

Socket Programming

PLab 2a – SMTP Client, PLab 2b – ICMP Pinger, PLab 2c – Web Proxy

Release Date: November 9, 2017

Due Date: December 5, 2017 (11:59pm)

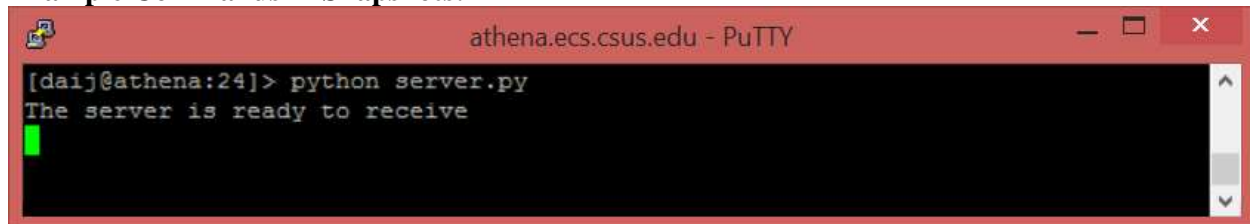
Goal: Practice makes perfect! Socket programming assignments are to help you review and apply your conceptual knowledge from this class.

Attention: Code plagiarism is absolutely **NOT** allowed! Please prepare for a **demonstration** of running your program in front of the instructor/grader and answer their questions.

Instructions: Please follow the lab instruction below to develop your client/server solutions with Python. If you prefer C or Java implementation, that's OK. If you choose to do so, the caveat is that there is more help if you do it in Python.

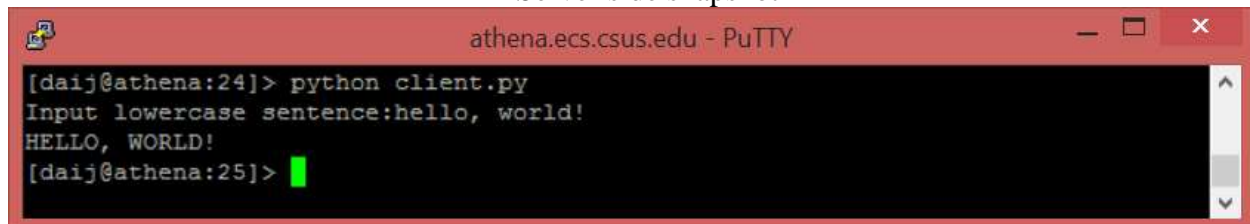
Create a separate submission for each of the labs: SMTP Client, ICMP Pinger, and Web Proxy

Example Commands in Snapshots:



```
athena.ecs.csus.edu - PuTTY
[daij@athena:24]> python server.py
The server is ready to receive
█
```

Server side snapshot



```
athena.ecs.csus.edu - PuTTY
[daij@athena:24]> python client.py
Input lowercase sentence:hello, world!
HELLO, WORLD!
[daij@athena:25]> █
```

Client side snapshot

Deliverable: A project report, an **electronic submission** to Canvas, is expected to include both your **source code** and some **screenshots** that can help you demonstrate your work (**commands, operations, results** and **analysis**). Code plagiarism is absolutely **NOT** allowed! Please also prepare for a **demonstration** of running your program in front of the instructor/grader and answer their **questions** (which are about your code). Your grade will be based on both the report and your performance during demonstration.

Requirement: The report will all be evaluated based on the following grading criteria.

Report Correctness, Completeness, Clarity	20%+15%+15%
Demonstration Correctness, Completeness, Question	20%+15%+15%

PLab 2a: SMTP Lab

By the end of this lab, you will have acquired a better understanding of SMTP protocol. You will also gain experience in implementing a standard protocol using Python.

Your task is to develop a simple mail client that sends email to any recipient. Your client will need to connect to a mail server, dialogue with the mail server using the SMTP protocol, and send an email message to the mail server. Python provides a module, called `smtpplib`, which has built in methods to send mail using SMTP protocol. However, we will not be using this module in this lab, because it hide the details of SMTP and socket programming.

In order to limit spam, some mail servers do not accept TCP connection from arbitrary sources. For the experiment described below, you may want to try connecting both to your university mail server and to a popular Webmail server, such as a AOL mail server. You may also try making your connection both from your home and from your university campus.

Code

Below you will find the skeleton code for the client. You are to complete the skeleton code. The places where you need to fill in code are marked with `#Fill in start` and `#Fill in end`. Each place may require one or more lines of code.

Additional Notes

In some cases, the receiving mail server might classify your e-mail as junk. Make sure you check the junk/spam folder when you look for the e-mail sent from your client.

What to Hand in

In your submission, you are to provide the complete code for your SMTP mail client as well as a screenshot showing that you indeed receive the e-mail message.

Skeleton Python Code for the Mail Client

```
from socket import *

msg = "\r\n I love computer networks!"
endmsg = "\r\n.\r\n"

# Choose a mail server (e.g. Google mail server) and call it mailserver
mailserver = #Fill in start    #Fill in end

# Create socket called clientSocket and establish a TCP connection with mailserver
#Fill in start
#Fill in end
recv = clientSocket.recv(1024).decode()
print(recv)
if recv[:3] != '220':
    print('220 reply not received from server.')

# Send HELO command and print server response.
heloCommand = 'HELO Alice\r\n'
clientSocket.send(heloCommand.encode())
recv1 = clientSocket.recv(1024).decode()
print(recv1)
if recv1[:3] != '250':
    print('250 reply not received from server.')

# Send MAIL FROM command and print server response.
# Fill in start
# Fill in end

# Send RCPT TO command and print server response.
# Fill in start
# Fill in end

# Send DATA command and print server response.
# Fill in start
# Fill in end

# Send message data.
# Fill in start
# Fill in end
# Message ends with a single period.
# Fill in start
# Fill in end

# Send QUIT command and get server response.
# Fill in start
# Fill in end
```

Optional Exercises

1. Mail servers like Google mail (address: smtp.gmail.com, port: 587) requires your client to add a Transport Layer Security (TLS) or Secure Sockets Layer (SSL) for authentication and security reasons, before you send MAIL FROM command. Add TLS/SSL commands to your existing ones and implement your client using Google mail server at above address and port.
2. Your current SMTP mail client only handles sending text messages in the email body. Modify your client such that it can send emails with both text and images.

PLab 2b: ICMP Pinger Lab

In this lab, you will gain a better understanding of Internet Control Message Protocol (ICMP). You will learn to implement a Ping application using ICMP request and reply messages.

Ping is a computer network application used to test whether a particular host is reachable across an IP network. It is also used to self-test the network interface card of the computer or as a latency test. It works by sending ICMP “echo reply” packets to the target host and listening for ICMP “echo reply” replies. The “echo reply” is sometimes called a pong. Ping measures the round-trip time, records packet loss, and prints a statistical summary of the echo reply packets received (the minimum, maximum, and the mean of the round-trip times and in some versions the standard deviation of the mean).

Your task is to develop your own Ping application in Python. Your application will use ICMP but, in order to keep it simple, will not exactly follow the official specification in RFC 1739. Note that you will only need to write the client side of the program, as the functionality needed on the server side is built into almost all operating systems.

You should complete the Ping application so that it sends ping requests to a specified host separated by approximately one second. Each message contains a payload of data that includes a timestamp. After sending each packet, the application waits up to one second to receive a reply. If one second goes by without a reply from the server, then the client assumes that either the ping packet or the pong packet was lost in the network (or that the server is down).

Code

Below you will find the skeleton code for the client. You are to complete the skeleton code. The places where you need to fill in code are marked with `#Fill in start` and `#Fill in end`. Each place may require one or more lines of code.

Additional Notes

1. In “receiveOnePing” method, you need to receive the structure ICMP_ECHO_REPLY and fetch the information you need, such as checksum, sequence number, time to live (TTL), etc. Study the “sendOnePing” method before trying to complete the “receiveOnePing” method.
2. You do not need to be concerned about the checksum, as it is already given in the code.
3. This lab requires the use of raw sockets. In some operating systems, you may need administrator/root privileges to be able to run your Pinger program.
4. See the end of this programming exercise for more information on ICMP.

Testing the Pinger

First, test your client by sending packets to localhost, that is, 127.0.0.1.

Then, you should see how your Pinger application communicates across the network by pinging servers in different continents.

What to Hand in

You will hand in the complete client code and screenshots of your Pinger output for four target hosts, each on a different continent.

Skeleton Python Code for the ICMP Pinger

```
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 = ord(string[count+1]) * 256 + ord(string[count])
        csum = csum + thisVal
        csum = csum & 0xffffffff
        count = count + 2

    if countTo < len(string):
        csum = csum + ord(string[len(string) - 1])
        csum = 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 "Request timed out."

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

        #Fill in start
        #Fetch the ICMP header from the IP packet
        #Fill in end
        timeLeft = timeLeft - howLongInSelect
        if timeLeft <= 0:
            return "Request timed out."

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)
    data = struct.pack("d", time.time())
    # Calculate the checksum on the data and the dummy header.
    myChecksum = checksum(str(header + data))

    # Get the right checksum, and put in the header
    if sys.platform == 'darwin':
        # Convert 16-bit integers from host to network byte order
        myChecksum = htons(myChecksum) & 0xffff
```

```

else:
    myChecksum = htons(myChecksum)

header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
packet = header + data

mySocket.sendto(packet, (destAddr, 1)) # AF_INET address must be tuple, not str
# Both LISTS and TUPLES consist of a number of objects
# which can be referenced by their position number within the object.

def doOnePing(destAddr, timeout):
    icmp = getprotobyname("icmp")
    # SOCK_RAW is a powerful socket type. For more details: http://sock-raw.org/papers/sock\_raw

    mySocket = socket(AF_INET, SOCK_RAW, icmp)

    myID = os.getpid() & 0xFFFF # Return the current process i
    sendOnePing(mySocket, destAddr, myID)
    delay = receiveOnePing(mySocket, myID, timeout, destAddr)

    mySocket.close()
    return delay

def ping(host, timeout=1):
    # timeout=1 means: If one second goes by without a reply from the server,
    # the client assumes that either the client's ping or the server's pong is lost
    dest = gethostbyname(host)
    print("Pinging " + dest + " using Python:")
    print("")
    # Send ping requests to a server separated by approximately one second
    while 1 :
        delay = doOnePing(dest, timeout)
        print(delay)
        time.sleep(1) # one second
    return delay

ping("google.com")

```

Optional Exercises

1. Currently, the program calculates the round-trip time for each packet and prints it out individually. Modify this to correspond to the way the standard ping program works. You will need to report the minimum, maximum, and average RTTs at the end of all pings from the client. In addition, calculate the packet loss rate (in percentage).
2. Your program can only detect timeouts in receiving ICMP echo responses. Modify the Pinger program to parse the ICMP response error codes and display the corresponding error results to the user. Examples of ICMP response error codes are 0: Destination Network Unreachable, 1: Destination Host Unreachable.

Internet Control Message Protocol (ICMP)

ICMP Header

The ICMP header starts after bit 160 of the IP header (unless IP options are used).

Bits	160-167	168-175	176-183	184-191
160	Type	Code	Checksum	
192	ID		Sequence	

- **Type** - ICMP type.
- **Code** - Subtype to the given ICMP type.
- **Checksum** - Error checking data calculated from the ICMP header + data, with value 0 for this field.
- **ID** - An ID value, should be returned in the case of echo reply.
- **Sequence** - A sequence value, should be returned in the case of echo reply.

Echo Request

The echo request is an ICMP message whose data is expected to be received back in an echo reply ("pong"). The host must respond to all echo requests with an echo reply containing the exact data received in the request message.

- Type must be set to 8.
- Code must be set to 0.
- 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.
- The data received by the echo request must be entirely included in the echo reply.

Echo Reply

The echo reply is an ICMP message generated in response to an echo request, and is mandatory for all hosts and routers.

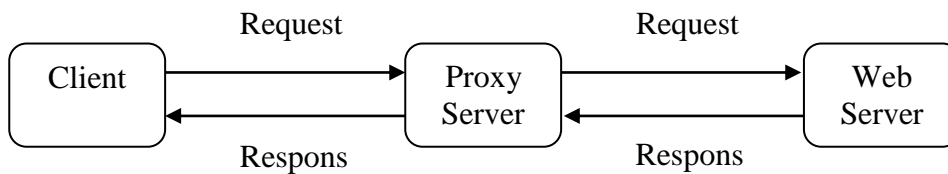
- Type and code must be set to 0.
- The identifier and sequence number can be used by the client to determine which echo requests are associated with the echo replies.
- The data received in the echo request must be entirely included in the echo reply.

PLab 2c: HTTP Web Proxy Server

In this lab, you will learn how web proxy servers work and one of their basic functionalities – caching.

Your task is to develop a small web proxy server which is able to cache web pages. It is a very simple proxy server which only understands simple GET-requests, but is able to handle all kinds of objects - not just HTML pages, but also images.

Generally, when the client makes a request, the request is sent to the web server. The web server then processes the request and sends back a response message to the requesting client. In order to improve the performance we create a proxy server between the client and the web server. Now, both the request message sent by the client and the response message delivered by the web server pass through the proxy server. In other words, the client requests the objects via the proxy server. The proxy server will forward the client's request to the web server. The web server will then generate a response message and deliver it to the proxy server, which in turn sends it to the client.



Code

Below you will find the skeleton code for the client. You are to complete the skeleton code. The places where you need to fill in code are marked with **#Fill in start** and **#Fill in end**. Each place may require one or more lines of code.

Running the Proxy Server

Run the proxy server program using your command prompt and then request a web page from your browser. Direct the requests to the proxy server using your IP address and port number.

For e.g. `http://localhost:8888/www.google.com`

To use the proxy server with browser and proxy on separate computers, you will need the IP address on which your proxy server is running. In this case, while running the proxy, you will have to replace the “localhost” with the IP address of the computer where the proxy server is running. Also note the port number used. You will replace the port number used here “8888” with the port number you have used in your server code at which your proxy server is listening.

Configuring your Browser

You can also directly configure your web browser to use your proxy. This depends on your browser. In Internet Explorer, you can set the proxy in Tools > Internet Options > Connections tab > LAN Settings. In Netscape (and derived browsers such as Mozilla), you can set the proxy in Tools > Options > Advanced tab > Network tab > Connection Settings. In both cases you need to give the address of the proxy and the port number that you gave when you ran the proxy server. You should be able to run the proxy and the browser on the same computer without any problem. With this approach, to get a web page using the proxy server, you simply provide the URL of the page you want.

For e.g. `http://www.google.com`

What to Hand in

You will hand in the complete proxy server code and screenshots at the client side verifying that you indeed get the web page via the proxy server.

Skeleton Python Code for the Proxy Server

```
from socket import *
import sys

if len(sys.argv) <= 1:
    print('Usage : "python ProxyServer.py server_ip"\n[server_ip : It is the IP Address Of Proxy Server]')
    sys.exit(2)

# Create a server socket, bind it to a port and start listening
tcpSerSock = socket(AF_INET, SOCK_STREAM)
# Fill in start.
# Fill in end.
while 1:
    # Start receiving data from the client
    print('Ready to serve...')
    tcpCliSock, addr = tcpSerSock.accept()
    print('Received a connection from:', addr)
    message = # Fill in start.          # Fill in end.
    print(message)
    # Extract the filename from the given message
    print(message.split()[1])
    filename = message.split()[1].partition("/")[2]
    print(filename)
    fileExist = "false"
    filetoUse = "/" + filename
    print(filetoUse)
    try:
        # Check whether the file exist in the cache
        f = open(filetoUse[1:], "r")
        outputdata = f.readlines()
        fileExist = "true"
        # ProxyServer finds a cache hit and generates a response message
        tcpCliSock.send("HTTP/1.0 200 OK\r\n")
        tcpCliSock.send("Content-Type:text/html\r\n")
        # Fill in start.
        # Fill in end.
        print('Read from cache')
    # Error handling for file not found in cache
    except IOError:
        if fileExist == "false":
            # Create a socket on the proxyserver
            c = # Fill in start.          # Fill in end.
            hostn = filename.replace("www.", "", 1)
            print(hostn)
            try:
                # Connect to the socket to port 80
                # Fill in start.
                # Fill in end.
                # Create a temporary file on this socket and ask port 80 for
                the file requested by the client
                fileobj = c.makefile('r', 0)
                fileobj.write("GET "+"http://" + filename + " HTTP/1.0\n\n")
                # Read the response into buffer
                # Fill in start.
                # Fill in end.
                # Create a new file in the cache for the requested file.
                # Also send the response in the buffer to client socket and
                the corresponding file in the cache
                tmpFile = open("./" + filename, "wb")
                # Fill in start.
                # Fill in end.
            except:
                print("Illegal request")
        else:
            # HTTP response message for file not found
            # Fill in start.
            # Fill in end.
```

```
# Close the client and the server sockets
tcpCliSock.close()
# Fill in start.
# Fill in end.
```

Optional Exercises

1. Currently the proxy server does no error handling. This can be a problem especially when the client requests an object which is not available, since the "404 Not found" response usually has no response body and the proxy assumes there is a body and tries to read it.
2. The simple proxy server supports only HTTP GET method. Add support for POST, by including the request body sent in the POST-request.
3. *Caching*: A typical proxy server will cache the web pages each time the client makes a particular request for the first time. The basic functionality of caching works as follows. When the proxy gets a request, it checks if the requested object is cached, and if yes, it returns the object from the cache, without contacting the server. If the object is not cached, the proxy retrieves the object from the server, returns it to the client and caches a copy for future requests. In practice, the proxy server must verify that the cached responses are still valid and that they are the correct responses to the client's requests. You can read more about caching and how it is handled in HTTP in RFC 2068. Add the simple caching functionality described above. You do not need to implement any replacement or validation policies. Your implementation, however, will need to be able to write responses to the disk (i.e., the cache) and fetch them from the disk when you get a cache hit. For this you need to implement some internal data structure in the proxy to keep track of which objects are cached and where they are on the disk. You can keep this data structure in main memory; there is no need to make it persist across shutdowns.