Assignment 3
CSE 130: Principles of Computer System Design, Spring 2020

Due: Friday, June 6 at 9:00PM

Goals

The goal for Assignment 3 is to create a load balancer that will distribute connections over a set of servers such as the one that you implemented in the previous assignment, while ignoring non-responding servers. It will still need concurrency to be able to service multiple requests at the same time. It will use the health check implemented in that assignment 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.

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 `loadbalancer` using `make`.

Programming assignment: creating a load balancer

You are going to create a load balancer that can distribute connections over a set of servers, while checking that they are still operational.

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.

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 load balancer must be able to work with a 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. It should also be able to connect to a server and request the `healthcheck` resource. It should not need to do any HTTP processing when forwarding connections to a server, but it will need some minimal processing to be able to request a health check and retrieve the number of entries and errors in the logging file.

You can assume the HTTP client will always send requests that can be processed directly by the HTTP server. You cannot assume that the HTTP server will always be responsive. Part of your program implementation should deal with identifying whether the servers being balanced are still operational.

We expect that you will reuse code from your previous assignments, but you do not need to add the `httpserver` to your submission. Remember also that your code for this assignment must be developed in `asgn3`.

Your load balancer 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 $N = 4$ and another optional parameter `-R` defining how often (in terms of requests) the load balancer should retrieve the health check from the servers. That is, the load balancer should request a health check of all servers when it starts and every R requests after that or every X seconds, whichever comes first. By default the load balancer should inspect the servers after every $R = 5$ requests. X will be defined by your as part of your design process, see the section on Load Balancing.

Those are some examples of how to execute the load balancer, in all examples the load balancer will be receiving connections on port 1234. Notice how the optional parameters can appear in any position and in any order, but the port number of the load balancer is always the first mandatory argument. If any of the parameters is not a positive integer the program must exit.

- ./loadbalancer 1234 8080 8081 8743
- ./loadbalancer -N 7 1234 8080 -R 3 8081
- ./loadbalancer 1234 -N 8 8081
- ./loadbalancer 1234 9009 -R 12 7890
- ./loadbalancer 1234 1235 -R 2 1236 -N 11

Connection Forwarding

In order to act as a load balancer, 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. For this assignment the load balancer does not need to process the content received from the client and can redirect it to a server straightaway. It will need, however, to ensure that the server sends a response before a finite amount of time, or send a 500 error response to the client instead. You can have a default 500 response string ready to be sent in that case.

How long the load balancer should wait before sending the error response is a design decision and should be explained in your design document. It must, however, be no longer than five (5) seconds due to the timed nature of our testing. You can also assume that if the server has started responding then the load balancer does not need anymore to send a error response and can just forward everything received from the server to the client.

You must not assume that the client will send all the data then wait. The client might send some data, wait for something from the server, then send more data. 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 load balancer must close the connection to the other side of the exchange. You can use `select(2)` to identify which sockets have data to read from, or you can use non-blocking sockets.

You can also assume that all servers will be running with access to the same files and no requests will be made that would cause contention between servers. For testing we will turn servers off and back on. Make sure your server won't have bind problems in that case.

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 every X amount of time or after every R requests received, whichever happens first. How long X is a design decision, but R will be defined by parameter and used for testing purposes. If X is too short, your load balancer might send too many health checks, but if it is too large then it might miss on server failures. Again, X must not be longer than five (5) seconds due to the timed nature of our testing. X can be the same as the server timeout, but does not need to be the same if that helps your design. Explain your design decisions in your design document.

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 positive response during a health check probe. For reference, a health check probe can fail either with the server not responding, if the response code is not 200, or if the body does not fit the `<errors>\n<entries>` format. Remember that although you have to parse the health check, your load balancer does not need to parse the server's response in normal operation.

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 load balancer. Again, you can also assume that all servers will be running with access to the same files and no requests will be made that would cause contention between servers. The contents of the messages exchanged between external clients and the HTTP servers must not be a factor in the load balancing.

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 load balancer 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?
- This load balancer does no processing of the client request. What improvements could you achieve by removing that restriction? What would be cost of those improvements?
- Using your `httpserver` from Assignment 2, do the following:

  - Place eight different large files in a single directory. The files should be around 400 MiB long.
  - Start two instances of your `httpserver` in that directory with four worker threads for each.
  - Start your `loadbalancer` with the port numbers of the two running servers and maximum of eight connections.
  - 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 substitute one of the instances of `httpserver` for `nc`, which will not respond to requests but will accept connections. Is there any difference in performance? What do you observe?

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 `loadbalancer`.
- 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/x1VhrdLXXsg7HLy68`). 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. You can also use `select` or `pselect` to detect time outs from the HTTP servers, or you can use non-blocking sockets.
- Work through the design for the load balancer *carefully* before you start to implement it. Make sure you've answered any questions you might have in advance.
- Your load balancer will be tested with a HTTP server that follows the instructions from Assignment 2 but might have issues. You should use your `httpserver` for testing, but do not assume that your load balancer will be graded with it. You can also use programs such as `nc` to simulate a non-responsive HTTP server.

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.

