CSE 150 Final Project
Simple Web Server

Project Goals:

In this project you will build a simple web server in Python. After completing this lab you will have learned some of the basic concepts of socket programming: how to open and create a socket, bind to a specific address and port, listen for client connections, and send messages through the socket interface. Additionally, you will practice with the HTTP message format by processing HTTP requests and sending HTTP response messages.

Your web server will handle one HTTP request at a time. The server should accept and parse incoming HTTP GET requests, locate and read the requested object from the server's file system, create an HTTP response message which includes the required HTTP status line and specified headers. If the requested web object is located, it is included directly in the body of the response message and the response is sent to the client via the socket interface. If the requested object is not located, the server should respond with the appropriate error message (see requirements below).

Submission and Due Dates:

This assignment is divided into two parts, each with its own deadline as follows:

- Part 1: Due Saturday March 4, 2023
- Part 2: Due Sunday March 12, 2023
- Early Birds: Due Wednesday March 8, 2023 at 9pmT
    - Those students who are ready to demo their project during Week 9 can earn 15 points **extra credit** in the lab category of the course grading rubric. Note: code must be submitted by the above date to qualify for Early Bird. No late submissions for Early Bird extra credit.
- NO LATE SUBMISSIONS: there no no late submissions accepted for the project. Please don't ask - we cannot accept late submissions. Submit whatever code you have completed by the above date for partial credit.
- In person project demos to TA:
    - Early Birds: Week 9 (scheduled during last 30 minutes of lab)
    - Regular: Week 10 (entire lab period is dedicated to demos)

What to Submit

- Part 1: 2 files (as described in Part 1)
    - server code <CruzID>MyServerPart1.py
    - <CruzID>Part1Output.txt

- Part 2: 4 files (as described in Part 2)
    - server code <CruzID>MyServerPart2.py
    - <CruzID>SocketOutput.csv
    - <CruzID>HTTPResponses.txt
    - <CruzID>Questions.pdf

Requirements and Guidelines:

Environment:

- Python3 and VM: The program must be written in Python3 and run on the Mininet VM. Code that is not developed on your Mininet VM might not be compatible with our test scripts and therefore will not pass the tests.
- HTTP: Server and Client use HTTP 1.1
- File System: A file system is provided to facilitate your testing (see File System below).

Server:

- **Allowed** libraries: socket, argparse, sys, os

  Your server MUST process the HTTP GET request and send the corresponding requested response message through the socket interface.

  > For this project, with respect to sockets and HTTP, **you may not use any network library** other than Python's socket, an interface for the low-level socket communication. High-level APIs such as http and urllib are ***not* allowed**. Please direct questions to your TA.

- Server program name: `<CruzID>MyServer.py` where CruzID is your UCSC email username
- Server Listen/Welcome Socket: This socket listens on the loopback address and the port number specified by the user on the command line. This socket stays open until the program is terminated.
- Connection Socket: When a client request comes in, open a connection socket. When the server's response is sent, close the connection socket.
- Request Pipelining: The server processes one request at a time and is NOT required to process pipelined HTTP requests.
- Allowed Client Requests: Your web server accepts **HTTP GET** requests only. You do not need to process other types of requests. If a different request method is received, return the appropriate status code in the server's response (see Response Status Codes)..

- Response Status Codes: As needed, the Server returns these status codes and phrases in the status line of its response.
    - `200 OK, 404 FILE NOT FOUND, 501 NOT IMPLEMENTED and 505 HTTP Version Not Supported`

- Command-line Arguments: Your program should accept a single port number and the path to the server's root file directory

  Usage: `MyServer.py [-p port] [-d directory]`

  Example: **`<CruzID>MyServer.py`** `-p 80 -d /home/username/web`

  Invalid Command-line arguments:

  Terminate the program if these events occur:

    - If an Invalid socket is entered, print an error message to `stderr`
    - if the specified root directory does not exist, print an error message to `stderr`

- Allowed Ports: Allow any integer port number that is within the valid range

- Files: Your server should be able to accept requests for both text and binary files. Additionally, the server must work with files of up to 1Gbyte as long as the computer has enough free space.

  You can assume that the file extension indicates the file type. Expected file types are: .csv, .png, .jpg, .gif, .zip, .txt, .html, .doc and .docx. Please refer to common MIME types.

- Error Checking: Your program must check for the following error conditions:
    - Socket Errors: The socket operations should handle any errors or exceptions thrown. Server application should continue to listen for subsequent client requests, and not crash.
    - File Errors: Print an error message to `stderr` and exit the program if the specified root directory (entered on the command line) does not exist.
    - Input Errors: Check that user enters a valid port number, and print an error message to `stderr` error if there is an error.
    - Resource not Available errors: Return 404 response code if requested resources are not present in the webserver's filesystem
    - Unsupported HTTP Methods: You are only required to handle HTTP GET requests. Return Status code: 501 Not Implemented
    - Unsupported HTTP Version: You are only required to handle HTTP 1.1 requests. Return Status code: 505 HTTP Version Not Supported

- Closing your program: Your program must close gracefully – make sure to close any open sockets before exiting the program.

- Client Requests: HTTP GET only

Web Server Skeleton Code

The textbook also provides basic code for a simple server written in Python (see Resources - Chapter 2 Section 7). This is an excellent example and you can follow the given structure.

File System

Here is a link to a zip file of the file system. Unzip/install this file in the root directory for your web server. This file system includes several HTML and binary files with which you can test your web server. Feel free to test with your own files as well.

Part 1: Reading a Text File and Writing HTTP Response

Server Info and Creating the HTTP Response Message

Part 1 is the first step to create your webserver. This step focuses on reading from the command line, reading from a file, forming an HTTP response message and writing the response to `stdout`. Note: Part 1 **does not** create or open sockets - this is done in Part 2.

Instructions for Part 1:

1. Bare bones Server: Create a bare-bones Python script for your server.

2. Read from the command line: Read the port number and root directory from the command line (see error checking above). **`argparse`** is allowed.
    a. Check input for valid port numbers: allow registered port numbers, but warn users if one was entered. Terminate program if port number out of range was entered and print error message to stderr.
    b. Print to **stdout:** server's port number and root directory in this format: <Port:> entered port number, <Root directory>: entered root directory path

3. Create a simple HTML file: named `HelloWorld.html` that displays the message "Hello World, I am X" where X is your full name and student ID. Put the file into the directory specified on the command line.

4. Form HTTP Response: Pretend an HTTP GET message requesting the file `HelloWorld.html` was received. Your server reads the file and forms a **complete** HTTP Response Message that indicates success and includes the requested file in the body of the response.

a. Include these HTTP headers in your message: `Content-Length, Content-Type, Date and Last-Modified.` Note: Headers are name:value pairs – the value should match expected values. None of the values should be hard-coded – your code needs to determine them. You can assume that the file extension indicates the file type.

5. Print HTTP response message to **stdout**
    a. Print the HTTP Response message to **stdout** (we are not using sockets until Part 2). Don't forget the body of the response includes the requested object. Note: Later allow for files to be binary-like image files, i.e., don't assume text files (line-oriented). Research how the read operation should be done.

Submission for Part1:

- Your server code: `<CruzID>MyServerPart1.py`
- Output file: `<CruzID>Part1Output.txt`
  `Run <CruzID>MyServerPart1.py > <CruzID>Part1Output.txt` (Note: You need to enter the command line arguments and replace CruzID with your own ID.)

Part 2: Your final Web Server (processes HTTP GET requests)

Your fully functioning web server should:

- Observe all of the Requirements and Guidelines at the beginning of this document. Review as needed.
- Accept Port number and directory from the user on the command line
- Open a listening/welcome socket and print to **stdout:** "Welcome socket created: <IP Address>, <Port> (no hard coding → determine from socket's attributes)
- When a client connection is established, print to **stdout: "**Connection requested from <Incoming IP address>, <Source port> **.** When this socket is closed**,** print a message to **stdout: "**Connection to <Incoming IP address>, <Source port> is closed.
- Accept and process client HTTP GET requests one at a time
    - Process the received HTTP message and respond appropriately over the connection socket.
    - For example, if an HTTP GET is received, check the server's root directory for the file requested
        - If the requested file is found, form an HTTP response and return to the client a successful HTTP response message. 4 headers are required: `Content-Length, Content-Type, Date and Last-Modified.` Note: Headers are name:value pairs – the value should match expected values. None of the values should be hard-coded – your code needs to determine them. You can assume that the file extension indicates the file type. You can consider

also including the header Connection: close because your server processes a single request and then closes the connection

- If the requested file is not found, return the appropriate error message to the client. Specifically, when the requested file is not present in the server's file system, the server should return a "404 Not Found" message to the client.
- Other errors can be formed using the Status Codes indicated in the Guidelines.
- For responses other than 200 OK, return headers as appropriate for your implementation. The Date: header should always be included.

- After writing the response to the socket, close the connection socket.

Output Files:

1. CSV file: <CruzID>SocketOutput.csv
   Once your server responds to an HTTP request message, write the following to a CSV file (.csv extension) using this format (it is OK to redirect the output to stdout for a file):

   "Client request served", "4-Tuple:", {server_ip}, {server_port}, {client_ip}, {client_port}, "Requested URL", {url}, {status line}, "Bytes transmitted:", {value of Content-Length}

   Example output to `stdout`:
   Client request served,4-Tuple:,127.0.0.1,80,192.60.3.1,9045, Requested URL,./HelloWorld.html,HTTP/1.1 200 OK,Bytes transmitted:, 50

2. Text file: <CruzID>HTTPResponses.txt
   Print the status line and headers of all the HTTP Response messages sent by your server. Do not include the requested object.

Submission for Part2:

- Your server code `<CruzID>MyServerPart2.py`
- Run your server for the following requested objects
  - A text file
  - A small binary file
  - A large binary file

  and send the output to the following 2 files:
  - <CruzID>HTTPResponses.txt – Text file as described above.
  - <CruzID>SocketOutput.csv – CSV file as described above
- <CruzID>Questions.pdf: Answers to Questions and Discussion in a PDF named

Getting Started

Part 1

1. Carefully read through the Requirements and Guidelines
2. Read through the Useful Resources section of this document
3. Read Section 2.7 of the textbook Computer Networks: A Top-Down Approach on sockets. This section is extremely useful!
4. Review the format of HTTP Request messages and Server responses (class notes, text book and the diagrams below in Useful Resources).

Part 2

Before you start on the server code, get familiar with the provided file system. Install the file system on the VM and unzip it. Then run the simple Python web server to familiarize yourself with the files by accessing a simple text file with wget and also in the browser. Next you can try a small binary file (like dog1.jpg).

Once you have thought about file placement and the directory in which the server starts up etc., It is time to get your own server going!

Testing:

1. Testing the Basic Server with wget and a simple file

- Start up your web server and enter command line parameters
- `HelloWorld.html`
    - In another terminal, enter wget `http://IP:Port/HelloWorld.html` where `IP` and `Port` are the IP address and port number entered on the command line
    - Verify that the `HelloWorld.html` file is properly received
- If everything looks good in wget and the output to **stdout** is also correct, move on to testing with the browser and other files in the file system.

2. Testing the Web Server with a Browser on your VM

Open the browser and provide the corresponding URLs for files stored in the web server's root directory.

For example, type in the browser: `http://A:B/HelloWorld.html`

where A is the server's `IP` Address and B is the port number on which the server is listening

Note: If you omit a specified port number the browser will assume port `80` and you will get the document only if your server is listening on port `80`. It is fine to use port 80 on the VM, but if you are developing on your own machine you will not be allowed to use it.

3. Testing the Web Server with the provided file system

All files aren't text! In addition to text files, you need to write binary files to your connection socket. This means you must read bytes from the file and write bytes into the socket.

To verify the binary image files are downloaded correctly, it is easiest to use the browser. For example, to verify an image, such as dog2.jpg, request the file in the browser. For binary files that are not images, executing 'diff' will allow you to determine if the file downloaded is the same as the original file stored at the server (it should be!).

Make sure you are able to request all files in the file system. Try Example1.html and eventually the large binary file. (Remember that your server is not required to support pipelined requests - you cannot control the browser, but you can give Example1.html a try.)

Finally, make sure you are checking for the errors outlined in the Guidelines.

Grading

Project Demo: Each student will meet with their TA during an assigned time slot. You will be asked to run your program and discuss your code design. The demo is **mandatory and is the only way to receive a grade for the project.** Missing the demo will result in a score of 0. Once you sign up for a demo slot this is your commitment and it cannot be missed without prior rescheduling and agreement with your TA.

Rubric:

Each submission will be tested to make sure it works properly and handles errors. Grades are allocated using the following guideline:

Item	100 Points Total
Part 1: Functionality and correct output	15
Part 2: Functionality including error handling Preprocessing tests performed prior to demo	45
Questions and Discussion	10
Live TA Demo	30

Useful Resources

- HTTP and Server:
  - Computer Networks: A Top Down Approach Chapter 2 Section 7 HTTP and Creating Socket
  - Lecture material on HTTP and steps for downloading web pages
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types
- Writing and Reading from Socket:

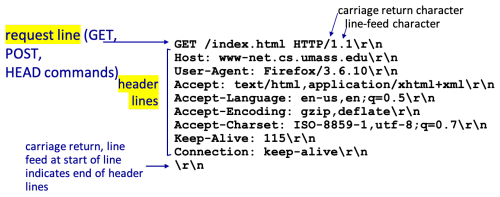
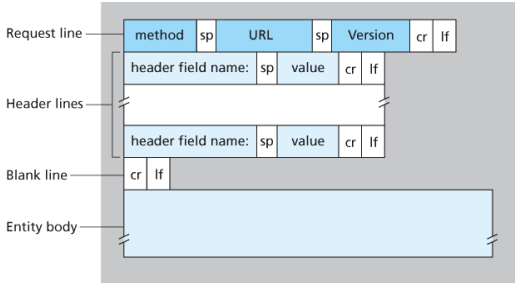
  https://pythontic.com/modules/socket/send

  https://www.geeksforgeeks.org/socket-programming-python/

  https://docs.python.org/3/library/socket.html

  https://docs.python.org/3/library/socket.html#creating-sockets

  https://www.ietf.org/rfc/rfc2553.txt
- HTTP requests and responses:
  - https://www.rfc-editor.org/rfc/rfc9110.html#name-status-codes
  - https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html
  - https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html

HTTP Request message:	HTTP Request programmer's view:
	

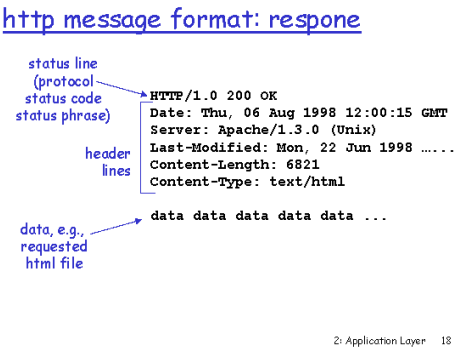
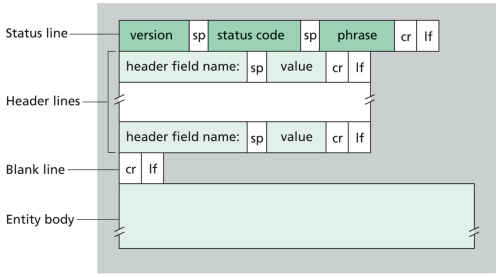
HTTP Response message:	HTTP Response programmer's view:
	

Figure 2.9 ♦ General format of a response message

- Exception Handling in Python
  - https://www.programiz.com/python-programming/exception-handling

Honor Code

All code must be developed independently. All the work must be your own - please remember to cite any sources in your code (comments). MOSS will be run on all submissions and any instances of plagiarism will result in a score of 0.

Questions and Discussion

1. What is a protocol?
2. What types of messages does your server accept?
3. What types of responses does your server return?
4. How does your program terminate? What happens to the socket?
5. When you think about writing this web server, did you need to know anything about TCP? Explain your answer.
6. Think about reading and writing to a file. Compare this operation to reading and writing with a socket. What is the difference? Is one way easier?
7. Follow the complete message exchange in Wireshark when HelloWorld.html is requested (include screenshots)
   a. Highlight the exchange in Wireshark that shows the socket being opened, document requested and finally socket closed
   b. How many bytes were transferred? How many bytes in the file were transferred?
   c. Highlight the above in screenshots.
   d. Include a screenshot of the full page loaded in the browser.
8. Follow the complete message exchange in Wireshark when Example2.html is requested.
   a. Highlight the exchange in Wireshark that shows the socket being opened, documents requested and finally socket closed
   b. How many files were transferred? How many bytes in each file?
   c. Highlight the above in screenshots.

d. Include a screenshot of the full page loaded in the browser

9. As an application layer programmer, what API did you rely upon to complete this assignment? What do you think of it?

10. What error handling cases did you implement?

11. If you were to extend your server to be a "real" web server, enumerate the changes/additions you think you would need to make.

FAQ

- **Do I have to use the VM for this lab?**
  - Yes. Your program will be evaluated on the VM, so even if you choose to develop on your host machine, please verify your program on the VM for Python3 compatibility. Some syscalls may behave differently on your host machine as well.
- **What libraries are allowed?**
  - socket, argparse, sys, os
- **Should I use python2 or python3?**
  - Python3
- **What types of HTTP Requests must be accepted by the web server?**
  - Your web server accepts and responds to HTTP GET requests only. You do not need to process other types of requests, but should return the proper response status code indicating the error condition.
- **What if I cannot attend my demo?**
- **What file should be supported?**
  - Expected file types are: .csv, .png, .jpg, .gif, .zip, .txt, .html, .doc and .docx. Please refer to common MIME types.
- **What if I cannot attend my demo?**
  - Your project grade will be 0. The completed demo to your TA is the only way to receive credit for the project.