Computer Networks
Fall 2013/14

Lab 1 – Web Server

Lab1 must be submitted using the course website by **28-11-2013.**
>>> No late submissions will be accepted.

Building a Multi-threaded Web Server over TCP

Goals:

- Multi-threaded web server over TCP.

Major Milestones (Web Server):

- A config.ini file that contains the following values for the server:
  - **port** – the port that the server listens to: **use port 8080.**
  - **root** – the root directory of the files: **use c:\serverroot**
  - **defaultPage** – The default page to return to the client in case a folder is mentioned in the HTTP request: **use index.html**
  - **maxThreads** – maximum responding threads allowed: **use 10.**
  - Notice: The config.ini file must be located at the server's root folder (where your .class files reside)
  - config.ini file format:

    port=[port number]
    root=[root direrctory]
    defaultPage=[default page]
    maxThreads=[max threads]

- Return the following HTTP Response codes:
  - 200 OK (everything is okay)
  - 404 Not Found (file not found)
  - 501 Not Implemented (HTTP request is not implemented/supported)
  - 400 Bad Request (server failed to parse/understand the request)
  - 500 Internal Server Error (something went wrong)
  - You may use any other response codes as you see fit

- Use "content-type: text/html", "content-type: image", "content-type: icon" for HTML, image files and icons respectively, in a response. In all the other cases, "content-type: application/octet-stream" should be used.

- Image files: .bmp, .gif, .png, .jpg.

- Icon files: .ico

- Support the HTTP methods GET and POST, and support getting parameters for both methods (store in a data structure). Also, support the methods OPTIONS, HEAD and TRACE.
  Print to the console window (System.out.println()) the HTTP request header received from the client and HTTP response header that is being sent back by the server to the client.

- Support "transfer-encoding: chunked". **Return as chunked only if the request contains the HTTP header "chunked: yes"**.

- The server's default page should return the HTML page you have written in exercise 1 with a new HTML form that contains a text area, checkbox and a submit button. Clicking the submit button causes the client to request params_info.html using a POST HTTP request:
  - The page should return to the client an HTML page that contains a table with the parameters the client sent by submitting the form (parameter name and parameter value).
  - Every parameter you cannot parse should be simply ignored.
  - **Important:** params_info.html must work also if the parameters are being sent using GET HTTP. If parameters are being sent by both GET and POST, use both. In case of 2 parameters of the same name, use the first one you parse.

- The server **must not** crash! (Exceptions must be handled and server should be able to recover).
  For example, if an exception/problem occurs while loading the server (for instance, reading the config.ini), the server should print to the console a user-friendly message and shutdown gracefully.

- Limit the number of threads that can be spawned to maxThreads.

- Client must not be able to surf "outside" the server's root directory

- Add favicon.ico to your website

- **Important:** All HTTP URL handling must be implemented by your own classes. Your classes must work directly with the sockets without any built-in classes that do the job for you!

Goals in Detail:

1. The server should be multi-threaded:

In case of a single threaded server, when Alice connects to the server, the server is busy responding to Alice instead of listening to the welcome socket (ServerSocket), therefore if Bob performs a request to the server **while** the server is responding to Alice, the server will not respond to Bob until it finishes handling Alice's request.

In a multi-threaded server, as soon as Alice connects to the server, a new thread is being created to respond a-synchronically, thus allowing the main thread of the server get back and listen on the welcome socket. This way, the server is able to get Bob's request faster.

You **can use the code from the recitation** as the basis for your multi-threaded server.

2. A config.ini file that contains the following values for the server:

"port": The port that the server listens on **(use port 8080).**
If the port value is "8080", then the welcome socket (ServerSocket) will listen to port 8080.

"root": The root directory of the files **use c:\serverroot\)**
Example:
Assume root=c:\wwwroot\

Assuming the client's browser requests the page: http://localhost/**index.html**,
Then the request can be:
GET /index.html HTTP/1.0[CRLF]
[CRLF]

So the file the server should return is:
[root]\index.html → c:\wwwroot\index.html

"defaultPage": The default page to surf to, in case no page is found in the HTTP request.
(**use index.html)**

Example:
Assume defaultPage=index.html and root=c:\wwwroot\

Assuming the client's browser requests the page: http://localhost/ .
The request can be:
GET / HTTP/1.0[CRLF]
[CRLF]

So, the page the server would look for without defaultPage is:
c:\wwwroot\
But that is a directory – not a file! So, the default page value tells the server which page (=file) to return if the client is not mentioning a specific page.

The page the server is looking for is: c:\wwwroot\index.html.

"maxThreads": Maximum responding threads. **(use 10)**

Example:

Each connection is being handled by a different thread. This means that if there are "maxThreads" connections, the server has used all its resources to respond to a new connection. A new connection will not be responded to until one of the threads will finish its job.

3. Examples:

Examples:

Requesting "C:\wwwroot\Index.html" is a page.
So the "root" value in the config file is "c:\wwwroot\".

(1) Request:
GET /index.html?x=1&y=2 HTTP/1.0[CRLF]
[CRLF]

Server's actions:

- Print using System.out.println() the request.
- Check which HTTP method is being used → in this case the method is "GET".
  If unknown method - return "501 Not Implemented".
- Check if the file c:\wwwroot\index.html exists.
  If not, return the error code: "404 Not Found".
- Read the file's content.
- Create HTTP response header:
  HTTP/1.1 200 OK[CRLF]
  content-type: text/html[CRLF]
  content-length: <the length of index.html>[CRLF]
  [CRLF]
- Print the header.
- Send full response to client (including the page content).

(2) Request:
POST /index.html?x=1&y=2 HTTP/1.0[CRLF]
Content-length: 3
[CRLF]
z=8

Server's actions:

- Print, using System.out.println(), the request header (without the string "z=8").

- Check which HTTP method is being used → in this case it is "POST".
  If unknown method, then return the code: "501 Not Implemented".

- Check if the file c:\wwwroot\index.html exists:
  If not, return error "404 Not Found".

- Read the file's content.

- Create HTTP response header:
  HTTP/1.1 200 OK[CRLF]
  content-type: text/html[CRLF]
  content-length: <the length of index.html>[CRLF]
  [CRLF]

- Print the header.

- Send full response to client (including the page content).

5. Do not allow users to surf "outside" the server's root directory.

Assume the following situation:
Root in config.ini is:
root=c:\wwwroot\

And you have the following important file:
C:\passwords\mypasses.txt

So surfing to http://localhost:8080/../passwords/mypasses.txt
Will return the page:
c:\wwwroot\..\passwords\mypasses.txt → c:\passwords\mypasses.txt

6. Add Favicon.ico to your website.

Favicon is the icon of a website, and can be seen by supported browsers (most up-to-date browsers). It is the icon you can see at the address bar in the browser.
When your browser surfs to a specific page in a website it requests the favicon.ico icon, and if it gets one, the browser will display the icon.
By default, the browser will look for favicon at the root directory of the website (for example: www.google.com/favicon.ico).

Make your own favicon.ico at: <LINK>

How to start testing:

First, run the server. The port that is mentioned in the config.ini should be openned, and the server should start listen to it (as already mentioned, the port that should be used is 8080).

Now, open your browser and surf to http://localhost:8080/ (or to http://127.0.0.1:8080/).
At this point, the browser will send an HTTP GET request to the server, and the server should parse the request and return the correct HTTP response.

To test an HTTP POST request, it is recommend to use the "HackBar" that was presented in the recitation. Using this Firefox add-on you can easily perform HTTP POST request.

Another way to test HTTP POST request is by using an HTML Form like the one added to the HTML you wrote in Exercise 1 with method="POST".

```
<form name="input" action="index.html" method="get">
```

You can check the webpage http://www.w3schools.com/html/html_forms.asp, where HTML forms are explained.

Do not forget to test the cases line the one where the requested page does not exist, such as http://localhost:8080/pageDoesntExist.html, and other cases that were not written here specifically (just QA your server as good as you can).

In case you are testing using a "localhost" server, using Wireshark will not help.
To get Wireshark to work, you have to run the server on your friend's computer and connect through a real network.
Another way is to use Firefox extension "Live HTTP Headers" that was presented in the recitation. It will show only the Headers of the requests and the response, even when working with the localhost.

Tips:

Notice, the following tips are merely recommendations.
You can write the web server however you like, as long as it works correctly!

- Initially write (or use the code from the recitation) a basic multi-threaded webserver.
  The main thread is the one listening on the welcome socket (ServerSocket) in an infinite loop.
  Every new connection opens a new thread that returns HTTP response that is hard-coded (just like in the code from the recitation).
- Test that server using a browser to see that you're getting the right response.

- Create the config.ini, and read its content. Make sure that your server listens on the port that is written in the config.ini file.

- Create a new class called "HTTPRequest".
  The new class represents an HTTP request.
  It receives in its constructor the HTTP request header and parses:
  - Type (GET/POST…)
  - Requested Page (/ , /index.html , etc.)

- Is Image – if the requested page has an extension of an image (jpg, bmp, gif…)
- Content Length that is written in the request
- Referer – The referer header
- User Agent – the user agent header
- Parameters – the parameters in the request (for example java.util.HashMap<String,String> to hold the parameters).

- Use that class to parse the request.
- After parsing the request – read the requested file (if it exists), and generate the proper response.
- Remember, if the HTML page contains embedded objects, the browser will request them right after it receives the HTML page.
- Code to read from File to byte[]

```
private byte[] readFile(File file)
{
	try
	{
		FileInputStream fis =  new FileInputStream(file);
		byte[] bFile = new byte[(int)file.length()];

		// read until the end of the stream.
		while(fis.available() != 0)
		{
			fis.read(bFile, 0, bFile.length);
		}

		return bFile;
	}
	catch(FileNotFoundException e)
	{
		// do something
	}
	catch(IOException e)
	{
		// do something
	}
}
```

- Java File class API: http://java.sun.com/j2se/1.3/docs/api/java/io/File.html
- Java HashMap: http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html

Exception Handling:

The program must not crash!

Use exception handling to recover. If you cannot recover, print what happened to the console and close the application gracefully.

Notice that possibility of crashing will drop **many** points!

Traces (a.k.a. print to console):

The following traces are **required**:

- Listening port (on startup).
- The HTTP requests arriving to the server.
- The HTTP response header returning to the browser.

You are **encouraged** to use additional traces in your code!
Traces will help you debug your application and perform a better QA.
Make sure your traces actually mean something that will help to understand better what is going on during runtime. **That can and will save you a lot of time!**

Bonus:

Feel free to add more functionality to the application as you see fit!
Good ideas will be **rewarded** with points and world-wide fame (score may be over a 100)!!!

Please write and explain all the implemented bonuses in a file named bonus.txt.
Notice that bonuses that will not be mentioned in bonus.txt will not be checked, that is, they will be ignored!

Important: We don't guarantee in advance that for extra-functionality, extra-points will be rewarded. Also, the idea is to ***add*** a new functionality ***not replace a requested functionality*** – that will lead to dropping of points! In other words, do not ignore things we require, and implement other things instead and call it a bonus! We cannot make it any clearer than that!

Discussion Board:

Please, use the forum of the course to ask questions when something is unclear.

You must make sure that you check the discussion board. If there are unclear matters about Lab 1 that were raised in the discussion board, **our answers are mandatory**, and apply to **everyone**!

You can register to receive e-mail notification from the forum.

A word from your checker:

- **Do not** create your own packages, all your classes **must be in the same default package**, and compile while all sources in the same directory.
- If you are coding using a mac, make sure you remove all hidden directories generated by your IDE.
- You can implement the lab using up to JDK1.6 .

- The checking of your code will be done from console, make sure your server works without IDE before submission!

Remember - A happy checker is a merciful checker!

Submission:

- You **must** submit a batch file called "compile.bat" - The batch file compiles your code successfully. If the batch fails to compile, **many** points will be dropped.
- You **must** submit a batch file called "run.bat" - The batch file starts your server. If the batch fails to start the server, **many** points will be dropped.

Submit the lab following the course submission rules.
The submitted zip should contain:

- Sources (*.java)
- config.ini
- compile.bat
- run.bat
- Directory called "serverroot" containing your non-code files (your HTMLs + extra files/images etc.)
- bonus.txt with the bonuses you implemented – if applicable.
- **readme.txt** that explains what exactly you have implemented and the role of each class in your server.

Notice: not submitting one of the above requested files (exclude bonus.txt) will lead to dropping of points!

That's it ☺ !
Good Luck!