Assignment 3
CSE 130: Principles of Computer System Design, Fall 2021

Due: Friday, December 3 at 11:59PM

Goals

The goal for Assignment 3 is to create an HTTP reverse proxy with load balancing and caching. It will distribute connections over a set of servers similar[1] to the one that you implemented in the previous assignment, while ignoring missing servers and caching results for fast response when applicable. It will still need concurrency to be able to service multiple requests at the same time. It will use the health check implemented in assignment 2 to keep track of the performance of these servers and decide which one will receive the incoming connection. This will require that your program is able to act as a server to the incoming connections, while acting as a client of the existing servers.

We will provide you with a working copy of httpserver with some modified features that you can use to test your proxy. The starter code will also be augmented to show how to start a connection as a client. Your main goal is to implement the reverse proxy, which will support GET requests, persistent connections, load balancing and caching.

As usual, you must have a design document and writeup along with your `README.md` in your `git` repository. Your code only needs to build `httpproxy` using `make`.

Programming assignment: creating an HTTP proxy

You are going to create an HTTP proxy that can distribute connections over a set of servers, while checking that they are still operational, and can cache resources to reduce requests to the servers. You will only need to handle GET requests in this assignment, your proxy must reply with response error 501 to any other types of requests.

Design document

Before writing code for this assignment, as with every other assignment, you must write up a design document. Your design document must be called `DESIGN.pdf`, and must be in PDF. You can easily convert other document formats, including plain text, to PDF.

Your design should describe the design of your code in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, non-trivial algorithms and formulas, and a description of each function with its purpose, inputs, outputs, and assumptions it makes about inputs or outputs.

Write your design document *before* you start writing code. It'll make writing code a lot easier. Also, if you want help with your code, the first thing we're going to ask for is your design document. We're happy to help you with the design, but we can't debug code without a design any more than you can.

You **should** commit your design document *before* you commit the code specified by the design document. You're encouraged to do the design in pieces (*e.g.*, overall design and detailed design of HTTP handling functions but not of the full server), leaving the code for the parts that aren't well-specified for later, after designing them. We **expect** you to commit multiple versions of the design document; your commit should specify *why* you changed the design document if you do this (*e.g.*, "original approach had flaw X", "detailed design for module Y", etc.). We want you to get in the habit of designing components before you build them.

Since a lot of the logic in Assignment 3 is similar to Assignment 2, we expect you are going to "copy" a good part of your design from your Assignment 2 design. This is fine, as long as it's *your* Assignment 2 you are copying from, and will let you focus on the new stuff in Assignment 3. We expect the design document to contain discussions of *why* your system is thread-safe or how else it implements concurrency. Which variables are shared between threads? When are they modified? Where are the critical regions? If not using threads, how do you implement concurrency? These are some of the things we want to see. You should also discuss your design choices for the assignment of connections to servers and detection of problems in the servers.

[1] it has one extra header in the response

Program functionality

You may not use standard libraries for HTTP; you have to implement this yourself. You have already been provided with a sample starter code that contains the networking system calls you need to create a server-side socket, which you used to build Assignment 1. You will also be provided with the code necessary to start a client-side socket. You may use standard file system calls, but not any `FILE *` calls except for printing to the screen (*e.g.*, error messages). Note that string functions like `sprintf()` and `sscanf()` aren't `FILE *` calls.

Your HTTP proxy must be able to work with an HTTP server that follows all requirements from Assignment 2. Your program must be able to handle multiple clients at the same time concurrently, however it does not need to use the same dispatcher-worker model as in the previous assignment. It will have to receive connections from clients, similar to how your HTTP server operates, and create connections to the HTTP servers, acting as a client this time, except when it detects an error or has a cached response (described in a later section of this assignment). You will be provided with a function which you can use to open a connection as a client and will return a socket that you can use in the same way that you have used server sockets. It should also be able to connect to a server and request the `healthcheck` resource. The proxy will need to process the request from the client before forwarding it to the server or responding from cache, and it will need to process the response from the server to forward it to the client properly. It will also need to send healthcheck requests to the server and process the response.

You can assume the HTTP client will always send requests that can be processed directly by the HTTP server and you can assume there will be no timeouts. This does not mean that it will only receive valid requests, only that they can be processed. If your proxy finds a bad request it must respond with error code 400 and end the connection. Likewise, since your proxy will only forward GET requests, anything else will result in 501 error responses.

You cannot assume that an HTTP server will always be available, but if available it will be responsive. If not available you will not be able to connect to it. Part of your program implementation should deal with identifying whether the servers being balanced are operational, which is described in the Load Balancing section later in this document.

We expect that you will reuse code from your previous assignments, but you **must not** add your `httpserver` or the provided one to your git repository. Remember also that your code for this assignment must be developed in `asgn3`.

Your HTTP proxy will take as parameters one port number that will be used to receive connections from the HTTP clients, followed by the port numbers for the servers. At least one server port number is required. It can also take one optional parameter `-N` defining the number N of parallel connections it can service at the same time, with a default value of $N = 5$ and another optional parameter `-R` defining how often (in terms of requests) the proxy should retrieve the health check from the servers. That is, the proxy should request a health check of all servers when it starts and every R requests after that. The design of how your proxy handles the update of server conditions while balancing requests from clients must be included in your design document. By default the proxy should inspect the servers after every $R = 5$ requests. Both N and R must be positive integers.

There are also two more parameters which are used to configure the cache: `-s`, a non-negative integer that specifies the capacity of the cache, and `-m`, the maximum file size in bytes to be stored. The default values for `-s` and `-m` will be 3 and 1024. Any non-negative integer is a valid value for these parameters, a value of zero (0) on either of them indicates that there will be no caching. Your proxy can also support an optional flag `-u` to modify the cache behavior for extra credit, this will be further described in the caching section of this assignment.

Below are some examples of how to execute the proxy. Notice how the optional parameters can appear in any position and in any order, but the port number of the proxy is always the first mandatory argument, and there must be at least one other port number for it to work. If any of the parameters is an invalid value the program must exit.

- `./httpproxy 1234 8080 8081 8743`

  This starts the proxy on port 1234 to balance load between three servers, on ports 8080, 8081, and 8743. The proxy will be able to handle 5 parallel connections, will run the healthcheck on the servers every 5 requests and will cache up to 3 transferred files with size lower or equal to 1024 bytes.

- `./httpproxy -N 7 1234 8080 -R 3 8081`

  This starts the proxy on port 1234 to balance load between two servers, on ports 8080, and 8081. The proxy will be able to handle 7 parallel connections, will run the healthcheck on the servers every 3 requests and will cache up to 3 transferred files with size lower or equal to 1024 bytes.

- `./httpproxy 1234 -N 8 8081 -s 5`

  This starts the proxy on port 1234 to proxy for one server on port 8081. The proxy will be able to handle 8 parallel connections, will run the healthcheck on the servers every 5 requests and will cache up to five transferred files with size lower or equal to 1024 bytes.

- `./httpproxy 1234 9009 -R 12 7890`

  This starts the proxy on port 1234 to balance load between two servers, on ports 9009, and 7890. The proxy will be able to handle 5 parallel connections, will run the healthcheck on the servers every 12 requests and will cache up to 3 transferred files with size lower or equal to 1024 bytes.

- `./httpproxy 1234 1235 -R 1 1236 -N 11 -m 2048`

  This starts the proxy on port 1234 to balance load between two servers, on ports 1235, and 1236. The proxy will be able to handle 11 parallel connections, will run the healthcheck on the servers every request and will cache up to 3 transferred files with size lower or equal to 2048 bytes.

- `./httpproxy 9090 8080 -s 4`

  This starts the proxy on port 9090 to proxy for one server on port 8080. The proxy will be able to handle 5 parallel connections, will run the healthcheck on the servers every 5 requests and will cache up to 4 transferred files with size lower or equal to 1024 bytes.

- `./httpproxy 8383 -s 1 -m 10000000 -R 3 7384 3456`

  This starts the proxy on port 8383 to balance load between two servers, on ports 7384, and 3456. The proxy will be able to handle 5 parallel connections, will run the healthcheck on the servers every 3 requests and will cache up to 1 transferred file with size lower or equal to 10000000 bytes.

Connection Forwarding

In order to act as a reverse proxy, your program must transparently forward the requests received from the clients to the servers, and similarly receive the responses from the servers and forward them to the clients. Transparently means that the client will not be aware that it is communicating with a proxy. For this assignment the proxy will have to process the received request and identify the type of request and the resource name. If it identifies an error at this stage, it will reply to the client directly with the proper error response. If the request is a GET and the resource is in the cache, it can respond with the cached content (this will be further described later). Otherwise, it can forward the request with no modifications to the appropriate server (selecting the appropriate server will be described later).

Keep in mind that the client might send further requests after receiving a response from the server, so the proxy will need to process the response in order to identify that it is complete, so it can check the client for more requests. You must send all data received from the client to the server, and all data received from the server to the client, until one of them closes the connection. Then the proxy must close the connection to the other side of the exchange. That is, if the client closed the connection to the proxy, the proxy will close the connection to the server, and if the server closed the connection to the proxy, the proxy will close the connection to the client.

You can also assume that all servers will be running with access to the same files. For testing we will turn servers off and back on.

Load Balancing

The purpose of using a load balancer is to distribute the requests received across multiple servers. In order to do so your program will probe the monitored servers for their `healthcheck` resource at start of operation and after every R requests received.

The load balancer should prioritize the server that has received the least amount of requests so far, and use the success rate as a tie-breaker. If more than one server would still qualify, any can be chosen to receive the request. If a server fails to respond to a health check request or a request from a client it should be marked as problematic and excluded from balancing until it returns a successful response during a health check probe. For reference, a health check probe can fail either with the server not connecting, if the response code is not 200, or if the body does not fit the `<errors>\n<entries>\n` format.

You can and should use internal variables to estimate server load between health check probes, however your design **must not** ignore the health check probes as the servers might receive requests external to the proxy. Tracking the internal state of the servers, that is, how many requests each server has received, and how many resulted in errors, is important so you can balance requests between healthcheck queries. Again, you can also assume that all servers will be running with access to the same files, so the only factors that are relevant to selecting a server to receive a request are the number of already received requests and how many of these resulted in errors. The contents of the messages exchanged between external clients and the HTTP servers **must not** be a factor in the load balancing.

Caching

When the proxy receives a GET request for a resource that is cached, it must verify that the cached copy is not obsolete by sending a HEAD request to the appropriate server and checking the Last-Modified header line, whose value is a date/time for when the requested object was last modified. **The provided HTTP server already implements this header so you can use it directly.** An example of the Last-Modified header line is given below:

```
Last-Modified:  Wed, 21 Oct 2015 07:28:00 GMT
```

The definition of the Last-Modified header line can be found here: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Last-Modified`

Your reverse proxy should compare the last modified date with the age of the stored object. If the stored object is the same age or newer, the proxy can respond directly to the client without forwarding to a server. Otherwise the proxy should forward the request to the appropriate server as usual.

Your proxy has a number of parameters defining how the cache will work. The first parameter, specified by the option s, defines how many items the cache can hold. That is just the number of files that can exist in the cache at once. If another file were to be added to the cache once it reaches the maximum number of files, then the new file should replace one of the existing files in the cache, according to a replacement policy, which will be First-In-First-Out for this assignment. That means that when removing an element from a full cache, the element that has been in the cache for the longest time will be the one to be replaced.

The second parameter, specified by the option m, defines the maximum size of a file to be cached. That is the size in bytes, as present in the Content-Length header line of a response to a GET request. Your proxy should only add to its cache files that are equal in size or smaller than this value. What if the file is larger? In that case the file will not be stored in the cache.

While more realistic implementations of caching can use efficient data structures and hashes to allow quick access to the desired data, this is not a requirement in this assignment and simple data structures will suffice.

You can, however, implement an alternative replacement policy for extra credit. This alternative policy will be Least-Recently-Utilized (LRU), and it will be enabled with the `-u` flag, which will **not** be followed by a parameter. In this policy the element to be replaced in the cache is the one that was accessed the longest ago. To illustrate how this policy differs from the previous one, consider four requests for resources A, B, A and C in this order, when the proxy can hold only two elements in the cache. Under FIFO, A would be replaced because it was added before B. Under

LRU, B would be replaced because A was accessed after B. Again, this policy is optional and will be worth 5 points in extra credit, so that your final grade can go up to 105.

README and Writeup

Your repository must also include a README file (`README.md`) writeup (`WRITEUP.pdf`). The README may be in either plain text or have MarkDown annotations for things like bold, italics, and section headers. **The file must always be called `README.md`**; plain text will look "normal" if considered as a MarkDown document. You can find more information about Markdown at `https://www.markdownguide.org`.

The `README.md` file should be short, and contain any instructions necessary for running your code. You should also list limitations or issues in `README.md`, telling a user if there are any known issues with your code.

Your `WRITEUP.pdf` is where you'll describe the testing you did on your program and answer any short questions the assignment might ask. The testing can be unit testing (testing of individual functions or smaller pieces of the program) or whole-system testing, which involves running your code in particular scenarios.

For Assignment 3, please answer the following questions:

- For this assignment, your proxy distributed load based on number of requests the servers had already serviced, and how many failed. A more realistic implementation would consider performance attributes from the machine running the server. Why was this not used for this assignment?
- Using the provided `httpserver`, do the following:
  - Place eight different large files in a single directory. The files should be around 400 MiB long, adjust according to your computer's capacity.
  - Start two instances of your `httpserver` in that directory with four worker threads for each.
  - Start your `httpproxy` with the port numbers of the two running servers and maximum of eight connections and disable the cache.
  - Start eight *separate* instances of the client *at the same time*, one GETting each of the files and measure (using `time(1)`) how long it takes to get the files. The best way to do this is to write a simple shell script (command file) that starts eight copies of the client program in the background, by using `&` at the end.
- Repeat the same experiment, but turn off one of the servers. Is there any difference in performance? What do you observe?
- Using one of the files that you created for the previous question, start the provided server with only one thread, then do the following:
  - Start your proxy without caching.
  - Request the file ten times. How long does it take?
  - Now stop your proxy and start again, this time with caching configured so that it can store the selected file.
  - Request the same file ten times again. How long does it take now/
- What did you learn about system design from this class? In particular, describe how each of the basic techniques (abstraction, layering, hierarchy, and modularity) helped you by simplifying your design, making it more efficient, or making it easier to design.

Submitting your assignment

All of your files for Assignment 3 must be in the `asgn3` directory in your `git` repository. When you push your repository to GITLAB@UCSC, the server will run a program to check the following:

- There are no "bad" files in the `asgn3` directory (*i.e.*, object files).
- Your assignment builds in `asgn3` using `make` to produce `httpproxy`.
- All required files (`DESIGN.pdf`, `README.md`, `WRITEUP.pdf`) are present in `asgn3`.

If the repository meets these minimum requirements for Assignment 3, there will be a green check next to your commit ID in the GITLAB@UCSC Web GUI. If it doesn't, there will be a red X. **It's OK to commit and push a repository that doesn't meet minimum requirements for grading.** However, we will only *grade* a commit that meets these minimum requirements.

Note that the *minimum* requirements say nothing about correct functionality—the green check only means that the system successfully ran `make` and that all of the required documents were present, with the correct names. **You must submit the commit ID you want us to grade via Google Form (`https://forms.gle/eR9g3yW82t4FXt3z8`). This must be done before the assignment deadline.**

Hints

- Start early on the design. This program builds on top of the techniques you used on the previous assignments, but is different.
- Reuse your code from previous assignments. No need to cite this; we expect you to do so. But you must do your work in the `asgn3` directory.
- Go to section for additional help with the program. This is especially the case if you don't understand something in this assignment!
- You'll need to use (at least) the system calls from previous assignments.
- Work through the design for the HTTP proxy *carefully* before you start to implement it. Make sure you've answered any questions you might have in advance.
- Your proxy will be tested with the same `httpserver` that we will provide to you.

Grading

As with all of the assignments in this class, we will be grading you on *all* of the material you turn in, with the distribution of points as follows: design document (20%); functionality (70%); writeup (10%).

If you submit a commit ID that cannot pass the minimum test in the pipeline or modify `.gitlab-ci.yml` in any way, your maximum grade is 5%. Make sure you submit a commit ID that can obtain a green checkmark in the minimum test.

