Second Assignment - Datanet

Nikolaj Dybdahl Rathcke
University of Copenhagen, DIKU
Universitetsparken 1
Copenhagen, Denmark
rfq695@alumni.ku.dk

ABSTRACT

This is an implementation of a web server using socket programming that is able to serve files, more specifically a implementation of the HTTP GET method.
This is completed in the programming language **Ruby**. It is a very simple web server, only supporting the GET method and raising few error codes in case something goes wrong. The finished product is able to serve files if they exist or if a directory is accessed, it will serve an index file in that directory if one exists. Otherwise, a list of the files and folders will be shown.
The implementation succesfully implements these features, but there are other issues, most importantly security issues that will have to be looked at before the server could go live.

1. INTRODUCTION

In this assignment, a HTTP compatible web server using socket programming is implemented.
The web server should be able to serve files, which it does by listening for a request on a socket. In this implementation it listen on port 1339. When a request is recieved, the request is read and then parses data accordingly to the request. That is, it can act on the request or it can produce an error. This is implemented in the language **Ruby** and is discussed in depth in the following section.
The entire implementation can be seen in appendix A.

2. WEB SERVER

2.1 Design

It will be written in the programming language **Ruby** using only the standard library `socket`, where the `TCPSocket` is taken in use.
It will work by receiving a request, verifying if it is a GET request since this will be the only supported method. Then, treating it as a string, it will get the information needed to serve the right files to the client. Before the file is served, it passes a header.
Errors codes will be implemented but only 404 in case a wrong directory or file is accessed and 400 if it is a bad request.
The webserver will not support persistent connections.

2.2 Implementation

The web server is only able to process HTTP GET requests. By listening for requests, it will handle the request as a string. First verifying it is actually a GET request, and then simply get the path that is requested. This is then applied to the underlying file system. It will then either show the content of the file in the path if a file can be accessed directly. Alternatively, if a folder is accessed (URL ending in '/'), it will look for an "index.html" file and serve this file if it exists in the directory. Otherwise it will list all entries in that directory. Appropiate errors codes are returned if a request does not match either of these cases.
These are the codes that can occur in the response headers.

- 404 (Not Found) is raised if a non existent path is accessed.
- 400 (Bad Request) is raised if it is an unsupported request.
- 200 (OK) is given if the request could be fullfilled.

The connection to the client is always closed afterwards, no matter if it is a bad request or if the request could be processed.

2.3 Reponse Headers

The response header is very simple, only providing the appropiate code and indicating that the connection will be closed, which it always will be as it does not support persistent connections.

2.4 Later Implementations

The first check is if it is a GET request. This makes it easy to implement other methods in the future as this check can simply be replaced by a switch statement that reacts to the given method.
Another idea to extend the web server, is, in the current implementation the headers are hardcoded. It is possible to make a function that generates these responses, thus avoiding having to write it everytime a request has been processed. It also makes it easier to to parse more info in the response header.
All in all this will provide a better overview.

2.5 Tests

The tests are used with the handed out example website ”jqapi”.
The tests that were completed include using a browser and trying to see if it behaved as expected in the cases when requesting:

- Files that exist (should serve the file).
- Files that do not exist (should raise a 404).
- Folders that exists with an index.html file (should serve the index file).
- Folders without an index.html file (should display contents of the folder).
- Folders that do not exist (should raise a 404).

The same was tried when using telnet, which is a virtual terminal connection. All these behaved as intended. The response header also worked as intended.
A stress test where there are lots of requests, thus raising the traffic, was unfortunately not tried but it would have given a good indication of the stability of the server.

3. CONCLUSIONS

Even though the implementation works as expected, there are still some issues that should be looked at. Especially security is not thought of at all, which means it is likely very easy to 'break into' things on the web server that should obviously not be allowed.
Also testing could have been thorough since it, as of now, only is known to work as intended for a single client.
So even though the task described in 2.1 is implemented, these flaws make the web server quite unreliable and useless, hence the web server (despite the very few functionalities) can not be set live, before these things are looked into.

APPENDIX
A. SOCKET.RB

```
require 'socket'

serv = TCPServer.new('localhost', 1339)

loop do
  Thread.start(serv.accept) do |s|
      req = s.gets
      req = req.split
      if (req[0] == "GET")
        if (req[1][-1] == "/")
          dir = "." + req[1]
          if !File.exist?(dir)
            s.write "HTTP/1.1 404 Not Found\r\nConnection: close\r\n\r\n"
            s.puts "Page not found: 404"
            s.close
          end
          path = dir + "index.html"
          if File.exists?(path)
            file = File.new(path, "r")
            s.write "HTTP/1.1 200 OK\r\nConnection: close\r\n\r\n"
            while (line = file.gets)
              s.puts line
            end
            file.close
          else
            s.write "HTTP/1.1 200 OK\r\nConnection: close\r\n\r\n"
            Dir.entries(dir).each do |f|
              if File.directory?(f) and !(f =='.' || f == '..')
                s.puts("#{f}/")
              else
                if !(f =='.' || f == '..')
                  s.puts("#{f}")
                end
              end
            end
          end
        else
          path = "." + req[1]
          if File.exists?(path)
            file = File.new(path, "r")
            s.write "HTTP/1.1 200 OK\r\nConnection: close\r\n\r\n"
            while (line = file.gets)
              s.puts line
            end
            file.close
          else
            s.write "HTTP/1.1 404 Not Found\r\nConnection: close\r\n\r\n"
            s.puts "Page not found: 404"
          end
        end
      else
        s.write "HTTP/1.1 400 Bad Request\r\nConnection: close\r\n\r\n"
        s.puts "Bad request: 400"
      end
    s.close
  end
end
```