University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	**NS3**			
Coursework Number	**Summative Exercise 1**			
Deadline	**Time:**	**9:00am**	**Date:**	**27 February 2012**
% Contribution to final course mark	**16%**			
Solo or Group ✓	**Solo**	✓	**Group**	
Anticipated Hours	**12**			
Submission Instructions	**Submit via Moodle, in a .tar.gz archive formatted as instructed in the NS3 Lab 3 handout.**			
Please Note: This Coursework cannot be Re-Done				

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://webapps.dcs.gla.ac.uk/ETHICS for all coursework
UNLESS submitted via Moodle

NS3 Lab 3 – Filtering Web Proxy

Dr Colin Perkins
School of Computing Science
University of Glasgow
http://csperkins.org/teaching/ns3/

25 January 2012

Introduction

The laboratory sessions for Networked Systems 3 (NS3) will introduce you to network programming in C on Unix/Linux systems. There are weekly labs for this course, during which you will complete several exercises. These exercises will build on your knowledge of C programming and pthreads from the Advanced Programming 3 course last semester, and on the material in the NS3 lectures. There are a mixture of formative and summative exercises. The formative exercises are intended to give you practice in programming networked systems in C; they are not assessed. The two summative exercises are assessed, and are worth a total of 20% of the marks for this course.

This is NS3 lab 3, an exercise to build a filtering web proxy in C. It comprises one summative exercise that should be completed during the timetabled laboratory sessions in weeks 3–7 of the semester, and during other hours as necessary. This work is assessed, and is worth 16% of the marks for this course.

Background

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 port 80 of the web server, sends an HTTP request for the requested web page over that connection, reads the response back, and then displays the page. Both HTTP requests and responses are text-based, making the network protocol relatively 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 (the current version is `HTTP/1.1`). For example, a

browser would send the method `GET /index.html HTTP/1.1` to retrieve the page `/index.html` from a 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 fetch the main University web page (`http://www.gla.ac.uk/index.html`), a browser could 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). The example above is a minimal HTTP request, and the browser will usually include other header lines, in addition to the `Host:` header, to control the connection, indicate support for particular file formats and languages, convey cookies, and so on.

On receiving an HTTP request for a web page that exists, a web server will reply with a `HTTP/1.1 200 OK` response, followed by several more header lines providing information about the response, a blank line, and then the body of the page. The headers lines should include a `Content-Length:` header, which specifies the size of the body of the page in bytes. As with the request, each header line ends with a carriage return followed by a new line, and the headers are separated from the body with a blank line. An example of the type of response that is sent 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">
<head>
...
</body>
</html>
```

In this example, the "Content-Length:" is 15694 bytes, meaning that there are exactly 15694 bytes in the body (starting with the "<" of the "<!DOCTYPE" line, and finishing with the ">" of the "</html>" line.

If a request is made for a non-existing file, the server will respond with a 404 "file not found" error. This will have a "Content-Type:" header of "text/html", and the body of the response contains the error page to be displayed to the user.

```
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.

Web Proxy Servers

A web proxy server is a middlebox that is interposed into HTTP connections, and can inspect, process, and manipulate the HTTP requests and responses. A web proxy can act as a cache, saving local copies of certain responses so it can reply directly to future requests without having to contact the server, saving time and upstream network bandwidth (this is the reason the University runs a web proxy cache). Alternatively, a web proxy can act as a filter, deciding which requests to forward on to the server, and which to respond to with a 404 error, as if the page cannot be found. This latter behaviour can be used to remove advertising, or to censor pages that the operator of the proxy thinks are undesirable.

To a web browser, a proxy appears as a web server: it accepts connections, reads requests, and sends responses in much the same way as any other web server would. To a web server, a proxy appears like a browser: it makes requests and reads responses. Internally, a proxy is implemented as a hybrid, and works much like a server and browser connected back-to-back. For each request it receives, it either replies directly to the browser like a server would, or it makes an onward connection to the actual web server, forwards the client's request to the server, reads the server's response, and sends it on to the client.

Summative Exercise 1: Filtering Web Proxy

The goal of this summative exercise is to build a web proxy server that can filter out advertising, or other unwanted content. It will introduce you to the operation of the

HTTP protocol, and give you practice building a moderately complex networked application in the C programming language.

You should write a program, `fwp`, which acts as a filtering web proxy. This should run on a machine in the Boyd Orr 720 lab, and accept TCP connections on port 8080. For each incoming connection that it accepts, it should make a corresponding outgoing connection to the University's web proxy (using hostname wwwcache.dcs.gla.ac.uk, TCP port 8080). Once both connections are established, your program should enter a loop, repeatedly reading HTTP requests from the incoming connection ready for processing. For each HTTP request, your proxy should inspect the method (the first line of the request). If the method is anything other than "GET", then the request should be forwarded unchanged on the outgoing connection. If the method in the request is "GET", then the requested URL should be extracted from the method line and parsed. If the requested URL matches a banned-site list contained in your proxy, then your proxy should respond to the browser with a locally generated 404 error page, and should not forward the request. Otherwise, if the URL in the GET method doesn't match the banned-site list, your proxy should forward the request to the University's web proxy using the outgoing connection.

Your proxy should read, parse, and buffer the response to each request that it sent on to the University web proxy. To read the full response, you'll need to read in the headers, stopping when you reach a blank line, then check the "Content-Length:" header to determine the length of the response content, then read that many bytes of data following the headers. Note that, due to the way TCP congestion control behaves, data may be delivered in arbitrary sized chunks, and it may take more than one `read()` call to get all the data.

After reading the response, you should check the "Content-Type:" header. If the content type is different to "text/html", the response from the server should be sent unchanged to the browser that made the request. If the content type is "text/html", the response from the server should be sent to the browser after first replacing all words on a banned-word list you maintain with an equivalent number of "*" characters (or with other words, if you prefer).

When your proxy notices that an incoming connection has been closed by the browser, it should close that connection itself, and also close the corresponding outgoing connection to the University's web proxy. Similarly, if the University's web proxy closes the outgoing connection you made to it, then your proxy should close both that connection and the corresponding incoming connection.

Your proxy should be implemented in a multithreaded manner, so it can handle multiple connections simultaneously. It should print a log of the actions it performs to standard output, for debugging purposes. This log should record threads created and destroyed, connections opened and closed, details of requests forwarded and denied, and details of any banned word filtering performed.

Your proxy should send a `HTTP/1.1 500 Internal Server Error` to the client, and close the incoming and outgoing connections, if it encounters a problem.

Your proxy must be written in C, and must run on the Linux machines in the level 3 laboratory. 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.

Test your proxy by changing the proxy settings in your browser to connect to your proxy (localhost, port 8080), then browse the web as normal. Check that browsing works as expected. Then, add some sites to your banned-sites lists to check that your proxy successfully prevents access to those sites (you might try to filter out advertising from a site you use). Finally, check that you can censor certain words within other websites.

Submission

You should prepare an electronic copy of your source code and Makefile (do not submit compiled binaries) archived as a `.tar.gz` file that expands into a directory named after your 7-digit matriculation number followed by "-submission1". For example, if your matriculation number is 0301234, your archive should expand to create a directory "0301234-submission1" with your files inside. You can create the archive using a command such as:

```
tar cvzf 0301234-submission1.tar.gz 0301234-submission1/
```

Ask one of the lab demonstrators if you are unsure how to create the archive. If your archive is formatted correctly, you should see something like the following when running the `tar ztf` command:

```
$ tar ztf 0301234-submission1.tar.gz
0301234-submission1/Makefile
0301234-submission1/fwp.c
$
```

(the `0301234-submission1/` prefix shows that the archive expands into a sub-directory with the appropriate name for this matriculation number).

This work is assessed, and is worth 16% of the marks for this course. Submissions should be made via Moodle. The deadline for submissions is 9:00am on Monday 27 February 2012. As per the Code of Assessment policy regarding late submissions, submissions will be accepted for up to 5 working days beyond this due date. Any late submissions will be marked as if submitted on time, yielding a band value between 0 and 22; for each working day the submission is late, the band value will be reduced by 2. 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, or that are in archives which do not meet the above guidelines will be penalised two bands. This penalty will be applied in addition to any late submission penalty.