Computer Networks & Distributed Processing (Spring 2014)
Project 2: Concurrent Web Proxy

Due: Mon June 16 @ 11:59pm

You will implement a concurrent web proxy server, a program that we all know too well which acts as a broker between an actual web server and a browser. The goal of this project is *not* building a feature-complete proxy server. Our goal is more modest: be able to browse standard websites on your browser via this proxy server. However, we'd like your proxy to be a download accelerator, which helps speed up downloading big files. This is often done by creating multiple simultaneous connections of different file segments. Your proxy server will cache small and moderate-sized web objects.

The end product should be callable via

```
./cproxy -p<port> [-c<cache size in bytes>] [-s<max obj size to cache in bytes>]
```

At minimal, you should be able to browse standard websites with the proxy after setting your browser's HTTP proxy setting to point to the machine's IP and this port.

1 Logistics

- For this project, you can work individually or in a team of two.[1] You can discuss ideas/design in general terms with other people taking the class, not just your project partner. You may help debug your friends code—though, this must not mean sharing code with another team.

  You will be interviewed following turning the project in.

- This project is to be implemented in Python, Go, C/C++. There will be a demo day, and there will likely be a contest (details to follow), so your choice of language might determine how fast/scalable your code will be.

- Use of a version-control system (VCS) such as Git, Mercurial, SVN, etc. is strongly encouraged. If you need a place to sync your repository, BitBucket (`www.bitbucket.org`) is a reasonable option, which is currently free for a small team.

- You'll hand in a tar ball or a zip file named `p2.{tar|tgz|zip}`. When you're ready but before the deadline, upload this file to Canvas under the assignment section.

  Be sure to include a README file with instructions on how to operate your program. If you're using Go, include in the README how to build your binary.

  Importantly, this README file will describe and document your design (if someone wants to pick up your program, s/he should be able to understand how it works by reading this and the comments). Did I mention, comment your code?

[1] Unless you really like working by yourself, it's generally good to have a project partner.

2 Assumptions and Requirements

- You only have to handle HTTP (not HTTPS nor other protocols). More precisely, it should work correctly with basic GET and PUT.
- Your proxy will start streaming the data to the user, without waiting until the whole file has been downloaded. When operating in the download-accelerator mode, your proxy will start serving data as soon as that particular chunk becomes available—use chunk transfer encoding if you wish.

  Tip: Is there any implication to your caching implementation if you don't know the object size a priori?
- Only perform download acceleration on objects larger than what's specified by `-s` and when the length is known.
- You don't have to handle requests for persistent connection; however, feel free to take advantage of it in your download accelerator if you want to.
- A proxy server shouldn't crash because the target servers/clients close your connections prematurely. *A crashing proxy server is believed to be a bad omen.*
- You should be able to serve the following pages/objects through your proxy: `www.nytimes.com`, `http://www.w3.org/Protocols/rfc2616/rfc2616.html`, and friendly servers providing Linux ISO files.
- Caching (see below) is required, but you will not serve stale contents. Your code will be tested if you cache objects.
- **Between correctness and speed, choose correctness before speed:** Your code must first be correct and then it's optionally—though preferably—fast.

3 Technical Notes & Tips

- It helps to start out small but keep in mind the big picture. In this case, you might want to begin with a sequential proxy server—and gradually make it concurrent. Think *from the beginning*, "if the sequential server looks like this, what would it take to make it concurrent?" as this will shape how you implement the sequential version. Don't forget that you'll implement object caching.
- There are over 160 occurrences of the keyword "proxy" in RFC2616. The RFC specifies precisely the behaviors. To start off, a GET request to your proxy server looks like this:

  ```
  GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1
  ```

  (There will be additional things in the header, too.)
- When the proxy receives a connection request from a client (typically a web browser), the proxy should accept the connection, read the request, verify that it is a valid HTTP request, and parse it to determine the server that the request was meant for. Then, it will open a connection to that server, send it the request, receive the reply, and forward the reply to the browser.
- Some random thoughts on the Internet about Web Proxy Servers: `https://www.mnot.net/blog/2011/07/11/what_proxies_must_do`
- Some of what you wrote for Project 1 can be reused here.

- For the download accelerator, you should reuse ideas you have seen before to manage how many connections to open at a time.
- **Testing:** You should test your code thoroughly, performing both end-to-end tests (e.g., use it to download a file to see if you get what you expected) and unit testing of parts. It's expected that your program will be reasonably robust.
- **Thread Safety:** Be very careful when accessing shared variables from multiple threads. Make sure you have enumerated race conditions: for example, while one thread is utilizing a cache entry object, another thread might be trying to remove this entry. At the same time, only perform locking as necessary because it can degrade performance significantly.

4 Caching Web Objects

Your proxy server will will cache recently-accessed objects in main memory. HTTP defines what proxy servers should do when caching objects—follow the rule. The basic idea is that your server will remember objects downloaded before and to re-serve them, make sure they aren't stale.

Memory Limit. If your proxy stored every object that was ever requested, it would require an unlimited amount of memory. To avoid this, we will establish a maximum cache size specified when the program is invoked (see the command-line parameter `-c`).

You should use a simple least-recently-used (LRU) cache replacement policy when deciding which objects to evict. One way to achieve this is to mark each cached object with a timestamp every time it is used. When you need to evict one, choose the one with the oldest timestamp. Note that reads and writes of a cached object both count as "using" it.

Object Size Limit. Some web objects are much larger than others. To avoid deleting all the objects in your cache to store one giant one, we will set a maximum object size as determined by the command-line parameter `-s`.

5 Resources

In addition to RFCs, these are some websites that contain further information about concurrent programming in Python and Go.

- Thread synchronization: `http://effbot.org/zone/thread-synchronization.htm`
- A rather comprehensive set of slides on Python Concurrency: `http://www.slideshare.net/dabeaz/an-introduction-to-python-concurrency`
- Async I/O: `http://effbot.org/librarybook/asyncore.htm`
- Go's Concurrency: `http://www.golang-book.com/10/index.htm`

If you're switching to Go, it helps to know that sockets are pretty much the same way you've seen except they have different function names (oh, well—that's what happens when you're picking up a different, though similar, framework). See, for example, this site for ideas: `http://jan.newmarch.name/go/socket/chapter-socket.html`. I'm also making our familiar upper-caser server available on pastebin: `http://pastebin.com/V2kTdZ4a`