Networked Systems (H) Laboratory Exercise 2: TCP/IP Networking in C – A Simple Web Server

Dr Colin Perkins
School of Computing Science
University of Glasgow
https://csperkins.org/teaching/2018-2019/networked-systems/

17 January 2019

Introduction

The laboratory exercise last week introduced you to network programming using the Berkeley Sockets API, by implementing a simple networked "Hello, world" application. In this exercise, you will extend and debug a simple web server application that was previously implemented in C using Berkeley Sockets.

This is a formative exercise and is not directly assessed. Note, however, that one of the questions on the final exam for this course will relate to the material covered in this exercise.

Background

In this lab you will debug and extend a simple web server. A web server is a program that listens for, accepts, and responds to requests made by web clients (i.e., browsers) using the hypertext transport protocol (HTTP) to deliver web pages. To complete this exercise you must understand the basic operation of the HTTP protocol, and the way it's implemented in the sample web server.

The Hypertext Transport Protocol

A web browser uses the Hypertext Transport Protocol (HTTP) to retrieve pages from a web server. The browser makes a TCP/IP connection to the web server, sends an HTTP request for the desired web page over that connection, reads the response back, and then displays the page. HTTP requests and responses are text-based, making the network protocol human-readable, and straight-forward to understand.

In version 1.1 of HTTP, a request comprises a single line command (the "method"), followed by one or more header lines containing additional information. To retrieve a page, a web browser uses the `GET` method, specifying the page to retrieve and the version of the HTTP protocol used (this exercise uses the `HTTP/1.1` protocol). For example, a browser would send the method `GET /index.html HTTP/1.1` to retrieve the page `/index.html` from a server. Following the `GET` method line will be a sequence of header lines, giving more information about the request and the capabilities of the browser. One of these header lines will be a `Host:` header, giving the name of the site, for example `Host: www.gla.ac.uk`, used to allow a single servers to host more than one site. After the headers is a blank line, indicating end of request.

For example, to fetch the main University web page (`http://www.gla.ac.uk/index.html`), a browser would make a TCP/IP connection to `www.gla.ac.uk` port 80, and send the following request:

```
GET /index.html HTTP/1.1
Host: www.gla.ac.uk
```

Note that each line ends with a carriage return ('\r') followed by a new line ('\n'), and the whole request ends with a blank line (i.e., a line containing nothing but the \r\n end of line marker). The example above is a minimal HTTP request. A web browser will usually include other headers, in addition to the `Host:` header, to control the connection, indicate support for particular file formats and languages, to convey cookies, and so on.

When it receives an HTTP GET request for a web page that exists, a web server will reply with a `HTTP/1.1 200 OK` response, followed by more header lines providing information about the response, a blank line, and then the contents of the page to the displayed. The headers lines should include a `Content-Length:` header, which specifies the size of the page in bytes, and a `Content-Type:` header that describes the format of the page. The server can also include other header lines, to specify additional information about the response. As with the request, each header line ends with a carriage return followed by a new line. Finally, a blank line separates the headers from the page content. An example of a response follows ("..." indicates omitted text):

```
HTTP/1.1 200 OK
Date: Tue, 12 Jan 2010 11:18:30 GMT
Server: Apache/1.3.34 (Unix) PHP/4.4.2
Last-Modified: Tue, 12 Jan 2010 09:59:31 GMT
ETag: "1a-3d4e-4b4c4803"
Accept-Ranges: bytes
Content-Length: 15694
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <title>University of Glasgow :: Glasgow, Scotland, UK</title>
</head>
<body>
  ...
</body>
</html>
```

In this example, the "Content-Length:" is 15694 bytes and the "Content-Type:" is text/html, meaning that there are exactly 15694 bytes of HTML text in the body of the response (starting with the "<" of the "<!DOCTYPE" line, after the blank line indicating end-of-header, and finishing with the ">" of the "</html>" line).

The "Content-Type:" header will take different values depending on the type of file returned. Commonly used values are:

Filename:	Content-Type:
*.html, *.htm	`Content-Type: text/html`
*.css	`Content-Type: text/css`
*.txt	`Content-Type: text/plain`
*.jpg, *.jpeg	`Content-Type: image/jpeg`
*.gif	`Content-Type: image/gif`
*.png	`Content-Type: image/png`
(unknown)	`Content-Type: application/octet-stream`

The IANA maintains the master list of standard content type values. It is available from their website at `http://www.iana.org/assignments/media-types`.

If the browser requests a non-existing file, the server will respond with a `HTTP/1.1 404 Not Found` response. In this case, the body of the response is the error page to display, and the headers give information about the error. An example might be:

```
HTTP/1.1 404 Not Found
Date: Tue, 20 Jan 2009 10:31:56 GMT
Server: Apache/2.0.46 (Scientific Linux)
Content-Length: 300
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
<title>404 Not Found</title>
...
</body>
</html>
```

Other types of response are possible, distinguished by the numeric code in the first line of the response.

A browser will open one or more connections to a server for each web page. On each connection, the browser will send a request, then wait for and process the response from the server. It may then send another request on the same connection, perhaps after waiting some time, or close the connection. If the server wants the browser to close the connection immediately, it can include a `Connection: close` header in its response to the request.

A Sample Web Server

The zip archive associated with this lab exercise contains the source code for a simple web server, as an example of a networked application using TCP/IP. The server is written in C, using the Berkeley Sockets API, and runs on the Linux machines in the lab. The zip archive also contains a Makefile, and a small sample website.

The web server implementation uses a single thread to `accept()` connections from new clients. It transfers each new connection to a worker thread from a thread pool for processing, by passing the file descriptor to the thread. Once a worker thread gets the file descriptor, it enters a loop where it reads the request, sends the response, and repeats until the browser closes the connection. The thread then blocks until it's passed a new connection. The web server is a minimal implementation, that contains only the bare minimum features needed to serve a simple website.

Download the file `lab02.zip` from the course website. Extract the zip archive, to create a directory called `lab02`. Open a shell in that directory, and run `make` to build the web server. Then, run `./wserver` to start the server. The server listens for connections on port 8080 of the host on which it's running. For example, if you run the server on host `bo720-1-01u.dcs.gla.ac.uk`, it will be reachable by the URL `http://bo720-1-01u.dcs.gla.ac.uk:8080/`. Browse the sample website using the server you have just built, by running a browser and opening the appropriate URL. Review the provided code, and familiarise yourself with its operation. You will notice that the web server provided is buggy and incomplete, and will not correctly serve image files. If you don't understand the HTTP protocol, or its implementation in the server, then ask one of the demonstrators.

Formative Exercise 2: Web Server

In Formative Exercise 2 you will debug and extend the web server provided in the `lab02.zip` file. There are three parts to the exercise, as follows:

1. Debug the web server, by modifying the file `wserver.c` so it correctly sends JPEG format images back to the browser. The sample website provided with the server contains some JPEG format images. You have successfully solved this part of the exercise when you can browse this sample

site, served by the web server, and have your browser display all the images, and indicate that it has finished downloading the page.

2. Modify the server code, in `wserver.c`, so that if a browser requests a URL representing a directory, and the file `index.html` exists in that directory, the server returns a redirect asking the browser to fetch the `index.html` file instead. This is done by sending a `307 Temporary Redirect` HTTP response. For example, if the browser requests `http://example.com/foo/`, the server would redirect it to `http://example.com/foo/index.html` by sending a response like:

```
HTTP/1.1 307 Temporary Redirect
Location: /foo/index.html
Content-Length: 143
Content-Type: text/html

<html>
  <head>
    <title>Redirected</title>
  </head>
  <body>
    <p>Redirecting ...</p>
  </body>
</html>
```

   The `Location:` header gives the location to which the browser is being redirected. The HTML body is for backwards compatibility, and is displayed by old browsers that don't understand the redirect response. Ensure the redirect works whether or not the URL representing the directory ended in a `/` character.

3. Modify the server code, in `wserver.c`, so that if a browser requests a URL representing a directory, and the file `index.html` does not exist in that directory, it returns instead a dynamically generated HTML page (a `200 OK` response) that contains a listing of the contents of the corresponding directory. Each entry in the listing should be a link to the corresponding file or directory. Be careful to avoid race conditions when generating the directory listing.

   Hint: use the `opendir()`, `readdir()`, and `closedir()` functions from the Linux standard library to read a directory listing.

To complete this exercise you will need to modify the `wserver.c` file included in the zip archive. You do not need to change any of the other files in the archive, and should make only the minimum number of changes required. Ensure any modifications you make to the `wserver.c` file match the code style of the surrounding code.

Your modified `wserver.c` file should compile cleanly, without any warnings or errors, using the provided `Makefile`, and run on the Linux machines in the labs.

This is a formative exercise and is not directly assessed. You do not need to submit your solution for assessment. Note, however, that one of the questions on the final exam for this course will focus on the HTTP protocol and the principles of operation of a simple web server so it is to your benefit to complete this exercise.

- + -