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

Assignment 1: Design Document

1.0 Introduction

In this assignment, we are designing a server, which parses HTTP requests. It will be serving a client (curl will be used for this purpose and testing). The port number shall be the only online parameter to the server.

There are only three HTTP requests that the server needs to handle: GET, PUT, and HEAD.

GET fetches the contents of a file named in the request. The server sends a response followed by the requested data. PUT specifies a file name to be created on the server and it also sends the data to be written/overwritten to the file. HEAD is very similar to GET except no data is returned by the server, only an informative response header.

There are two phases to the design of this server.

1. Establish a communication path between the server and the client, which would allow requests and responses to travel back and forth between the server and the client.
2. Once the communication has been established between the client and the server, the server must be able to parse the requests, check for error conditions in these requests, and return the appropriate status codes and/or the data.

The server shall accept one command-line parameter, which will be a port number (i.e 8080).

1.1 Goals and objectives

The overall goals and objectives are successful processing of the PUT, GET, and HEAD requests once the communication has been established between the server and the client. Through “aggressive error-checking”, the following error/status conditions should be checked for each request: 200 (OK), 201 (Created), 400 (Bad Request), 403 (Forbidden), 404 (Not Found), and 500 (Internal Server Error).

1.2 Statement of scope

The major inputs to the server are the PUT, GET, and HEAD requests. From a developer’s point of view, once a socket connection has been established between the server and the client, the major processing will include parsing the request as they come in, error-checking them, and appropriately processing each request. Receiving the requests will take place in a loop in the main() routine. Requests will be sent to a parseRequest() function, which will do partial error-checking to see if the proper command names (PUT, GET, or HEAD) are present in the

request and then call the appropriate routines (processPut(), processGet(), and processHead()) which will perform further parsing, error-checking, and serving the appropriate response.

1.3 Major constraints

This program must be written in C. The developer is not permitted to use any FILE * calls, or use any standard HTTP libraries. The permitted system calls are: socket, bind, listen, accept, send, recv, open, read, write, and close. The functions dprintf for printing to a file descriptor and sscanf may be used.

2.0 Data design

A 4 KiB buffer will be used to retrieve data from the client and process accordingly. The starter code contains a sockaddr_in structure named “server_addr” which is then populated as appropriately for the socket connection, including the port number that will be passed to the server from the command line.

2.1 Internal software data structure

The data structures that will be passed between the components of the software are the 4 KiB buffers, the socket descriptor used for communicating with the client, and the file descriptor used for reading and writing files on the server-side.

2.2 Global data structure

There are no global data structures.

2.3 Temporary data structure

Upon receiving PUT requests, files are created on the server-side and upon receiving GET and HEAD requests, these files will be read.

2.4 Database description

No databases are used in this program.

3.0 Architectural and component-level design

A description of the program architecture is presented.

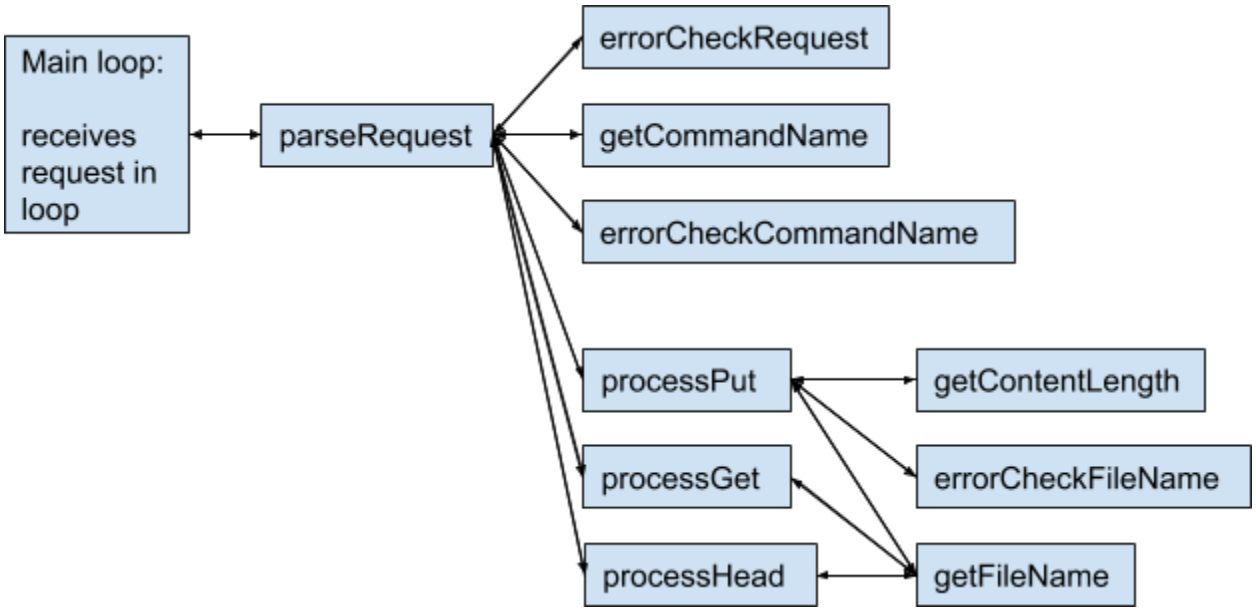
The following functions are the main components in the server design:

main(), parseRequest(), errorCheckRequest(), getCommandName(), errorCheckCommandName(), processPut(), processGet(), processHead(), getFileName(),

errorCheckFileName(), and getContentLength(). A description of the components is discussed in further detail in 3.2.1.

3.1 Architecture diagram

A pictorial representation of the architecture is presented below.



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().

parseRequest(): In this routine, minimal error-checking will be done only to check the command names PUT, GET, and HEAD. An error-response will be returned if the an error is detected, otherwise, depending on the command, the 4 KiB buffer, and the socket descriptor that were received from main() will be passed to processGet(), processPut(), or processHead() as appropriate.

processGet(): Here, we will parse the 4 KiB buffer passed to us by parseRequest() even further, “aggressively” looking for error conditions in the request. This routine 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.

processPut(): The 4 KiB buffer passed to us by parseRequest() will be parsed even further, looking for error conditions in the request. This routine 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.

processHead(): This function is very similar to processGet(), except that it does not contain any data.

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.

Responses sent by the routines above are of the form:

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

3.2.1 Component processing detail

main() calls parseRequest() with two parameters: request and the client socket descriptor.

parseRequest() calls the function errorCheckRequest() with the parameter request.

Inside errorCheckRequest(), we return -1 if the request is NULL, if the length of the request exceeds 4 KiB, or if we do not come across a colon. Additionally, while our token is not NULL, we traverse the request to keep a count of the number of instances of Content-Length. Should

the number of Content-Length occurrences be greater than 1, we return -1. If none of the above error conditions are met, errorCheckRequest() returns 1.

If errorCheckRequest() returns -1, we expect a 400 Bad Request. Otherwise, we will be calling getCommandName() with the parameter request.

getCommandName() will retrieve the name of the command in the request via delimiters. Next, we will be calling errorCheckCommandName with the command name as its parameter to check its validity.

Inside errorCheckCommandName(), we will be returning 1 if the string matches the criteria for a valid command name, otherwise return -1. If errorCheckCommandName() returns -1, this is a 400 Bad Request.

Should the command names be valid, and errorCheckCommandName() returned 1, we can now proceed to call the individual procedures that handle PUT, GET, and HEAD requests.

Inside processPut(), we are getting the file name and validating it via getFileName() utilizing delimiters and errorCheckFileName() to ensure the legal length of a resource name and that the resource name begins with a slash. If there is an error in the file name, we will return a 400 Bad Request. Next, we will be calling getContentLength(). Inside getContentLength(), we are ensuring that every token is a digit. If the content-length is -1, we return a 400 Bad Request. Then, we ensure that the file is valid by using errno and checking EISDIR, EACCES, and ENOENT, and generating the appropriate responses. In the case of EISDIR and EACCES, we would return a 403 Forbidden response. In the case of ENOENT, we would return a 404 Not Found. Otherwise, return a 500 Internal Server Error. Finally, we write the number of bytes to the socket via recv() and write(). With a successful request, we print the content-length along with 201 Created.

processGet() is very similar to processPut in terms of error-checking. The successful checks would generate 200 OK response, and the contents of the file would be printed.

processHead() is identical to processGet(), except we don’t print the content of the file.

3.2.2 Design constraints for components

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

4.0 Testing Issues

Test strategy and preliminary test case specification are presented in this section.

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.

(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

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.

Positive tests:

GET a zero sized file
GET a small text file
GET a small binary file
GET a large text file
GET a large binary file
PUT a small text file
PUT a small binary file
PUT a large binary file
Test HEAD request for a proper response header with 200 OK
Test if PUT requests receive response code 201 (Created) in response header
PUT effectively overwrites an existing file (generates 201 Created)
PUT a zero-length file
HEAD a zero-length file

Negative tests:

Test for 400 error on bad resource (request-target) name
Test for 403 error on resource without permission (using PUT)
Test for 403 error on resource without permission (using GET)
Test for 403 error on resource without permission (using HEAD)
Test for 404 error for resource that doesn't exist (using GET)
Test for 404 error for resource that doesn't exist (using HEAD)
Test for 400 error on file exceeding 27 ASCII characters
Test for a file consisting of the *allowed* characters
Test for 400 error on file without slash
Test for a file with more than one Content-Length
Test for a file with no Content-Length

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
```

4.2 Expected software response

The expected results from testing are specified, below.

GET a zero sized file → 200 OK
GET a small text file → 200 OK
GET a small binary file → 200 OK
GET a large text file → 200 OK
GET a large binary file → 200 OK
PUT a small text file → 201 Created
PUT a small binary file → 201 Created
PUT a large binary file → 201 Created
Test HEAD request for a proper response header → 200 OK
Test if PUT requests receive response code 201 in header → 201 Created
PUT effectively overwrites an existing file → 201 Created
PUT a zero-length file → 201 Created
HEAD a zero-length file → 200 OK
Test for 400 error on bad resource (request-target) name → 400 Bad Request
Test for 403 error on resource without permission (using PUT) → 403 Forbidden
Test for 403 error on resource without permission (using GET) → 403 Forbidden
Test for 403 error on resource without permission (using HEAD) → 403 Forbidden
Test for 404 error for resource that doesn't exist (using GET) → 404 Not Found

Test for 404 error for resource that doesn’t exist (using HEAD)	→ 404 Not Found
Test for 400 error on file exceeding 27 ASCII characters	→ 400 Bad Request
Test for a file consisting of the *allowed* characters	→ 400 Bad Request
Test for 400 error on file without slash	→ 400 Bad Request
Test for a file with more than one Content-Length	→ 400 Bad Request
Test for a file with no Content-Length	→ 400 Bad Request