# Networked Systems 3: Laboratory Task 2

Dr Colin Perkins

17 February 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 2, to write a basic web server.

## 2 Basic Web Server

The aim of this laboratory exercise is to write a simple web server. This will extend your knowledge of network programming, by illustrating how basic server applications are implemented. The first version of your server will support only the minimum parts of HTTP required to send a single response to the client; as you proceed further through the exercise you will add more HTTP features, and add support for multiple requests and multiple simultaneous clients. You should attempt all parts of this exercise.

### 2.1 Initial Implementation

Your web server should bind to port 8080, listen for HTTP requests sent by a web browser, and return appropriate responses to the client (either the web page requested, or an error message).

On accepting a connection from the client, your server should parse the request. The first line of the request (up to the initial `\r\n`) determines the type of request. If the request begins `GET` followed by a filename and `HTTP/1.1` then you should parse that request to retrieve the name of the file requested. The filename should be interpreted as being relative to the directory in which your server was started (i.e., if the server was run from directory `/users/staff/csp` and received the request `GET /index.html HTTP/1.1`, it would return the contents of `/users/staff/csp/index.html`). Your server should also check the value of the `Host:` HTTP header sent by the client, to ensure it matches the current hostname (use the `gethostname()` function to find the hostname). After parsing the request to determine the filename, and checking the `Host:` header, your server should respond with the appropriate HTTP headers, followed by the data (the contents of the file). It should then close the connection.

If the hostname matches, and the requested file exists, a success ("200 OK") response should be sent, followed by the requested file, then the connection should be closed. An example of a minimal successful response, returning an HTML page, is as follows:

```
HTTP/1.1 200 OK
Content-Type: text/html
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                      "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
...
```

(the "..." indicates that the output has been truncated in these notes – your server should return a complete web page).

If the requested file doesn't exist, a "404 File Not Found" response should be generated. An example "404 File Not Found" response is as follows (the headers indicate that an error has occurred, body of the response is displayed by the browser):

```
HTTP/1.1 404 Not Found
Content-Type: text/html
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                      "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title> 404 Not Found </title>
</head>
<body>
<p> The requested file cannot be found. </p>
</body>
</html>
```

If the hostname of the server doesn't match the `Host:` header, or if the request does not start with `GET`, or if your server doesn't understand the request for some other reason, you should send a "400 Bad Request" response. If your server fails for some other reason, it should send a "500 Internal Server Error" response.

In all cases, the first line of the response indicates the version of HTTP used (`HTTP/1.1`) and whether the response succeeded (`200 OK`) or failed (e.g., `404 Not Found`). This is followed by several header lines giving information about the response, a blank line, and then the actual data requested. Two header lines are essential: the `Content-Type:` header line tells the browser the format of the data – the first version of your server should use `text/html` for everything – and the `Connection: close` header tells the client that you will close the connection after sending the data.

Write a simple `index.html` file, and test your server by retrieving this file using both the web browser you wrote in laboratory task 1, and another browser (e.g. FireFox). The URL you need to give to your browser will depend on the host you're using: for example, if you are using host `bo720-1-01` you would browse to `http://bo720-1-01.dcs.gla.ac.uk:8080/index.html`. Note the presence of the port specifier (`:8080`) in the URL, this tells the browser to use this non-standard port, rather than the standard port 80 (Linux restricts access to ports 0–1023 to privileged processes, so you need to use this non-standard port). You may need to modify the web browser you wrote earlier to allow the port to be specified as part of the URL.

Check that your server correctly responds to both valid and invalid requests.

## 2.2 Handling Multiple Sequential Connections

You will recall that the `accept()` function returns a file descriptor for the newly open connection, leaving the file descriptor of the listening socket untouched. A server may therefore accept a new connection, read the request, send its response, and close the connection, all without disturbing the listening socket. Extend your web server to use this feature to accept and serve multiple connections, one after the other, rather than exiting after serving a single connection. Don't forget to increase the backlog in the `listen()` call, so multiple connections can be waiting.

Write a simple web site, comprising multiple HTML pages, for your server to host. Browse this site using your favourite web browser to check that your server correctly responds to multiple requests.

## 2.3 Specifying the Content Type

Extend your website to include some images, in both JPEG and GIF format, linked from the HTML pages. To make the browser recognise these images, you'll need to include an appropriate `Content-Type:` header in the response sent by your server. The `Content-Type:` should be chosen according to the extension of the filename:

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

You'll need to parse the filename, to determine the extension, and then fill in the `Content-Type:` appropriately when constructing your response.

## 2.4 Handling Multiple Requests per Connection

Forcing a web browser to open a new TCP connection for each request is inefficient when multiple files are retrieved from a single web server. To avoid this inefficiency, HTTP allows several requests to be sent on a single connection. If the server *does not* include a `Connection: close` header in its response, the client can keep the connection open, and may send additional HTTP requests to the server. To allow the client to distinguish data from multiple requests, the server must include a `Content-Length:` header specifying the size of each response's data in bytes. You can retrieve the size of a file using the `fstat()` function:

```
#include <sys/stat.h>
...
struct stat    fs;
int            fd = open(filename, O_RDONLY);
...
if (fstat(fd, &fs) == -1) {
    // Error...
}
printf("file size = %d\n", fs.st_size);
```

Update your web server to support multiple requests per connection, only closing the connection when the client does so (the `read()` function will return zero when the connection is closed). Demonstrate that this works using a standard web browser (e.g. Firefox) by printing details of each request handled by the server.

## 2.5 Handling Multiple Connections in Parallel

Extend your web server to accept multiple connections in parallel. This should improve its scalability and response time on multicore systems, and on systems with relatively slow disks compared to the speed of their network connection. There are two parts to this task: introducing concurrency to your web server by starting a new thread for each connection, and then extending the server with a thread pool, to hide thread creation overheads. Both parts should be implemented using the pthreads API functions.

### 2.5.1 One thread per connection

Introduce concurrency by extending your server so that it starts a new thread to process each network connection accepted on a socket. That is, when the `accept()` function completes, start a new thread to process the newly accepted connection. The new thread should accept the file descriptor for the new connection as a parameter in the `pthread_create()` call. Once created, the thread will process HTTP requests until the connection is closed by the client, then it will close the connection socket, and exit. The main thread should remain open and accepting new connections.

Test your system to demonstrate that it can successfully handle multiple connections in parallel. A standard web browser such as Firefox should open multiple connections to a site if you have enough content for it to fetch, or you could modify the simple web browser you wrote in the first laboratory session to use threads and download several pages at once. Create a web page containing several (at least 10) images, and test concurrent browsing using this page (print the thread identifier, returned by `pthread_self()` and filename when returning a response, to shows that it's working in parallel).

### 2.5.2 Thread pool

Starting a new thread for each connection can be inefficient, since threads take some time to start. A more scalable approach creates a pool of worker threads before accepting any connections, and passes each new connection to the next free thread in the pool. Such a system comprises a single controller thread, with a pool of worker threads. The controller maintains a list of workers, along with a state variable for each indicating if it is busy of idle. Newly accepted connections are passed to the first idle worker in the list, with new connections not being accepted until there is an idle worker thread.

Implement such a system using pthreads to create the worker threads. Take care to provide appropriate locking when manipulating condition variables shared between controller and workers. Ensure your workers block on the condition variable while waiting for new work, rather than continually polling. Test your system, to demonstrate that it works correctly.

## 2.6 IPv6

Your server has, until now, been listening for connections on IPv4 only. Extend it to listen for IPv6 connections as well. To do this, you'll need to open an additional socket and bind it to the IPv6 unspecified address (`in6addr_any`, defined in `<netinet/in.h>`), and process requests sent to that socket as well.

Test this using the web browser you wrote in the previous laboratory task, and using a regular web browser. Your machine is unlikely to have an IPv6 address entered in the DNS, but you can browse to it directly. Running the `ifconfig` command from a terminal will should you the IP addresses of your system. The IPv6 address will look something like "fe80::21e:c9ff:fe4a:52e7" (it may be listed with a trailing "/64" which you can ignore), and you can connect using a URL like `http://[fe80::21e:c9ff:fe4a:52e7]:8080/index.html`.

# 3 Notes

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.

This exercise is worth 10% 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 "-lab2". For example, if your matriculation number is 0301234, your archive should expand to create a directory "0301234-lab2" 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 lab2", 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 that does not have a matching declaration of originality, or where the MD5 sum written on the 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 Friday 12 March 2010. This is a hard deadline; late submissions will be awarded a mark of zero, unless accompanied by a valid special circumstances form.