Despina Patronas
CSE 130

Design Document Assignment 2: httpserver

Introduction

I. **Goals and Objectives:**
This assignment is to implement *httpserver* as a multi-threaded HTTP server.

II. Usage and Scope of *httpserver*

i. *httpserver* responds to GET and PUT commands for files

ii. *httpserver* responds to HEAD command on existing files

iii. *httpserver* reponds to malformed / erroneous requests w/o crashing.

iv. The port number is specified on the command line as a mandatory user arg.
Port number must be positive 16 bit integer. Server closes with invalid port #

v. *Httpserver* accepts command line args: -N threads, -l log_file.

vi. *httpserver* supports persistent connection, able to respond to multiple requests that arrive in the same connection.

vii. *httpserver* persistently stores files in the same directory the server is run.
Ie: server can be restarted with the same files
or run on a directory that already has files.

viii. *httpserver* exhibits HTTP protocol

1. Sends files via http
2. Receives files via http
3. Parse simple simple HTTP requests
    a. Use string functions

ix. Example of HTTP Protocol

1. Client sends request to Server : EITHER (a), (b), or (c)
    a. Send a file: client -> server **(PUT – file into server)**
    b. Request a file: client <- server **(GET – file from server)**
    c. Send metadata about file: client <-server **(HEAD)**

2. Server does the following:
    a. Receives (a), (b), or (c) UNSUCCESSFULLY
        i. Respond with error message
            1. Server closes current client connection
                a. Idles until another client connects

    b. Receives (a), (b), or (c) successfully
        i. Responds to client request appropriately
            1. Idles for:
                a. Another request from current client
                b. OR if Client closes connection:
                    i. Idles until another connection

III. Design

httpserver may NOT use HTTP libraries.
Httpserver may NOT use FILE * calls (except for printing error messages)

HTTPS REQUESTS:

i. PUT: ***Assume headers are no longer than 1KiB. Body may be any size.***
***Client* connects and sends (ASCII text) request:** *Note: \r\n == CRLF == \n*

PUT /Object HTTP/1.1*(\r\n)*	--request
Host: 10.0.0.5:8080*(\r\n)*	--header
Content-Length: 3*(\r\n)*	--size of body sent
(\r\n)	--empty line
message for Object	--body

1. 1st Part:
   a. Ascii text
      i. Request. *(3 components space separated parts)*
         1. commands = GET, PUT, HEAD
         2. /Object = Should match a valid object name.
         3. HTTP = version used. ***Asgn uses HTTP/1.1***

      ii. Header. *Must check host & value are present and valid*
         Must be present in all requests
         1. HOST:value *(value should be valid)*
         2. Content-length: Num (**required in PUT**)

      iii. One empty line (only CRLF) ends 1st part

2. 2nd Part: *(only present in PUT)*
   a. Note: this doesn't have to be ascii
      i. Body contents for PUT file
      (PUT must have content-length header, else are invalid)

      sent data may include any bytes, including NUL (\0).

***Server* Response to successful PUT:**

HTTP/1.1 201 Created(\r\n)	--HTTP version status_code code_descr.
Content-Length: 8(\r\n)	--All responses match Content-Length
\r\nCreated\n	--Necessary whether successful or not.

1) read the specified number of bytes
2) Create a file with name "Object" *(truncate if file already exists)*
3) Write contents of the clients body to file Object
4) Send client a response according to the success of operation
5) Waits for the next another request *(from current or new client)*

---

i. **HTTPS LOGGING:**

Logging is done for each request while a log file argument exists.

1. While log is valid, a named log argument will persist throughout server runs, appending to it upon passing it into the server execution arguments.

2. Logs contain information about the request line from client as well as content length and a hex representation of first 1K or less bytes process.

3. Logs also document failed requests.

ii. **HTTPS HEALTH CHECK:**

1. Health check diagnoses the number of failed logs per total requests and returns that information back to client.

2. The health check can only be performed by get, no file is saved on server, it is generated upon request based on the current log file which exists on running server.

3. Running health check without a running log produces a 404 error

4. Running health check with PUT or HEAD produces a 403 error.

iii. **GET:**

***Client* connects and sends request:**

Note: no content-length or body

GET /Object HTTP/1.1*(\r\n)*	--request
Host: 127.0.0.1:1234*(\r\n)*	--header
(\r\n)	--empty line

***Server* Response to successful GET:**

HTTP/1.1 200 OK*(\r\n)*
Content-Length: sizeof_file_requested(\r\n)
file_conents_requested

1) Send client a response according to the success of operation
2) Waits for the next another request *(from current or new client)*

---

iv. **HEAD:**

***Client* connects and sends**

Note: no content-length or body

HEAD /Object HTTP/1.1*(\r\n)*	--request
Host: 127.0.0.1:1234*(\r\n)*	--header
(\r\n)	--empty line

***Server* Response to (HEAD similar to GET):**

HTTP/1.1 200 OK
Content-Length: sizeof_file_requested (\r\n)
file_conents_requested

Note: Valid resource names are up to 19 ASCII characters (upper, lower-case alphabet (52 characters), digits 0–9 (10 characters), dot (.) and underscore (_), for total of 64 possible characters. The resource names are preceded in requests by slash (/) which does not count towards the 19 characters limit. If a request includes an invalid name, the server must fail the request and respond accordingly. **Regex for resource names.**

The following status codes used in assignment:

Code	Message	Usage
200	OK	To be used when a response can be responded to successfully, if no other code is more appropriate;
201	Created	To be used if a successful response resulted in the creation of a file – note that PUT can be used to create a file, but it can also overwrite a file;
400	Bad Request	To be used when a request cannot be parsed correctly, or it has problems that result in failure to respond, and no other code is more appropriate;
403	Forbidden	To be used if the request is for a valid resource name, but the server does not have the permissions to access it;
404	File Not Found	To be used the request is for a valid resource name, but the server cannot find it (e.g., the file doesn't exist);
500	Internal Server Error	To be used if the request is valid but the server cannot comply because of internal problems (e.g., error when allocating memory for a data structure that would necessary to process the request), and no other response code is appropriate;
501	Not Implemented	To be used if the request is properly formatted, but uses a request type that the server has not implemented (that is, not GET, PUT, or HEAD).

v. Send()

Usage: *ssize_t send(int sockfd, const void *buf, size_t len, int flags);*
Sends message to another socket when connected ie: recipient is known.
On success, returns number of bytes sent.
On error, -1 is returned, and errno is set appropriately.

vi. Recv()

Usage: *ssize_t recv(int sockfd, const void *buf, size_t len, int flags);*
Receives message from another socket when connected ie: recipient is known.
On success, returns number of bytes received.
On error, -1 is returned, and errno is set appropriately.

***vii.* Open()**

Usage: *int fd = open(const char *filename, int flags);*
If successful, returns a file descriptor for the opened file,
otherwise, returns -1

viii. **Read()**

Server uses directory which it is run to Read files from GET (sent files)
Server reads HEAD requests.

Usage: *int r_count = read(int fd, void *buff, int count);*

If successful, returns number of bytes read from file descriptor to buff (0 for End of File (EOF)), otherwise returns -1.

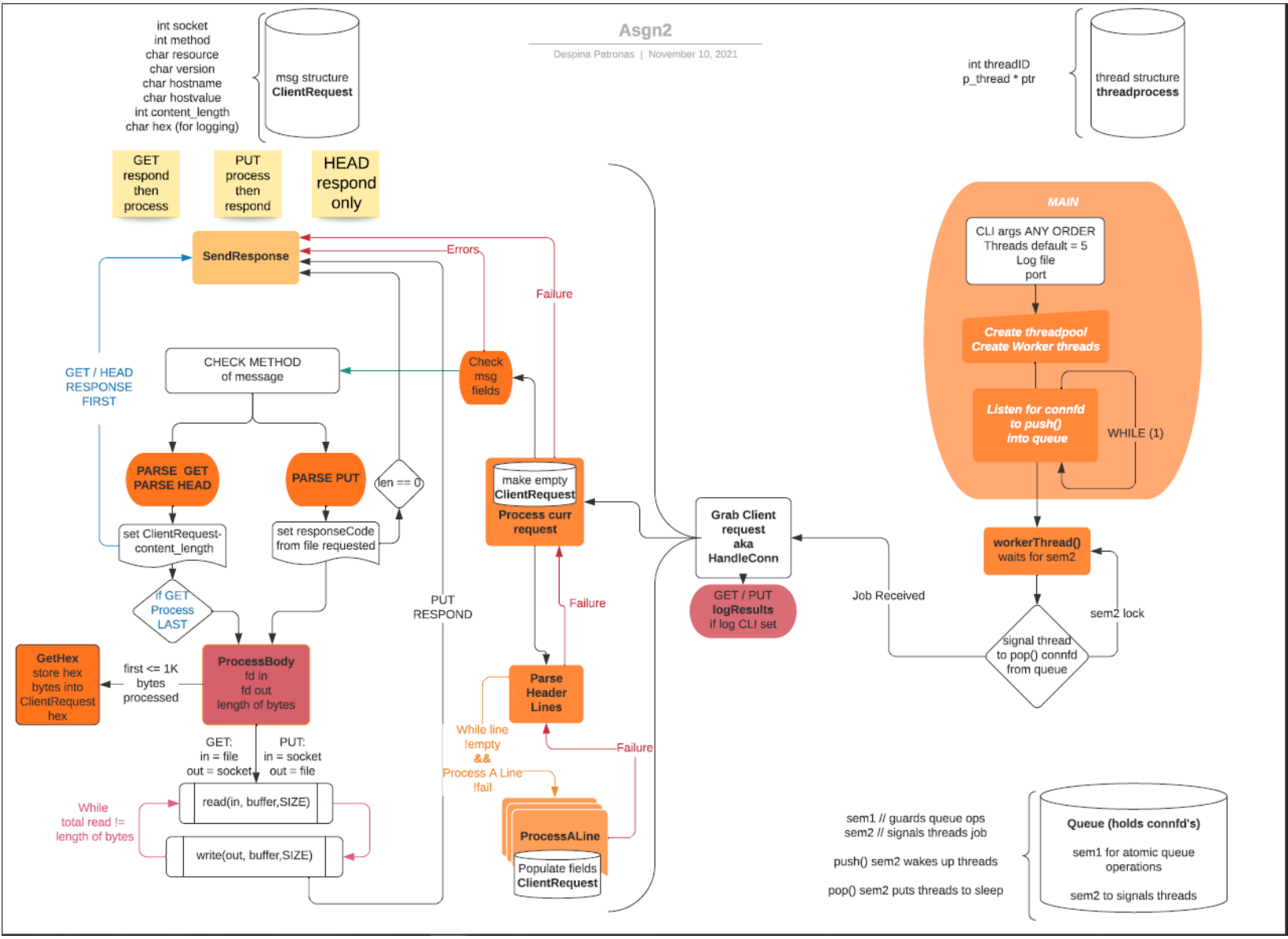
shoulders: Read will be done in a loop, with constant buffer size until EOF. Note: the size which you read in is subjective and performance will vary based on your implementation.

ix. **Write()**

Server uses the directory which it is run to Write files from PUT requests

Usage: *int w_count = write(int fd, void * buff, int count);*

If successful, returns number of bytes written to file descriptor from buff, otherwise returns -1.


Asgn2
Despina Patronas | November 10, 2021
int socket
int method
char resource
char version
char hostname
char hostvalue
int content_length
char hex (for logging)
msg structure
ClientRequest
int threadID
p_thread * ptr
thread structure
threadprocess
GET
respond
then
process
PUT
process
then
respond
HEAD
respond
only
SendResponse
Errors
Failure
MAIN
CLI args ANY ORDER
Threads default = 5
Log file
port
Create threadpool
Create Worker threads
Listen for connfd
to push()
into queue
WHILE (1)
GET / HEAD
RESPONSE
FIRST
CHECK METHOD
of message
Check
msg
fields
PARSE GET
PARSE HEAD
PARSE PUT
len == 0
make empty
ClientRequest
Process curr
request
Grab Client
request
aka
HandleConn
workerThread()
waits for sem2
set ClientRequest-
content_length
set responseCode
from file requested
if GET
Process
LAST
PUT
RESPOND
Failure
GET / PUT
logResults
if log CLI set
Job Received
sem2 lock
signal thread
to pop() connfd
from queue
GetHex
store hex
bytes into
ClientRequest
hex
first <= 1K
bytes
processed
ProcessBody
fd in
fd out
length of bytes
Parse
Header
Lines
While line
!empty
&&
Process A Line
!fail
Failure
GET:
in = file
out = socket
PUT:
in = socket
out = file
While
total read !=
length of bytes
read(in, buffer,SIZE)
write(out, buffer,SIZE)
ProcessALine
Populate fields
ClientRequest
sem1 // guards queue ops
sem2 // signals threads job
push() sem2 wakes up threads
pop() sem2 puts threads to sleep
Queue (holds connfd's)
sem1 for atomic queue
operations
sem2 to signals threads