Lab 2 – HTTP and SMTP

Assistants

Junchao Wang (J.Wang2@uva.nl)

Rex Valkering (r.a.valkering@uva.nl)

Daniël Louwrink (daniel.louwrink@student.uva.nl)

Daan Kruis (daan.kruis@student.uva.nl)

Lab dates

09 and 14 Sep. 2016

Deadline

Sep. 15 at 23:59 CEST

Total points

20 + 5 bonus

This lab must be done individually

Abstract

In this assignment, you will learn the basics of TCP socket programming and the HTTP and SMTP protocols, which are essential for browsing the web and sending email.

Preparation

For this assignment you must use Python 2. This should already be installed on your computer. If it is not, you can download it from https://www.python.org/ or you can install it using your package manager if you are running a GNU/Linux-based operating system.

Read the documentation on the socket module included with the standard Python distribution. Python's socket module is a direct translation of the Berkeley Sockets API for C. You can find examples on socket programming in Python in your textbook, section 2.6 or you can use Google. You can find the documentation for the socket module at http://docs.python.org/library/socket.html.

Submission

IMPORTANT: Make sure your code is PEP8 compliant.

You can check your code by installing a PEP8 checker such as the command-line utility `pep8` or http://pep8online.com. Points will be subtracted if your code is badly formatted.

Submit your working code in an archive called `lab2-<last_name>_<first_letter_of_first_name>.zip` (or `tar.gz`). For example, `lab2-Louwrink_D.zip`. It should contain the following files:

- `server.py` (task 1 + 2)
- `cgi-bin/test.py` (task 2)
- `cgi-bin/trace.py` (task 2)
- `cgi-bin/email.py` (task 3, if you implemented SMTP)
- `public_html/*`

Please do not forget to include your name, your student number and a short description of the program at the top of each file.

Task 1 – Basic HTTP Server (12 pts)

In this first task you will need to implement a simple web server that can handle a single client at a time (you will also learn how to handle multiple clients). You must parse HTTP requests sent by the client (web browser) and send appropriate responses. You may ignore any request headers for now. You can read more about the HTTP specification at http://tools.ietf.org/html/rfc2616.

HTTP implements a variety of methods including GET, POST, PUT, DELETE and many more. In this task you will only need to respond to GET requests. If the client tries any other method, you should follow the HTTP specification and throw a 501 error (Not Implemented).

When you receive a request for a file, your server should go looking for that file. If the file is found, you should serve it and respond with a code 200 OK, which means everything went well. If you could not find the file, respond with a 404 Not Found.

After the response is sent close the connection and wait for the next client. HTTP responses normally contain a number of response headers in order to control the transaction. Implement (at least) the following response headers: `Connection`, `Content-Type`, `Content-Length` and `Server`. For the `Server` header, you can make up a nice name for your web server.

To help you test, we have included a sample website in the `public_html` directory with an index.html which you can try to access.

Tips

- If you're a bit confused as to what kind of request the browser will send you, create a socket and point your browser at it. It will make a request and the socket will allow you to see exactly how the request is built.
- You're not allowed to use the HTTP module in Python for obvious reasons.
- The standard page includes an image. Because this image triggers a separate HTTP request, this is a real good way to check if your server can handle multiple connections well.
- Be sure to use the right line endings!

Task 2 – Common Gateway Interface (8 pts)

In this task you will make your website dynamic. CGI (common gateway interface, http://tools.ietf.org/html/rfc3875) is a standard interface for web servers to serve dynamic content rather than static pages. It works by executing scripts as external processes and redirecting `stdout` as a response back to the client. The parameters are given by a number of environment variables that the script can read and process. In this case, we will use CGI to use Python scripts to generate content.

Your task is to implement a subset of CGI, as follows. Whenever the client requests a URI starting with `/cgi-bin`, you must execute the requested Python script from the `/cgi-bin` directory instead of sending its contents directly (this is important for when you start saving your SQL passwords and what not in there, you don't want to accidentally serve those). The HTTP server must output the HTTP status line, but the headers and contents of the response are output separately by the script. The script outputs by printing to `stdout`, so the server must pipe the script process's `stdout` to the client socket.

You must pass the following environment variables to the script:

- `DOCUMENT_ROOT:` The `public_html` directory.
- `REQUEST_METHOD`: The HTTP request method that was used to access the script.
- `REQUEST_URI`: The requested URI, without query string.
- `QUERY_STRING`: The query string following the URI, if any. For example, in `/cgi-bin/hello.py?name=pietje` the part before the ? is the URI, and the part after is the query string.

- PATH: The standard PATH environment variable copied from the HTTP server's own environment.
- REMOTE_ADDR: The IP address of the visitor.

  For testing, write a script accessed from /cgi-bin/test.py that outputs all environment variables passed to it. You should then be able to view that file by pointing your browser to http://localhost:8080/cgi-bin/test.py (adjusting the port is required).

Secondly, create a Python program trace.py that takes an IP address as a GET argument and outputs the traceroute to that IP address through the CGI. You should use the traceroute built in to your system, don't try to implement your own.

Tips

- Use the subprocess module to execute the Python CGI scripts.
- For some real *Inception*-like programming, you could create a Python script that executes the fortune program and pipes the output into cowsay. Who knows, a cow giving good advise through the internet could just be the next big thing.

Task 3 (BONUS) – Authenticated SMTP (5 bonus pts)

In this task you will implement *sending* mail using the SMTP protocol (http://tools.ietf.org/html/rfc5321).

To get a feel for how SMTP works, you might want to try an SMTP transaction yourself. You can use the UvA server. In a terminal, try nc smtp.uva.nl 587. Try the example below:

```
> nc smtp.uva.nl 587
..... response from the server ...
> HELO myserver.mydomain.nl
.... response from the server ...
> MAIL FROM my.account@student.uva.nl
.... reponse from the server .....
> QUIT
```

We have included a web form to send mail in the file /public_html/email.html. Please take your time to inspect the file. Most notable, look for the line that opens the form tag and take note of the action and method attributes. This tells your browser what to do when you click the send button at the bottom. In this case, your browser will send a GET request with the required data to /cgi-bin/email.py.

You will need to write a CGI script in Python which will send mail using SMTP, including authentication. Again, Python includes an SMTP module but you're not allowed to use it. You will once more need to use socket programming to create a connection between you and the SMTP server.

The task is to write a CGI script to send emails using the SMTP protocol including authentication. Use the STARTTLS (http://tools.ietf.org/html/rfc3207) and AUTH (http://tools.ietf.org/html/rfc4616) commands to send emails over an encrypted and authenticated channel.

Use smtp.uva.nl as a test server on port 587. Use the username and password fields in the email web form for the authentication. At every step in the protocol you must print the server's responses, check the response codes and quit in case of an error.

If all goes well, you should be able to send mail through your very own SMTP client which runs on your very own HTTP server which are connected through your very own CGI. Well done!

Tips:

- You will use TLS to secure the channel. For once, you won't have to implement it yourself! Just import the ssl module and use the wrap_socket function from there.

- You may want to test your Python SMTP implementation without connecting it to your CGI first. To do this, you can add the `if __name__ == '__main__':` clause to your file. Any code that is in that block will only be executed is you execute the Python file directly (or through the CGI, as that also directly executes the script). Here, you should insert the code that reads the arguments from `sys.argv`.