Natalie Petrosian
CSE 130 Principles of Computer Systems Design
Prof. Peter Alvaro

Assignment 2: Design Document

1.0 Introduction

In this assignment, we are modifying the HTTP server that was previously implemented in Assignment 1. We will be adding three additional features: multi-threading, logging and a health check. As a result, the server will be able to handle multiple requests at once, display a log of requests (if prompted), and display information regarding the health of the server (if prompted).

1.1 Goals and objectives

The goals for Assignment 2 are to modify the HTTP server with the three additional features described in section 1.0. Multithreading makes it possible for our server to process requests from multiple clients at once. Logging enables the server to print out a record of each request and its data in the form of hex values. The health check will allow the client to request a resource that will respond with information regarding the failure/success rate of the server. This information provided by the health check is derived from the log. Synchronization techniques will be utilized to ensure the service of multiple requests at once, and avoid the intermixing of log entries from a multitude of threads.

1.2 Statement of scope

The major inputs to the server are the simultaneous processing of PUT, GET, and HEAD requests. The server creates `N` threads when it begins (where the default is `N = 4`), but the argument `-N` tells the server to use the number of threads specified after the `-N` argument. For example, `-N 8` tells the server to use 8 worker threads. Likewise, the program must log each request to the log file if the `-l` option is given as an argument, followed by a file name. Requests must be logged in a manner where an entry is not interrupted by another request entry. To ensure there is no intermixing of logging entries, use `pwrite()`. Alternatively, we can use `dprintf()` along with a logger offset to successfully log requests without overlay. Finally, the performance of the server can be measured through the number of errors and requests generated from performing a GET on a special file named `healthcheck`.

1.3 Major constraints

This program must be written in C. The developer is not permitted to use standard libraries for HTTP or FILE * calls except for printing to the screen. The developer may use sprintf() and sscanf(). The function dprintf() for printing to a file descriptor may be used.

2.0 Multithreading

The three functions designed to handle PUT, GET, and HEAD requests were implemented from Assignment 1, and they are `processPut()`, `processGet()`, and `processHead()`. The first step is to turn these functions into threads. In order to do so, the return value and the parameters will need to be altered. The return values of all three of these functions will be changed to `void*`. In Assignment 1, all three functions accepted the parameters: request and client socket ID. In order to become a thread for Assignment 2, these two parameters need to be embedded in a single parameter structure: `void *obj`. The request and client socket ID attributes will become a part of the *thread structure* utilized throughout the program. The thread structure is discussed in section 3.0. The dispatcher function, parseRequest(), is also going to be made into a thread.

In order to turn `parseRequest()`, `processPut()`, `processGet()`, and `processHead()` into threads, we must create a global thread structure. To lock and protect critical sections of the code during multithreading, a pointer to `pthread_mutex_t` will be included in the structure. To account for the logging requirement, the name of the log file, the log file descriptor, and the log file offset are defined inside the structure. To account for the health check, the total number of requests and the total number of failed requests are added to the thread structure. The maximum number of threads, as specified in the command line, is defined in the structure as well. The socket ID will also be a part of this structure.

The dispatcher (parseRequest()) shall spawn threads as needed but needs to maintain a count of the running threads and assure that the number of threads does not exceed the number specified after the -N. If there are already N threads running and more requests are coming in, the dispatcher must wait until a running thread returns before starting another one. Hence, maintaining no more than N threads running at a time. If -N is not specified at the command line, the number of threads defaults to 4.

2.1 Logging

For logging, we will be using `dprintf()` and a logger offset. Per a discussion on the Piazza forum and for simplicity of implementation, I have chosen this design over `pwrite()`. This offset, which will be defined in the thread structure, is pertinent in that subsequent calls will not overlay preceding writes to the logger file.

2.2 Health Check

The health check is a special GET request. Therefore, this will be constructed in the `processGet()` function. Once we have retrieved the file name, we are going to check via `strcmp()` if the file name is “healthcheck”. If so, we need to increment the total number of requests, and write a ratio of failures to total number of requests to the socket. We must ensure that a resource name “healthcheck” sent with a PUT or HEAD command will return a 403 Forbidden error code.

3.0 Architectural and component-level design

In `processPut(), processGet(), and processHead()`, we now define a parameter that is of the newly created thread type. After the declarations, we need to perform a lock to denote the entrance of the critical section. The section is unlocked just before the functions return. Locks are strategically placed to protect critical code in the functions `processPut(), processGet(), and processHead()` because they are “shared” locations, meaning they will be visited by multiple threads. We cannot allow these threads to interfere with one another during their respective executions.

Throughout these functions, in every instance where we return an error, we must increment a counter keeping track of the number of failures. Likewise, in `parseRequest()`, we need to increment the total number of requests when `processPut(), processGet()`, and `processHead()` is called.

3.2.Processing narrative (PSPEC) for component n

main(): This function will establish the proper socket connection for communication between the client and server. A 4 KiB buffer will be used upon receiving a request. The 4 KiB buffer and the socket descriptor will be passed to a routine called parseRequest(). A structure of type ThreadArg described above will be created in main() and passed to the dispatcher, parseRequest().

parseRequest(): In this thread, error-checking will be done to check the command names PUT, GET, and HEAD. An error-response will be triggered and logged if an error is detected, otherwise, depending on the command, the structure that was received from main() will be passed to processGet(), processPut(), or processHead() threads as appropriate.

processGet(): This thread will parse the request embedded in the thread structure passed to us by parseRequest() even further, “aggressively” looking for error conditions in the request. This thread will make a call to the open() system call to open a file for reading and sending its information to the client. If the open() system call fails, appropriate responses should be sent to the client. All errors will be logged to a log file if specified. Nominally, this thread writes the contents of the resource file to the socket for the client to receive. It also writes a hex representation of the data in the file to the log file. If the GET request specifies the resource name “healthcheck”, this thread will return X\nY to the client, where X is the total number of failures, and Y is the total number of requests.

processPut(): This thread will parse the request embedded in the thread structure, passed to us by parseRequest(). It will continue looking for error conditions in the request. This thread will make a call to the open() system call to open a file for writing/overwriting the data, from the client, into the file. If the open() system call fails, appropriate responses should be sent to the client. If a PUT command is used on the special resource “healthcheck”, the program will return a 403 Forbidden error code and log the error condition into the log file, if specified. If no errors

are detected, it logs the successful message to the log file. All pertinent failures will be logged to the log file, if specified. Nominally, this thread reads the data sent by the client and writes it into the resource file specified by the client. It also writes a hex representation of the data in the file to the log file.

processHead(): This function is very similar to processGet(), except that it does not contain any data. If a HEAD command is used on the special resource “healthcheck”, the program will return a 403 Forbidden error code and increment the total failure count.

getFileName(): This function will parse the request using strtok() to retrieve and return the file name.

getCommandName(): This function will parse the request using strtok() to retrieve and return the first token. It expects the first word of the request to be the command.

getContentLength(): This function will parse the request using strtok(). It finds the string “Content-Length” and retrieves the token following it. The colon (:) will be treated as a delimiter.

errorCheckCommandName(): This function checks if the command name is valid (i.e PUT, GET, or HEAD). If the command is valid, return 1. Otherwise, return -1.

errorCheckFileName(): This function checks the validity of every character in the file name to assure the file name starts with a slash (/), the length of the file name excluding the slash is within the legal limit (27), and checks if every character in the file name is either an ASCII character (isalpha) or an integer (isdigit), a dash(-), or an underscore (_). It returns 1 if all conditions are valid, otherwise -1.

errorCheckRequest(): This function checks the validity of every request. Tokenize for carriage return and line feed and make sure that all tokens have a colon. Skip the first token, which contains the command and file name and get the second token, which is most likely the header. While the token is not NULL, increment a counter for the instance of “Content-Length” to track the number of occurrences. Return -1 if the header does not contain a colon and if the number of occurrences of “Content-Length” is greater than 1. Otherwise, return 1.

dumpHexString(): This routine accepts a buffer, the file descriptor of the log file, and a position integer. It utilizes dprintf() to print a hex representation of the data into the log file. For large files, the position integer is used to format multiple calls and maintain the order of the bytes printed.

Responses sent by the routines above are of the form:

```
HTTP/1.1 200 OK\r\nContent-Length: 0\r\n\r\n
```

3.3 Design constraints for components

Standard libraries for HTTP and FILE * calls may not be used in any component.

4.0 Testing Issues

This program should be tested with erroneous and correct inputs in the three categories of PUT, GET, and HEAD requests. The following error conditions should be recreated during testing:

(1) 400 (Bad Request)

This error condition must be generated by violating the rules of legal resource names and command names as well as duplicating Content-Length headers or excluding Content-Length headers when expected.

(2) 403 (Forbidden)

This error condition must be generated by creating a file name which denies the user read or write permissions on a file name, and attempting to overwrite it in a PUT command or retrieving information from it via GET and HEAD commands. Additionally, this error condition will be generated if the resource “healthcheck” is utilized in the HEAD and PUT commands.

(3) 404 (Not Found)

This error condition must be generated by utilizing non-existent resource names in the requests to PUT, HEAD, and GET.

(4) 500 (Internal Server Error)

This error condition must be generated when an error arises which does not meet the criteria of the previous error conditions listed above.

4.1 Classes of tests and software responses

Two classes of tests should be performed on this program. Positive tests, which expect error-free executions of legal requests, and negative tests, which test the proper response of the server to erroneous conditions.

The following is a subset of tests that the program should handle appropriately.

GET a small binary file (with __-- in the name)
GET a large binary file
PUT a small binary file
PUT a large binary file
Test HEAD request for a proper response header with 200 OK

Test for 400 with a bad HTTP version (HTTP 1.0)
Test if server creates a default number of threads
Test if server can read optional arguments in any order
Test if server can handle multiple GET requests simultaneously
Test if server can handle many requests greater than the number of threads
Test if server can log a single valid request
Test if server truncates log files every time it starts
Test if server can log many valid requests sent simultaneously
Test if server can log many valid and invalid requests (400) with multiple requests sent
Test if server can log many valid and invalid requests (400 and 404) with multiple requests sent
GET request to ‘healthcheck’ with no logging enabled (404 error)
GET request to ‘healthcheck’ with logging enabled
Test 403 error for a request to ‘healthcheck’
Test logging of many valid and invalid requests including a request to ‘healthcheck’

The following cURL commands are used to test for PUT, GET, and HEAD, respectively.

```
curl -v -T fileToSend.txt http://localhost:8080/ --request-target fileName

curl -s http://localhost:8080/fileName

curl --head http://localhost:8080/fileName
```