Networked Systems 3 Exercise 1: Web Client

Dr Colin Perkins

27 January 2010

1 Introduction

The laboratory sessions for Networked Systems 3 aim to introduce you to network programming in C on Linux systems. There are weekly labs for this course, in which you will complete several exercises. These exercises are assessed, and count for 20% of the marks for this course in total. The exercises are as follows:

Exercise 0:	Warm-up exercise – networked "Hello, world!"
Exercise 1:	Write the world's most basic web browser
Exercise 2:	Write a basic web server

Two weeks are allocated to Exercise 0, three weeks to Exercise 1, and four weeks to Exercise 2. The exercises build on your knowledge of C programming and pthreads, from the AP3 course last semester, and the introduction to the Berkeley Sockets API given at the start of this course. This is exercise 1, to write the world's most basic web browser.

2 Background

A web browser uses the HyperText Transport Protocol (HTTP) to retrieve pages from a web server. The browser makes a TCP/IP connection to port 80 of the web server, sends an HTTP request over that connection, and reads the response back. Both HTTP requests and responses are text-based, making the network protocol relatively straight-forward to understand.

Requests in HTTP comprise a single command line, followed by one or more header lines containing additional information. To retrieve a page, use the `GET` command, specifying the page to retrieve, and the version of the HTTP protocol used (the current version is `HTTP/1.1`). For example send `GET /index.html HTTP/1.1` to retrieve the page `/index.html` from the server. The `GET` request must be followed by a header to specify the name of the web site, for example `Host: www.gla.ac.uk` (in case there are several sites hosted on the same server). The headers are followed with a blank line, to indicate the end of the request. For example, to retrieve the main page from the University web server (`http://www.gla.ac.uk/index.html`), you 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 is terminated by a blank line (i.e. a line containing nothing but the \r\n end of line marker).

On receiving a request such as this, the web server should reply with an `HTTP/1.1 200 OK` response, several header lines providing information about the response, a blank line, and then the HTML content of the page. As with the request, each line ends with a carriage return followed by a new line. Finally the server will close the connection. An example of the type of response you can expect follows ("..." indicates that some text has been elided).

```
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" lang="en">
<head>
...
</body>
</html>
```

If a request is made for a non-existing file, the server will respond with a 404 "file not found" error, for example:

```
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.

3 The World's Most Basic Web Browser

The aim of this exercise is to write the world's most basic web browser program. This should be written in C using the Berkeley Sockets API, and should be developed in stages as explained below. You have three weeks to complete this exercise.

3.1 Basic Protocol Operation

Having seen the basic operation of HTTP, your first task is to write a C program to retrieve a page from the main University web site. You should write a program `wbrowser` that connects to the University web server; sends an

HTTP request as shown in Section 2; retrieves and prints the complete HTTP response, headers, and the HTML data sent by the server; then closes the connection. Your `wbrowser` program should take a single URL as its command line argument, indicating which server to contact and what page to retrieve. For example, if started using the command:

```
wbrowser http://www.gla.ac.uk/index.html
```

your program should make a TCP/IP connection to the server named www.gla.ac.uk on port 80, request the file `/index.html`, and display the data sent back from the server. To test your program, retrieve and display several pages from the University website.

Don't forget to specify the correct `Host:` header and filename in the request you send to the server. Be sure to test the case where a URL of the form `http://www.gla.ac.uk/` is provided, and also look at the results returned if you request a page that doesn't exist. Make sure you understand what you are sending to the server, and what it sends in response. Ask for help if you don't understand the operation of the protocol.

3.2 Web Proxies

The University requires that all requests for off-campus web pages go through a caching web proxy, rather than being made directly. To demonstrate this, use your `wbrowser` program to try to retrieve the BBC news website (`http://news.bbc.co.uk/`), and look at the response you get.

Extend your `wbrowser` program to take a single URL as its command line argument, and retrieve it by making a direct connection to the server if the hostname part of the URL ends in `gla.ac.uk`, or retrieve it using the web proxy otherwise. The web proxy server is called wwwcache.gla.ac.uk. You can retrieve pages using the proxy by making a TCP/IP connection to it on port 8080 and sending a request in the same way you do to retrieve pages directly, except that the complete URL must be specified on the "GET" line of the request. For example, if started as `wbrowser http://news.bbc.co.uk/` your program would connect to wwwcache.gla.ac.uk port 8080, and send a request as follows:

```
GET http://news.bbc.co.uk/ HTTP/1.1
Host: news.bbc.co.uk
```

However, if started as `wbrowser http://www.gla.ac.uk/` your program will make a direct connection to `www.gla.ac.uk`, in exactly the same way as the earlier version.

Test your program using several different web-sites, both on and off campus, and check that it can retrieve pages both using the web proxy, and directly (for on-campus servers).

3.3 Parsing the Response (1)

The response you get from the web server is split into two parts: headers, and body. The headers come first, starting with a response code (the `HTTP/1.1 ...` line), and continuing with header lines followed by a single blank line. Each header will be on a line of its own, and after the response code, each header will be of the form `Header-Name: value`. Following the headers, the body comprises the rest of the response. The headers tell you if the request was successful, and provide information about the body.

The next part of this exercise is to begin processing the headers. You should extend your `wbrowser` program to parse the headers in the response it receivers from the server, and display only the body of the response, with none of the headers. Depending on how the server has been configured, there are two ways to do this.

In the simplest case, the server will include a `Content-Length:` header, telling you the size (in bytes) of the body. In this case, your program will parse the data returned by the the server, reading the `Content-Length:` header as it does so, and then finding the blank line that marks the end of the headers. It will then read, and display, the body of the response. The number of bytes of data to read from the server for the body will be exactly the value specified in the `Content-Length:` header. The connection should be closed once the appropriate amount of data has been read for the body. For example, if your program received a response like that beginning `HTTP/1.1 200 OK` shown on page 2 of this handout, it would parse the headers until it reached the end of the terminating blank line, then read and display exactly 15694 bytes of body data. The body will start with the `<!DOCTYPE` line, and finish at the end of `</html>` in this case.

Alternatively, the server may use chunked transfer encoding. In this case, the `Content-Length:` header will be missing, and a `Transfer-Encoding: chunked` header will be present instead. The body of the response, after the blank line signalling the end of the headers, will comprise a number of chunks of data. Each chunk begins with a count of the amount of data that follows in hexadecimal, followed by an end of line marker (`\r\n`), then a chunk of the actual data. The end of the data is indicated by a chunk with size zero. When reading a chunked body, you need to reassemble the data from the separate pieces, removing the chunk headers before you display the body. Wikipedia (`http://en.wikipedia.org/wiki/Chunked_transfer_encoding`) describes the chunked transfer encoding in more detail.

Test your program with several web sites, to demonstrate that it correctly prints only the body data returned from the server, and none of the headers, for servers using both the `Content-Length:` header and the chunked transfer encoding.

3.4 Parsing the Response (2)

Another important header is the `Content-Type:` line. This tells you the format of the data. Common values for the Content-Type you might see are `text/html` for an HTML web page, `text/css` for a web style sheet, `image/jpeg` for a JPEG format image, and `audio/mpeg` for an MP3 audio file. A complete list of valid content types can be found at `http://www.iana.org/assignments/media-types/`.

When retrieving an image it makes no sense for your program to print the content, since the image is not textual data. Instead, the content should be stored in a file, which can then be viewed using an appropriate image viewer. Extend your `wbrowser` code to parse the `Content-Type:` header of the response. If the content type is `text/plain` or `text/html` it should display the body as before; otherwise it should ask the user for a filename, and store the body data in a file with that name.

Test your program with several web-sites, retrieving both HTML web pages, and other content, such as images. Check that any stored images can be displayed using an appropriate image viewer.

3.5 Parsing the Content

Displaying the raw HTML code of a web page is not very user-friendly, and the results are hard to read. The final part of this exercise is to extend your program to recognise if the data has content type `text/html` and if so, parse the HTML code in the body to strip out the HTML tags, and print the body as plain text.

If the HTML body of a web page is correctly formatted, it should start with an `<html>` tag (all tags are case-insensitive), followed by `<head>...</head>` then `<body>...</body>`, and finally a closing `</html>`. The data between `<body>` and `</body>` is the content to be displayed. This content can be displayed as plain text by removing all the tags (anything between < and > inclusive), replacing closing tags (starting </ and ending >) with a new line, and printing the remaining content. For example, if you run your program as `wbrowser http://csperkins.org/test.html` the complete response from the server should look something like:

```
HTTP/1.1 200 OK
Content-length: 157
```

```
Content-type: text/html
Server: Zeus/4.2
Date: Tue, 20 Jan 2009 11:46:42 GMT
Accept-ranges: bytes
Last-modified: Tue, 20 Jan 2009 11:45:13 GMT
Set-cookie: X-Zeus-Mapping-3ade68de=192.168.217.160; path=/
Via: 1.1 proxy-server1
Proxy-agent: Sun-Java-System-Web-Proxy-Server/4.0

<html>
<head>
<title> Test page </title>
</head>
<body>
<h1> This is a test </h1>

<p>  The quick brown fox jumped over the lazy dog. </p>
</body>
</html>
```

and your program should display:

```
Test page

This is a test

The quick brown fox jumped over the lazy dog.
```

Test your program with a range of web-sites, and see what is displayed. If you understand HTML code, you may want to parse more of the response, and display pages more accurately (e.g. you could word-wrap the text, or show `<ul>` lists as bullet lists). You may also want to strip out JavaScript (everything between `<script` and `</script>`) to make pages more legible.

4 Notes and Submission

You are *required* to write a simple Makefile to compile your code, rather than running the compiler by hand. You are also *strongly advised* to enable all compiler warnings (at minimum, use `gcc -W -Wall`), and to fix your code so it compiles without warnings. Compiler warnings highlight code which is legal, but almost certainly doesn't do what you think it does. Use them to help you find problems.

The standard C library contains a number of functions for parsing strings which might be useful in this exercise. In particular, you are advised to pay attention to the `strtok()` and `sscanf()` functions (see Chapter 7 and Appendix B.3 of Kernighan and Ritchie's book for details).

This exercise is worth 6% of the marks for this course. You will be assessed on the quality, correctness, and readability of your code and your Makefile.

You are required submit an electronic copy of your source code and Makefile, archived as a `.tar.gz` file which expands into a directory named after your 7-digit matriculation number followed by "-lab1". For example, if your matriculation number is 0301234, your archive should expand to create a directory "0301234-lab1" with your files inside. *If you do not know how to create such an archive, ask one of the lab demonstrators*. These archives should be submitted as email attachments, with title "Submission of NS3 lab1", to csp@dcs.gla.ac.uk from your university email address.

You must also submit a printed and signed Declaration of Originality form for this exercise to the teaching office. Complete and sign this form in the usual way, then write the MD5 checksum of your code archive on the front of the form (the MD5 checksum is the 32 character hexadecimal code generated by running the `md5sum` program with the name of your archive file as its argument). Any submission where the MD5 sum written on your declaration of originality form does not match that of the submitted archive will receive zero marks.

Both your code and the declaration of originality must be submitted not later than 12:00pm on Tuesday 16 February 2010. This is a hard deadline; late submissions will be awarded a mark of zero, unless accompanied by a valid special circumstances form.