Project 2 - Parallel HTTP Server

Contents

Project Objectives

In this project, you will be upgrading the simple web server created in Project 1 to support concurrent (parallel) downloads. In doing so, you will gain:

- Hands-on experience with parallel programming (threads or processes) in the Python 3 programming language
- Hands-on experience with a broader set of HTTP/1.1 features

Requirements

All of the requirements specified in Project 1 are still valid. If part of your original Project 1 implementation was incomplete or buggy, you should fix it, either before starting this project, or during this project.

The high-level goal of this project is the following:

Your web server must be upgraded to support concurrent requests from multiple web browsers. Simply put, if your website hosted a single 100GB file, and 1, 10, 100, ... 1000 clients tried to download the same file at the same time, their downloads should all make forward progress. In contrast, your original solution in Project 1 (assuming you did not provide any parallel code) would result in the first client downloading the entire file, followed by the second client downloading the entire file, etc...

There are a variety of methods to support concurrent requests in your server. Either of the following methods is acceptable:

1. Use **multiple processes**, each process handling a single active connection. Because processes are "heavy" and take a non-trivial amount of time to launch, your solution should launch a "pool" of processes during initialization, and re-use processes in the pool for each active socket.
2. Use **multiple threads** (in a single process), where each thread handles a single active connection. Because threads are "lightweight", you can launch and kill threads for each active socket.
   1. *Python note: In this interpreted language, all threads compete for the single global interpreter lock (GIL). The effect is that only 1 thread can be running Python code at a time, and thus the performance benefits of threads in Python are often minimal or non-existent. A straight C implementation using threads would not have this bottleneck. (See: Python Threads and the GIL, Understanding the GIL)*

In addition, the following features are also required of your web server:

- **Persistent connections**: In HTTP/1.1, a web server should leave a client connection open after serving a request. The client has the option to send additional HTTP requests over the already-open socket. This reduces the latency of each request, because a new TCP connection does not have to be established. (See: Wikipedia entry) The client socket should stay open for 30 seconds, or a similar, finite, length of time.
  - Note: The `Content-Length` header (see below) is critical to the correct operation of persistent connections!
  - Note: The `Connection: Close` header should not be sent when using persistent connections! (If set, the browser might close the socket and never send another request on this connection)
  - Note: In Firefox, for the testing of **Checkpoint 1 **only****, you can restrict the browser to only use 1 persistent connection, instead of trying to open multiple parallel connections to the server and failing when all but one of those connections stalls out. To set, go to "about:config" (as a URL), then ignore the warning, search for `network.http.max-persistent-connections-per-server` and change the value from 6 to 1. **Be sure to set it back before Checkpoint 2!**
- **User-configuration recv() size**: It is a "best practice" to choose a recv() size that is a power of 2 and is reasonably large, but not excessively large. 64kB (64*1024) is a good default choice that is large enough to accommodate incoming data from the network. However, for testing purposes, your program should be run with *small* recv() sizes (e.g. 100 bytes), to ensure that it is properly looping and calling recv() until all data is obtained. Implement a new command line argument, `--recv=????`, that will allow the instructor to vary this parameter as desired. If the command line argument is not set, default to a reasonable recv() size.
- **Graceful shutdown**: Rather than abruptly terminating when the server administrator does a CTRL-C, your web server should capture the CTRL-C signal and do a graceful shutdown. All active sockets should be allowed to finish their current request before being closed by the server.
  - Tip: Get stuck in a loop while testing your signal handler? Use `CTRL-Z` to "background" your web server, then do a `ps` to get the process ID (PID) of the server, and then do a `kill -9 <PID>` to kill the server abruptly.
- **HEAD requests**: The HTTP HEAD method produces a response identical to HTTP GET, but without the response body (i.e. file data). This is frequently used by browsers to check on file creation/modification dates, and thereby determine if their local cached item is current, or if a newer version should be fetched from the server.

- **Headers**: The following response headers must be produced by your web server:
    - Date
    - Server
    - Content-Length
    - Content-Type (the python mimetypes module should be able to guess a reasonable MIME type based on file extension)
    - Last-Modified
    - Expires (set an expiration time of 12 hours in the future)
- **Verbose / Silent Modes**: The default behavior of your web server should be **silent** under normal operation. Add a `--verbose` command-line option to enable debugging output to be printed to the console. It is up to you how much debugging output to produce beyond a minimum of 1 line per URL request. (Real servers often allow you to vary the amount of debugging output produced, from minimal to extreme).

Restrictions

The same restrictions as specified in Project 1 apply here.

Functionality Testing

The same test strategy as specified in Project 1 applies here.

In addition to the test website, you should also employ additional testing methods:

- Place a large file in your web server. While the file is downloading via a web browser, press CTRL-C in the server. Does the download complete before the server gracefully terminates?
- Place a large file in your web server. Download the file in parallel using several different web browsers. Do all downloads appear to make forward progress concurrently?

WARNING: I will **definitely** test your web browser with a large, multi-GB file in order to easily view concurrent downloads. If your web server does something sloppy, like calling `f.read()` *once* to read the entire contents of the file into an array, points will be deducted if Python crashes on my machine. (Maybe I'll even lower the amount of memory in my virtual machine to something small, like 512MB, so there's no way the file could fit into an array at once...)

Performance Testing

After writing a web server that accepts parallel requests, you should **benchmark** its performance, and compare against the original web server in Project 1. Ideally, we want to produce a table or graph that shows the number of pages served by your web server per second as the number of concurrent clients varies from 1 to infinity, or at least until the web server begins to suffer under heavy load.

There are **many** web server benchmarking tools. How do we choose a suitable tool? They fall into two main categories:

- "Classic" designs: These tools (including FunkLoad, ab aka "Apache Bench", Siege, JMeter, and httperf) are flexible and generate a wide range of measurements. However, if poorly configured, they may be slower than the web server they are attempting to profile!
- "Modern" designs: These tools (including weighttp and wrk) use the same parallel, event-driven code style employed by the most sophisticated web servers. Although they produce only simple results (simple == streamlined), they can create a blizzard of requests sufficient to saturate most web servers.

For this project, I don't expect our Python-based web server to be particularly fast. Thus, a classic tool should be sufficient. For this project, we will be using **Siege** for its simplicity and support of HTTP/1.1.

Siege *is a multi-threaded http load testing and benchmarking utility. It was designed to let web developers measure the performance of their code under duress. It allows one to hit a web server with a configurable number of concurrent simulated users. Those users place the webserver "under siege." Performance measures include elapsed time, total data transferred, server response time, its transaction rate, its throughput, its concurrency and the number of times it returned OK.*
From: https://www.joedog.org/siege-manual/

To get started, install Siege on the performance measurement computer.

```
sudo apt-get install siege
```

Then, run your web server in **SILENT mode** (no debugging output on the screen to slow things down!)

Verify that your computer is otherwise idle before running the test.

Run 1 simulated user at a time for 60 seconds to ensure basic functionality. Each simulated user will load the page `index.html` on your server, and then immediately exit and repeat the request.

```
siege --concurrent=1 --benchmark --time=60s http://localhost:8080/index.html
```

Now, increase the number of concurrent users, and complete the following table for 60 second tests. (Complete the table up to the point that you have excessive errors and the benchmark test no longer runs)

Number of concurrent users:	**1**	**8**	**64**	**128**	**256**	**512**

(60 second test)						
Response time						
Transaction rate						
Throughput						
Concurrency						
Successful transactions						
Failed transactions						
Longest transaction						
Shortest transaction						

Be sure to do the test suite twice - once for your original web server, and once for your new web server.

Debugging Note:

- Siege sends a "Connection: Close" header in its HTTP request, indicating that your web server should not use persistent connections, and instead should close the socket after sending a single reply. Depending on how you have coded your web server and tested for client socket closing, you may need to explicitly handle this header in order for Siege to run and download more than zero files from your web server.

Tip: If you really want to get good measurements, this **blog post** has helpful advice on how to setup your experiments in order to avoid gathering misleading results. Useful tips include:

1. Verify that the web server is configured for SILENT mode (no debugging output on screen!)
2. Do not run the performance tester on same machine as your web server, as they will compete for resources. Instead, use two different computers.
3. Verify that the network connecting the web server machine and performance tester machine is not congested before running the test. (Translation: Do NOT run this test over Wi-Fi! Instead, run it over two adjacent computers plugged into the same, preferably gigabit, Ethernet switch).
4. Verify that the web server machine and performance tester machine are otherwise idle before running the test

But, it is not required for you to follow this testing methodology in Project 2.

Resources

See the main resource page for links that helped me when developing my solution.

Checkpoints

This project has two weekly checkpoints due, in addition to the final project deadline. **Checkpoints give the instructor an opportunity to review your in-progress work, and, if problems are found, provide helpful feedback in advance of the project deadline.** Checkpoints are graded as full credit, half credit, or no credit.

- **Checkpoint 1:**
    - Server supports **persistent connections**, where a client can use one socket to sequentially request multiple files.
    - Server generates all the required HTTP response headers (Content-Length is necessary for persistent connections).
    - Server supports command line argument to vary recv() size
- **Checkpoint 2:**
    - Server supports silent and verbose operation modes
    - Server supports **parallelism** (thread or process based), where a client can use multiple sockets to request multiple files in parallel.

Submission

There are slight differences between Python 3.x versions (3.3, 3.4, and 3.5). To ensure I use the same version of Python while grading that you did during development, include the following version-checking code during your program's initialization.
Note: Replace "3,4" with the version number of Python that you used.

```
import sys
if not sys.version_info[:2] == (3,4):
 print("Error: need Python 3.4 to run program")
 sys.exit(1)
else:
 print("Using Python 3.4 to run program")
```

Your submitted project must include:

- Web server code
- PDF table of Siege performance test results for both original web server *and* parallel web server

If your Python program is just a single `.py` file, simply upload it to Canvas directly, along with the other requested files.

Otherwise, if your program contains multiple source code files, create a .tar.gz compressed archive and upload that. To create the archive, assuming your files are in the folder "project2", run:

```
$ tar -cvzf project2.tar.gz project2
```

Once created, upload this archive file to the corresponding Canvas assignment and submit. To extract your archive, I will run:

```
$ tar -xvf project2.tar.gz
```