# Foreword

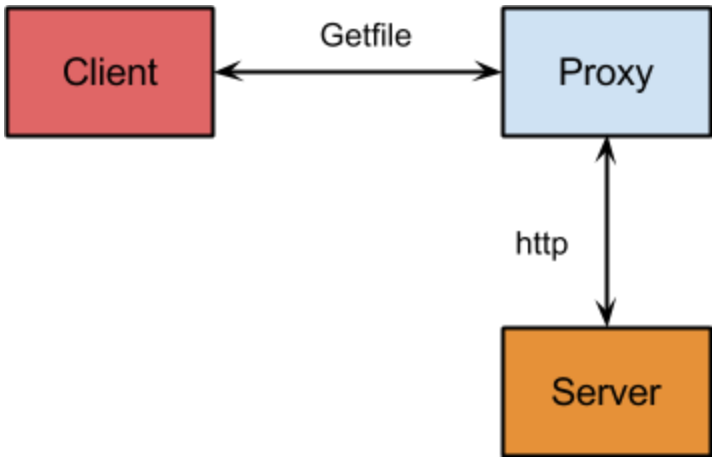
This project has two parts, plus an extra-credit portion. In the first part, you will convert an implementation of a getfile server into a getfile proxy that converts incoming Getfile requests into http requests for a server on the internet. In the second part, you will implement a simple cache that communicates with the proxy via shared memory. For extra credit, you may integrate the code you wrote from the first two parts with your own code from Project so that you have an end-to-end implementation of a getfile proxy with cache built from your own source code.

# Directions

1. Download the starter code in the tarball ud923-project3.tar.gz from the "Downloadables" section of one of project's morsels the on the Udacity site.
2. Run 'tar -zxf ud923-project3.tar.gz' to untar the file.
3. Begin programming by **modifying only those specified below.**
4. Turn in your code by copying and pasting onto the Udacity programming quizzes.

# Part I

To convert the getfile server into a proxy, you only need to replace the part of the code that retrieves the file from disc with code that retrieves it from the web. Using the gfserver library provided in the starter code, this is as easy as registering a callback. Documentation can be found in the gfserver.h file. To implement the callback you should use the libcurl's "easy" C interface.




Your proxy server must use a boss-worker multi-threading pattern, allowing it to serve multiple connections at once, and it must support the command line arguments listed below.

---

```
usage:
      webproxy [options]
```

```
options:
      -p port for incoming requests
      -t number of worker threads
      -s server address(e.g. "localhost:8080", or
      "http://example.com")
```

---

Note that you don't have to write your own http server. Workloads are provided for files that live on an amazon S3 instance http://s3.amazonaws.com/content.udacity-data.com. Concatenate this with one of the paths found in workload.txt to create a valid url.

Here is a summary of the relevant files and their roles.

- **Makefile** - (do not modify) file used to compile the code. Run 'make webproxy' to compile your code.
- **gfserver.h** - (do not modify) header file for the library that interacts with the getfile client.
- **gfserver.o** - (do not modify) object file for the library that interacts with the getfile client.
- **handle_with_curl.c** - (modify) implement the handle_with_curl function here using the libcurl library. On a 404 from the webserver, this function should return a header with a Getfile status of GF_FILE_NOT_FOUND.
- **handle_with_file.c** - (not submitted) illustrates how to use the gfserver library with an example.
- **gfclient_download** - a binary executable that serves as a workload generator for the proxy. It downloads the requested files, using the current directory as a prefix for all paths.
- **gfclient_measure** - a binary executable that serves as a workload generator for the proxy. It measures the performance of the proxy, writing one entry in the metrics file for each chunk of data received.
- **webproxy.c** - (modify) this is the main file for the webproxy program. Only small modifications are necessary.
- **workload.txt** - (not submitted) this file contains a list of files that can be requested of http://s3.amazonaws.com/content.udacity-data.com as part of your tests.

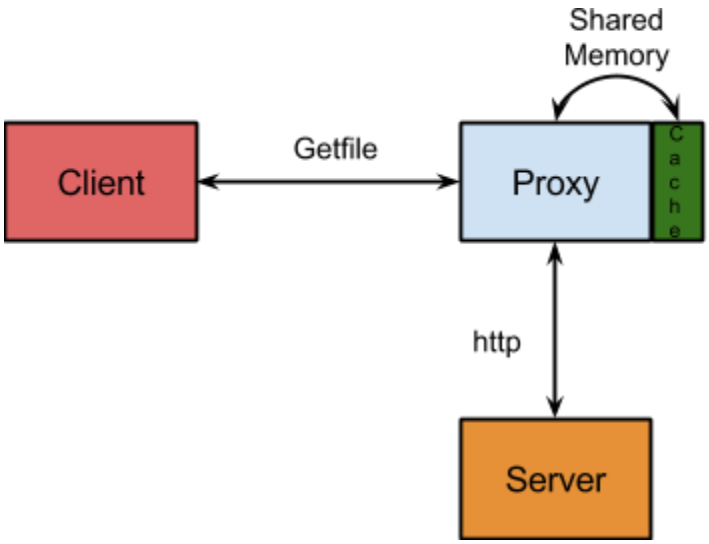
To turn-in this portion of the assignment, you should navigate to

https://www.udacity.com/course/viewer#!/c-ud923/l-2906788540/m-3729408566

and copy/paste your code in and submit. The site will run a "build verification test" that will help catch more obvious mistakes. A grader will review your work after the submission deadline.

# Part II

The objective of this second part of the project is for you to gain experience with shared memory-based IPC. You will implement a cache process that will run on the same machine as the proxy and communicate with it via shared memory.




Because the goal is to give you experience with shared memory IPC and not cache replacement policy or cache implementations, you will access the contents of the cache through an API (simplecache.h) that abstracts away most details of the cache implementation. Your focus will be on relaying the contents of the cache to the proxy process by way of shared memory.

You can use either System V or the POSIX API to implement the shared-memory component and/or other IPC mechanisms (e.g., message queues or semaphores). Please see the list of reference material. If it works on the course's VM, then it will work in the Udacity build verification test.

The command line interface for the proxy should include two new components: the number of segments and the size of the segments to be used for the interprocess communication. Instead of automatically relaying requests to the http server, the proxy should query the cache to see if the file is available in local memory already. If the file is available there, it should send the cached version. In the real world, you would then have the proxy query the server to obtain the file. For the Udacity quiz submission, however, **you need only check the cache**.

In the interprocess communication, **the proxy should be responsible for creating and destroying the shared memory (or message queues) over which the file content is transferred.** This is good practice because in a scenario where the cache is connected to more than one client (or proxy servers in this case) it makes it easier to ensure that

- a non-responsive client doesn't degrade the performance of the cache
- a malicious client doesn't corrupt another's data.

From the cache's perspective this falls under the "don't mess with my memory" adage.

The cache will have to set up some communication mechanism (socket, message queue, shared memory) by which the proxy can communicate its request along with the information about the communication mechanism (shared memory name, id, etc.) For the purposes of this project, this mechanism does not have to be robust to misbehaving clients.

Neither the cache daemon nor the proxy should crash if the other process is not started already. For instance, if the proxy cannot connect to the IPC mechanism over which requests are communicated to the cache, then it might delay for a second and try again.

It is not polite to terminate a process without cleaning up, so proxy process (and perhaps the cache as well depending on your implementation) must use a signal handler for both SIGTERM (signal for kill) and SIGINT (signal for Ctrl-C) to first remove all existing IPC objects -- shared memory segments, semaphores, message queues, or anything else -- and only then terminate.

The command line interface for the proxy process should be as follows.

```
usage:
      webproxy [options]
options:
      -n number of segments to use in communication with cache.
      -z the size (in bytes) of the segments.
      -p port for incoming requests
      -t number of worker threads
      -s server address(e.g. "localhost:8080", or "example.com")
      -h print a help message
```

The command line interface for the cache process should be as follows.

```
usage:
      simplecached [options]
options:
      -t the number of threads running in the process
      -c a filename to be passed to the initialization of the
         simplecache library.
      -h print a help message
```

Depending on your implementation, it may be necessary to hard-code something about how the proxy communicates its requests to the cache (e.g. a port number or a name of shared memory segment). This is acceptable for the purpose of the assignment.

Here is a summary of the relevant files and their roles.

- **cached_files** - (not submitted) a directory containing files that will be stored in the cache. Use in conjunction with locals.txt.
- **gfserver.h** - (do not modify) header file for the library that interacts with the getfile client.
- **gfserver.o** - (do not modify) object file for the library that interacts with the getfile client.
- **handle_with_cache.c** - (modify) implement the handle_with_cache function here. It should use one of the IPC mechanisms discussed to communicate with the simplecached process to obtain the file contents. You may also need to add an initialization function that can be called from webproxy.c
- **handle_with_file.c** - (not submitted) illustrates how to use the gfserver library with an example.
- **locals.txt** - (not submitted) a file telling the simplecache where to look for its contents.
- **Makefile** - (do not modify) file used to compile the code. Run 'make webproxy' to compile your code.
- **shm_channel.[ch]** - (modify) you may use these files to implement whatever protocol for the IPC you decide upon (e.g., use of designated socket-based communication, message queue, or shared memory ).
- **simplecache.[ch]** - (do not modify) these files contain code for a simple, static cache. Use this interface in your simplecached implementation. See the simplecache.h file for further documentation.
- **simplecached.c** - (modify) this is the main file for the cache daemon process, which should receive requests from the proxy and serve up the contents of the cache using the simplecache interface. The process should use a boss-worker multithreaded pattern, where individual worker threads are responsible for handling a single request from the proxy.
- **steque.[ch]** - (do not modify) you may find this steque (stack and queue) data useful in implementing the the cache. Beware this uses malloc and may not be suitable for shared memory.
- **gfclient_download** - a binary executable that serves as a workload generator for the proxy. It downloads the requested files, using the current directory as a prefix for all paths.
- **gfclient_measure** - a binary executable that serves as a workload generator for the proxy. It measures the performance of the proxy, writing one entry in the metrics file for each chunk of data received.
- **webproxy.c** - (modify) this is the main file for the webproxy program. Only small modifications are necessary. In addition to setting up the callbacks for the gfserver

library, you may need to pre-create a pool of shared memory descriptors, a queue of free shared memory descriptors, associated synchronization variables, etc.
- **workload.txt** - (not submitted) this file contains a list of files that can be requested of https://s3.amazonaws.com/content.udacity-data.com as part of your tests.

To turn-in this portion of the assignment, you should navigate to

https://www.udacity.com/course/viewer#!/c-ud923/l-2906788540/m-3738778755

and copy and paste your code in.

# Extra Credit

Integrate the code you wrote from the first two parts into your source code from Project 1 to build a complete getfile-to-http proxy server with cache. Turn into T-square a zip file with all of the source code and a README file that documents your code and how to compile and run it.

# References

### Relevant Lecture Material

- P3L3 Inter-Process Communication
  - SysV Shared Memory API
  - POSIX Shared Memory API

### Sample Source Code

- Signal Handling Code Example
- Libcurl Code Example

# Rubric

### Part I: Sockets (35 points)

### Proxy Server: Sockets

- Correctly send client requests to server via curl
- Correctly send responses back to client using gfserver library

### Part II: Shared Memory (55 points)

### Proxy Server: Shared Memory

- Creation and use of shared memory segments
- Segment ID communication
- Cache request generation
- Cache response processing

- Synchronization
- Proper clean up of shared memory resources

## Cache: Shared Memory

- Segment ID communication
- Proxy request processing
- Proxy response processing
- Synchronization

## Report (10 points)

- Summary of the project design in README.md
- Any observations

## Extra Credit (+10 points)

- Modify Project 1 to integrate the Project 3 components, and submit via T-square