Lab 1

Internet Protocols

Lab Objective: *Learn how TCP and HTTP facilitate communication between computers.*

The Internet Protocol Suite is the set of communication protocols which underly most computer networks. Because TCP (Transmission Control Protocol) and IP (Internet Protocol) are two of the oldest and most important protocols, the entire suite is sometimes called TCP/IP. There many protocols in the suite are divided into four abstraction layers:

1. Application: Software that utilizes transport protocols to move information between computers. This layer includes protocols important for email, file transfers, and browsing the web.

2. Transport: Protocols that define basic high level communication between two computers. The two most common protocols in this layer are TCP and UDP. TCP is by far the most widely used due to its reliability. UDP, however, trades reliability for low latency.

3. Internet: Protocols that handle routing and movement of data on a network.

4. Link: Protocols that deal with local networking hardware such as routers and switches.

TCP

TCP dictates how computers connect to each other, exchange bits of information called packets, and then close the connection. TCP/IP is very reliable, ordered, and error-checked.

Specifically, TCP creates network *sockets*. These sockets send and receive data packets. While we would normally think of sending data between two different machines, two sockets on the same machine can communicate via TCP as well. Using the Python `socket` module, we will demonstrate how to create a network socket (the *server* to listen for incoming data, and how to create a second socket (the *client*) to send data.

Creating the Server

First, create a new socket object. The socket object will be able to listen for and accept incoming connections from other sockets.

The two input arguments specify the socket type. Further description of socket types is in the python documentation.

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

We then define an address and a port for the socket to listen on. A port is analogous to a mailbox on the computer. There are 65535 available ports. Of those, about 250 are commonly used. Certain ports are have pre-defined uses. For example:

- 0 to 1023: Special reserved ports
- 80, 443: Commonly used for web traffic
- 25, 110, 143, 465: Commonly used for email servers

We also specify an address for the host, which is analogous to the "mailing address" of the machine on which the server is running. The address may be set to the computer's IP address, to `'localhost'`, or to an empty string. We bind the socket to the port and address, then call `listen` to tell it to listen for connections.

```
address = '127.0.0.1' #the local machine
s.bind((address, 33498)) #bind to an arbitrary port number
s.listen(1)
```

Next we tell the socket what do with incoming connections. Once a connection is made, the `accept` method returns the connection, which is itself a socket object. The connection object receives data in blocks, so we specify a block size in bits. Data is received in string form.

The connection object can also send back data. In the code below, the connection simply echoes back whatever data it receives. After all the data has been received, we close the connection.

```
size = 2048 #block size
conn, add = s.accept() #conn is our new socket object for receiving/sending data
print "Accepting connection from:", addr
while True:
    data = conn.recv(size) #read 20 bytes from the incoming connection
    #terminate the connection if data stops coming in (no more blocks to receive)
    if not data:
        break
    conn.send(data)
conn.close()
```

Command	Description
`bind((address, port))`	Binds to a port and an address.
`listen`	Starts listening for requests.
`accept`	Accepts a connection from a client, and returns a new socket object and a connection address.
`recv(size)`	Reads and returns a block of incoming data.
`send(data)`	Sends data to the client.
`close`	Closes the socket.
`gethostname`	Returns the host name of the machine.
`getsockname`	Returns the socket's own address.

Table 1.1: Table of Socket Commands

Creating the Client

The client socket will connect to our server. We create the socket the same way as before:

```
import socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

We specify the address of the server, and the port (this needs to be the same port on which the server is listening). We then connect to the server.

```
ip = '127.0.0.1'
port = 33498
client.connect((ip, port))
```

Once connected, the client can send and receive data. Unlike the server, the client sends and reads the data itself instead of creating a new connection socket. When we are done with the client, we close it.

```
size = 2048 #block size
msg = "Trololololo, lolololololo, ahahahaha."
client.send(msg)

print "Waiting for the server to echo back the data..."
data = client.recv(size)
print "Server echoed back:", data

client.close()
```

To see the client and server communicate, open a terminal and run the server. Then run the client in a separate terminal.

Problem 1. Write a file called `simple_server.py`. When run, this file creates a server socket, accepts a connection and then reads incoming data. The server should append the current time onto each data block, then send it back to the client. (Look at `time.strftime('%H:%M:%S')` for formatting the current time.)

Also write a file called `simple_client.py`. In this file, create a client socket and connect to the server created in `simple_server.py`. The client should send a message to the server and print the server's response.

Problem 2. Write a file called `rps_server.py`, which plays rock-paper-scissors with a client. The server should accept a connection, and while the connection is open, cycle through the following loop:

- Receive a move ("rock","paper", or "scissors").
- Generate a random move of its own. Print both moves.
- Determine who won the round.
- Send "you win", "you lose", or "draw" back to the client, depending on the outcome of the round. If the move is invalid, send back "invalid move."
- If the client won, break the loop and close the connection.

Also write a file called `rps_client.py`. The client should connect to the server and then enter a while loop. In the loop, the client sends the server a move and prints the server's response. The client should break the loop and close once it receives a "you win" back from the server.

Although these examples are simple, we use a similar pattern for every transfer of data over TCP. For simple connections, the amount of work the programmer has to do can be minimal. However, imagine trying to request a complicated web page. We would have to manage possibly hundreds of connections. We would naturally want to use a higher level protocol that takes care of the smaller details for us.

HTTP

HTTP stands for Hypertext Transfer Protocol. The protocol is centered around a request and response paradigm. A client makes a request to a server and the server replies with response. HTTP is an application layer networking protocol. This means it is a higher level protocol than TCP, taking care of many of the small details of TCP for us. It usually relies on the underlying TCP protocol to provide networking capabilities. There are several methods defined for HTTP, but the two most common are GET and POST. GET requests are typically used to request information from a server. POST requests are sent to the server with the intent of modifying the state of the server. We can send additional information with both GET and POST requests.

Every HTTP request consists of two parts: a header and a body. The headers contain important information about the request such as the type of request, encoding, among other things. We can add custom headers to any request to provide

additional information. The body of the request contains the requested data. The body of a request may or may not be empty.

We can setup an HTTP connection in Python as demonstrated below. We will encourage you to use the Requests library instead of the modules in the standard library. However, the code below is illustrative of the steps in making an HTTP connection

```
import httplib
conn = httplib.HTTPConnection("www.example.net")
conn.request("GET", "/")
resp = conn.getresponse()
if resp.status == 200:
    headers = resp.getheaders()
    data = resp.read()
conn.close()
print headers
print data      # long string
```

We start by creating a connection to specific host. Then we make a request. In this case, we use GET request. The host we are connected to will respond and we retrieve the response. We will need to check the status of the response to know if our request was processed successfully. A status code of 200 means that everything went alright. We can now attempt to read the data of the response. At the end we explicitly close the connection.

This exchange is greatly simplified by the Requests library

```
import requests
r = requests.get("http://www.example.net")
r.close()
print r.headers
print r.content
```

Now, lets demonstrate various things we can do with HTTP requests. We will use a web service called HTTPBin which is very helpful in developing applications that make HTTP requests. When making a GET request, we can send along a list of parameters. These parameters should be a Python dictionary.

```
>>> data = {'key1': 0, 'key2': 1}
>>> r = requests.get("http://httpbin.org/get", params=data)
>>> print r.content
```

When we post to a server, we have the option of sending data. This data can be a file object, a dictionary or a string. To send our data via post, we first serialize it to JSON and then send the resulting string to the request.

```
>>> p = requests.post("http://httpbin.org/post", data=json.dumps(data))
>>> print p.content
```

Problem 3. Included is nameserver.py. This simple server has allows clients to send the last name of a famous computer scientist with GET and returns the corresponding first name. For example, running the following code results

in the message 'Grace' from the server.

```
r = requests.get("http://localhost:8000?lastname=Hopper")
```

Expand the functionality of the server so that a client can also use GET to obtain a dictionary of everybody whose last name starts with a given string. Then add a method so that clients can add a person to the dictionary (or modify an existing entry) using PUT.