Lab 11

# Web Technologies 1: Internet Protocols

**Lab Objective:** *The internet has strict protocols for managing communications between machines. In this lab, we introduce the basics of TCP, IP, and HTTP, and create a server and client program. We then apply this understanding to navigation of online infrastructure and to data collection.*

The internet is comparable to a network of roads connecting the buildings of a city. Each building represents a computer and the roads are the physical wires or wireless pathways that allow for intercommunication. In order to properly navigate the road, you must use the correct kind of vehicle and follow the established instructions for travel. If these requirements are not met, you are not allowed to navigate the road to another building. There are also vehicles of different capabilities and sizes which can retrieve items from particular buildings. In a similar fashion, the internet has specific rules and procedures, called protocols, which allow for standardized communication within and between computers. By understanding these protocols, we can more easily navigate the world web to collect data, create web dependent programs, and interact with other network connected computers.

The most common communication protocols in computer networks are contained in the Internet Protocol Suite. Among these are TCP (Transmission Control Protocol) and IP (Internet Protocol), which are two of the oldest and most important protocols. In fact, they are so important that the entire suite is sometimes referred to as TCP/IP. However, there are many other protocols in the suite, all of which 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.

For this lab, we will focus on understanding and utilizing TCP/IP and HTTP, which are the most common Transport and Application protocols, respectively.

## TCP

TCP is specifically used to establish a connection between computers, exchange bits of information called *packets*, and then close their connection. Built for host computers on an IP network, TCP allows for a very reliable, ordered, and error-checked stream of data bytes (eight binary bits).

Specifically, TCP creates network *sockets* that are used to 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 (a *server* to listen for incoming data, and how to create a second socket (a *client*) to send data.

### Creating a Server

A *server* is a program that provides functionality to *client* programs. Oftentimes, these server programs run on dedicated computer hardware that we also refer to as servers. These servers are fundamental to the modern networks we work on and provide services such as file sharing, authentication, webpage information, databases, etc. To create a server, we 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 can be found in the python documentation.

```python
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 = '0.0.0.0' # Default address that specifies the local machine and allows↩
      connection on all interfaces
s.bind((address, 33498)) # Bind the socket to a port 33498, which was arbitrarily↩
      chosen
s.listen(1) # Tell the socket to listen for incoming connections
```

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 (as a string) in blocks, so we specify a block size in bits.

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 of 20 bytes
conn, addr = 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
    if not data: # Terminate the connection if data stops arriving (no more ↩
         blocks to receive)
        break
    conn.send(data) # Send the data back to the client
conn.close()
```

We can also close the server by using the KeyBoardInterrupt (`ctrl+c`).

**ACHTUNG!**

When running the code above, you will see the program hang on the line,

```
conn, addr = s.accept()
```

As mentioned above, the `accept()` method does not return until a connection is made. Therefore for the code above to execute in its entirety, a client needs to connect to the server. Creating a client is addressed in the next section.

## Creating a Client

A *client* is a program that contacts a server in order to receive data or functionality. We use many client programs which include web browsers, mail programs, online video games, etc. We will create a new client socket to connect to our server. We follow a similar process as we did for the server socket:

```
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 = '0.0.0.0'
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 of 20 bytes
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.

| 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 11.1: Table of Socket Commands

**Problem 1.** Write a file called `simple_server.py`. When run, this file should create a server socket, accept a connection and then read incoming data. The server should append the current time onto each data block, then send it back to the client.
(Hint: use `time.strftime('%H:%M:%S')` to format the current time nicely.)

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, requesting a complicated webpage would require us to manage possibly hundreds of connections. Naturally, we would want to use a higher level protocol that takes care of the smaller details for us.

## HTTP

HTTP stands for Hypertext Transfer Protocol, which is an application layer networking protocol. It is a higher level protocol than TCP and takes care of many of the small details of TCP for us. It also relies on underlying TCP protocol to provide network capabilities. The protocol is centered around a request and response paradigm. A client makes a request to a server and the server replies with response. There are several methods, or requests, defined for HTTP servers. The three most common of which are GET, POST, and PUT. 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. PUT requests are used to add new pieces of data to the server.

Every HTTP request consists of two parts: a header and a body. The headers contain important information about the request including: the type of request, encoding, and a timestamp. We can add custom headers to any request to provide additional information. The body of the request contains the requested data and 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 using `httplib`.

```python
import httplib
conn = httplib.HTTPConnection("www.example.net")
conn.request("GET", "/")
resp = conn.getresponse() # Server response message
if resp.status == 200:
    headers = resp.getheaders()
    data = resp.read()
conn.close() # After collecting the information we need, we close the connection
print headers
print data      # Long string full of HTML code from the webpage
```

The above codes starts by creating a connection to specific host. We then make a request, which in this case was a GET request. The host we are connected to then responds and we retrieve the response. In order to know if our request was successfully processed, we need to check the response message from the server. A status code of 200 means that everything went alright. We can now read the data of the response and then explicitly close the connection.

This exchange is greatly simplified by the `requests` library:

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

We will now examine an example of a GET request with a list of parameters in the form of a Python dictionary. We will use a web service called HTTPBin which is very helpful in developing applications that make HTTP requests.

```python
>>> data = {'key1': 0, 'key2': 1}
>>> r = requests.get("http://httpbin.org/get", params=data)
>>> print r.content # Our parameters now show up in the args() of the get request
```

For more information on the `requests` library, see the documentation at `http://docs.python-requests.org/`.

**Problem 3.** The file `nameserver.py` is an example of a simple HTTP server using the Python module `BaseHTTPServer`. It allows clients to send the last name of a famous computer scientist with GET and then returns the corresponding first name. For example, running the following code results in the message `"Grace"` from the server. The *?* signifies the start of a *query* in the URL.

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

Expand the functionality of the server to do the following:

- Obtain a list of everybody whose last name starts with a given string. For example, a query of `lastname=B` would return all the names whose last name starts with the letter `"B"`.

- Similarly, obtain a list of everybody in the list of names with `lastname =AllNames`.
- Then add a method `do_PUT()` so that clients can add a person to the dictionary (or modify an existing entry) using PUT with a query of `firstname` and `lastname`. Multiple query fields separated using the `"&"` character.

The format of the list of names returns from the server should be:

```
LASTNAME, FIRSTNAME
LASTNAME, FIRSTNAME
   ...      ...
```

Though simple, this HTTP server was the original idea for Instant Messaging. The instant messaging client on your computer sends your name, IP address, and port number to the web server.

This server then advises your contacts that you are online. If they wish to message you, all communication then happens between your port and IP address and the port and IP address of the other user without passing through the main name server.

**Problem 4.** We will now practice chatting through HTTP. To do so, create a new HTTP server in a file called `chatserver.py` using the same method as Problem 3. This new server should store a dictionary with keys of names and values as lists of messages. The GET method should return in this format:

```
Name1:
    Message 1
    Message 2
Name2:
    Message 1
    ...
...
```

The PUT method should accept a `name` and a `message` and store them in your dictionary.

Once you have successfully created your server, run it through a terminal at your localhost IP address. You may then PUT your `name` and `ipaddressport` on the HTTP server provided by your instructor. To find your own IP address, run in a separate terminal, `ifconfig` (for Mac or Linux) or `ipconfig` on Windows command prompt. You may use GET requests of `AllNames` to the nameserver to retrieve the names of other active users. To send them a message, use PUT requests to their respective IP address and port.

Sample requests are as follows:

```
# Request names and IP addresses from nameserver.
r = requests.get("http://ipaddress:portnumber?name=AllNames")
```

```python
# Send a message to another persons chat server.
r = requests.put("http://ipaddress:portnumber?name=Thomas&message=Hello ↩
    World")
```

To receive credit for this problem, please do the following:

1. PUT your name, IP address, and port number on the provided name-server.
2. Send a message with your name to the *TA* user IP address and port.
3. Accomplish any additional specified by your instructor.

To receive credit for this problem, send a message with your name to the *TA* user IP address. When you have finished chatting, update your IP address to an empty string in the name server.

**Achtung!**

This problem requires a level of remote network access. If the network doesn't allow you to connect, use SSH to remote into a network location where this can be done.