Computer Networks Laboratory

CSE 324 Lab Experiment # 8 Fall 2017

Introduction to Socket Programming

1. Objectives

- Introduction to **Socket Programming.**
- To create a **Client-Server application** using **Socket Programming** in **Python.**

2. Background

2a. Socket Basics

A *network socket* is an endpoint of an inter-process communication flow across a computer network. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents. Today, most communication between computers is based on the internet protocol; therefore most network sockets are *internet sockets*. To create a connection between machines, Python programs import the **socket** module, create a socket object, and call the object's methods to establish connections and send and receive data. Sockets are the endpoints of a bidirectional communications channel.

In this lab, you will learn the basics of socket programming for TCP connections in Python: how to create a socket, bind it to a specific address and port, as well as send and receive a HTTP packet. You will also learn some basics of HTTP header format.

2b. Hints

Here are a few hints that may help you as you write the program.

- ✓ You have to choose a server port to connect to. Ports from 1-1023 are mostly used for certain services and require administrative privileges. Use port numbers greater than at least 1023.
- ✓ Close your sockets cleanly before exiting the program. If you abort the program, the port may not be freed.
- ✓ You can run all of the processes on the same machine. For your machine, just use localhost. You can use ifconfig (unix) or ipconfig (windows) to determine the IP address for testing across multiple machines.
- ✓ Be wary of overzealous firewalls stopping your connections - try temporarily disabling firewalls if you find your connections timeout or are denied.

2c. Create a Socket

This first thing to do is create a socket. The **socket.socket** function does this. Run **socketExample1.py**. The code will create a socket with **Address Family** : **AF_INET** (this is IP version 4 or IPv4), **Type : SOCK_STREAM** (this means connection oriented TCP protocol).

2d. Connect to a Server

We connect to a remote server on a certain port number. So we need 2 things , IP address and port number to connect to. So you need to know the IP address of the remote server you are connecting to. Here we used the ip address of **google.com** as a sample. Run **socketExample2.py**. It creates a socket and then connects. Try connecting to a port different from port 80 and you should not be able to connect which indicates that the port is not open for connection. This logic can be used to build a port scanner.

2e. Sending data to a Server

Function sendall will simply send data. Let us send some data to google.com. Run **socketExample3.py**. In the above example , we first connect to an ip address and then send the string message "**GET / HTTP/1.1\r\n\r\n**" to it. The message is actually an "**http command**" to fetch the **mainpage of a website**. Now that we have send some data, it's time to receive a reply from the server. So let us do it.

2f. Sending data to a Server and receiving data from the Server

Function recv is used to receive data on a socket. In the following example we shall send the same message as the last example and receive a reply from the server. Run **socketExample4.py**. Google.com replied with the content of the page we requested. Quite simple! Finally, we close the socket.

3. Sample Server and Client Code

A Simple Server

To write Internet servers, we use the **socket** function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server. Now call **bind(hostname, port)** function to specify a *port* for your service on the given host. Next, call the *accept* method of the returned object. This method waits until a client connects to the port you specified, and then returns a *connection* object that represents the connection to that client.

```
#server.py
# TCP Server Code
host="127.0.0.1"            # Set the server address to variable host
port=4444                   # Sets the variable port to 4444
from socket import *        # Imports socket module
s=socket(AF_INET, SOCK_STREAM)
s.bind((host,port))         # Binds the socket. Note that the input to
                            # the bind function is a tuple
s.listen(1)                 # Sets socket to listening with a  queue
                            # of 1 connection
print "Listening for connections.. "
q,addr=s.accept()     # Accepts incoming request from client and returns
                      # socket and address to variables q and addr
data=raw_input("Enter data to be send:  ")
                 # Data to be send is stored in variable data from user
q.send(data)          # Sends data to client
s.close()
# End of code
```

A Simple Client

Now we will write a very simple client program which will open a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's *socket* module function. The **socket.connect(hosname, port)** opens a TCP connection to *hostname* on the *port*. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file. The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits:

```
#Client
# TCP Client Code
host="127.0.0.1"       # Set the server address to variable host
port=4444               # Sets the variable port to 4444
from socket import *    # Imports socket module
s=socket(AF_INET, SOCK_STREAM)  # Creates a socket
s.connect((host,port))          # Connect to server address
msg=s.recv(1024)   # Receives data upto 1024 bytes, stores in a var msg
print ("Message from server : " + msg.strip().decode('ascii'))
s.close()                              # Closes the socket
# End of code
```

Now run this ***server.py*** in background and then run above ***client.py*** to see the result.

Output:

Step 1: Run *server.py*. It would start a server in background.
Step 2: Run *client.py*. Once server is started run client.
Step 3: Output of *server.py* generates as follows:

```
C:\PythonDir\>python server.py
        Listening for connections..
        Enter data to be send:  cse323
```

Step 4: Output of *client.py* generates as follows:

```
C:\ PythonDir\>python clients.py
        Message from server : cse323
```

4. Exercise: A web Server

You will develop a **web server** that handles **one HTTP request** at a time. Your web server should accept and parse the HTTP request, get the requested file from the server's file system, create an **HTTP response message** consisting of the requested file preceded by header lines, and then send the response directly to the client. If the requested file is not present in the server, the server should send an HTTP "**404 Not Found**" message back to the client.

Skeleton Code

Below find the skeleton code for the Web server. 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 Server

Put an **HTML file** (e.g., HelloWorld.html) in the **same directory** that the server is in. Run the server program. Determine the IP address of the host that is running the server (e.g., 128.238.251.26). From another host, open a browser and provide the corresponding URL. For example:

```
http://128.238.251.26:6789/HelloWorld.html
```

'HelloWorld.html' is the name of the file you placed in the server directory. Note also the use of the port number after the colon. You need to **replace this port number** with whatever port you have used in the server code. In the above example, we have used the port number **6789**. The browser should then display the contents of **HelloWorld.html**. If you omit ":6789", the browser will assume port 80 and you will get the web page from the server only if your server is listening at port 80. Then try to get a file that is not present at the server. You should get a "404 Not Found" message.

4. Submit the files

You will hand in the **complete server code** along with the **screen shots of your client browser**, verifying that you actually receive the contents of the HTML file from the server.

Skeleton Python Code for the Web Server

```
#import socket module
from socket import *
serverSocket = socket(AF_INET, SOCK_STREAM)
#Prepare a sever socket
#Fill in start
#Fill in end
while True:
#Establish the connection
print 'Ready to serve...'
connectionSocket, addr = #Fill in start #Fill in end
try:
message = #Fill in start #Fill in end
filename = message.split()[1]
f = open(filename[1:])
outputdata = #Fill in start #Fill in end
#Send one HTTP header line into socket
#Fill in start
#Fill in end
```

```
#Send the content of the requested file to the client
for i in range(0, len(outputdata)):
connectionSocket.send(outputdata[i])
connectionSocket.close()
except IOError:
#Send response message for file not found
#Fill in start
#Fill in end
#Close client socket
#Fill in start
#Fill in end
serverSocket.close()
```

Optional Exercises (Extra Credit up to 3)

1. Currently, the web server handles **only one HTTP request at a time**. Implement a **multithreaded server** that is capable of serving **multiple requests simultaneously**. Using threading, first create a **main thread** in which your modified server listens for clients at a fixed port. When it receives a TCP connection request from a client, it will set up the TCP connection through another port and services the client request in a separate thread. There will be a separate TCP connection in a separate thread for each request/response pair.

2. Instead of using a browser, **write your own HTTP client** to test your server. Your client will connect to the server using a TCP connection, send an HTTP request to the server, and display the server response as an output. You can assume that the **HTTP request sent is a GET method**.

The client should take **command line arguments** specifying the **server IP address or host name**, **the port** at which the server is listening, and **the path** at which the requested object is stored at the server. The following is an input command format to run the client.

```
client.py server_host server_port filename
```