httpserver

The HTTPParse class:

The main goal of this class is to handle the parsing and sorting of an HTTP header and body. ServerManager will be creating HTTPParse objects to parse requests sent to the server, then use the data parsed by HTTPParse to respond to these requests.

Private Members:

- index: An integer representing the current index in our character stream of the request. Utilized by GetWord().
- request: The string HTTP request from the client. This can either be a header or a body
- requestLength: The string length of request
- requestType: A string representing the type of HTTP request given
- filename: The filename given in the request header
- contentLength: The content length given in the request header
- char* GetWord()
- int PutAction()
- int GetAction()

Public Members:

- body: The buffer for the data send in the HTTP request's body
- HTTPParse()
- ~HTTPParse()
- int ParseRequestHeader(char*)
- int ParseRequestBody(char*)
- int GetRequestType()
- int ContentLength()

char* HTTPParse::GetWord():

This function will return the next word in the stream of *request*. It uses *index* as it's marker for where the last GetWord() call left off in *request*. A word is defined to be a substring of characters, starting from *index*, up to the next space or newline. When the next word is returned, the index is set to the beginning of the next word.

int HTTPParse::PutAction():

This function is designed to finalize the processes of a PUT request. It is called after parsing has completed and the body of the request has been acquired from the client. It will save a file of a specified name with specified data to the server. The function returns a code based on whether or not the put request was handled successfully (This code will be interpreted and handled by ServerManager).

```
int HTTPParse::PutAction()
    fd = open(filename, write only | create if it does not exist | remove any data inside)
    if (fd < 0)
        return 403
    if (write(fd, body, length of body) != length of body)
        return 500
    if (close(fd) < 0)
        return 500
    return 201
```

int HTTPParse::GetAction():

This function is designed to finalize the processes of a GET request. It is called after parsing has completed. It will open a specified file on the server, save its data to *body*, and set *contentLength* to the size of *body*. The function returns a code based on whether or not the put request was handled successfully (This code will be interpreted and handled by ServerManager).

```
int HTTPParse::GetAction()
    fd = open(filename, read only)
    exists = access(filename)
    if (fd < 0)
        if (exists < 0)
```

```
                return 404
            else
                return 403
        buffer[8]
        Read the data in the file at fd into the buffer 1 byte at a time, then
    concatenate the string *body* with the contents of buffer, until it reaches EOF
        *contentLength* = size of body
        if (close(fd) < 0)
            return 500
        return 200
```

HTTPParse::HTTPParse()

This is the constructor for HTTPParse. It initializes the member variables of the class.

```
HTTPParse::HTTPParse()
    *index* = 0
    *requestLength* = 0
    *contentLength* = -1
    *requestType* = NULL
    *filename* = NULL
    *contentLength* = 0
```

HTTPParse::~HTTPParse()

This is the destructor for HTTPParse. It deletes *requestType* and *filename*.

int HTTPParse::ParseRequestHeader(char*)

This function parses an HTTP header. It will return a code to be handled by ServerManager once it completes its task. If this is the first iteration of a PUT request, it will return 0 to identify that ServerManager is still waiting for the body of the HTTP request.

```
int HTTPParse::ParseRequestHeader(char* r)
    *index* = 0
    *request* = r
    *requestLength* = length of *request*
    *requestType* = GetWord()
```

```
        filename = GetWord()
        if (filename[0] == '/')
            remove that character from the string
        else
            return 400
        if (length of filename isn't 10)
            return 500
        httpTitle = GetWord()
        if (httpTitle != "HTTP/1.1")
            return 400
        delete httpTitle
        if (GetRequestType() == -1)
            return 500
        if (GetRequestType() == 0)
            return GetAction()
        if (GetRequestType() == 1)
            contentLengthHeader
            while (contentLengthHeader != "Content-Length: " && index <
    requestLength)
                delete contentLengthHeader
                contentLengthHeader = GetWord()
            if (index < requestLength)
                contentLength = (int) GetWord()
            delete contentLengthHeader
        return 0
```

int HTTPParse::ParseRequestBody(char*):

This function parses an HTTP body (from a put request). It will return a code to be handled by ServerManager once it completes its task.

```
    int HTTPParse::ParseRequestBody(char* r)
        index = 0
        request = r
        requestLength = length of request
        if (contentLength > requestLength)
            return 500
        body = request
        return PutAction()
```

int HTTPParse::GetRequestType():

This function returns an integer based on the value of *requestType*. If its NULL, return -1, if its GET, return 0, if its PUT, return 1.

int HTTPParse::GetContentLength():

This function returns the value stored in *contentLength()*.

The ServerManager class:

The purpose of this class is to handle requests sent by the client and send them to HTTPParsers for interpretation. Once the requests are interpreted, ServerManager responds to the client accordingly

Private Members:

- listen_fd: The Listen file descriptor
- comm_fd: The Communication file descriptor
- char* GenerateMessage(int, int): Returns a message based on the message code and Content Size
- unsigned long GetAddress(char*): Returns an Internet address based on specific parameters

Public Members:

- ServerManager(): The constructor of ServerManager
- void Setup(char*, unsigned short): The main process of ServerManager

char* ServerManager::GenerateMessage(int, int):

```
char* ServerManager::GenerateMessage(int message, int contentLength)
    if (message == 0)
        return NULL
    else if (message == 200)
        return "HTTP/1.1 200 OK \r\nContent-Length: " + contentLength
+ "\r\n"
    else if (message == 201)
```

```
                return “HTTP/1.1 201 OK\r\nContent-Length: 0\r\n\r\n”
        else if (message == 400)
                return “HTTP/1.1 400 Bad Request\r\nContent-Length: 0\r\n\r\n
        else if (message == 403)
                return “HTTP/1.1 403 Forbidden\r\nContent-Length: 0\r\n\r\n”
        else if (message == 404)
                return “HTTP/1.1 404 Not Found\r\nContent-Length: 0\r\n\r\n”
        else if (message == 500)
                return “HTTP/1.1 500 Internal Server Error\r\nContent-Length:
0\r\n\r\n”
        else
                return NULL
```

unsigned long ServerManager::GetAddress(char*):

This function is designed to take an input string referencing an internet address and convert it into a working integer address.

ServerManager::ServerManager():

```
ServerManager::ServerManager()
        listen_fd = 0
        comm_fd = 0
```

void ServerManager::Setup(char*, unsigned short):

The purpose of this function is to create a socket server, then wait for HTTP requests. Once the HTTP requests are received, they are parsed by HTTPParsers. After parsing, the function then determines the correct action to take depending on the request, and respond to the client accordingly

```
unsigned long ServerManager::Setup(char* address, unsigned short port)
        struct sockaddr_in servaddr
        memset(&servaddr, 0, sizeof servaddr)
        listen_fd = socket(AF_INET, SOCK_STREAM, 0)
        if (listen_fd < 0)
                Socket error
        servaddr.sin_family = AD_INET
        servaddr.sin_addr.s_addr = GetAddress(address)
        servaddr.sin_port = htons(port)
        if (bind(listen_fd, (struct sockaddr*) &servaddr, sizeof servaddr) < 0 )
                bind error
        if (listen(listen_fd, 500) < 0)
                listen error
        while(1)
                waiting = “waiting for connection\n”
                write waiting to stdout
                comm_fd accept(listen_fd, NULL, NULL)
                if (comm_fd < 0)
                    accept error
                buf[SIZE]
                HTTPParse* parser = new HTTPParse()
                while(1)
                    n = recv(comm_fd, buf, SIZE, 0)
                    if (n < 0) recv error
                    if (n <=0) break;
                    int message
                    if (parser.GetRequestType() == 1)
                        message = parser.ParseRequestBody(buf)
                        clear buf
                        msg = GenerateMessage(message, 0)
                        send(comm_fd, msg, length of msg, 0)
                        delete parser
                        parser = new HTTPParse()
                    else
                        message = parser.ParseRequestHeader(buf)
                        clear buf
                        if (message != 0)
                        msg = GenerateMessage(message,
        parser.GetContentLength())
                        if (message == 200)
```

```
                    send(comm_fd, msg, length of msg, 0)
                else
                    send(comm_fd, msg, length of msg, 0)
                delete parser
                parser = new HTTPParse()
```

main.cpp:

This file's purpose is to call main() and get our server running. It takes 2 arguments, one for the server's Internet address and one for the port for it to be active on. If nothing is given as an argument for the port, it defaults the port to 8080.

```
main(size of array, array of program arguments)
        if size of array < 2
            return
        if size of array > 3
            return
        address = element 2 of the array
        port = 8080
        if the size of the array > 2
            port = element 3 of the array
        ServerManager serverManager
        serverManager.Setup(address, port)
        Return
```

Testing

The testing done on this server consisted of running various curl commands to create connections to the server. As I made each function, I tested its functionality with temporary print functions of the members the program was currently working with, and initialized the testing by curling to the specified port and address. I used different types of curl commands to test (GET, PUT, multiple GETs, GET into a file, etc).

Question: What happens in your implementation if, during a PUT with a Content-Length, the connection was closed, ending the communication

early? This extra concern was not present in your implementation of dog. Why not? Hint: this is an example of complexity being added by an extension of requirements (in this case, data transfer over a network).

If, during a put, connection is closed, the processes of HTTPParse will be deleted and the server will start checking for another client. This case was not in my implementation of dog because dog did not have to handle the latency from sending information over an internet connection. Since that takes much more time than local processes in a computer, a case in which connection is cancelled in the middle of a process opens up.