Task 2.1: Web Server

For the second task of this practical, you will be building a simple HTTP web server. Web Servers are a fundamental part of the Internet; they serve the web pages and content that we are all familiar with. You should have learned about web servers and the operation of the HTTP protocol in Lecture 3: *Web & HTTP*. Fundamentally, a web server receives a HTTP GET request for an object (usually a file), located on the web server. Once it receives this request, the web server will respond by returning this object back to the requester.

As with the previous task, we will be using network sockets to build our application and to interact with the network. The Web Server differs from the ICMP Ping application in that it will bind to an explicit socket, identified by a *port* number. This allows the Web Server to listen constantly for incoming requests, responding to each in turn. HTTP traffic is usually bound for port 80, with port 8080 a frequently used alternative. For the purposes of this application, we suggest you bind to a high numbered port above 1024; these are unprivileged sockets, which reduces the likelihood of conflict. For interest, application developers can register port numbers with the Internet Assigned Numbers Authority (IANA), reserving them for their application's use:

https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

The application you build should respond to HTTP GET requests, and should be built to HTTP/1.1 specification, as defined in RFC2616. These requests will contain a *Request-URI*, which is used to define the path to the object requested. For example, a request with a URI of 127.0.0.1:8000/index.html, will serve a file name index.html found in the same directory as the Python script itself. The URI is broken down as follows:

- 127.0.0.1: Hostname of web server
- 8000: Port number that web server has bound to
- index.html: File to be served

On successfully finding and loading the file, it will be sent back to client with the appropriate header. This will contain the *Status-Code* 200, meaning that the file has been found OK, and that it will be delivered to the client as expected. Your implementation needs only serve files from the same directory in which the Python script is executed.

Implementation Tips

As before, we have provided skeleton code that can be used to aid you in this task. This can be found on the course's Moodle page. It contains suggested functions, as well as an overview of functionality to be implemented by each. These are given as comments and are to be treated as **guidance only**. Note that you may have to change the parameters passed to each function as you advance with the task. An example HTML file (index.html) is also provided in the same location. The following Python library and its documentation may also serve as a pointer to helpful functions:

https://docs.python.org/3/library/socket.html

https://docs.python.org/3/library/socketserver.html

As a baseline, your implementation needs only to be single-threaded. This allows a maximum of one request to be handled at a time.

Debugging and Testing

To test your web server application, you must generate a valid request. There are a number of tools to achieve this. For example, the wget utility can be used to generate a request (presuming your web server is running on port 8000):

```
wget 127.0.0.1:8000/index.html
```

wget is a free tool to download files and crawl websites via the command line. To run wget, you need to download and install manually.

An equally valid and simpler method is to use a web browser. Put an HTML file (e.g., index.html) in the same directory that the server is in. Run the server program. Open a browser and provide the corresponding URL. For example,

127.0.0.1:8000/index.html

Then, the browser should display the contents of index.html. Note that you need to replace the port number after the colon with whatever port you used in the server code.

If you are unsure about what a HTTP request should look like, Wireshark can again be used to inspect packets. This includes both the HTTP request and response. This will help you debugging the form and structure of your requests, identifying any issues that may be present. If you are still using Wireshark from the previous task, make sure to remove the icmp filter! http can be used instead. It will also be necessary to capture packets on the loopback interface (lo), rather than the external interface (eth0).

If you wish to observe how a Web Server should behave (and examine the packets generated by such), Python provides a handy way of starting a very simple HTTP server implementation:

```
python -m SimpleHTTPServer # for Python 2

python -m http.server # for Python 3
```

Requests to this server can be made using the methods described previously.

Marking Criteria

For this task, you will be awarded marks for building a functioning Web Server, capable of handling requests for content. You should be able to demonstrate that, given a request, the Web Server will return the correct file, as well as producing a well-formed response header with protocol version and response code set correctly.

Potential additional features include:

- Binding the Web Server to a configurable port, defined as an optional argument
- When a requested file is not available on the server, return a response with

the status code *Not Found* (404)

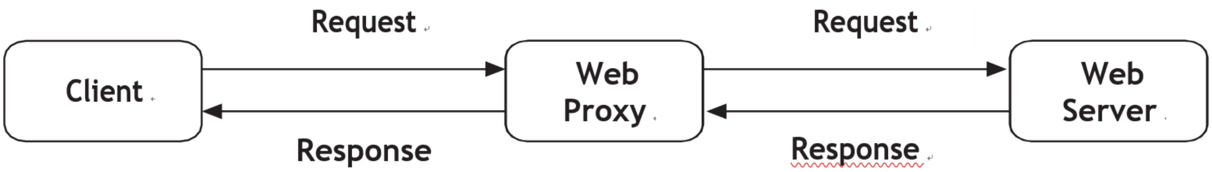
- Create a multithreaded server implementation, capable of handling multiple concurrent connections
- Write your own HTTP client to query the web server (this should be submitted as an additional standalone Python file)

Please note that the features mentioned above are considered supplementary; you do not have to complete them all, and you can still receive a satisfactory mark without completing *any* of them. They are intentionally challenging and designed to stretch you.

Task 2.2: Web Proxy

Building on the Web Server described in Task 2.1, this task is concerned with building a Web Proxy. This operates in much the same way as a web server, with one significant difference: once configured to use the Proxy Cache application, a client will make all requests for content **via** this proxy.

Normally, when we make a request (without a Web Proxy), the requests travels from the host machine to the destination. The Web Server then processes the request and sends back a response message to the requesting client. However, when we use a Web Proxy, we place this additional application between the client and the web server. Now, both the request message sent by the client, and the response message delivered by the web server, pass through the Web Proxy. In other words, the client requests the objects via the Web Proxy. The Web Proxy will forward the client's request to the web server. The web server will then generate a response message and deliver it to the proxy server, which in turn sends it to the client. The message flow is as below:



As with the Web Server, your Web Proxy application is only expected to handle HTTP/1.1 GET requests. Similarly, the Web Proxy will also bind to a specific port (this can be the same as the Web Server), and continue to listen on this port until stopped.

Debugging and Testing

As with Task 2.1, there are a number of ways to test your Web Proxy. For example, to generate requests using wget, we can use the following:

```
wget lancaster.ac.uk -e use_proxy=yes -e http_proxy=127.0.0.1:8000
```

This assumes that the Web Proxy is running on the local machine and bound to port 8000. In this case, the URL requested from the proxy is lancaster.ac.uk.

A web browser can also be used to the same effect. Many web browsers support the use of a web proxy through configuration. For example, the Chromium browser can be started in the following way to utilise the Web Proxy:

```
chromium-browser --proxy-server="127.0.0.1:8000"
```

A caveat when testing your Web Proxy: some websites have enabled HTTP Strict Transport Security (HSTS) (RFC6797). This forces clients (including both wget and a web browser) to use HTTPS rather than HTTP. HTTPS is a secure version of HTTP, but we will consider this out of scope for this practical. To check if a website has this feature enabled, use the following tool, and ensure that the website you are using for testing purposes is **not** listed:

https://hstspreload.org/

As with the other tasks, Wireshark can be used to capture and investigate packets sent to and from your proxy. As the proxy will be receiving local requests from the web browser, as well as making external requests to fetch content, it is necessary to capture packets on both the external (eth0) and loopback (lo) interfaces.

Marking Criteria

For this task, the majority of marks will be awarded for demonstrating a working Web Proxy. You are expected to show the functionality of such through the use of either wget or a properly configured web browser. Note that you are not expected to demonstrate the Web Proxy using a website with HSTS enabled (see above).

Additional marks will be awarded for the following aspects:

- Binding the Web Proxy to a configurable port, defined as an optional argument
- Support for other HTTP request types (PUT, DELETE, etc.)
- Object caching: A typical Web Proxy will cache the web pages each time the client makes a particular request for the first time. The basic functionality of caching works as follows. When the proxy gets a request, it checks if the requested object is cached, and if yes, it returns the object from the cache, without contacting the server. If the object is not cached, the proxy retrieves the object from the server, returns it to the client and caches a copy for future requests. In practice, the proxy server must verify that the cached responses are still valid and that they are the correct responses to the client's requests. You can read more about caching and how it is handled in HTTP in RFC2068. Add the simple caching functionality described above. You do not need to implement any replacement or validation policies. Your implementation, however, will need to be able to write responses to the disk (i.e., the cache) and fetch them from the disk when you get a cache hit. For this you need to implement some internal data structure in the proxy to keep track of which objects are cached and where they are on the disk. You can keep this data structure in main memory; there is no need to make it persist.

As before, please note that the features mentioned above are considered supplementary; you do not have to complete them all, and you can still receive a satisfactory mark without completing *any* of them. They are intentionally challenging and designed to stretch you.