Socket Programming Lab #3: ICMP Pinger

Full program for *client.py* provided below:

```python
from socket import *
import os
import sys
import struct
import time
import select
import binascii

ICMP_ECHO_REQUEST = 8


def checksum(string):
    csum = 0
    countTo = (len(string) // 2) * 2
    count = 0

    while count < countTo:
        thisVal = (string[count + 1]) * 256 + (string[count])
        csum += thisVal
        csum &= 0xffffffff
        count += 2

    if countTo < len(string):
        csum += (string[len(string) - 1])
        csum &= 0xffffffff

    csum = (csum >> 16) + (csum & 0xffff)
    csum = csum + (csum >> 16)
    answer = ~csum
    answer = answer & 0xffff
    answer = answer >> 8 | (answer << 8 & 0xff00)
    return answer


def receiveOnePing(mySocket, ID, timeout, destAddr):
    timeLeft = timeout

    while 1:
        startedSelect = time.time()
        whatReady = select.select([mySocket], [], [], timeLeft)
        howLongInSelect = (time.time() - startedSelect)
        if whatReady[0] == []:  # Timeout
            return (None, None)

        timeReceived = time.time()
        recPacket, addr = mySocket.recvfrom(1024)

        #Fill in start
        # Fetch the ICMP header from the IP packet

        # ICMP header is 8 bytes long and starts after bit 160 of the IP header (starts at byte 20)
        type, code, checksum, id, sequence = struct.unpack("bbHHh", recPacket[20:28])

        if type == 0 and id == ID:
            data = struct.unpack("d", recPacket[28:])[0]   # data in ICMP reply is time that ICMP request was sent
            timeDelay = timeReceived - data

            return (timeDelay, (type, code, checksum, id, sequence, data))
```

```
59
60          #Fill in end
61
62          timeLeft = timeLeft - howLongInSelect
63          if timeLeft <= 0:
64              return (None, None)
65
66
67  def sendOnePing(mySocket, destAddr, ID):
68  # Header is type (8), code (8), checksum (16), id (16), sequence (16)
69
70      myChecksum = 0
71      # Make a dummy header with a 0 checksum
72      # struct -- Interpret strings as packed binary data
73      header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
74      data = struct.pack("d", time.time())
75
76      # Calculate the checksum on the data and the dummy header.
77      myChecksum = checksum(header + data)
78
79      # Get the right checksum, and put in the header
80      if sys.platform == 'darwin':
81          # Convert 16-bit integers from host to network byte order
82          myChecksum = htons(myChecksum) & 0xffff
83      else:
84          myChecksum = htons(myChecksum)
85
86      header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
87      packet = header + data
88      mySocket.sendto(packet, (destAddr, 1))  # AF_INET address must be tuple, not str
89
90      # Both LISTS and TUPLES consist of a number of objects
```

```
91      # which can be referenced by their position number within the object.
92
93
94  def doOnePing(destAddr, timeout):
95      icmp = getprotobyname("icmp")
96
97      # SOCK_RAW is a powerful socket type. For more details: http://sockraw.org/papers/sock_raw
98      mySocket = socket(AF_INET, SOCK_RAW, icmp)
99
100     myID = os.getpid() & 0xFFFF  # Return the current process i
101     sendOnePing(mySocket, destAddr, myID)
102     result = receiveOnePing(mySocket, myID, timeout, destAddr)
103     mySocket.close()
104     return result
105
106
107 def ping(host, timeout=1):
108 # timeout=1 means: If one second goes by without a reply from the server,
109 # the client assumes that either the client's ping or the server's pong is lost
110
111     dest = gethostbyname(host)
112     resps = []
113     print("Pinging " + dest + " using Python:")
114     print("")
115
116     # Calculate vars values and return them
117     # Send ping requests to a server separated by approximately one second
118     for i in range(0, 5):
119         result = doOnePing(dest, timeout)
120         resps.append(result)
121         time.sleep(1)  # one second
122
123     return resps
124
```

```
125
126 if __name__ == '__main__':
127     ping("google.co.il")
```

Lines 48-60 are the added code to the skeleton code. Line 52 unpacks the ICMP header from the received IP packet. Bytes 20-28 of the IP packet are unpacked as a 5-tuple of data types (integer, integer, integer, integer, double) where the relative 5-tuple corresponds to the ICMP packet's type, code, checksum, ID, and sequence number.

```
48         #Fill in start
49         # Fetch the ICMP header from the IP packet
50
51         # ICMP header is 8 bytes long and starts after bit 160 of the IP header (starts at byte 20)
52         type, code, checksum, id, sequence = struct.unpack("bbHHh", recPacket[20:28])
```

Line 54 validates the ICMP reply, checking if it is of type 0 and if the ID number matches up. If so, then line 55 unpacks the data field of the ICMP packet. We can expect the data field to contain the timestamp the ICMP echo request packet was sent. Using the timestamp from the data field, line 56 calculates the round-trip time the ICMP echo request packet was sent to the time the corresponding ICMP echo reply packet was received. Line 58 returns a tuple containing the round-trip time as the first entry and a 6-tuple containing the ICMP echo reply packet type, code, checksum, ID number, sequence number, and data field.

```
54         if type == 0 and id == ID:
55             data = struct.unpack("d", recPacket[28:])[0]   # data in ICMP reply is time that ICMP request was sent
56             timeDelay = timeReceived - data
57
58             return (timeDelay, (type, code, checksum, id, sequence, data))
59
60         #Fill in end
```

Optional Exercise #1

Full program for *ping_statistics.py* below:

```python
from socket import *
import os
import sys
import struct
import time
import select
import binascii

ICMP_ECHO_REQUEST = 8


def checksum(string):
    csum = 0
    countTo = (len(string) // 2) * 2
    count = 0

    while count < countTo:
        thisVal = (string[count + 1]) * 256 + (string[count])
        csum += thisVal
        csum &= 0xffffffff
        count += 2

    if countTo < len(string):
        csum += (string[len(string) - 1])
        csum &= 0xffffffff

    csum = (csum >> 16) + (csum & 0xffff)
    csum = csum + (csum >> 16)
    answer = ~csum
    answer = answer & 0xffff
    answer = answer >> 8 | (answer << 8 & 0xff00)
    return answer


def receiveOnePing(mySocket, ID, timeout, destAddr):
    timeLeft = timeout

    while 1:
        startedSelect = time.time()
        whatReady = select.select([mySocket], [], [], timeLeft)
        howLongInSelect = (time.time() - startedSelect)
        if whatReady[0] == []:  # Timeout
            return (None, None)

        timeReceived = time.time()
        recPacket, addr = mySocket.recvfrom(1024)

        #Fill in start
        # Fetch the ICMP header from the IP packet

        # ICMP header is 8 bytes long and starts after bit 160 of the IP header (starts at byte 20)
        type, code, checksum, id, sequence = struct.unpack("bbHHh", recPacket[20:28])

        if type == 0 and id == ID:
            data = struct.unpack("d", recPacket[28:])[0]   # data in ICMP reply is time that ICMP request was sent
            timeDelay = timeReceived - data

            return (timeDelay, (type, code, checksum, id, sequence, data))
```

```
59
60          #Fill in end
61
62          timeLeft = timeLeft - howLongInSelect
63          if timeLeft <= 0:
64              return (None, None)
65
66
67  def sendOnePing(mySocket, destAddr, ID):
68  # Header is type (8), code (8), checksum (16), id (16), sequence (16)
69
70      myChecksum = 0
71      # Make a dummy header with a 0 checksum
72      # struct -- Interpret strings as packed binary data
73      header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
74      data = struct.pack("d", time.time())
75
76      # Calculate the checksum on the data and the dummy header.
77      myChecksum = checksum(header + data)
78
79      # Get the right checksum, and put in the header
80      if sys.platform == 'darwin':
81          # Convert 16-bit integers from host to network byte order
82          myChecksum = htons(myChecksum) & 0xffff
83      else:
84          myChecksum = htons(myChecksum)
85
86      header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
87      packet = header + data
88      mySocket.sendto(packet, (destAddr, 1))  # AF_INET address must be tuple, not str
89
90      # Both LISTS and TUPLES consist of a number of objects
```

```
91      # which can be referenced by their position number within the object.
92
93
94  def doOnePing(destAddr, timeout):
95      icmp = getprotobyname("icmp")
96
97      # SOCK_RAW is a powerful socket type. For more details: http://sockraw.org/papers/sock_raw
98      mySocket = socket(AF_INET, SOCK_RAW, icmp)
99
100     myID = os.getpid() & 0xFFFF  # Return the current process i
101     sendOnePing(mySocket, destAddr, myID)
102     result = receiveOnePing(mySocket, myID, timeout, destAddr)
103     mySocket.close()
104     return result
105
106
107 def ping(host, timeout=1):
108 # timeout=1 means: If one second goes by without a reply from the server,
109 # the client assumes that either the client's ping or the server's pong is lost
110
111     dest = gethostbyname(host)
112     resps = []
113     print("Pinging " + dest + " using Python:")
114     print("")
115
116     # Ping statistics
117     maxRTT = averageRTT = packetLoss = packetReceived = 0
118     minRTT = sys.maxsize
119
120     # Calculate vars values and return them
121     # Send ping requests to a server separated by approximately one second
122     for i in range(0, 5):
123         result = doOnePing(dest, timeout)
124         resps.append(result)
125
```

```
126         if result[0] != None:
127             minRTT = min(minRTT, result[0])
128             maxRTT = max(maxRTT, result[0])
129             averageRTT += result[0]
130             packetReceived += 1
131         else:
132             packetLoss += 1
133 
134         time.sleep(1)  # one second
135 
136     packetLoss = (packetLoss / len(resps)) * 100
137 
138     if packetLoss == 100:
139         minRTT = 0
140     else:
141         averageRTT = (averageRTT / packetReceived) * 1000
142         maxRTT *= 1000
143         minRTT *= 1000
144 
145     print("{} packets transmitted, {} packets received, {:.1f}% packet loss".format(len(resps), packetReceived, packetLoss))
146     print("rtt min/avg/max = {:.3f}/{:.3f}/{:.3f} ms".format(minRTT, averageRTT, maxRTT))
147 
148     return resps
149 
150 
151 if __name__ == '__main__':
152     ping("google.co.il")
```

ping_statistics.py is a copy of *client.py* in all except for lines 116-146. In lines 117-118, variables for the maximum RTT, minimum RTT, average RTT, amount of packets lost, and amounts of packets received are initialized.

```
116     # Ping statistics
117     maxRTT = averageRTT = packetLoss = packetReceived = 0
118     minRTT = sys.maxsize
119 
```

Lines 122-134 contain a for-loop that sends and retrieves the results of 5 ICMP pings. Lines 126-132 update the variables for maximum RTT, minimum RTT, average RTT, amount of packets lost, and amounts of packets received with each iteration of the for-loop. The ping statistics variables maximum RTT, minimum RTT, average RTT, and amounts of packets received will only update if the results of the server pong are not lost. The variable for amounts of packets lost will update if the results of the server pong are lost.

```
120     # Calculate vars values and return them
121     # Send ping requests to a server separated by approximately one second
122     for i in range(0, 5):
123         result = doOnePing(dest, timeout)
124         resps.append(result)
125 
126         if result[0] != None:
127             minRTT = min(minRTT, result[0])
128             maxRTT = max(maxRTT, result[0])
129             averageRTT += result[0]
130             packetReceived += 1
131         else:
132             packetLoss += 1
133 
134         time.sleep(1)  # one second
```

Line 136 calculates the packet loss percentage. In lines 138-138, if the packet loss percentage results in 100% then the variable for minimum RTT is set to 0. Otherwise, average RTT, minimum RTT, and maximum RTT are calculated to be milliseconds. Lines 145-146 print the ping statistics to the user interface.

```
136     packetLoss = (packetLoss / len(resps)) * 100
137 
138     if packetLoss == 100:
139         minRTT = 0
140     else:
141         averageRTT = (averageRTT / packetReceived) * 1000
142         maxRTT *= 1000
143         minRTT *= 1000
144 
145     print("{} packets transmitted, {} packets received, {:.1f}% packet loss".format(len(resps), packetReceived, packetLoss))
146     print("rtt min/avg/max = {:.3f}/{:.3f}/{:.3f} ms".format(minRTT, averageRTT, maxRTT))
```

Optional Exercise #2

Full program for *decoding_errors.py* below:

```python
from socket import *
import os
import sys
import struct
import time
import select
import binascii

ICMP_ECHO_REQUEST = 8


def checksum(string):
    csum = 0
    countTo = (len(string) // 2) * 2
    count = 0

    while count < countTo:
        thisVal = (string[count + 1]) * 256 + (string[count])
        csum += thisVal
        csum &= 0xffffffff
        count += 2

    if countTo < len(string):
        csum += (string[len(string) - 1])
        csum &= 0xffffffff

    csum = (csum >> 16) + (csum & 0xffff)
    csum = csum + (csum >> 16)
    answer = ~csum
    answer = answer & 0xffff
    answer = answer >> 8 | (answer << 8 & 0xff00)
    return answer


def printErrorMessage(type, code):
    errorMessages = {
        3: {
            0: "Net is unreachable",
            1: "Host is unreachable",
            2: "Protocol is unreachable",
            3: "Port is unreachable",
            4: "Fragmentation is needed and \"Don\'t Fragment\" was set",
            5: "Source route failed",
            6: "Destination network is unknown",
            7: "Destination host is unknown",
            8: "Source host is isolated",
            9: "Communication with destination network is administratively prohibited",
            10: "Communication with destination host is administratively prohibited",
            11: "Destination network is unreachable for type of service",
            12: "Destination host is unreachable for type of service",
            13: "Communication is administratively prohibited",
            14: "Host precedence violation",
            15: "Precedence cutoff is in effect"
```

```
        },
        5: {
            0: "Redirect datagram for the network (or subnet)",
            1: "Redirect datagram for the host",
            2: "Redirect datagram for the type of service and network",
            3: "Redirect datagram for the type of service and host"
        },
        11: {
            0: "\"Time to Live\" exceeded in transit",
            1: "Fragment reassembly time exceeded"
        },
        12: {
            0: "Pointer indicates the error",
            1: "Missing a required option",
            2: "Bad length"
        }
    }

    print("Error:", errorMessages[type][code])


def receiveOnePing(mySocket, ID, timeout, destAddr):
    timeLeft = timeout

    while 1:
        startedSelect = time.time()
        whatReady = select.select([mySocket], [], [], timeLeft)
        howLongInSelect = (time.time() - startedSelect)
        if whatReady[0] == []:  # Timeout
            return (None, None)

        timeReceived = time.time()
        recPacket, addr = mySocket.recvfrom(1024)

        #Fill in start
        # Fetch the ICMP header from the IP packet

        # ICMP header is 8 bytes long and starts after bit 160 of the IP header (starts at byte 20)
```

```
        type, code, checksum, id, sequence = struct.unpack("bbHHh", recPacket[20:28])

        if type == 0 and id == ID:
            data = struct.unpack("d", recPacket[28:])[0]   # data in ICMP reply is time that ICMP request was sent
            timeDelay = timeReceived - data

            return (timeDelay, (type, code, checksum, id, sequence, data))

        elif type != 0:
            printErrorMessage(type, code)
            return (None, None)

        #Fill in end

        timeLeft = timeLeft - howLongInSelect
        if timeLeft <= 0:
            return (None, None)


def sendOnePing(mySocket, destAddr, ID):
# Header is type (8), code (8), checksum (16), id (16), sequence (16)

    myChecksum = 0
    # Make a dummy header with a 0 checksum
    # struct -- Interpret strings as packed binary data
    header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
```

```
118     data = struct.pack("d", time.time())
119
120     # Calculate the checksum on the data and the dummy header.
121     myChecksum = checksum(header + data)
122
123     # Get the right checksum, and put in the header
124     if sys.platform == 'darwin':
125         # Convert 16-bit integers from host to network byte order
126         myChecksum = htons(myChecksum) & 0xffff
127     else:
128         myChecksum = htons(myChecksum)
129
130     header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
131     packet = header + data
132     mySocket.sendto(packet, (destAddr, 1))  # AF_INET address must be tuple, not str
133
134     # Both LISTS and TUPLES consist of a number of objects
135     # which can be referenced by their position number within the object.
136
137
```

```
138 def doOnePing(destAddr, timeout):
139     icmp = getprotobyname("icmp")
140
141     # SOCK_RAW is a powerful socket type. For more details: http://sockraw.org/papers/sock_raw
142     mySocket = socket(AF_INET, SOCK_RAW, icmp)
143
144     myID = os.getpid() & 0xFFFF  # Return the current process i
145     sendOnePing(mySocket, destAddr, myID)
146     result = receiveOnePing(mySocket, myID, timeout, destAddr)
147     mySocket.close()
148     return result
149
150
151 def ping(host, timeout=1):
152 # timeout=1 means: If one second goes by without a reply from the server,
153 # the client assumes that either the client's ping or the server's pong is lost
154
155     dest = gethostbyname(host)
156     resps = []
157     print("Pinging " + dest + " using Python:")
158     print("")
159
160     # Calculate vars values and return them
161     # Send ping requests to a server separated by approximately one second
162     for i in range(0, 5):
163         result = doOnePing(dest, timeout)
164         resps.append(result)
165         time.sleep(1)  # one second
166
167     return resps
168
169
170 if __name__ == '__main__':
171     ping("127.0.0.1")
```

decoding_errors.py is like that of *client.py* except that there is a new *printErrorMessage* on lines 35-72 that will print an error message to the user interface in the event that an ICMP error reply packet is received.

In the *printErrorMessage* function, lines 36-70 declare the variable *errorMessages* that contains a nested dictionary of all possible error messages for all ICMP error type-code pairs. Line 72 prints the error message to the user interface.

```
35  def printErrorMessage(type, code):
36      errorMessages = {
37          3: {
38              0: "Net is unreachable",
39              1: "Host is unreachable",
40              2: "Protocol is unreachable",
41              3: "Port is unreachable",
42              4: "Fragmentation is needed and \"Don\'t Fragment\" was set",
43              5: "Source route failed",
44              6: "Destination network is unknown",
45              7: "Destination host is unknown",
46              8: "Source host is isolated",
47              9: "Communication with destination network is administratively prohibited",
48              10: "Communication with destination host is administratively prohibited",
49              11: "Destination network is unreachable for type of service",
50              12: "Destination host is unreachable for type of service",
51              13: "Communication is administratively prohibited",
52              14: "Host precedence violation",
53              15: "Precedence cutoff is in effect"
54          },
55          5: {
56              0: "Redirect datagram for the network (or subnet)",
57              1: "Redirect datagram for the host",
58              2: "Redirect datagram for the type of service and network",
59              3: "Redirect datagram for the type of service and host"
60          },
61          11: {
62              0: "\"Time to Live\" exceeded in transit",
63              1: "Fragment reassembly time exceeded"
64          },
65          12: {
66              0: "Pointer indicates the error",
67              1: "Missing a required option",
68              2: "Bad length"
69          }
70      }
71
72      print("Error:", errorMessages[type][code])
```

The *printErrorMessage* function is later called in the *receiveOnePing* function. Lines 100 checks if the ICMP reply packet is not of type 0, which means that is an ICMP error reply. If so, then line 101 calls the *printErrorMessage* and passes the ICMP reply packet's type and code as parameters. Line 102 returns the tuple (None, None).

```
94          if type == 0 and id == ID:
95              data = struct.unpack("d", recPacket[28:])[0]   # data in ICMP reply is time that ICMP request was sent
96              timeDelay = timeReceived - data
97
98              return (timeDelay, (type, code, checksum, id, sequence, data))
99
100         elif type != 0:
101             printErrorMessage(type, code)
102             return (None, None)
```