Computer Science 667/867
Internet Application Design and Development
(spring 2012)
Web Server Project

---

1. Overview

You are to design and write a web server program. The web server you create will not be a complete web server, however, it will implement enough features (as summarized in the grading summary below) that it can be used to support and serve a basic web site. Your web server will be able to:

1. Parse HTTP Requests (generally sent from browser to server) [Part I]
2. Generate and send HTTP Responses (generally sent from server to browser) [Part I]
3. Handle multiple requests "simultaneously" through the use of threads [Part II]
4. Execute server-side processes to handle CGI's and other server-side scripts [Part II]
5. Support a simple authentication [Part II]
6. Support a simple caching [Part II]
7. Support persistent connection (extra credit)

   Note: Phase I is the minimum requirement to be considered as "working web server". Anyone who couldn't submit a working Phase I will not be able to join a term project team.

This is a two-person team project. The due date for the complete project is posted on course web site. Part I is due by midnight of Feb 19th (Sun). You need to get started immediately. If you don't submit the part I, then your part II can not get any credit. Your grading will be done at thecity, so we recommend you to make sure your program is tested on thecity before submission.

2. Grading Summary

This is a breakdown of how your web server will be graded. Requirements for the individual points below are given under the Requirement Details section. (The grading chart is attached at the end)

Phase	Requirement	Weight
I	Parsing HTTP Request	10%
I	Generating Response with plain text and html	10%
I	Handling httpd.conf file	10%
I/II	Handling Images and other MIME types	10%
II	CGI	20%
II	Multithreading	10%
II	Handling 304 status code (Caching)	5%
II	Handling 401/403 status code (Authentication)	5%
II	Performance and Code Review	10%

II	Documentation	10%

Phase I is required to be completed and submitted by given due date, and the complete version (phase I & II) is due by March 11th midnight. I am giving extra time (2 more days) for documentation, so you can focus on programming and then generate quality documentation afterwards. Submit your documentation by March 13th midnight. Interactive grading will start from the next week. A guideline for good documentation will be provided separately.

3. Requirement Details

A. General Requirements

You are required to design all of your own classes for the web server project. You are NOT ALLOWED to use any of the J2EE classes for implementation. This means that objects like the Request and Response (if you choose to have such objects) MUST be designed and implemented by your own code; you are not allowed to use the HttpRequest and HttpResponse objects. If you have any question about whether or not your implementation complies with these guidelines, feel free to ask!

To be able to handle request and response properly, your server need to read in and follow the configuration directives in httpd.conf. The web server, at startup, will read a file called httpd.conf. This file contains information about the server root (the directory where the server is installed), the port the server is listening on, the document root (the path containing the documents that the server is to deliver), the directory and filename in which the web server is to log information, the default cgi-bin directory (the path containing executable scripts), and a list of aliases (alternative names for specific paths). To specify this information, the file will use the following keywords:

Keyword	**Description**
ServerRoot	Provides the path to the root of the server installation.
Listen	Provides the port number that the server will listen on for incoming requests.
DocumentRoot	Provides the path to the root of the document tree.
LogFile	Provides the path AND filename for the logfile.
ScriptAlias	Provides the path to the root of the cgi-bin directory.
Alias	Maps a symbolic path in a URL to a real path

Every time the server performs an action, it should record the action in the log file specified in the httpd.conf file. The format for information recorded in the log file should follow the Common Log Format discussed in class. More information about this format can be found at http://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format.

The httpd.conf file also includes authentication information using <limit> and <directory> tags. There are a few different approaches for authentications (Apache supports 2 methods and other web servers also provide slightly different approaches. You may choose one of them and document and demonstrate during interactive grading. Due to many variations available, you are required to submit clear documentation that explains the authentication method you choose. An example of the httpd.conf file can be found if you install Apache web server.

B. Parse HTTP Requests (generally sent from browser to server)

Your server should be able to appropriately respond to the following request methods: GET, HEAD, POST, and PUT. In case of headers, your server needs to convert them to environment variables.

The general format for the first line in a request message is (a description of each item follows):
METHOD SP REQUEST-URI SP HTTP-VERSION SP CRLF

Item	Description
SP	represents a space in the line of text.
CRLF	represents a carriage return followed by a line feed.
METHOD	Method specifies the type of request being made. In HTTP v1.1, valid methods include: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, and CONNECT. Descriptions of the various methods can be found in sections 9.2 - 9.9 of the HTTP v1.1 RFC.
REQUEST-URI	The Request URI has the traditional URL syntax with optional parameters. The Request URI may be part of a query, in which case it would be terminated by a question mark, and the question mark would be followed by a string of parameters.
HTTP-VERSION	This gives the version of the HTTP Protocol being used. This will be either HTTP/1.0 or HTTP/1.1.

An example of an HTTP Request follows:

```
GET /this/is/a/path/url?param1=John&param2=something HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.01b6C[en](X11;I;Linux 2.0.30i586)
Pragma: no-cache
Host: localhost:8080
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

C. Generate and send HTTP Responses (generally sent from server to browser)

For this requirement, your server should be able to appropriately generate response messages with the following response codes: 200, 201, 304, 400,401,403, 404. Field names that must be included in EVERY response from the server include Date and Server. Other headers are required in order to properly respond to certain requests, including Content-Type and Content-Length. You are encouraged to ask questions if you are unsure as to what headers need to be implemented.

The web server, at startup, also needs to support diverse formats' of media files. A file called mime.types contains a list of MIME types and associated file suffixes. When the server sends back the response to the browser, it compares the file suffixes with the mime type and attaches

the mime type to Content-Type response header. Therefore, you should read in these mime types and maintain them in some data structure.

The general format for the first line in a response message is (a description of each item follows):
HTTP-VERSION SP STATUS-CODE SP REASON-PHRASE CRLF

Item	Description
SP	represents a space in the line of text.
CRLF	represents a carriage return followed by a line feed.
HTTP-VERSION	This gives the version of the HTTP Protocol being used. This will be either HTTP/1.0 or HTTP/1.1.
STATUS-CODE	A 3 digit integer representing the status of the response.
REASON-PHRASE	A phrase that gives a description of the status code.

These are response phrases that are required to be implemented.

200 OK
201 Successfully Created
304 Not Modified
400 Bad Request
401 Unauthorized
403 Forbidden
404 Not Found
500 Internal Server Error

An example of an HTTP Response follows:

> HTTP/1.1 200 OK
> Date: Thu, 18 Sep 1997 19:34:18 GMT
> Server: Apache/1.1.3
> Content-Type: text/html
>
> [This would be the body of the response message. Notice the new line between the headers and the body.]

The HTTP header fields have the following general format (Note the colon ':' in the header field line):
FIELD-NAME:[FIELD-VALUE]CRLF

Item	Description
CRLF	represents a carriage return followed by a line feed.
FIELD-NAME	There are 19 different request field-names described in the protocol. There are 9 different response field-names. 9 field-names are provided as general header field-

	names. Finally, an additional 11 "entity headers" are defined in the protocol that allow both the server and the browser to specify additional information in the message. All of the field names in the protocol are described in section 14 of the HTTP v1.1 RFC.
FIELD-VALUE	The value corresponding to the field name. The format of this value varies depending on the field-name.

Some examples of field names, followed by the type of field name are:

Content-Length	(entity)
Content-Type	(entity)
From	(request)
Host	(request)
Referer	(request)
User-Agent	(request)
Age	(response)

D. Handle multiple requests "simultaneously" through the use of threads

Your server should be able to simultaneously handle multiple requests. One way to test the functionality of your web server's multithreading capability is to request a LARGE file (for example, a large jpeg). Right after you have requested this file, in another browser window, request a small text file (for example, the document root/index.html file). Assuming the image is large enough, you should receive the index.html file before the image has finished downloading to the client browser.

E. Execute server-side processes to handle CGI's and other server-side scripts

Both Perl scripts and Python scripts will be tested. If you are using the test case available, insure that you have set the correct paths to the appropriate interpreters on the system you are testing on!

F. Support a simple authentication

For this requirement, your web server should be able to appropriately generate the 401 and 403 response messages. Please note that this includes the use of additional header lines (WWW-Authenticate, Authorization).

G. Support a simple caching

For this requirement, your web server should be able to appropriately generate the 304 response message. Please note that this includes the use of additional header lines (Expires/Age/Last-Modified).

H. Support persistent connection (extra credit)

You will learn the performance gain by employing persistent connection. Make your server support persistent connection as an extra credit feature.

[Clarifications] Parsing httpd.conf Configuration file:

Minimum Requirements:

1) Web Server must be able to handle in parsing the following variables in the configuration file:
    a. ServerRoot
    b. ServerAdmin
    c. DocumentRoot
    d. Listen
    e. LogFile
    f. ScriptAlias
    g. Alias
    h. DirectoryIndex
    i. AccessFileName or <directory> tag
    j. MaxThreads
    k. KeepAlive On/Off (Extra Credit)
    l. KeepAliveTimeOut (Extra Credit)

Note: the value of each variable is separated by a whitespace. Use absolute path (or full path) to specify the location of your files. eg. (ServerRoot /home/class/csc667/WebServer). You web server will be able to handle any line that begins with the # sign which is a comment.

Below is an example of the httpd.conf file:

```
ServerRoot /home/class/csc667/WebServer/
ServerAdmin wmac01@thecity.sfsu.edu
DocumentRoot /home/class/csc667/Webserver/htdocs/
Listen 8096
LogFile /home/class/csc667/Webserver/htdocs/ldir/hd.log
ScriptAlias /cgi-bin/ /home/class/csc667/Webserver/cgi-bin/
ScriptAlias /script/ /home/class/csc667/Webserver/cgi-bin2/
ScriptAlias /sa1/ /home/class/csc667/Webserver/htdocs/cdir1/
ScriptAlias /sa2/ /home/class/csc667/Webserver/htdocs/cdir2/
Alias /ab/  /home/class/csc667/Webserver/htdocs/ab1/ab2/
Alias /da1/ /home/class/csc667/Webserver/htdocs/ddir1/
Alias /da2/ /home/class/csc667/Webserver/htdocs/ddir2/
Alias /~wmac01/ /home/student/wmac01/public_html/index.html
DirectoryIndex index.html index.htm
MaxThreads 10
KeepAlive On
KeepAliveTimeOut 20
```

You can secure your directory using <directory> tag.
<directory /home/class/csc667/Webserver/htdocs/admin/>
AuthType Basic
AuthName Admin Protected Area
AuthUserFile /home/class/csc667/Webserver/.users
Require user admin
</directory>

<directory /home/class/csc667/Webserver/htdocs/general/>
AuthType Basic
AuthName General Protected Area
AuthUserFile /home/class/csc667/Webserver/.users
Require valid-user
</directory>

OR You can secure a directory by putting the authentication information inside
the .htaccess file. If you are using this method, use the keyword AccessFileName
AccessFileName .htaccess

Parsing mime.types file:

Your web server will be able to parse the given sample mime.types file that can be downloaded from the website (under the link called "Web Server Skeleton").

Note: Your web server will be able to handle any line that begins with the # sign which is a comment.

Generate and Send HTTP Responses:

Your web server should be able to appropriately generate response messages with the following response codes:

1) 200 - OK
2) 201 - Created
3) 304 - Not Modified
4) 400 - Bad Request
5) 401 - Unauthorized
6) 403 - Forbidden
7) 404 - Not Found
8) 500 - Internal Server

[Hints]

Step 1 - Getting Started

This section is intended to give you an idea of where to begin your web server project. This section is intended only to get you started, and to give you some ideas about how to debug as you go. You may design Java classes such as HTTPMsg object (extended to HTTPRequest and HTTPResponse), HTTPReader (reads an input stream from a file or socket, parses it, and stores to an HTTPRequest or HTTPResponse), MyResponse (generates a response that will be sent back to the browser - could be an error message if a requested resource is not found, or it could be an html page or image).

The classes above are for REFERENCE ONLY. You DO NOT need to follow the same implementation!

When your web server is started, it behaves in the following way:

0. First your server should parse HTTP Request. But the request doesn't have to come from Network. You can make a fake request file and use file inputstream to process the request. Later converting from file input stream to socket stream can be done with only one line change. This clearly separates your debugging process into socket programming and parsing programming. Parsing process requires to do the below:
    a. Method
    b. URI with query string or arguments
1. Read in headers
2. The server reads the mime.types file and the httpd.conf file, and creates a structure that holds all of the information.
3. The server interprets and responds to any command line options.
4. If there is a -i filename option, then your server will read the contents of the file which should contain a legal HTTP Request. If there is no filename specified, then it is assumed that requests come across the network to the specified port.
5. The server parses the request, which should request a specific html document. The server retrieves the document, places headers at the beginning, and outputs each line to the display. For example, after reading a request, your web server might display:

    The method was GET
    The request URL was /mydocs/myfile.html
    The query string was name1=value1&name2=value2
    The following headers were included:
    Content-Type: text/html
    Referer: /domain/pub/x.html
    The body was EMPTY

    The Response included the following lines:
    Return value 200
    Content-Type: text/html
    Content-Length: 2345
    Message body is:
    <HTML> ... </HTML>

Return to Section Table of Contents

Step 2 - Adding Functionality

This section is intended to give you an idea of where to begin your web server project. This section is intended only to get you started, and to give you some ideas about how to debug as you go. YOU ARE NOT REQUIRED TO SUBMIT THE WORK COMPLETED HERE AS A SEPARATE PROJECT. The

only submission will be the ENTIRE, working server. This section focuses on building on the work done in Step 1 by enabling the web server to work on a port.

Instead of reading a request from a file, as the web server in Step 1 does, your web server should now open a socket. The web server will "listen" for requests on this socket, and for each request it will parse out the HTTP Headers and the URL. The web server will use the URL to find a file in the directory relative to the directory the server is running in. For example, if the DocumentRoot for your server is assignment_html, and the request URL is /a/b/c, the server should look in assignment_html/a/b/c for the default file (since no file is specified in the URL). The server should then either return the URL to the client and close the socket, or send an appropriate error message.

The above description lists the process that your web server will go through in order to service one request. In order to continue serving requests, the web server, upon startup, should open a socket, and enter a while loop that continually listens to the port for any requests. When a request arrives at the socket, your web server will parse the request, extracting the necessary information, and return the document that is requested. There are a number of classes to be aware of to do this (the Java API is your friend):

- Socket
- ServerSocket
- FileInputStream
- BufferedInputStream
- OutputStream
- BufferedOutputStream
- BufferedReader
- BufferedWriter

Of course, which you choose to use depends on you.

Return to Section Table of Contents

Step 3 - Working With CGI and Scripts

To manage CGI, you server will fork a sub process and execute a program specified by the script file. Any results from this process will be passed back to the browser. You will need to determine the appropriate values for the environment variables and make sure that the web server passes them to the sub process. Note that you may need to pass some of these as properties to your server, as Java does not allow you to query the environment directly.

Return to Section Table of Contents

Gotchas to Watch For

When parsing paths, keep in mind that Java gives you a handy, system-INDEPENDENT method of determining which character is used as a path separator. Take a look at the Java API for the File class. Because you are writing your web server in Java, the server should run just as well on ANY platform.

Return to Section Table of Contents

Examples

Return to Section Table of Contents

Programming Hints - Simple Server

The following is a sample structure of a simple web server in Java. It is for your reference only and you are NOT required to follow it, neither the structure nor the classes. DO NOT ask for any explanation about the sample code!

```
ServerSocket ding = null;
Socket dong = null;
try {
// Read in "conf/httpd.conf" and make an object that handles all information about it
// HttpdConf, ReadHMsg are classes you have to implement.
// ServerSocket, Socket are availalbe Java classes.
HttpdConf hcf = new HttpdConf();
hcf.readHttpd("conf/httpd.conf");
ding = new ServerSocket(hcf.getListenPort());
System.out.println("Opened socket " + hcf.getListenPort());
while (true) {
// keeps listening for new clients, one at a time
try {
   dong = ding.accept(); // waits for client here
}
catch (IOException e) {
   System.out.println("Error opening socket");
   System.exit(1);
}

try {
   // Connection is built, so read stream from the socket and parse request
   ReadHMsg htr = new ReadHMsg(dong);
   // Apply aliases to the request using httpd configuration
   htr.setHttpdConf(hcf);
   // process request (get file or execute CGI)
   htr.processRequest();

   // Generate a response
   htr.writeOutput();
}

catch (NullPointerException e){
System.out.println("Error writing output");
}

}
}

catch (IOException e) {
System.out.println("Error opening socket");
System.exit(1);
}
```

Return to Section Table of Contents

Programming Hints - Running CGIs

CGI receives input from standard input and sends the output to standard output. There are several things you have to do in order to execute a CGI from the server:

- Set up a list of environment variables
- Create a process or thread to execute the CGI with the preset environment variables
- Pass the network input from the browser to the standard input
- Capture the standard output of the CGI and send back to the network to the client.

There are several to accomplish this task, e.g. you can redirect the socket input to the standard input of the child process or use some temporary input and output files. You should have learnt the techniques you need in Operating System class. The following are some classes, method or functions you might want to you. You should read the documenation to know the details. Again, you DO NOT need to follow and there might be some better methods. [This information is accurate with JDK 1.4. There is an equivalent class that performs the same task better at latest JDK]

- class **Runtime** - to excute a program in a child process
- class **Process** - can be obtained from Runtime
- method **Process.getOutputStream()** - to obtain the output stream process
- (Note: the name "Output" may confuse you)
- method **Process.getInputStream()** - to obtain the input stream of a process
- (Note: the name "Input" may confuse you)
  method **Socket.getOutputStream()** - to obtain an output stream of a socket

---

Return to Section Table of Contents

httpd.conf

```
DocumentRoot /usr/local/httpd/html
ScriptAlias /cgi-bin/ /usr/local/httpd/cgi-bin
Alias /mydocs/ /usr/local/temp/
ServerRoot /auto/home-scf-00/csci667/webserver
Listen 8080
LogFile /auto/home-scf-00/csci571/webserver/logs/log.txt
```

The document root above indicates that a request to http://host/index.html would get the file located at /usr/local/httpd/html/index.html.
The script alias above indicates that a request to http://host/cgi-bin/test-date?arg would execute the script at /usr/local/httpd/cgi-bin/test-date with argument arg.
The alias above mean that a request to http://host/mydocs/index.html would get the file located at /usr/local/temp/index.html.
The server root entry above means that the server class files are located at /auto/home-scf-00/csci667/webserver.
The listen entry above means that the web server is to listen to port 8080 when waiting for client requests.
The log file entry above means that all action performed by the server are to be recorded in the log file located at /auto/home-scf-00/csci571/webserver/logs/log.txt.

Return to Section Table of Contents

mime.types

```
text/plain txt
text/html htm html
audio/* wav
```

image/gif gif
image/jpeg jpg jpeg

This is a very small subset of the possible mime types. There are many locations on the internet where you will be able to find a comprehensive listing of mime types. The above listing indicates that a file on the server with the file extension "txt" (i.e. a file named something.txt) will be returned to the client with the Content-Type of text/plain.

Last Name, First Name **GRADING REPORT CARD**				
Phase	**Requirement**	**Points Earned**	**Maximum Points**	**Comments**
1	Parsing HTTP Request		10 %	
1	Generating Response with plain text and html		10 %	
1	Handling httpd.conf with keywords: ServerRoot, ServerAdmin, DocumentRoot, Listen, LogFile, ScriptAlias, Alias, DirectoryIndex, AccessFileName, MaxThreads		10 %	
1	Handling other MIME types such as (.sgml, .xml, .doc, xls, .pdf, .exe, unknown mime types =>7%). Handling images (.jpg, .gif, .png =>3%)		5 %	
2	CGI (birthday.pl, printenv.pl, python script)		5 %	
2	CGI (showcgi.html)		5 %	
2	Multithreading		10 %	55
2	Generating Log file with proper format		2 %	
2	Handling 200 (GET / POST) OK		4 %	61
2	Handling 201 (PUT) Created		5 %	
2	Handling 304 (Caching) Not Modified		5 %	
2	Handling 400 (Bad Request)		1 %	
2	Handling 401 (Authentication) Unauthorized		5 %	
2	Handling 403 (Permission) Forbidden		2 %	
2	Handling 404 (Page not found)		1 %	
2	Handling 500 (Internal Server Error)		1 %	
2	Documentation		10 %	
Extra Credit (Part 2)	Handling encryption/decryption password file (Authentication) using MD5		5 %	
Penalty	Late Submission (-10% per day)		0	0 days late
	Maximum Points Earned:			