Design Document

Author: Miguel Avalos

Student ID: 1704078

Project Description

httpproxy is a program that creates an http proxy server that receives requests from clients and caches files of the GET requests if they can fit. It will forward HEAD and PUT requests automatically to the main server. For a GET request, the httpproxy will send a response with the cache file if the file exists and is uptodate from the server. If not, the program will forward the GET request to the main server. THe, it will forward the response from the main server to the client and (if the file fits in the cache and the response wasn't an error) will save the updated file on the cache.

Program logic

httpproxy will listen for a connection from a client. Once it receives one:
Processing requests from a connection:

A. A request from the connection will be parsed. The header of the request will be put into a buffer. The command and filename will be extracted into their own separate buffers.
B. If it is a PUT or HEAD request:
    a. The request will automatically be forwarded to the main server. Additionally in a put command, the content length will be extracted to avoid accidentally sending part of another request in a persistent connection. The response received from the main server will not be cached, but only forwarded to the client.
C. If the request was a GET request, the proxy server will try to match the filepath to one of the cached responses.
    a. If no match was found, then the GET request will be forwarded to the main server and the response will be saved to the proxy server's cache and forwarded to the client.
    b. If there is a match, then a HEAD request with the same filepath will be sent to the main server to see if the cached version is up to date.
        i. If the cached version is up to date, then the cached response is sent to the client.
        ii. If the cached version is not to date, then the GET request will be forwarded to the main server. The response will update the cached version (if not an error or file size fits in cache) and be forwarded back to the client.
D. If the connection is closed by the client at any time during this process, the server will not send a response and the connection will be closed from httpproxy's side.
E. If a bad request error message is received from or confirmed by the main server, then the proxy server will close the connection to the client.

F. The httpproxy will use a new connection with httpserver for every request received even in a persistent connection.

Data Structures

A boolean global variable that records what policy will manage the cache.

A character array will be used as a buffer to store data received from a client and data from files that will be sent to the client.

slot struct

- char* file- points to allocated space
- char name[17] - allocates space for file name
- bool empty - will prevent program from reading empty cache space
- int length - describes the length of the data stored in the cache
- struct tm time- (imported from time.h library) stores the last modified time for the stored data

c_manage struct

- slot* slots - points to a slot array that stores cache slots
- int* order - points to a int array used for determining replacement order
- int c_slots - shows the number of available slots
- int file_size - max file size
- int filled - number of non-empty slots

Functions

- int strtoint(char* input)
  will convert a valid string to an integer. input must only contain digit characters. If not, a negative -1 is returned.

- int filerror(int option)
  This function is responsible for determining what error occurred while interacting with a file and returning an integer value that will be used as a code for the response to the client. The caller must call this function using the errno(3) return value as the argument for option.
  - If the program lacks the permissions to access the file, then 403 will be returned.
  - If the file doesn't exist, then 404 will be returned.
  - For any other error, 500 will be returned.

- void gettime(struct tm* t, char* source)
  initializes every field in t and extracts the time from source.

- void err_response(int code, int conn_fd, bool body)
  This function is responsible for sending http responses to the client if an error occurs. If **body** is true, a body will be sent containing a brief description of the error.
  The responses vary by code:
  - 400 - Bad Request
  - 403 - Forbidden

- 404 - Not Found
- 500 - Internal Server Error
- 501 - Not Implemented

- void sortorder(int* arr, int position, int length, int offset)
  It is responsible for offsetting an int array at indexes that have stored values that are greater than position.

- int readline(int conn_fd, char* line)
  It is responsible for reading a header line. Will return -2 if the connection is closed prematurely.

- bool fullR(int source_fd, int* length, char* first, char* multi_buf, struct tm* times)
  Extracts the important components of a http response: stores content-length in length, stores the first line in first, and stores Last-Modified in times. Returns true if successful.

- bool splitter(int source_fd, int dest_fd, char* store, int bytes)
  Receives data from the main server. The data is simultaneously sent to the client and stored in the cache. Returns true if successful.

- bool bridge(int source_fd, int dest_fd, int bytes)
  Forwards data from one network connection to another. Returns true if successful.

- int newentry(int server_fd, int client_fd, char* filename, char* date, int length, c_manage* cache)
  Finds a suitable place to store a new cache entry.
  A. Will store in empty spaces first before following the replacement policy.
     a. In FIFO, it will set order[slot_number] = # of filled slots(including this one)
     b. In LRU, it will set order[slot_number] = 1.
  B. If all slots are full, it uses int* order from c_manage to determine the most suitable slot to overwrite.
     a. For FIFO, order[slot_number] == 1 indicates that its the oldest non-empty slot in the cache and will be set to c_slots.
     b. For LRU, order[slot_number] == c_slots refers to the recently least used resource and will be set to 1.
  C. Once the best slot is found, the file data is simultaneously sent to the client and stored in the cache. The rest of the order positions will be rearranged/offset to fit the new entry, not having any duplicates, and has no value gaps in order. Returns the index in the order and slots array in the cache.

- bool incache(int server_fd, int client_fd, int place, c_manage* cache, char* header)
  Is called if a matching file exists on the server. HEAD request is sent to the main server before this function is called and the resulting header response is passed as one of its arguments. Returns true if the slot was emptied.
  A. If the response contains a 400 error, then a 400 response is sent to the client.

B. If the header contains any other error, then the slot is emptied and the error response is sent to the client.
C. Otherwise, the last modified time stamps are compared between the cache slot and the header.
    a. If the header is up to date, then the data in the cache is sent to the client.
    b. If it is not up to date, then a GET request is sent to the main server.
        i. If the file data fits in the cache, then the entry will be updated and the server response is forwarded to the client.
        ii. If not, the slot will be emptied. The server response is forwarded to the client.

- bool process_request(int conn_fd , int server_fd)
  This function is responsible for parsing the request for the client and calling the necessary function to complete the request if no errors were found with the request. It will log every request result if a log name is provided. A return value of true indicates that the connection needs to be closed. If the connection is closed by the client at any time during this process (including all function calls), the server will not send a response and the connection will be closed.

\

1. Attempt to get the header into a buffer of 4001 bytes (one space is dedicated to '\0' character, receive byte by byte to find a newline character that ends the first line.
2. Let bad be false.
3. Will check if all minimum components are present and valid. If at least one is not so, then set bad to true.
4. If the request is a GET:
    a. respond with a 400 error if bad is true and return true
    b. check the cache to see if a matching response is in it.
        i. If a match is not found:
            1. forward the response to the server
            2. If the response is an error or will not fit in cache, then forward the response to the client and return false
            3. Otherwise, save the file in the cache
        ii. If a match is found:
            1. send a head request to the server to see if the response is still up to date
                a. If the response is up to date, send the cached response to the client. (In LRU, associated order index is set to one)
                b. If the response is outdated, then forward the GET request to the server. If the response was valid and the file could fit in the cache, then update the cached response. Otherwise, empty the cache slot.
5. If the request is PUT request:
    a. if a valid Content-Length is missing, respond with a 400 error and return true

    b. it will be forwarded to the main server.
    c. The response received from the main server will be forwarded to the client.
6. If the request is HEAD request:
    a. it will be forwarded to the main server.
    b. The response received from the main server will be forwarded to the client.
7. If the request is an unimplement request:
    a. respond with a 400 error if bad is true
    b. Otherwise respond with a 501 error
    c. return true
8. Return false to caller if this point is reached

- void handle_connection(int client_fd, int server_fd)
  This is responsible for handling the connection to the client and httpserver.
    1. Will Call process_request to handle the request data sent and open a new server connection unless one the following happens:
        a. (Does not apply for the first request of established connection) The last request has a 400 error
        b. the connection was closed by client
        c. If there is no more data that will be received from the client
    2. Close the connection with the client and the server reverts back to listening for a new connection.

TESTING: Whole-System

- individual responses/connections: (Assignment 1 requesments)
    - testing persistent connections
        - check if PUT file, GET file, and HEAD file requests sent in the same connection in the order do not produce any errors or non-responses.
    - any type of request without errors works normally with any size file
    - bad request on on every implemented command:
        - file path is not 15 characters or not completely alphanumeric
        - Host component does not exist
        - http version is not 1.1
    - unimplemented request
    - unimplemented request that is also a bad request
    - no valid context length in a put request: doesn't exist, value is negative, contains non-digit characters
    - get, head, or put request on a file without the proper permission
    - get or head on a file that does not exist
- cache testing: will look in the log file of the httpserver to verify if the proper requests were sent and have printed debug messages to see what slots are being assigned/replaced. All files requested are on the server unless the scenario tests for the opposite. n >= 3.
    - 0 slots and/or 0 filespace cache: make sure that no file was cached and all get requests are forwarded to httpserver

- Passthrough: head and put requests do not interact with the cache at all.
- Cache storage: In a cache size of n slots, send get requests of n different files. Resend the same n get requests. Verify in the log file that the server has received only a get and a head request for each file. If not, caching has failed. Also verify for all get requests that the files received are identical to the server file.
- Cache rejecting: Does not store files that are too big or the response details an error. Send a request on the file that will result in one of these. Resend the request. check if the server has got only a get request for this request.
- Outdated file: Send a get request for a file. Check that it has been cashed. Modify the file. Resend a get request for that file. check if the server has gotten a head and get requests for this file request.
    - 400 responses will just send the response to the client, but not modify the cached file.
    - no error and able to be stored : Resend a get request again for that file. check if the server has got only a head request.
    - not enough space on cache or other error responses: Resend a get request again for that file. check if the server has got only a get request.
- Testing FIFO order: In a cache size of n slots, send get requests of n+1 different files. Resend the first request and there should be two GET requests for the first file name. Can be repeated with the next file until it cycles.
- Testing LRU order: In a cache size of n, send get requests of n different files. Resend the same requests again except for one request (x) that was in between the first request and the last request. Send a get request for a file that is unique for all the other files. Verify that it was stored on the cache. Resend a get request xth file. The proxy should have sent only a get request for this request.

ASSIGNMENT QUESTIONS:

Using a large file (e.g. 100 MiB — adjust according to your computer's capacity) and the provided HTTP server:

- Start the server with only one thread in the same directory as the large file (so that it can provide it to requests);
- Start your proxy with no cache and request the file ten times. How long does it take?
  It takes 6.883 seconds.
- Now stop your proxy and start again, this time with cache enabled for that file. Request the same file ten times. How long does it take?
  It takes 1.664 seconds.
- Aside from caching, what other uses can you consider for a reverse proxy?
  Handling the response to bad requests or unimplemented requests to only send valid and implemented requests to the main server.