Networks & Operating Systems Essentials 2 (NOSE 2)

Lab 2: IP addresses and the
Internet Control Message Protocol (ICMP)

The aim of this lab is to familiarise you with command-line parsing from a Python program, as well as with command-line tools that can be used to examine part of the network state of your computer, related to the Network layer of the OSI reference model. Specifically, we will look at IP addresses, the ICMP protocol, and tools for examining connectivity and routing to remote hosts.

Command Line Option Parsing in Python

The main focus of this session is on making sure that you know how to parse command line arguments from your Python programs. Command-line arguments are arguments that you pass to a program at invocation time like so:

- ipconfig /all
- arp /a
- …

In the above examples, the first word in every line is the name of the program to execute, and all subsequent words/terms are its arguments; e.g., ipconfig is executed with one argument ("/all"), arp is also executed with one argument ("/a"), etc.
Parsing of command-line arguments can be accomplished in a number of ways:

a. Through *sys.argv*
b. By using the *getopt* Python library/module
c. By using the *argparse* Python library/module

As a running example, in the following we will create a program that accepts two numerical arguments from the command line and prints their sum.

Preamble

To execute your code from the command line, you will need to be in an environment where Python is in the PATH variable for your shell to find. Fortunately, the lab PCs and also the Virtual Windows Desktop have Anaconda 3 installed. If you want to work on your own individual machine I would recommend to firstly install Anaconda 3 on your machine: (Windows: https://docs.anaconda.com/anaconda/install/windows/ , Mac: https://docs.anaconda.com/anaconda/install/mac-os/ , GNU/Linux: https://docs.anaconda.com/anaconda/install/linux/)

Windows users: To open a command prompt window with an appropriate Python environment, open your Start menu (click on the little Windows icon at the bottom left of your screen), then open up the Anaconda 3 menu, and choose either "Anaconda Powershell Prompt (Anaconda3)" or "Anaconda Prompt (Anaconda3)". The Python interpreter binary will be in the PATH variable (so you can just type "python X.py" to execute "X.py") and it will have access to all modules/libraries installed through Anaconda3. For ease of use, first navigate to where you have your scripts stored (e.g., if they are stored under "M:\NOSE2\lab2", type "M:" and hit Enter to change to the

M drive, then “cd NOSE2\lab2”). Ask your tutor for help if you have any issues with the above.

MacOS users: To open a command prompt with python3 (assuming that you have already installed the full Anaconda 3 distribution) you can do it by simply opening a Terminal from User/Applications/Utilities/Terminal. When you open your terminal type “which python3” and you should get the full path of where your python3 is installed (e.g., /Users/akm/opt/anaconda3/bin/python3). After you do that then navigate in the folder where your python script was saved. More info on navigating in your terminal can be found here : https://swcarpentry.github.io/shell-novice/02-filedir/index.html).

GNU/Linux users: You should already be familiar with the shell and how to verify your python3 installation and PATH. (if not, check this : https://linuxconfig.org/check-python-version). Then, navigate to the folder in which your script was saved as advised to the MacOS users.

sys.argv

This is the most simplistic and barebones method of the three. Every time a Python program is executed, the runtime stores all command-line arguments in a list called *argv* (part of the *sys* module), which you can access from your program just like any other list. The first element in the list is the name of the program. For example, write the following code, save it in a file, and try to execute it from the command line with various arguments:

```
import sys

print('#arguments: ' + str(len(sys.argv)))
for arg in sys.argv:
   print(arg)
```

As such, you can access all passed arguments and parse them in your code; e.g.:

```python
import sys

def usage():
   print('Usage: <int> <int>')
   sys.exit(1)


val1 = 0
val2 = 0

if len(sys.argv) != 3:
   print('Wrong number of arguments')
   usage()

try:
   val1 = int(sys.argv[1])
   val2 = int(sys.argv[2])
except ValueError:
   print('Argument is not an int')
   usage()

print('Sum: ' + str(val1 + val2))
```

The sys.argv approach outlined above works quite well when all arguments are positional; i.e., when the position of an argument in the list also signifies its semantics. However, it is often the case that we want to be able to provide arguments in an arbitrary order, or to omit certain arguments, etc. In this case, it is better to use one of the many command line parsing modules (two of which are outlined below); keep in mind, though, that these modules as well are merely providing functionality around sys.argv.

getopt

Assume that you want to be able to execute your Python program like so:

- python my.py -a 10 -b 20
- python my.py -a 10
- python my.py -b 20

The semantics are that we provide values for the first and second operand in the addition (a, b respectively), with 0 being the default value when none is defined for an operand.

Along comes the getopt module. The getopt() method looks like this:

- getopt.getopt(args, shortopts, longopts=[])

where:

- args is the list of command-line arguments,
- shortopts is a string specifying the option letters and whether they require a value or not. For example, a string of “a:b” means that the program accepts two options, called ‘a’ and ‘b’ (given as ‘-a’ and ‘-b’ on the command line). Additionally, the first of these also expects a value; i.e., the program would be executed like so:
    - python my.py -a x
    - python my.py -b

    - python my.py -b -a x
    - etc.
- longopts defines extended versions of the shortopts, which must be prepended with '--' on the command line.

The getopt.getopt() function then returns two elements: a list of (option, value) pairs, and a list of arguments not recognised/defined in the option strings above.
Back to our simple addition program:

```
import getopt
import sys

def usage():
   print("Usage: [-a|--operand_a <int>] [-b|--operand_b <int>"])
   sys.exit(1)

val1 = 0
val2 = 0

try:
   opts, vals = getopt.getopt(sys.argv[1:], "a:b:",
      ["operand_a=", "operand_b="])
except getopt.GetoptError as err:
   print(str(err))
   usage()

if len(vals) > 0:
   print("Unknown options: " + str(vals))
   usage()

try:
   for opt, val in opts:
      if opt in ("-a", "--operand_a"):
         val1 = int(val)
      elif opt in ("-b", "--operand_b"):
         val2 = int(val)
except ValueError:
   usage()

print(str(val1 + val2))
```

argparse

The above provides a good degree of flexibility to the user, but is not that much easier on the programmer. Thus, Python 3.x introduced a new module, called argparse, that aims to address some of the shortcomings of earlier offerings. To see this in action, here is code that implements the same functionality as above using argparse. The gist of it is that you first create a parser object, then you add arguments to it, then you parse the command line arguments. For each argument, you can define its short version, its long version, its type, a default value, a help message, whether it is required or optional, etc. The parser then handles such aspects as type checking, exception handling, etc., on its own, and even produces informative error messages and a handy help message when something goes wrong.

```
import argparse
import sys

val1 = 0
val2 = 0

ap = argparse.ArgumentParser()
ap.add_argument("-a", "--operand_a", type=int, default=val1)
ap.add_argument("-b", "--operand_b", type=int, default=val2)
args = vars(ap.parse_args())

print(str(int(args["operand_a"]) + int(args["operand_b"])))
```

Experiment with the above to familiarise yourselves with at least sys.argv and argparse. Provide different arguments, both correct and incorrect, to see how these modules handle errors and edge cases. You can also consult the relevant documentation at:

- https://docs.python.org/3/library/sys.html#sys.argv
- https://docs.python.org/3/library/getopt.html
- https://docs.python.org/3/library/argparse.html

Internet Control Message Protocol (ICMP)

The Internet Protocol (IP) provides addressing and end-to-end connectivity at Network layer of the OSI reference model. In order to achieve this functionality, it relies on a number of other protocols to provide it with routing and control functions. Examples of such protocols include the Routing Information Protocol (RIP) and Open Shortest Path First (OSPF) protocol, the Internet Group Management Protocol (IGMP), and the Internet Control Message Protocol (ICMP). In this lab we will focus on the latter.

ICMP is defined in RFC 792, with updates in RFC 950 and optional extensions in RFC 1191 (Path MTU Discovery), RFC 1256 (Router Discovery) and RFC 1393 (Traceroute using an IP option). In its basic form, it allows for error reporting and simple queries to be executed over an IP internet. ICMP messages are encapsulated in IP datagrams; that is, ICMP acts as a higher-level protocol than IP. However, ICMP is an integral part of IP and implemented by all networked devices that also implement the IP. It should also be noted that, although ICMP allows IP-enabled devices to report errors, it does not provide any reliability guarantees for IP; as discussed in the lectures, IP is best-effort and as such messages can be lost, repeated, reordered, etc. Keeping in line with the above, RFC 792 states that ICMP messages *may* be sent to report IP errors; in practice, though, most routers will almost always use ICMP message to do so. Just like IP comes in two versions (IPv4, IPv6), so does ICMP (sometimes referred to as ICMPv4 and ICMPv6); although the two versions have some differences, they are quite similar in operation and indeed in packet structure. Last, there are some safeguards in place to guarantee that the use of ICMP will not inadvertently affect the operation of the network: ICMP messages are never used to report errors on ICMP datagrams (avoiding infinite loops), and they are never sent in response to datagrams with a zero/broadcast/multicast source or destination IP addresses.

In general, every ICMP packet has a Type (8 bits) and a Code (8 bits), followed by a 16-bit checksum, followed by a 32-bit field that is operation-dependent, and followed by up to 568 bytes of data (576 bytes in total). The first two fields together (Type, Code) define the operation to be carried out by devices receiving the packet.

As alluded to earlier, ICMP messages are used either for error reporting, or for informational messages. Examples of error conditions conveyed by ICMP messages include:

- Destination unreachable (Type: 3): Generated when an IP datagram cannot be delivered to its destination (e.g., because no path can be found leading to it from the sender); the value of the Code field defines the exact error.
- Source quench (Type: 4): Generated when a receiving host is overwhelmed and requests that the sender slows down its sending rate.
- Redirect (Type: 5): Generated when a router detects that a better route exists between a sender and destination, possibly not including said router.
- Time exceeded (Type: 11): Generated when an IP datagram exceeds its Time-To-Live (TTL) field and is thus discarded before reaching its destination.
- Parameter problem (Type: 12): Generated when an IP datagram contained an invalid/unknown parameter, or when no other error type is applicable.

On the other hand, informational messages come in pairs: a query/request message and a response/reply message. Examples include:

- Echo (Type: 8 (request), 0 (reply)): Used to test connectivity to a remote device. Also known as PING/PONG by the name of a command-line program (ping) that uses these messages.
- Router discovery (Type: 10 (request), 9 (reply)): Used to request that any routers send updates of their existence and capabilities to hosts.
- Timestamp (Type: 13 (request), 14 (reply)): Used to exchange clock synchronization information.
- Address mask (Type: 17 (request), 18 (reply)): Used to query the subnet mask of a remote device.
- Traceroute (Type: 30): Used to discover the path (routers) between a sender and receiver.

Finally, the data contained in the message body depends on the Type/Code of the message, but quite often it includes the header and part of the body of the IP datagram that caused the generation of the ICMP packet[1]. As mentioned above, this part of the ICMP packet is usually limited to 568 bytes.

ICMP packets have also been used in the past for somewhat more nefarious reasons. The notorious "ping of death[2]" denial-of-service (DoS) attack consists of sending large

[1] For more details, please see: http://www.tcpipguide.com/free/t_ICMPMessageTypesandFormats.htm
[2] https://en.wikipedia.org/wiki/Ping_of_death

or specially malformed Echo Request packets to the target host. Also, ICMP packets have been as covert channels[3,4], allowing for data to be exchanged between two hosts without requiring the establishment of “proper” network (e.g., TCP) connections; unless deep packet inspection is performed by the firewall(s) on the path (which is quite seldom the case with ICMP), such channels can go unnoticed for a long time.

Ping and Traceroute

In this lab you will be using two command-line utilities; namely:

- ping: used to send an Echo request to a remote host
- tracert: used to discover the route to a remote host

The first program (ping) does pretty much what it says on the box. The sender creates an ICMP packet, sets its Type to 8 and Code to 0, then fills in the body with a data structure containing an 8-bit identifier and an 8-bit sequence number (so as to allow hosts to have multiple ping “sessions” active, while keeping track of responses and packet losses), plus some random data. The destination then responds with a copy of the incoming packet, only changing its Type to 0 (Echo reply).

The second program (tracert) is somewhat more interesting. Although there exists a Type value explicitly designed to support reporting of the routing path to a remote device, it was added quite later than the rest (defined in RFC 1393) by which time implementing the option required that a large number of networked devices be updated. As such, the tracert program (and its various counterparts/variants, such as traceroute, tracepath, etc.) take a different route (pun intended): the program sends a series of triples of packets with increasing values of TTL, starting from 1 and going up to a user-defined value (or a default of, usually, 30). As we know, on every hop the forwarding router will decrease the packet’s TTL by 1 and if said value reaches 0 the packet is discarded; in the latter case, the router will send an ICMP Time Exceeded (Type: 11) message back to the sending host. Consequently, the sender will receive a series of ICMP messages, each coming from a router one step further along the path to the final destination. Although the above mechanism is the same across the various implementations of traceroute, the actual content of the message sent differs; e.g., the default implementation on Unix-like operating systems uses UDP packets (sent to invalid port numbers – more on this in an upcoming lecture), Windows implementations use ICMP Echo Request messages, and most newer implementations have options to choose the protocol of the payload of outgoing packets (ICMP Echo Request, UDP, TCP SYN).

As a final note, although ICMP is indeed an integral part of IP, it is sometimes the case that network administrators choose to block ICMP packets (either specific types or all of them). This practice, often discredited as security-through-obscurity, can result in the above tools “misbehaving”; that is, they may indicate that a remote host is unreachable or that the path ends abruptly, when that is most definitely not the case.

[3] https://en.wikipedia.org/wiki/ICMP_tunnel
[4] https://www.exploit-db.com/docs/english/18581-covert-channel-over-icmp.pdf

Often a combination of these and other tools/tricks can help us troubleshoot such cases, but this is outside the scope of this lab.

Lab Tasks

In this lab, you are asked to perform the following tasks:

1. Open a command prompt window. If you are on a:
    a. **Windows**: To do this, you can press ⊞+ 'r' to bring up the "run command" windows, then type "cmd" (without the double quotes) and hit enter. Alternatively, you can search for the "Command Prompt" option through your Start Menu (usually located under Accessories).
    b. **Mac:** click the Launchpad icon in the Dock, type **Terminal** in the search field, then click **Terminal**. Or in the Finder, open the /Applications/Utilities folder, then double-click **Terminal**.
    c. **GNU/Linux** (in most distributions e.g., Ubuntu): you can find it using Ctrl + alt + t , or simply by clicking the terminal GUI on your desktop (if you are on GNU/Linux you should probably know this already!)
2. Ping and traceroute:
    a. **Windows:** Execute "ping /?" and "tracert /?". Read the help messages and familiarise yourselves with the various options and capabilities of the two programs.
    b. **Mac/GNU-Linux**: Execute "man ping" and "man traceroute". Read the help messages and familiarise yourselves with the various options and capabilities of the two programs.

3. Use "ping *some.host*" to test connectivity of your computer to a remote device. Substitute *some.host* with at least one external hostname of your choosing (e.g., www.bbc.co.uk or www.google.co.uk) and the IP addresses of a computer of some other student in your group. How do the two outputs compare? Is there anything unexpected in the output? Discuss.
4. Timed ping:
    a. **Windows:** "ping -t *some.host*" to send several Echo requests to at least two remote devices of your choosing. Hit Ctrl+C to get the relevant statistics when you consider enough responses have been received. Discuss any interesting observations on the output.
    b. **Mac/GNU-Linux:** "ping -t 10 *some.host*" to send several Echo requests to at least two remote devices of your choosing. Hit Ctrl+C to get the relevant statistics when you consider enough responses have been received. Discuss any interesting observations on the output.

5. Use "tracert" (on Windows) or "traceroute"(on Mac/GNU-Linux) to record the path to www.bbc.co.uk and www.gla.ac.uk. Compare the two outputs and discuss any interesting observations.

✓ In your free time, try tasks 3, 4 and 5 with devices connected to *eduroam* or to your home network. How does the output change compared to what you got in the lab? Why do you think this is so?

✓ In your free time, have a look at the below for yet more fun with traceroute: https://gist.github.com/ismaild/7659981

What to Submit

Nothing... ☺ You should discuss your questions and answers with your classmates and tutor at the lab; this will allow you to get direct feedback and may allow you to explore various options on the spot. Answers to the lab tasks will be posted on Moodle at the end of the week. You should then review them for any differences with your answers, and discuss any discrepancies with your classmates/tutor in the next lab.