CPSC 254
Fall 2015

Project #4 — HTTP server

Introduction

You will write a minimalist HTTP 0.9 server in Python. The program will accept incoming HTTP connections and answer requests according to the HTTP 0.9 standard. You may work in a group of up to 3 people, but if you choose to work with a group, you will need to utilize GitHub to update your project and each of you will write a separate paragraph in the deliverables about your experience using GitHub.

Minimal HTTP 1.0

The HyperText Transfer Protocol (HTTP) is a protocol for transferring files from a server to a client using TCP connections. The most recent version of HTTP is 1.1; in this assignment you will implement a server for HTTP version 0.9, which is slightly simpler.

Sessions

A HTTP server accepts incoming TCP connections on a designated port. The canonical HTTP port is 80, but a server may choose to use a different port instead. Your server should wait for connections on port 8000. The server and client exchange one piece of information in a session. A session involves the following steps:

1. The client opens a TCP connection to the server.
2. The server accepts the connection.
3. The client sends a request:
    a. A line of ASCII text with the word ”GET”, followed by a space, followed by the document address.
    b. A document address is a single word (no spaces) that specifies the path to a file (see below).
    c. The carriage return (\r) and newline (\n) whitespace characters.
4. The server sends a response: either the contents of the resource (file) specified by the document address, or an error message.
5. The server closes the TCP connection.

Document Addresses

The document address part of the GET line uses roughly the same format as Unix path names. A full URL looks like

http://www.fullerton.edu/index.html

The document address is everything after the http: part and the domain name. The document address for the above URL is

/index.html

More generally, a document address is an ASCII string comprised of one or more path segments.

A path segment is a forward slash (/) followed by one or more extended alphanumeric characters.

An extended alphanumeric character can be any of the following:

- an upper case letter (A-Z)
- a lower case letter (a-z)
- a digit (0-9)
- punctuation; any of the following: $ - @ . & + ! * ” ’ ( ) ,
- a % character followed by two hexadecimal digits (0-F)

For the purposes of this project you can treat any string that starts with a slash and contains no whitespace as a valid document address.

Requirements

Write a Python program that implements a HTTP 0.9 server. Your program should log its activities by printing messages about what it does to standard output.

Your program should listen for connections on port 8000, and respond to each according to the HTTP 0.9 protocol:

1. Read a request line
2. If the request is not valid (”GET”, then a space, then a valid document address):
    a. Send the string ”400 Bad Request” to the socket
    b. Print a message about a bad request to standard output.
3. If the document address is /STATUS:
    a. Send some text that says the server is running.
    b. Include the count of documents that have been successfully served so far.
4. Otherwise (a valid request for a file other than /STATUS):
    a. Remove the leading slash from the document address, and consider the resulting string to be a filename.

b. Try to open the filename.
c. If the filename exists and is readable, write it to the socket as a response. Print a message about serving the file to standard output. Increment the counter of files served so far.
d. Otherwise (i.e. opening the file triggers an exception), write the string ”404 Not Found” to the socket. Print a message about not finding the file to standard output.

5. Close the connection.

Modules

The Python library contains modules that implement HTTP servers, namely BaseHTTPServer and SimpleHTTPServer. You must not use these. You need to implement the above functionality using the Python socket module directly.

Deliverables

When you have your program working, submit the following (in a word doc or pdf:

1. Your program’s source code

2. A screenshot of Firefox displaying the following pages, served from your program. Each screenshot should show the URL you requested in the address bar, as well as the rendered page.
    a. The status page, showing 0 files served so far
    b. A valid HTML file (you can use the test1.html or test2.html from Project 3, or another file)
    c. A second valid HTML file
    d. The ”404 Not Found” message for a file that does not exist
    e. The status page, showing 2 files served so far

3. A transcript of connecting to the server with telnet, issuing a bad request, and getting the ”400 Bad Request” message.

4. The program’s standard output for all the above steps.

   * If you worked in a group, each of you will write a paragraph of your experience working with GitHub. Be sure to indicate who wrote which paragraph.

The first page of your submission should include your name and a title.

The following is an example of a valid standard output transcript, for item above:

```
Sent STATUS
Sent test1.html
Sent test2.html
Sent STATUS
Could not find nonsense.html
Bad request
```

Hints

Python documentation

The official Python documentation has technical documentation, and example code, for dealing with sockets:

http://docs.python.org/library/

HTTP documentation

You may want to refer to the official documentation for the HTTP 0.9 standard:

http://www.w3.org/Protocols/HTTP/AsImplemented.html

or the Wikipedia page on HTTP:

http://en.wikipedia.org/wiki/Http

Note that the Wikipedia page refers to HTTP 1.1, which is more complex than version 0.9.

Testing

When you are first getting your program to work, it is recommended to test it with the telnet command. With telnet it is easy to see whether the server is producing correct output.

Once the program seems to be correct according to telnet, try interacting with it using a web browser such as Firefox. The web address of your status page should look like

http://localhost:8000/STATUS

Keep in mind that Firefox caches web pages aggressively. It may display an old, cached version of your program's output, even if you press Reload. You can force Firefox to communicate with your server by restarting Firefox.

telnet

You may want to consult the manpage for telnet.

Flow control

Your program will probably need two nested loops: an outer loop that repeats for each incoming telnet session, and an inner loop that repeats for each line of text.

Reading a line

The socket.recv method reads all available characters at the moment it is called. When that socket is connected to a telnet session, each entered line becomes available at the same moment, so socket.recv will return one line of input text.

Stale ports

If you bind a server socket to a given port, such as 8000, then that port is unavailable to all other programs until your program closes the port. As a consequence, if your program crashes before it has a chance to close its socket, port 8000 will be unusable until you reboot.

While you're developing your program, a reasonable workaround is to simply use a different port if that happens. For example, if port 8000 is stale, modify your program to use 8001. If port 8001 becomes stale, use port 8002.

If you do this, make sure to change your program back to port 8000 before you submit your project.

Firewalls

If you have activated a network firewall, it may block incoming connections from reaching your daemon. You will need to deactivate your firewall or configure it to allow connections to your daemon. The default Ubuntu installation does not have an active firewall.