

Lab-Report

Report No:03

Course code: ICT-3207

Course title: Computer Networks

Date of Performance:24/01/2021

Date of Submission:

Submitted by

Name: Afrin Zaman & Mahfuza
Talukdar

ID:IT-18008 & IT-18009

3rd year 2nd semester

Session: 2017-2018

Dept. of ICT

Submitted To

Nazrul Islam

Assistant Professor

Dept. of ICT

MBSTU.

Theory :

Third-party libraries: Although the Python's standard library provides a great set of awesome functionalities, there will be times that you will eventually run into the need of making use of third party libraries. Can you imagine building a webserver from scratch? Or making a port to a database driver? Or, maybe, coming up with an image manipulation tool?. Third party libraries are welcome in a way that they prevent you from reinventing the something that exist. They save you time to focus on finishing and delivering your application.

Networking Glossary: Before we begin discussing networking with any depth, we must define some common terms that you will see throughout this guide, and in other guides and documentation regarding networking.

- **Connection:** In networking, a connection refers to pieces of related information that are transferred through a network. This generally infers that a connection is built before the data transfer (by following the procedures laid out in a protocol) and then is deconstructed at the end of the data transfer.
- **Packet:** A packet is, generally speaking, the most basic unit that is transferred over a network. When communicating over a network, packets are the envelopes that carry your data (in pieces) from one end point to the other. Packets have a header portion that contains information about the packet including the source and destination, timestamps, network hops, etc. The main portion of a packet contains the actual data being transferred. It is sometimes called the body or the payload.
- **Network Interface:** A network interface can refer to any kind of software interface to networking hardware. For instance, if you have two network cards in your computer, you can control and configure each network interface associated with them individually. A network interface may be associated with a physical device, or it may be a representation of a virtual interface. The "loopback" device, which is a virtual interface to the local machine, is an example of this.
- **LAN:** LAN stands for "local area network". It refers to a network or a portion of a network that is not publicly accessible to the greater internet. A home or office network is an example of a LAN.
- **WAN:** WAN stands for "wide area network". It means a network that is much more extensive than a LAN. While WAN is the relevant term to use to describe large, dispersed networks in general, it is usually meant to mean the internet, as a whole. If an interface is said to be connected to the WAN, it is generally assumed that it is reachable through the internet.
- **Protocol:** A protocol is a set of rules and standards that basically define a language that devices can use to communicate. There are a great number of protocols in use

extensively in networking, and they are often implemented in different layers. Some low level protocols are TCP, UDP, IP, and ICMP. Some familiar examples of application layer protocols, built on these lower protocols, are HTTP (for accessing web content), SSH, TLS/SSL, and FTP. Port: A port is an address on a single machine that can be tied to a specific piece of software. It is not a physical interface or location, but it allows your server to be able to communicate using more than one application.

- **Firewall:** A firewall is a program that decides whether traffic coming into a server or going out should be allowed. A firewall usually works by creating rules for which type of traffic is acceptable on which ports. Generally, firewalls block ports that are not used by a specific application on a server.
- **NAT:** NAT stands for network address translation. It is a way to translate requests that are incoming into a routing server to the relevant devices or servers that it knows about in the LAN. This is usually implemented in physical LANs as a way to route requests through one IP address to the necessary backend servers.
- **VPN:** VPN stands for virtual private network. It is a means of connecting separate LANs through the internet, while maintaining privacy. This is used as a means of connecting remote systems as if they were on a local network, often for security reasons.
- **Interfaces:** Interfaces are networking communication points for your computer. Each interface is associated with a physical or virtual networking device. Typically, your server will have one configurable network interface for each Ethernet or wireless internet card you have. In addition, it will define a virtual network interface called the "loopback" or localhost interface. This is used as an interface to connect applications and processes on a single computer to other applications and processes. You can see this referenced as the "lo" interface in many tools. Many times, administrators configure one interface to service traffic to the internet and another interface for a LAN or private network.
- **Protocols:** Networking works by piggybacking a number of different protocols on top of each other. In this way, one piece of data can be transmitted using multiple protocols encapsulated within one another. We will talk about some of the more common protocols that you may come across and attempt to explain the difference, as well as give context as to what part of the process they are involved with. We will start with protocols implemented on the lower networking layers and work our way up to protocols with higher abstraction.

Exercises:

4.1. Enumerating interfaces on your machine

Code:

```

import sys

import socket

import fcntl

import struct

import array

SIOCGIFCONF = 0x8912 #from C library sockios.h

STUCT_SIZE_32 = 32

STUCT_SIZE_64 = 40

PLATFORM_32_MAX_NUMBER = 2**32

DEFAULT_INTERFACES = 8

def list_interfaces():

    interfaces = []

    max_interfaces = DEFAULT_INTERFACES

    is_64bits = sys.maxsize > PLATFORM_32_MAX_NUMBER

    struct_size = STUCT_SIZE_64 if is_64bits else STUCT_SIZE_32

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    while True:

        bytes = max_interfaces * struct_size

        interface_names = array.array('B', '\0' * bytes)

        sock_info = fcntl.ioctl(

            sock.fileno(),

            SIOCGIFCONF,

            struct.pack('iL', bytes, interface_names.buffer_info()[0])

        )

```

```

outbytes = struct.unpack('iL', sock_info)[0]

if outbytes == bytes:
    max_interfaces *= 2
else:
    break

namestr = interface_names.tostring()

for i in range(0, outbytes, struct_size):
    interfaces.append((namestr[i:i+16].split('\0', 1)[0]))

return interfaces

if __name__ == '__main__':
    interfaces = list_interfaces()

    print( "This machine has %s network interfaces: %s."
          %(len(interfaces), interface))

```

Output:

```
This machine has 2 network interfaces: ['lo', 'eth0'].
```

Exercise 4.2: Finding the IP address for a specific interface on your machine

Code:

```

import argparse

import sys

import socket

import fcntl

import struct

import array

def get_ip_address(ifname):

```

```

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

return socket.inet_ntoa(fcntl.ioctl(

s.fileno(),

0x8915, # SIOCGIFADDR

struct.pack('256s', ifname[:15])

)[20:24])

if __name__ == '__main__':

#interfaces = list_interfaces()

parser = argparse.ArgumentParser(description='Python networking utils')

parser.add_argument('--ifname', action="store", dest="ifname",

required=True)

given_args = parser.parse_args()

ifname = given_args.ifname

print ("Interface [%s] --> IP: %s" %(ifname, get_ip_

address(ifname)))

```

Output:

Interface [eth0] --> IP: 10.0.2.15

Exercise 4.3: Finding whether an interface is up on your machine

Code:

```

import argparse

import socket

import struct

import fcntl

import nmap

```

```

SAMPLE_PORTS = '21-23'

def get_interface_status(ifname):

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    ip_address = socket.inet_ntoa(fcntl.ioctl(
        sock.fileno(),
        0x8915, #SIOCGIFADDR, C socket library sockios.h
        struct.pack('256s', ifname[:15]))[20:24])

    nm = nmap.PortScanner() nm.scan(ip_address, SAMPLE_PORTS)

    return nm[ip_address].state()

if __name__ == '__main__':

    parser = argparse.ArgumentParser(description='Python networking
    utils')

    parser.add_argument('--ifname', action="store", dest="ifname",
        required=True)

    given_args = parser.parse_args()

    ifname = given_args.ifname

    print ("Interface [%s] is: %s" %(ifname, get_interface_
    status(ifname)))

```

OUTPUT:

Interface [eth0] is: up

Exercise 4.4: Detecting inactive machines on your network

Code:

```

import argparse

import time

```

```

import sched

from scapy.all import sr, srp, IP, UDP, ICMP, TCP, ARP, Ether

RUN_FREQUENCY = 10

scheduler = sched.scheduler(time.time, time.sleep)

def detect_inactive_hosts(scan_hosts):
    """
    Scans the network to find scan_hosts are live or dead
    scan_hosts can be like 10.0.2.2-4 to cover range.
    See Scapy docs for specifying targets.
    """

    global scheduler

    scheduler.enter(RUN_FREQUENCY, 1, detect_inactive_hosts, (scan_
        hosts, ))

    inactive_hosts = []

    try:

        ans, unans = sr(IP(dst=scan_hosts)/ICMP(), retry=0, timeout=1)

        ans.summary(lambda(s,r) : r.sprintf("%IP.src% is alive"))

        for inactive in unans:

            print "%s is inactive" %inactive.dst

            inactive_hosts.append(inactive.dst)

        print "Total %d hosts are inactive" %(len(inactive_hosts))

    except KeyboardInterrupt:

        exit(0)

if __name__ == "__main__":

```



```
parser = argparse.ArgumentParser(description='Python networking utils')

parser.add_argument('--scan-hosts', action="store", dest="scan_
hosts", required=True)

given_args = parser.parse_args()

scan_hosts = given_args.scan_hosts

scheduler.enter(1, 1, detect_inactive_hosts, (scan_hosts, ))

scheduler.run()
```

OUTPUT:

```
$ sudo python 3_7_detect_inactive_machines.py --scan-hosts=10.0.2.2-4
Begin emission:
.*...Finished to send 3 packets.
.
Received 6 packets, got 1 answers, remaining 2 packets
10.0.2.2 is alive
10.0.2.4 is inactive
10.0.2.3 is inactive
Total 2 hosts are inactive
Begin emission:
*.Finished to send 3 packets.
Received 3 packets, got 1 answers, remaining 2 packets
10.0.2.2 is alive
10.0.2.4 is inactive
10.0.2.3 is inactive
Total 2 hosts are inactive
```

Exercise 4.5: Pinging hosts on the network with ICMP

Code:

```
import os

import argparse

import socket

import struct

import select
```

```

import time

ICMP_ECHO_REQUEST = 8 # Platform specific

DEFAULT_TIMEOUT = 2

DEFAULT_COUNT = 4

class Pinger(object):

    """ Pings to a host -- the Pythonic way """

    def __init__(self, target_host, count=DEFAULT_COUNT,
        timeout=DEFAULT_TIMEOUT):

        self.target_host = target_host

        self.count = count

        self.timeout = timeout

    def do_checksum(self, source_string):

        """ Verify the packet integrity """

        sum = 0

        max_count = (len(source_string)/2)*2

        count = 0

        while count < max_count:

            val = ord(source_string[count + 1])*256 + ord(source_
                string[count])

            sum = sum + val

            sum = sum & 0xffffffff

            count = count + 2

        if max_count < len(source_string):

            sum = sum + ord(source_string[len(source_string)-1])

```

```

sum = sum & 0xffffffff

sum = (sum>>16) + (sum & 0xffffffff)

sum = sum + (sum >> 16)

answer = ~sum

answer = answer & 0xffff

answer = answer >> 8 | (answer << 8 & 0xff00)

return answer

def receive_pong(self, sock, ID, timeout):
    """
    Receive ping from the socket.
    """
    time_remaining = timeout

    while True:
        start_time = time.time()

        readable = select.select([sock], [], [], time_remaining)

        time_spent = (time.time() - start_time)

        if readable[0] == []: # Timeout
            return

        time_received = time.time()

        recv_packet, addr = sock.recvfrom(1024)

        icmp_header = recv_packet[20:28]

        type, code, checksum, packet_ID, sequence = struct.unpack(
            "bbHHh", icmp_header
        )

```

```

if packet_ID == ID:
    bytes_In_double = struct.calcsize("d")
    time_sent = struct.unpack("d", recv_packet[28:28 +
        bytes_In_double])[0]
    return time_received - time_sent
    time_remaining = time_remaining - time_spent
    if time_remaining <= 0:
        return

```

We need a `send_ping()` method that will send the data of a ping request to the target host.

Also, this will call the `do_checksum()` method for checking the integrity of the ping data, as follows:

```

def send_ping(self, sock, ID):
    """
    Send ping to the target host
    """
    target_addr = socket.gethostbyname(self.target_host)
    my_checksum = 0
    # Create a dummy header with a 0 checksum.
    header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, my_
checksum, ID, 1)
    bytes_In_double = struct.calcsize("d")
    data = (192 - bytes_In_double) * "Q"
    data = struct.pack("d", time.time()) + data
    # Get the checksum on the data and the dummy header.

```

```

my_checksum = self.do_checksum(header + data)

header = struct.pack
( "bbHHh", ICMP_ECHO_REQUEST, 0, socket.htons(my_checksum),
ID, 1 )

packet = header + data

sock.sendto(packet, (target_addr, 1))

def ping_once(self):

icmp = socket.getprotobyname("icmp")

try:

sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
icmp)

except socket.error, (errno, msg):

if errno == 1:

# Not superuser, so operation not permitted

msg += "ICMP messages can only be sent from root user
processes"

raise socket.error(msg)

except Exception, e:

print "Exception: %s" %(e)

my_ID = os.getpid() & 0xFFFF

self.send_ping(sock, my_ID)

delay = self.receive_pong(sock, my_ID, self.timeout)

sock.close()

return delay

```

```

def ping(self):
    """
    Run the ping process
    """

    for i in xrange(self.count):
        print "Ping to %s..." % self.target_host,
        try:
            delay = self.ping_once()
        except socket.gaierror, e:
            print "Ping failed. (socket error: '%s')" % e[1]
            break

        if delay == None:
            print "Ping failed. (timeout within %ssec.)" % \
                self.timeout

        else:
            delay = delay * 1000

        print "Get pong in %0.4fms" % delay

    if __name__ == '__main__':
        parser = argparse.ArgumentParser(description='Python ping')
        parser.add_argument('--target-host', action="store", dest="target_
            host", required=True)

        given_args = parser.parse_args()

        target_host = given_args.target_host

        pinger = Pinger(target_host=target_host)

```

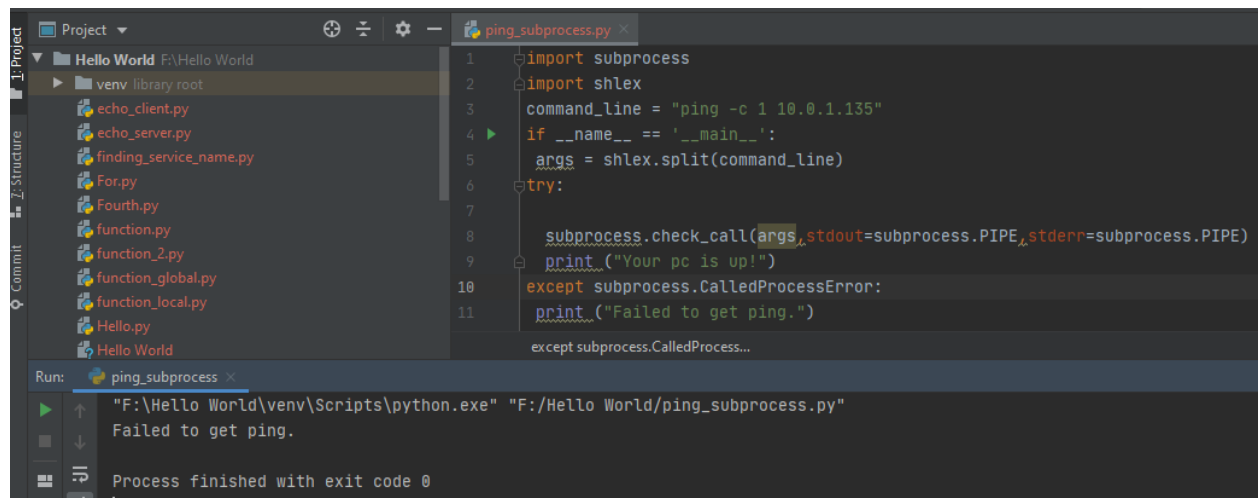
pinger.ping()

OUTPUT:

```
$ sudo python 3_2_ping_remote_host.py --target-host=www.google.com
Ping to www.google.com... Get pong in 7.6921ms
Ping to www.google.com... Get pong in 7.1061ms
Ping to www.google.com... Get pong in 8.9211ms
Ping to www.google.com... Get pong in 7.9899ms
```

Exercise 4.6: Pinging hosts on the network with ICMP using pc resources

Code:



```
1 import subprocess
2 import shlex
3 command_line = "ping -c 1 10.0.1.135"
4 if __name__ == '__main__':
5     args = shlex.split(command_line)
6     try:
7
8         subprocess.check_call(args, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
9         print("Your pc is up!")
10    except subprocess.CalledProcessError:
11        print("Failed to get ping.")
12    except subprocess.CalledProcessError:
```

Run: ping_subprocess x

"F:\Hello World\venv\Scripts\python.exe" "F:\Hello World\ping_subprocess.py"

Failed to get ping.

Process finished with exit code 0

Exercise 4.7: Scanning the broadcast of packets

Code:

```
from scapy.all import *
import os
captured_data = dict()
END_PORT = 1000
def monitor_packet(pkt):
    if IP in pkt:
        if not captured_data.has_key(pkt[IP].src):
            captured_data[pkt[IP].src] = []
            if TCP in pkt:
                if pkt[TCP].sport <= END_PORT:
                    if not str(pkt[TCP].sport) in captured_data[pkt[IP].src]:
                        captured_data[pkt[IP].src].append(str(pkt[TCP].sport))
            os.system('clear')
    ip_list = sorted(captured_data.keys())
    for key in ip_list:
```

```

ports=', '.join(captured_data[key])
if len (captured_data[key]) == 0:
print '%s' % key
else:
print '%s (%s)' % (key, ports)
if __name__ == '__main__':
sniff(prn=monitor_packet, store=0)

```

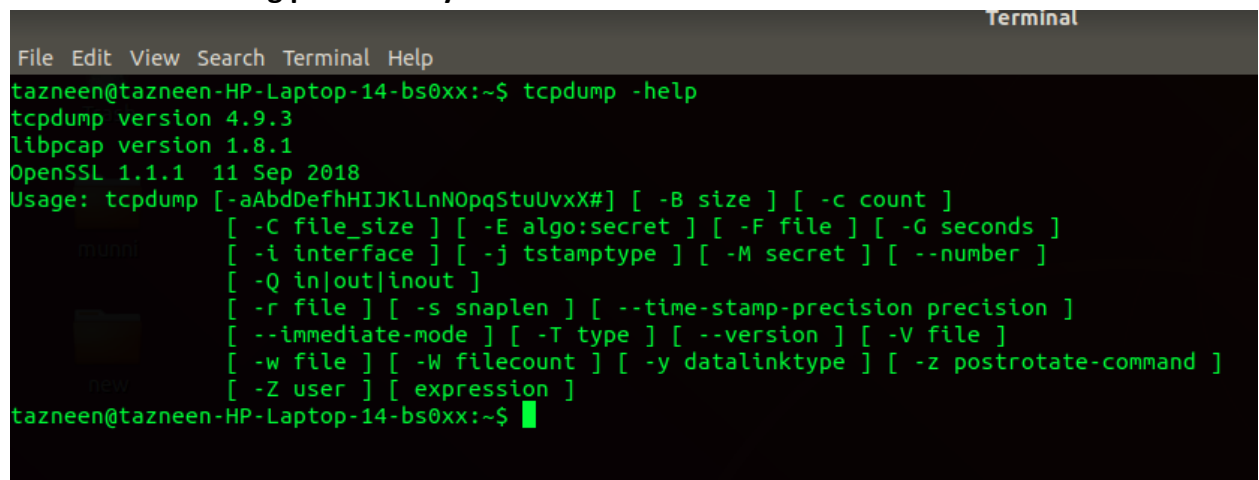
Output:

```

10.0.2.15
XXX.194.41.129 (80)
XXX.194.41.134 (80)
XXX.194.41.136 (443)
XXX.194.41.140 (80)
XXX.194.67.147 (80)
XXX.194.67.94 (443)
XXX.194.67.95 (80, 443)

```

Exercise 4.8: Sniffing packets on your network



```

Terminal
File Edit View Search Terminal Help
tazneen@tazneen-HP-Laptop-14-bs0xx:~$ tcpdump -help
tcpdump version 4.9.3
libpcap version 1.8.1
OpenSSL 1.1.1 11 Sep 2018
Usage: tcpdump [-aAbdDefhHIJKlLnNOpqStuUvxxX#] [-B size] [-c count]
               [-C file_size] [-E algo:secret] [-F file] [-G seconds]
               [-i interface] [-j tstamptype] [-M secret] [--number]
               [-Q in|out|inout]
               [-r file] [-s snaplen] [--time-stamp-precision precision]
               [--immediate-mode] [-T type] [--version] [-V file]
               [-w file] [-W filecount] [-y datalinktype] [-z postrotate-command]
               [-Z user] [expression]
tazneen@tazneen-HP-Laptop-14-bs0xx:~$

```

Discussion:

Compared to many languages, Python is easy to learn and to use. Its functions can be carried out with simpler commands and less text than most competing languages. And this might explain why it's soaring in popularity, with developers, coding students and tech companies.

It's not an exaggeration to say that Python plays a small part of all of our lives. It's one of those invisible forces with a presence in our mobile devices, web searches and gaming (and beyond). So it was an obvious choice for inclusion in our full stack coding bootcamp. Here's an introduction to the language itself, and some of the everyday but profound, things that Python is used for.

Python has a socket method that let programmers' set-up different types of socket virtually. After you defined the socket, you can use several methods to manage the connections. Some of the important server socket methods are:

- `listen()`: is used to establish and start TCP listener.
- `bind()`: is used to bind-address (host-name, port number) to the socket.
- `accept()`: is used to TCP client connection until the connection arrives.
- `connect()`: is used to initiate TCP server connection.
- `send()`: is used to send TCP messages.
- `recv()`: is used to receive TCP messages.
- `sendto()`: is used to send UDP messages
- `close()`: is used to close a socket.