University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	**Networked Systems (H)**			
Coursework Number	**Summative exercise 1**			
Deadline	**Time:**	**4:30pm**	**Date:**	**8 February 2016**
% Contribution to final course mark	**10%**			
Solo or Group ✓	**Solo**	✓	**Group**	
Anticipated Hours	**10**			
Submission Instructions	**Submit via Moodle, following instructions in the lab 2 handout.**			
Please Note: This Coursework cannot be Re-Assessed				

Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below.

The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

(i) in respect of work submitted not more than five working days after the deadline
   a. the work will be assessed in the usual way;
   b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.

(ii) work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

Penalty for non-adherence to Submission Instructions is 2 bands

You must complete an "Own Work" form via https://studentltc.dcs.gla.ac.uk/ for all coursework

Networked Systems (H) lab 2: Debug and extend a simple web server

Dr Colin Perkins
School of Computing Science
University of Glasgow
https://csperkins.org/teaching/2015-2016/networked-systems/

19/20 January 2016

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 summative exercise, that is worth 10% of the marks for the course.

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.

An HTTP 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>
...
</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`
*.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 8088 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:8088/`. 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.

Summative Exercise 1: Web Server

In Summative Exercise 1 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 ending in a `/`, and the file `index.html` exists in the corresponding directory, the server returns the contents of the `index.html` file instead (for example, if the browser requests `http://example.com/foo/`, the server should act as if it has requested `http://example.com/foo/index.html`).

3. Modify the server code, in `wserver.c`, so that if a browser requests a URL ending in a `/`, and the file `index.html` does not exist in that directory, it returns instead a dynamically generated HTML page 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. Hint: use the `opendir()`, `readdir()`, and `closedir()` functions from the Linux standard library.

To complete this exercise you will need to modify the `wserver.c` file included in the zip archive. Do not modify any of the other files in the archive, and 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 must compile cleanly, without any warnings or errors, using the provided `Makefile`, and run on the Linux machines in the labs.

Submission Instructions

Once you have completed the assignment, make a copy of the `wserver.c` file – containing your modifications – under a new name, with a dash and your 7-digit numeric matriculation number before the extension. For example, if your matriculation number is 1234567, you should run the command:

```
cp wserver.c wserver-1234567.c
```

to do this. Submit the resulting `wserver-`*`matric`*`.c` file (replacing *`matric`* with your matriculation number) via Moodle. Do not submit any other files.

This is an assessed exercise, worth 10% of the marks for this course. The deadline for submissions is 4:30pm on Monday 8 February 2016. The Code of Assessment allows late submission up to 5 working days beyond this deadline, subject to a penalty of 2 bands for each working day, or part thereof, the submission is late. Submissions received more than 5 working days after the due date will receive an H (band value of 0).

Submissions that are not made via Moodle, that have the wrong filename, or that otherwise do not follow the submission instructions will be subject to a 2 band penalty. This penalty is in addition to any late submission penalty.

Marking Scheme

We will mark submissions, and assign a band on the 22-point University of Glasgow scale, between A1 and H. Marking will be in accordance with the following grade descriptors:

Primary Grade	Description
A	Excellent submission. All parts of the exercise have been correctly implemented. New code has essentially no memory safety issues, buffer overflows, or security problems, and is essentially bug free.
B	Very good submission. Mostly complete implementation of all parts of the exercise; new code is largely correct, and generally free of memory safety issues, buffer overflows, and security problems.
C	Good submission. Either all parts of the exercise are attempted, and the new code is good quality; or, the submission is an essentially bug-free attempt at only some of the tasks.
D	Satisfactory submission, with some significant bugs or limitations in the new code.
E	Weak submission, with many significant bugs and/or limitations.
F	Poor submission, making only a limited attempt to complete the exercise; some serious implementation problems.
G	Very poor submissions, with only a minimal attempt to complete the exercise; serious flaws in implementation.
H	No attempt.

After marking, submissions will be returned along with a assigned band, and a brief written justification for the band.