Friday, Jan. 20
Winter 2017

CME 451 – Lab 2

Application-Layer Network Programming and Socket Programming [1]

Pre-Lab: While the material provided in this lab is intended to be self-contained, the interested reader is urged to also browse through the online tutorial of Python programming by Python Software Foundation, available here: `https://www.python.org/doc/`.

Verification: Exercises marked with an **asterisk** (∗) are required to be verified and signed off by a lab instructor during your assigned lab time. When you complete a step that requires verification, simply demonstrate the step to a lab instructor. Turn in the completed verification sheet to your TA at the end of the lab.

Code Submission: Exercises marked with a **dagger** (†) can be completed after your assigned lab time. Submit your code and relevant outputs electronically in one zip file (named <NSID>-<Lab#>) through the BlackBoard no later than a week after your lab. Enhance the readability of your code by adding explanatory comments wherever appropriate. If you are unsure about what is expected, ask your TA.

1 Introduction

Upon completion of this Lab, the students should be able to write simple application-layer network programs, to retrieve information from a website and perform data analysis.

In addition, you should learn the basics of socket programming for TCP connections in Python, including creating a socket, binding it to an IP address and port number, and sending and receiving HTTP packets. You also will create a simple web server and client using Python standard libraries.

2 Client-Side Web Programming in Python

The urllib.request module defines functions and classes that can be used to open URLs (mostly HTTP). In the Example 2.1, we get the usask.ca main page and display a portion of it.

Example 2.1 (*Downloading a webpage*)

Open the Python interactive shell and run the following lines.

```
>>> from urllib.request import urlopen
>>> data = urlopen('http://www.usask.ca/')
>>> print(data.read(380))
```

You should expect an output like this.

```
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">\n<html xmlns="http://www.w3.org/1999/xhtml" lang=
"en">\n<head>\n<title>University of Saskatchewan</title>\n    \n    \n    <meta content=
"Fri, 17 Oct 2014 10:14:01 -0440" name="date" />\n    \n    <meta content="text/html;
charset=utf-8" http-equiv="Content-Type" />'
```

[1] Lab handouts are available online on the Blackboard. Please report typos and send comments to francis.bui@usask.ca.

Once you download the webpage using `urlopen`, you can read it all at once into one big string by calling its `read` method. Here we read and displayed the first 380 bytes of the webpage. Note that urlopen returns a bytes object (signified by character b at the beginning of the output). This is because urlopen cannot automatically determine the encoding of the byte stream it retrieves. The meta tag at the end of the output shows that usask.ca website uses utf-8 encoding. We can use the same for decoding the bytes object to string as follows.

```
>>> print(data.read(380).decode('utf-8'))
```

The output will be like this.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
<head>
<title>University of Saskatchewan</title>


    <meta content="Fri, 17 Oct 2014 10:14:01 -0440" name="date" />

    <meta content="text/html; charset=utf-8" http-equiv="Content-Type" />
```

Now we want you to write Python scripts to retrieve some relevant information from the ECE department's website.

Exercise 2.1 (*Extracting info from a web document*) ∗
In this exercise, you will use Python to extract image filenames from a web document. Write a Python script to perform the following tasks.

(a) Download main page of the ECE website (*i.e.*, `http://engineering.usask.ca/ece/`) and saves it in a .html file, in a subdirectory called lab02 in your current working directory. (*Hint*: In order to create a directory with the necessary parent directories, use the function os.makedirs(). In order to open a file and write into it, use the functions open() and write(), respectively.)

(b) Display a list of the filenames of all the .jpg files in the webpage. (*Hint*: To this end, you need to search for the '.jpg' string in the webpage and extract the filenames.)

Exercise 2.2 (*Retrieving a collection of documents from the web*) †
In this exercise, you will use Python to retrieve documents and do some basic processing. First, browse `http://engineering.usask.ca/ece/syllabi-CME.php` to understand the basic structure of this page. Then, write a Python script to perform the following tasks.

(a) Count the number of *unique* hyperlinks in this page.

(b) Display a sorted (ascending order) list of the filenames of all the .pdf files.

(c) Retrieve all these .pdf files, and save them in a subdirectory called "PDF_files". (*Hint*: Use urllib.request.urlretrieve() to retrieve and save the files.)

3 The Socket Module

One of the important networking module in Python is `socket`. You can think of a socket as an "information channel" with a program on each end. There are basically two types of sockets: server sockets and client sockets.

After you create a server socket, you tell it to wait for connections. It will then listen at a certain network address (a combination of an IP address and a port number) until a client socket connects. The two can then communicate.

Technically, a socket is an instance of the `socket` class from the `socket` module. It is instantiated with up to three parameters: an address family (defaulting to `socket.AF_INET`), whether it is a stream (`socket.SOCK_STREAM`, the default) or a datagram (`socket.SOCK_DGRAM`) socket, and a protocol (defaulting to `0`, which should be okay).

Example 3.1 (*Simple Server & Client*)

A server basically does the following: it creates a socket, binds the socket to an address, listens for incoming connections, accepts the connections, receives the requests, processes them and responds accordingly. The script below describes a simple server.

```
import socket
import sys

# Create an AF_INET (IPv4), STREAM (TCP) socket
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('Socket created.')

# Get the local machine name and assign an arbitrary non-privileged port number
host, port = socket.gethostname(), 1234

try:
    # Bind the socket to the network address
    serversocket.bind((host, port))
except socket.error as msg:
    print('Binding failed. Error code: ' + str(msg[0]) + ', Message: ' + msg[1])
    sys.exit()
print('Binding complete.')

# Queue up to 5 requests
serversocket.listen(5)
print('Socket listening.')

while True:
    # Establish a connection - blocking call
    clientsocket, addr = serversocket.accept()
    print('Connection established.')
    clientsocket.close()
```

Function `socket.socket` creates a socket and returns a socket descriptor. Note that the address family `AF_INET` and type `SOCK_STREAM` indicate IP version 4 and connection oriented TCP protocol, respectively. If any of the socket functions fail then python throws an exception called `socket.error` which must be caught. A server socket uses its `bind` method to indicate which interface the server will listen on. You can get the name of the current host using the function `socket.gethostname`. The `bind` method is followed by a call to the `listen` method, for listening to a given address. The `listen` method takes a single argument, which is the length of its backlog, *i.e.*, the number of connections allowed to queue up, waiting for acceptance, before connections start being disallowed. Once a server socket is listening, it can start accepting clients, using the `accept` method. This method will wait until a client connects, and then it will return the client's socket and address in a tuple. Note that `address` itself is a tuple of the form `(host, port)`. After responding a client's request, the server can start waiting for new connections with another call to accept. This is usually done in an infinite loop.

Now have a look at the script below for a simple client.

```
import socket
import sys

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error as msg:
    print('Socket creation Failed. Error code: ' + str(msg[0]) + ', Message: ' + msg[1])
    sys.exit()
print('Socket created.')

host, port = socket.gethostname(), 1234

# Connect to the server
s.connect((host, port))
print('Successfully connected.')
s.close()
```

A client socket can connect to a server by using its `connect` method with the the server's address. To this end, the client requires the IP address of the server and the port number on which the service is listening. Note that the notion of *connection* only applies to `SOCK_STREAM` type of sockets.

What is discussed in Example 3.1 is called *blocking* or *synchronous* network programming. *Nonblocking* or *asynchronous* network programming is another form of server programming which uses threads to deal with several clients at once.

For transmitting data, sockets have two methods: `send` and `recv` (for "receive"). `send` can be called with a string argument to send data, and `recv` with a desired (maximum) number of bytes to receive data. Using this information and what you learned in Example 3.1, try the following exercise.

Exercise 3.1 (*Echo Server & Client*)

Write a script for a simple *echo server* that echoes back the data it receives to a client that sent it, along with the current time at the server. Then write a simple client to test your server. Note to start the server first if you run both scripts on the same machine.

Exercise 3.2 (*Web Server*) ∗

Using your script from the Exercise 3.1, program a simple *web server* that handles one HTTP request simultaneously. The web server needs to accept and parse the HTTP request, retrieve the requested file from the server's file system, generate an HTTP response, and then send the response message to the client. HTTP response should consist of one header line indicating the successful status of the HTTP request, followed by the requested file. Consult the IETF RFC 2616 (Hypertext Transfer Protocol – HTTP/1.1) to learn more about HTTP header format.[2] The server needs to generate an HTTP "404 Not Found" message if the requested file does not exist on the server.

To test your script, put a `.html` file in the same directory that the server script is in. Run the script. Determine the IP address of the host that is running the server. From another machine in the same subnet, open a browser and provide the corresponding URL, *i.e.*, `http://<ip_address>:<port_no>/<html_file>` (If you want to use the same machine as the client as well, use 'localhost' instead of the IP address). The browser should then display the contents of `.html` file. Then try to obtain a file that does not exist on the server. You should get the "404 Not Found" error message.

Exercise 3.3 (*Web Client*) †

Write an HTTP client, instead of using a web browser, to test your server in Exercise 3.2. Your client should connect to the server over a TCP connection, send an HTTP request to the server, and print out the server response as an output. A `GET` method can be used as the HTTP request sent. The client should take three command line arguments: the server IP address, the port number on which the server is listening, and the path to requested object on the server.

[2]Specifically, see section 10 (Status Code Definitions) and section 14 (Header Field Definitions).

CME 451 Lab 2 - Instructor Verification Sheet

Turn this page to your grading TA.

Name:

Student No.:

Instructor Verification: Exercise 2.1, steps (a) & (b)	
Verified:	Date & Time: