JSS MAHAVIDYAPEETA

JSS SCIENCE AND TECHNOLOGY UNIVERSITY

SRI JAYACHAMARAJENDRA COLLEGE OF ENGINEERING

MYSURU-570006

COMPUTER NETWORKS (IS620) PROJECT

Title: Ping Implementation Using Python

Submitted by:

Amarasimha G Hebbar *01JST18IS401*

Nagabharana K Jamadagni *01JST18IS405*

Srikanta Vaibhav N *01JST17IS063*

6th Semester

Dept. of Information Science & Engineering,

JSSSTU, Mysuru.

25th May, 2020

Table of Contents

1. Problem Statement

Ping is a popular networking application used to test from a remote location whether a particular host is up and reachable. It is also often used to measure latency between the client host and the target host. It works by sending ICMP echo request packets (i.e., ping packets) to the target host and listening for ICMP echo response replies (i.e., pong packets). Ping measures the RRT, records packet loss, and calculates a statistical summary of multiple ping-pong exchanges (the mini- mum, mean, max, and standard deviation of the round-trip times). Write your own Ping application in Python. This program will use ICMP. Follow the official specification in RFC 1739.

\- 9th Problem Statement,
IS620 Computer Networks - Laboratory Assignment, 16th May 2020.

2. Introduction:

The Internet is a world-wide network of computer networks. Millions of people rely on it to exchange information. Different Internet connections can have different performance reflecting how fast and reliable information can be exchanged between two end points. End-to-end performance measurement plays an important role in the development of the Internet. A popular two-way measurement tool is the ping utility:

Ping (Packet Internet Groper) is a computer-network software utility designed to test the reachability of a host on an Internet Protocol (IP) network. It is available in all operating systems that have networking capability, and in most embedded network administration software. The name comes from active sonar terminology that sends a pulse of sound and listens for the echo to discover objects under water.

Ping operates by sending **Internet Control Message Protocol (ICMP)** echo request packets to the target host and waiting for an ICMP echo reply. It measures the round-trip time for messages sent from the originating host to a destination computer that are echoed back to the source.

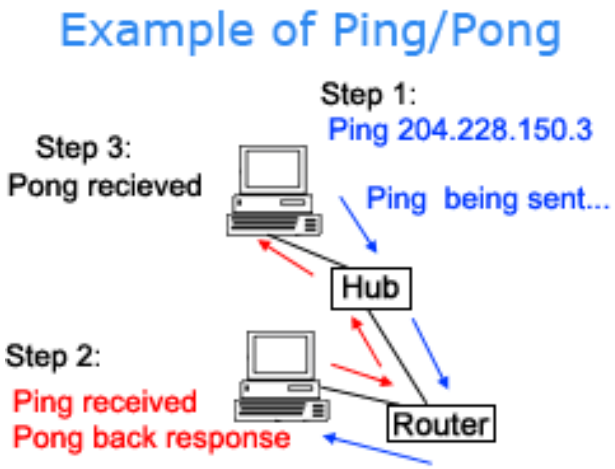

Illustration 1: Ping-Pong example

The program reports errors, packet loss, and a statistical summary of the results, usually including the minimum, maximum, the mean round-trip times, and standard deviation of the mean. The command-line options of the ping utility may include the size of the payload, count of tests, limits for the amount of network hops (TTL) that probes traverse, interval between the requests and time to wait for a response.

In this assignment, we implement Ping networking application in **Python v3.6.9** using **ICMPv6**, **IPv6** and following the specifications in **RFC 1739**.

3. Literature survey:

3.1 "The Ping Utility" originally by Mike Muuss, remarked by David Mills in 4.3BSD, at Ballistic Research Laboratory, sc. 4.3 under BSD licence, December 1983.

In this paper, the ping network utility was implemented in C using the Internet Control Message Protocol (ICMP) "ECHO" facility, it measures the round-trip-delays and packet loss across network paths. Authored at the U. S. Army Ballistic Research Laboratory and Modified at Uc Berkeley.

Later it was proposed that using ICMP echo packets for IP network diagnosis and measurements, prompted to create the utility to troubleshoot network problems and changed few arguments (eg. Changing arguments to inet_ntoa() to be struct in_addr instead of u_long).

The status of this paper/version was Public Domain and Distribution Unlimited. Bugs that were later reported were- more statistics could always have been gathered and this program has to run SUID to ROOT to access the ICMP socket.

3.2 "A pure py ICMP Ping using raw sockets", by Jens Diemer in PyPi projects repository, sc. 1d8e600, under GNU General Public License, October 2011.

This is an implementation of ICMP ping using raw sockets and an improvement to the previous versions. A simple command line interface (CLI) was added which was preceded by an improved functionality which includes:

- Constant time between pings, and internal times that consistently use milliseconds.
- Clarified annotations (e.g., in the checksum routine).
- Using unsigned data in IP & ICMP header pack/unpack unless otherwise necessary.
- Signal handling.
- Ping-style output formatting and stats.

The initial py ICMP ping work which is dated back to December 1997, included the checksum bytes in the wrong order when this was run under Solaris 2.X for SPARC but it worked right under Linux x86 for some reason.

Since it was not known what was wrong, the bytes were swapped always and then an htons() i.e., converting the unsigned short integer hostshort from host byte order to network byte order was done.

3.3 "Short term behaviour of Ping Measurements" by Xing Deng in his M.S. Thesis at University of Waikato, July 1999.

In this study, ICMP (Internet Control Message Protocol) packets were sent at high speed for a short period by the ping utility to various Internet addresses and their round-trip times (RTTs) were measured by ping and the tcp dump utility. The study has two purposes: to evaluate the accuracy of ping and tcp dump in terms of measuring packet wire round-trip times (WRTTs); and to identify short term high density ping traffic patterns and interpret them.

A careful comparison was done of different software and hardware configurations for measuring the WRTTs, as well as the repeatability of measurements on similar configurations. The results show that ping's own RTT measurements can contain significant source processing delay and so distort the analysis of traffic patterns; running tcpdump to capture the packet wire times increases the accuracy of WRTT measurement; running tcpdump on separate measurement machines in single-user mode with output to the RAM disks can generate consistent WRTT measurements.

Findings in the pattern analysis include: short term high density ping traffic patterns can sometimes be interpreted by buffering and queueing models; the size of measurement packets can significantly influence traffic behaviour; due to source host factors ping request packets maybe sent out of sequence.

4. Protocols and Standards followed:

We implement Ping networking application in Python v3.6.9 using **IP v6**, **ICMP v6,** and following the specifications in **RFC 1739**.

4.1 Internet Protocol version 6 (IPv6):

Internet Protocol version 6 (**IP** v6) is the most recent version of the Internet Protocol (IP), the communications protocol that provides an identification and location system for computers on networks and routes traffic across the Internet.

It is an Internet Layer protocol for packet-switched internetworking and provides end-to-end datagram transmission across multiple IP networks, closely adhering to the design principles developed in the previous version of the protocol, Internet Protocol Version 4 (IPv4).

IPv6 packet:

	Bits 0–3	**Bits 4–7**	**Bits 8–11**	**Bits 12–15**	**Bits 16–23**	**Bits 24–31**
Header (40 bytes)	Version	Traffic Class		Flow Label		
	Payload Length				Next Header	Hop Limit
	Source Address					
	Destination Address					

Illustration 2: IPv6 packet header

An IPv6 packet has two parts: a header and payload of user data.

- The header consists of control information for addressing and routing required for all packets. It occupies the first 40 octets (320 bits).
- The payload is typically a datagram or segment of the higher-level transport layer protocol, but may be data for an internet layer (e.g., ICMPv6). It must be less than 64kB without special options.

IPv6 is defined in RFC 2460 and RFC 8200, whereas the addressing architecture of IPv6 is defined in RFC 4291 and allows three different types of transmission: unicast, anycast and multicast.

IPv6 addresses are represented as eight groups, separated by colons, of four hexadecimal digits. The full representation may be simplified by several methods of notation; for example, *2001:0db8:0000:0000:0000:8a2e:0370:7334* becomes *2001:db8::8a2e:370:7334*.

4.2 Internet Control Message Protocol version 6 (ICMP v6):

Internet Control Message Protocol version 6 (**ICMP** v6) is the implementation of the Internet Control Message Protocol (ICMP) for Internet Protocol version 6 (IPv6). It is defined in RFC 4443. ICMPv6 is an integral part of IPv6 and performs error reporting and diagnostic functions (e.g., ping), and has a framework for extensions to implement future changes.

4.2.1 Message format of ICMP v6 packet (including IP v6 datagram):

<table>
<tr><th></th><th>Bits 0–3</th><th>Bits 4–7</th><th>Bits 8–11</th><th>Bits 12–15</th><th>Bits 16–23</th><th>Bits 24–31</th></tr>
<tr><td rowspan="4">Header
(40 bytes)</td><td>Version</td><td colspan="2">Traffic Class</td><td colspan="3">Flow Label</td></tr>
<tr><td colspan="4">Payload Length</td><td>Next Header</td><td>Hop Limit</td></tr>
<tr><td colspan="6">Source Address</td></tr>
<tr><td colspan="6">Destination Address</td></tr>
<tr><td rowspan="2">ICMP6
Header
(8 bytes)</td><td colspan="2">Type of message</td><td>Code</td><td colspan="3">Checksum</td></tr>
<tr><td colspan="6">Header Data</td></tr>
<tr><td>ICMP6
Payload
(optional)</td><td colspan="6">Payload Data</td></tr>
</table>

Illustration 3: ICMPv6 packet

Generic composition of an ICMP packet:

- IPv6 Header (in blue): *Next Header* set to 58 (ICMP6)

- ICMP Header (in red):
  - Type of ICMP message (8 bits)
  - Code (8 bits)
  - Checksum (16 bits), the 16-bit one's complement of the one's complement sum of the packet. For IPv4, this is calculated from the ICMP message starting with the Type field (the IP header is not included). For IPv6 this is calculated from the ICMP message, prepended with an IPv6 "pseudo-header".
  - Header Data (32 bits) field, which in this case (ICMP echo request and replies), will be composed of identifier (16 bits) and sequence number (16 bits).

- ICMP Payload: *payload* for the different kind of answers; can be an arbitrary length, left to implementation detail. However, the packet including IP and ICMP headers must be less than the maximum transmission unit of the network or risk being fragmented.

Basically the ICMP packet is encapsulated in IP protocol. Every ping program goes like this:

Ping (Program on the application layer) ⟶ Opens a 'raw' socket to IP Layer ⟶ IP layer packages ICMP packet and sends it.

4.2.2 Message format of Echo request:

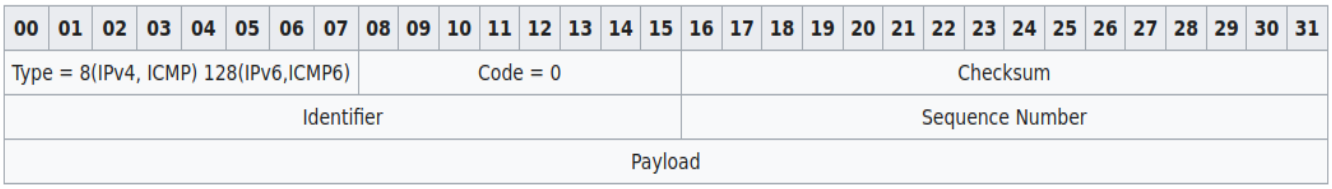

Illustration 4: Message format of ICMP Echo request

The *echo request* ("ping") is an ICMP/ICMP6 message.

The Identifier and Sequence Number can be used by the client to match the reply with the request that caused the reply. In practice, most Linux systems use a unique identifier for every ping process, and sequence number is an increasing number within that process. Windows uses a fixed identifier, which varies between Windows versions, and a sequence number that is only reset at boot time.

4.2.3 Message format of Echo reply:

The *echo reply* is an ICMP message generated in response to an echo request; it is mandatory for all hosts, and must include the exact payload received in the request.

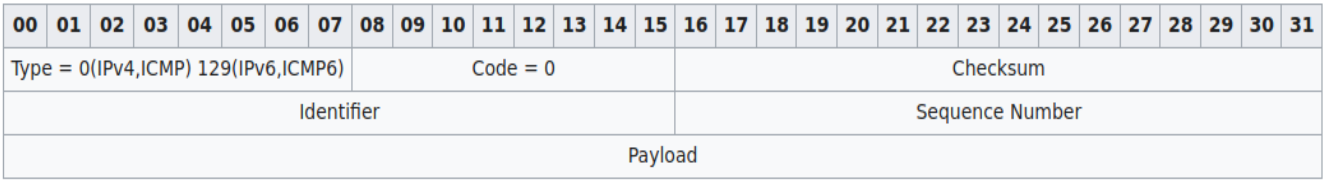

Illustration 5: Message format of ICMP Echo response

The *identifier* and *sequence number* can be used by the client to associate each echo request with its reply.

4.3 RFC 1739 - A Primer On Internet and TCP/IP Tools:

It is an introductory guide to some of the TCP/IP and Internet tools and utilities that allow users to access the wide variety of information on the network, from determining if a particular host is up to viewing a multimedia thesis on foreign policy. It also describes discussion lists accessible from the Internet, ways to obtain Internet documents, and resources that help users weave their way through the Internet.

Some of the TCP/IP utilities and applications mentioned in RFC 1739 are:

1. NSLOOKUP
2. PING
3. FINGER
4. TRACEROUTE
5. FTP
6. TELNET

The ping utility mentioned here is:

PING [-s] {IP_address | host_name} [size] [quantity]

The "-s" parameter tells the system to send an ICMP Echo message every second. The optional "size" parameter specifies that each message should be in *size* bytes in length (64 bytes is the default size); the optional "quantity" parameter indicates that this test will only send *quantity* no. of messages (the default is to run the test continuously until interrupted). Simple use of the command contains no optional parameters.

We implement this ping utility in our python program with the following parameters:
s = 1000ms or 1s,
size = 56 bytes, and
quantity = <until Ctrl+C is pressed> i.e., until a keyboard interrupt is received.

5. Algorithm:

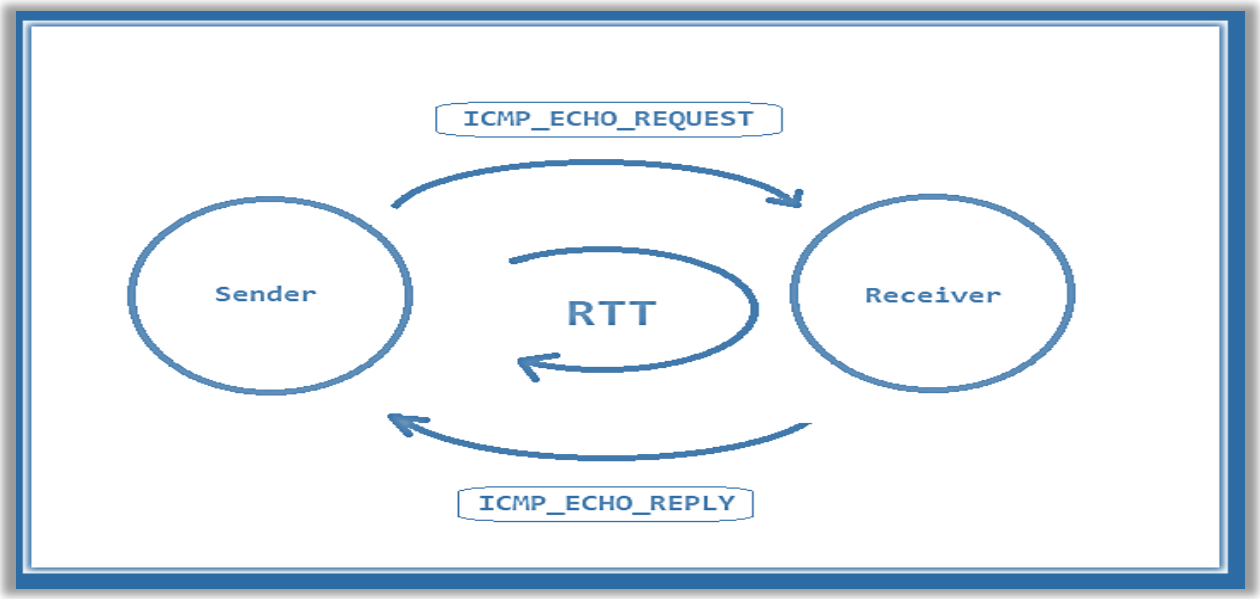

Illustration 6: Working of ICMP Ping-Pong

Algorithm PythonPing (host)

#To test the reachability of a host on an Internet Protocol (IP) network.
#Input: Host's name or IP address
#Output: Displays every ICMP echo response received and displays a brief summary of round-trip times and packet loss statistics.

Steps	Description
1.	**Input hostname from the user**
2.	**Perform a DNS lookup** DNS lookup is performed using **gethostbyname(hostname)**.The **gethostbyname( )** function converts a normal human readable website passed as parameter and returns the IPv4 address of the host in the form of decimal digits, formatted as four 8-bit fields separated by periods.
3.	Open a **Raw socket** using **SOCK_RAW** with protocol as **IPPROTO_ICMP**. Note: raw socket requires super user privileges, hence to run this code sudo is used.
4.	**Sending ping request:** **4.1** Set the **TTL option** to a value in the socket TTL value is set to limit the number of hops a packet can make. **4.2** Set the **timeout** of the recv function If timeout is not set, recv will wait forever, halting the loop.

4.3 Fill up the **icmp packet** as follows:

i. Set packet header type to ICMP_ECHO.

ii. Set id to pid of process

iii. Fill msg data part randomly.

iv. Calculate checksum and fill it in checksum field.

4.4 **Send** the packet.

5. **Receiving pong response:**

5.1 Wait for it to be **received**.

Note: If a packet is received it does not mean that the destination is working. An Echo reply means destination is OK. It is sent from the destination's OS kernel.

5.2 If **received successfully**, close the raw socket and display the packet size, the IP address from where it was received, ICMP sequence number, TTL, and the delay i.e., round-trip time.

5.3 If **timed out**, then display "Request Timed Out" and close the raw socket.

6. When Crtl + C is pressed, ping gives a report (Step 7). This interrupt is caught by an interrupt handler which just sets our pinging looping condition to false.

7. **Show statistics when pings are done, as follows:**

7.1 Display the number of **packets transmitted** and the number of **packets received.**

7.2 Calculate and display the **packet loss** in percentage, total **time** involved in sending the first packet and receiving the last packet including the waiting/lag time between each ping.

7.3 Calculate and display the **minimum, maximum, average** and **standard** deviation of round-trip times of all the ping-pong exchanges made.

6. Implementation:

Requirements necessary to run this program:

OS	: Windows10/Linux
Network Access	: Requires Internet connectivity (min 10-12 Kbps)
Privileges	: Requires Superuser/Administrator privilege
Python	: Version 3.6 and above

Python3 Code:

```
#!/usr/bin/env python

import os, sys, socket, struct, select, time, signal, numpy

if sys.platform == "win32":
    # On Windows, the best timer is time.clock()
    default_timer = time.clock
else:
    # On most other platforms the best timer is time.time()
    default_timer = time.time

#--------------------------------------------------------------------------#

# ICMP parameters
ICMP_ECHOREPLY  =    0 # Echo reply (per RFC4443 and RFC1739)
ICMP_ECHO       =    8 # Echo request (per RFC4443 and RFC1739)
ICMP_MAX_RECV   = 2048 # Max size of incoming buffer

MAX_SLEEP = 1000

class MyStats:
    thisIP   = "0.0.0.0"
    pktsSent = 0
    pktsRcvd = 0
    minTime  = 999999999
    maxTime  = 0
    totTime  = 0 #sum of each ping delays
    fracLoss = 1.0
    delayList = []
    hostName = ''
    startTime, endTime = 0,0

myStats = MyStats # Used globally

#--------------------------------------------------------------------------#
```

```
def checksum(source_string):

    countTo = (int(len(source_string)/2))*2
    sum = 0
    count = 0

    # Handle bytes by decoding as short ints
    loByte = 0
    hiByte = 0
    while count < countTo:
        if (sys.byteorder == "little"):
            loByte = source_string[count]
            hiByte = source_string[count + 1]
        else:
            loByte = source_string[count + 1]
            hiByte = source_string[count]
        sum = sum + (hiByte * 256 + loByte)
        count += 2

    # Handle last byte if applicable (odd-number of bytes)
    if countTo < len(source_string): # Check for odd length
        loByte = source_string[len(source_string)-1]
        sum += loByte

    sum &= 0xffffffff # Truncate sum to 32 bits
    sum = (sum >> 16) + (sum & 0xffff)    # Add high 16 bits to low 16 bits
    sum += (sum >> 16)                    # Add carry from above (if any)
    answer = ~sum & 0xffff              # Invert and truncate to 16 bits
    answer = socket.htons(answer)

    return answer

#-----------------------------------------------------------------------------#



def do_one(destIP, timeout, mySeqNumber, numDataBytes):
    """
    Returns either the delay (in ms) or None on timeout.
    """
    global myStats

    delay = None #delay means roundTrip time

    try:
        mySocket = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.getprotobyname("icmp"))
    except socket.error as e:
        print("failed. (socket error: '%s')" % e.args[1])
        raise # raise the original error

    my_ID = os.getpid() & 0xFFFF

    sentTime = send_one_ping(mySocket, destIP, my_ID, mySeqNumber, numDataBytes)
    if sentTime == None:
        mySocket.close()
        return delay

    myStats.pktsSent += 1;

    recvTime, dataSize, iphSrcIP, icmpSeqNumber, iphTTL = receive_one_ping(mySocket, my_ID, timeout)

    mySocket.close()

    if recvTime:
        delay = (recvTime-sentTime)*1000; myStats.delayList.append(delay)
        print("%d bytes from %s: icmp_seq=%d ttl=%d time=%0.3f ms" % (dataSize, socket.inet_ntoa(struct.pack("!I", iphSrcIP)), icmpSeqNumber,
iphTTL, delay))
        myStats.pktsRcvd += 1;
        myStats.totTime += delay
        if myStats.minTime > delay:
            myStats.minTime = delay
```

```
        if myStats.maxTime < delay:
            myStats.maxTime = delay
    else:
        delay = None
        print("Request timed out.")

    return delay

#-----------------------------------------------------------------------#
def send_one_ping(mySocket, destIP, myID, mySeqNumber, numDataBytes):
    """
    Send one ping to the given >destIP<.
    """
    destIP  =  socket.gethostbyname(destIP)

    # Header is type (8), code (8), checksum (16), id (16), sequence (16)
    myChecksum = 0

    # Make a dummy heder with a 0 checksum.
    header = struct.pack(
        "!BBHHH", ICMP_ECHO, 0, myChecksum, myID, mySeqNumber
    )

    padBytes = []
    startVal = 0x42
    for i in range(startVal, startVal + (numDataBytes)):
        padBytes += [(i & 0xff)]  # Keep chars in the 0-255 range
    data = bytes(padBytes)

    # Calculate the checksum on the data and the dummy header.
    myChecksum = checksum(header + data) # Checksum is in network order
    header = struct.pack("!BBHHH", ICMP_ECHO, 0, myChecksum, myID, mySeqNumber)
    packet = header + data
    sendTime = time.time()
    try:
        mySocket.sendto(packet, (destIP, 1)) # Port number is irrelevant for ICMP
    except socket.error as e:
        print("General failure (%s)" % (e.args[1]))
        return

    return sendTime

#-----------------------------------------------------------------------#
```

```
def receive_one_ping(mySocket, myID, timeout):
    """
    Receive the ping from the socket. Timeout = in ms
    """
    timeLeft = timeout/1000

    while True: # Loop while waiting for packet or timeout
        startedSelect = time.time()
        whatReady = select.select([mySocket], [], [], timeLeft)
        howLongInSelect = (time.time() - startedSelect)
        if whatReady[0] == []: # Timeout
            return None, 0, 0, 0, 0

        timeReceived = time.time()

        recPacket, addr = mySocket.recvfrom(ICMP_MAX_RECV)

        ipHeader = recPacket[:20]
        iphVersion, iphTypeOfSvc, iphLength, \
        iphID, iphFlags, iphTTL, iphProtocol, \
        iphChecksum, iphSrcIP, iphDestIP = struct.unpack(
            "!BBHHHBBHII", ipHeader
        )

        icmpHeader = recPacket[20:28]
        icmpType, icmpCode, icmpChecksum, \
        icmpPacketID, icmpSeqNumber = struct.unpack(
            "!BBHHH", icmpHeader
        )

        if icmpPacketID == myID: # Our packet
            dataSize = len(recPacket) - 28
            return timeReceived, dataSize, iphSrcIP, icmpSeqNumber, iphTTL

        timeLeft = timeLeft - howLongInSelect
        if timeLeft <= 0:
            return None, 0, 0, 0, 0
#---------------------------------------------------------------------------#
def dump_stats():
    """
    Show stats when pings are done
    """
    global myStats

    print("\n---- %s PYTHON PING Statistics ----" % (myStats.hostName))

    if myStats.pktsSent > 0:
        myStats.fracLoss = (myStats.pktsSent - myStats.pktsRcvd)/myStats.pktsSent

    print("%d packets transmitted, %d packets received, %0.1f%% packet loss, time %dms" % (myStats.pktsSent, myStats.pktsRcvd, 100.0 *
myStats.fracLoss, (myStats.endTime-myStats.startTime)+(myStats.pktsSent-1)*1000)) #(myStats.pktsSent-1)*1000 is the total of waiting/sleep/lag
time btw each ping.

    if myStats.pktsRcvd > 0:
        print("Round-Trip Times:\nMinimum = %0.3f ms\tMaximum = %0.3f ms\nAverage = %0.3f ms\tStandard Deviation = %0.3f ms" %
(myStats.minTime, myStats.maxTime, myStats.totTime/myStats.pktsRcvd, numpy.std(myStats.delayList)))
    return

#---------------------------------------------------------------------------#
```

```
def ping(hostname, timeout = 1000, numDataBytes = 56):
    """
    Ping to >destIP< with the given >timeout< and display
    the result.
    """
    global myStats
    myStats = MyStats() # Reset the stats
    mySeqNumber = 1 # Starting value

    try:
        destIP = socket.gethostbyname(hostname) # DNS lookup
        print("\nPYTHON PING %s (%s) with %d data bytes:" % (hostname, destIP, numDataBytes))
    except socket.gaierror as e:
        print("\nPYTHON PING: Unknown host: %s (%s)" % (hostname, e.args[1]))
        print()
        return

    myStats.hostName = hostname
    myStats.thisIP = destIP

    try:
        myStats.startTime = time.time()
        while True:
            delay = do_one(destIP, timeout, mySeqNumber, numDataBytes)

            if delay == None:
                delay = 0

            mySeqNumber += 1

            if (MAX_SLEEP > delay):
                time.sleep((MAX_SLEEP - delay)/1000)
    except KeyboardInterrupt:
        myStats.endTime = time.time()
        dump_stats()

#-----------------------------------------------------------------------------#

if __name__ == '__main__':

    hostname = input("Enter name/IP address of the host: ")
    ping(hostname)

#-----------------------------------------------------------------------------#
```

7. Result:

1. Pinging a known and valid host: *www.jssstuniv.com*

```
vaibhav@ubuntu-18-04-2:~/Desktop/CNproject2$ sudo python3 ping.py
Enter name/IP address of the host: www.jssstuniv.in

PYTHON PING www.jssstuniv.in (69.162.124.28) with 56 data bytes:
56 bytes from 69.162.124.28: icmp_seq=1 ttl=49 time=651.815 ms
56 bytes from 69.162.124.28: icmp_seq=2 ttl=49 time=401.384 ms
56 bytes from 69.162.124.28: icmp_seq=3 ttl=49 time=360.891 ms
56 bytes from 69.162.124.28: icmp_seq=4 ttl=49 time=320.321 ms
56 bytes from 69.162.124.28: icmp_seq=5 ttl=49 time=318.036 ms
56 bytes from 69.162.124.28: icmp_seq=6 ttl=49 time=316.233 ms
^C
---- www.jssstuniv.in PYTHON PING Statistics ----
6 packets transmitted, 6 packets received, 0.0% packet loss, time 5005ms
Round-Trip Times:
Minimum = 316.233 ms    Maximum = 651.815 ms
Average = 394.780 ms    Standard Deviation = 118.928 ms
```

Since the host is up and running, we obtain successful ping statistics result with 0% packet loss.

2. Pinging an unknown host that does not exist: *www.ThisWebsiteDoesNotExist.com*

```
vaibhav@ubuntu-18-04-2:~/Desktop/CNproject2$ sudo python3 ping.py
Enter name/IP address of the host: www.ThisWebsiteDoesNotExist.com

PYTHON PING: Unknown host: www.ThisWebsiteDoesNotExist.com (Name or service not
known)
```

In this case, we obtain an “Unknown host” message as a result of failure in DNS lookup.

3. Pinging a host which is down and not running:

```
vaibhav@ubuntu-18-04-2:~/Desktop/CNproject2$ sudo python3 ping.py
Enter name/IP address of the host: 1.2.3.4

PYTHON PING 1.2.3.4 (1.2.3.4) with 56 data bytes:
Request timed out.
Request timed out.
Request timed out.
Request timed out.
Request timed out.
^C
---- 1.2.3.4 PYTHON PING Statistics ----
5 packets transmitted, 0 packets received, 100.0% packet loss, time 4009ms
```

Here, we obtain “Request timed out” message as a result of no response from the host. Thus, there is 100% packet loss.

In all the above cases, we terminate the ping process by a keyboard interrupt Ctrl+C.

8. Conclusion:

In this assignment, Ping utility was successfully implemented in Python v3.6.9 to test from a remote location whether a particular host is up and reachable. Ping is often used to measure latency between the client host and the target host. It works by sending ICMP echo request packets (i.e., ping packets) to the target host and listening for ICMP echo response replies (i.e., pong packets).

Our ping program was able to send and receive ICMP packets and measure the RRT, record packet loss, and calculate a statistical summary of multiple ping-pong exchanges (the minimum, mean, maximum, and standard deviation of the round-trip times). There were no errors reported.

We employed IP v6 and ICMP v6 protocols and followed the specifications in RFC 1739. IPv6 is defined in RFC 2460 and RFC 8200, and ICMPv6 is defined in RFC 4443.

9. References:

"The Story of the PING Program". U.S. Army Research Laboratory. Archived from the original on 25 October 2019.

"RFC 4443 - Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification".

"RFC 1739 - A Primer On Internet and TCP/IP Tools - IETF Tools".

"IPv6 – Wikipedia".

Appendix: Abbreviations Used

IPv6: Internet **P**rotocol **v**ersion **6**

ICMPv6: Internet **C**ontrol **M**essage **P**rotocol **v**ersion **6**

PING: Packet **I**nter**N**etwork **G**roper

TCP: Transmission **C**ontrol **P**rotocol

RFC: Request **f**or **C**omments

RRT: Round-**T**rip **T**ime

TTL: Time **t**o **L**ive