# 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 relative path to server then www directory
  + default\_page – The default page to return to the client in case a folder is mentioned in the HTTP request: use index.html
  + max\_threads – maximum responding threads allowed: use 10.
  + Notice: The config.ini file must be located at the server’s root folder
  + config.ini file format:
    - port=[port number]
    - root=[root direrctory]
    - default\_page = [default page]
    - max\_threads = [max threads]
* Return the following HTTP Response codes:
  + 200 OK (everything is okay)
  + 403 Forbidden (use .htaccess file)
  + 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 Correct content-type for response
* 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 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 Correct HTTP header.
* The server should support custom redirections for example implement a redirection for /config/params\_info.html which will show an html table with all parms sent to page.
  + 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 parameters of the same name, use the first one you parse.
  + Every parameter you cannot parse should be simply ignored.
* Important! : 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 max\_threads.
* Implement Basic Security For the Web Server – for example Detect And Write To Log Directory traversal meaning Client must not be able to surf “outside” the server’s root directory and respond with a 403 forbidden.
* 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 must 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.

1. A config.ini file that contains the following values for the server:
   1. “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.
   2. “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
   3. “default\_page”: 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 default\_page 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.
   4. “max\_threads”: Maximum responding threads. (use 10)  
      Example: Each connection is being handled by a different thread. This means that if there are “max\_threads” 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.
2. Examples:
   1. 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 the request line. (debug print entire 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. (debug print entire response)  
      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] [CRLF]  
       z=8  
      Server’s actions:  
      Print the request header (without the string “z=8”). (debug print a lot ☺ )  
      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. (debug print entire response)  
      Send full response to client (including the page content).
   3. 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
   4. 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).
3. How to start testing:
   1. 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).
   2. 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.
   3. To test an HTTP POST request you can use something like “HackBar” Firefox add-on or just do it via a html form with post to params\_info.html.
4. Tips: Notice, the following tips are merely recommendations. You can write the web server however you like, as long as it works correctly!
   1. Initially 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. Test that server using a browser to see that you’re getting the right response.
   2. 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.
   3. Create a new class called “HTTPRequest”. The new class represents an HTTP request. It receives in its constructor the HTTP request header and parses:
      1. Type (GET/POST…)
      2. Requested Page (/ , /index.html , etc.)
      3. Is None HTML File – if the requested page has an extension
      4. Content Length that is written in the request
      5. Referer – The referer header
      6. User Agent – the user agent header
      7. Parameters – the parameters in the request

Use that class to parse the request.

* 1. 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.

1. Traces (a.k.a. print to console) The following traces are required:
   1. Listening port (on startup).
   2. The HTTP requests arriving to the server.
   3. The HTTP response header returning to the browser.
   4. **You are encouraged to use additional traces in your code in debug mode!**
2. Bonus:
   1. Feel free to add more functionality to the application as you see fit!
   2. Good ideas will be rewarded with world-wide fame  
      Please write and explain all the implemented bonuses in a file named bonus.txt.**the idea is to add a new functionality not replace a requested functionality.**

# Good Luck