



Mawlana Bhashani Science And Technology University

Lab-Report

Lab Report No: 03

Lab Report Name: Python for Networking

Course Title : Network Planning and Designing Lab

Course Code : ICT-3208

Group member ID: IT-18013 and IT-18028

Date of Performance: 04-02-2021

Date of Submission: 04-02-2021

Submitted by

Name: Anjom Nour Anika

ID: IT-18013.

Name: Fatema_Tuz_Jannat

ID: It-18028

3rd Year 2nd Semester

Submitted To

Nazrul Islam

Assistant Professor

Dept. of ICT

MBSTU.

Lab 3: Python for Networking

1. Objectives

The objective of the lab 3 is to:

1. Install python and use third-party libraries
2. Interact with network interfaces using python
3. Getting information from internet using Python

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 exit. They save you time to focus on finishing and delivering your application.

The following is a list of those third-party libraries with their download URLs used in

networking:

- ntplib: <https://pypi.python.org/pypi/ntplib/>
- diesel: <https://pypi.python.org/pypi/diesel/>
- nmap: <https://pypi.python.org/pypi/python-nmap>
- scapy: <https://pypi.python.org/pypi/scapy>
- netifaces: <https://pypi.python.org/pypi/netifaces/>
- netaddr: <https://pypi.python.org/pypi/netaddr>
- pyopenssl: <https://pypi.python.org/pypi/pyOpenSSL>
- pygeocoder: <https://pypi.python.org/pypi/pygocoder>
- pyyaml: <https://pypi.python.org/pypi/PyYAML>
- requests: <https://pypi.python.org/pypi/requests>
- feedparser: <https://pypi.python.org/pypi/feedparser>
- paramiko: <https://pypi.python.org/pypi/paramiko/>
- fabric: <https://pypi.python.org/pypi/Fabric>
- supervisor: <https://pypi.python.org/pypi/supervisor>
- xmlrpclib: <https://pypi.python.org/pypi/xmlrpclib>
- SOAPpy: <https://pypi.python.org/pypi/SOAPpy>
- bottlenose: <https://pypi.python.org/pypi/bottlenose>
- construct: <https://pypi.python.org/pypi/construct/>
- pyserial: <https://pypi.python.org/pypi/pyserial>

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.

1. 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.

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.

Methodology

Installing Python Third-party includes:

Python Third-party includes a setup.py file, it is usually distributed as a tarball (.tar.gz or .tar.bz2

file). The instructions for installing these generally look like:

- Download the file from website.
- Extract the tarball.
- Change into the new directory that has been newly extracted.
- Run `sudo python setup.py build`
- Run `sudo python setup.py install`

Exercises

When importing a module if there is an error it means that the module needs to be installed.

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:

```
$ 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 4.2: 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:

```

$ 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

```

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'

```

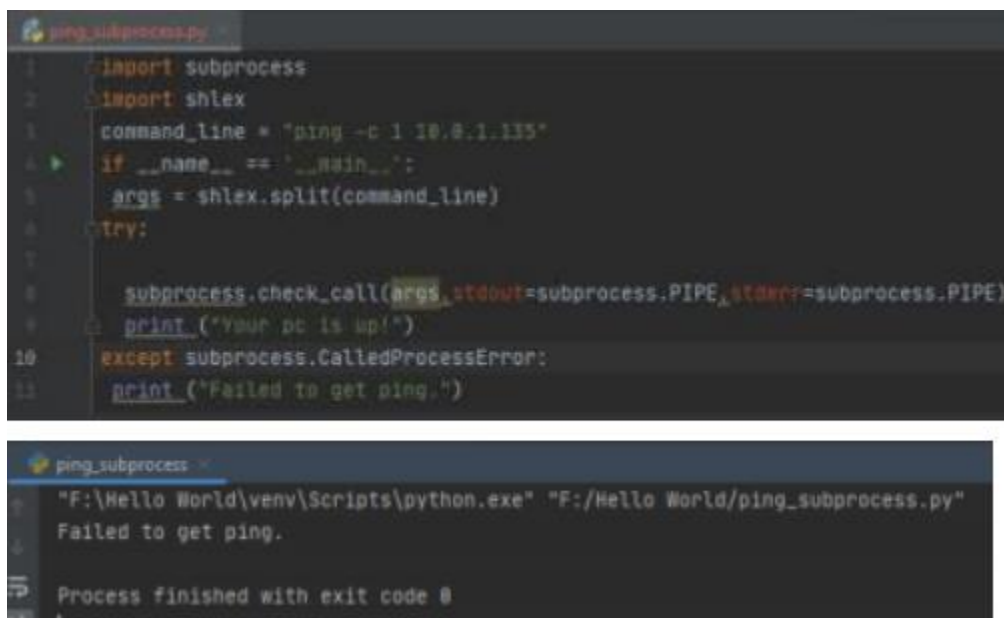
```

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
    Parser(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:



The top screenshot shows a code editor with the following Python 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.")

```

The bottom screenshot shows the terminal output of the script:

```

ping_subprocess
"F:\Hello World\venv\Scripts\python.exe" "F:/Hello World/ping_subprocess.py"
Failed to get ping.

Process finished with exit code 0

```

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:

```

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 4.5: 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 > 16: sum = (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
    Parser(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:

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

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:

```

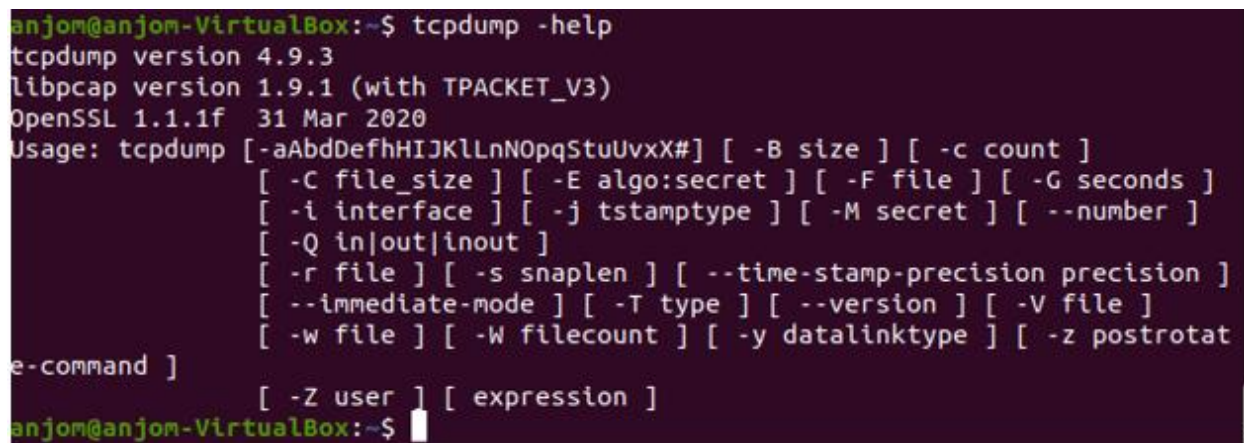
```

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:

Exercise 4.8: Sniffing packets on your network



```

anjom@anjom-VirtualBox:~$ tcpdump -help
tcpdump version 4.9.3
libpcap version 1.9.1 (with TPACKET_V3)
OpenSSL 1.1.1f 31 Mar 2020
Usage: tcpdump [-aAbdDefhHIIJKLlLnNOpqStuUvxx#] [-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 postrotat
e-command]
               [-Z user] [expression]
anjom@anjom-VirtualBox:~$

```

Conclusion:

There are two levels of network service access in Python. These are:

1. Low-Level Access

2. High-Level Access

In the first case, programmers can use and access the basic socket support for the operating system using Python's libraries, and programmers can implement both connection-less and connection-oriented protocols for programming. Application level network protocols can also be accessed using high-level access provided by

Python libraries. These protocols are HTTP, FTP, etc. A socket is the end-point in a flow of communication between two programs or communication channels operating over a network. They are created using a set of programming requests called socket API (Application Programming Interface). Python's socket library offers classes for handling common transports as a generic interface.

Sockets use protocols for determining the connection type for port-to-port communication between client and server machines. The protocols are used for:

1. Domain Name Servers (DNS)

2. IP addressing

3. E-mail

4. FTP (File Transfer Protocol) etc... 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:

1. `listen()`: is used to establish and start TCP listener.

2. `bind()`: is used to bind-address (host-name, port number) to the socket.

3. `accept()`: is used to TCP client connection until the connection arrives.

4. `connect()`: is used to initiate TCP server connection.

5. `send()`: is used to send TCP messages.

6. `recv()`: is used to receive TCP messages.

7. `sendto()`: is used to send UDP messages

8. `close()`: is used to close a socket. Sending messages back and forth using different basic protocols is simple and straightforward. It shows that programming takes a significant role in client-server architecture where the client makes data request to a server, and the server replies to those machines.