that documents and explains your code in detail, and any problems that you encountered during the project. Please use this Lab Report Template to write your report.