Networked Systems Architecture 3: Laboratory Task 1

Dr. Colin Perkins

16 January 2008

1 Introduction

The laboratory sessions for the NSA3 module aim to introduce you to network programming in C on Linux systems. There are eight weekly labs for this module, each two hours in length, in which you will complete four tasks. These tasks are not directly assessed, but the material covered in the labs will be examined as part of the final written examination. The four tasks are as follows:

Task 1:	Write the world's most basic web browser
Task 2:	Write a basic web server, accepting a single connection
Task 3:	Extend your web server to accept multiple connections in series, and process multiple requests per connection
Task 4:	Extend your web server to accept multiple connections in parallel using the pthreads API

Two weeks are allocated to each task.

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 the web server, sends an HTTP command over that connection, and reads the response back. HTTP commands and responses are textual, making it possible to fetch a web page by hand, manually typing the commands using the `telnet` program to open a TCP/IP connection to the web server. As an example, we will fetch the web page at `http://www.dcs.gla.ac.uk/index.html` by hand.

Open a terminal window, and start the telnet program with the address of the server and the desired port number as parameters: `telnet www.dcs.gla.ac.uk 80` (HTTP usually uses port 80). You should see output similar to the following ("`-->`" is my Linux shell prompt):

```
-->telnet www.dcs.gla.ac.uk 80
Trying 130.209.240.1...
Connected to crete.dcs.gla.ac.uk.
Escape character is '^]'.
```

This indicates that the telnet program has opened a TCP/IP connection to host crete.dcs.gla.ac.uk (the actual machine for which www.dcs.gla.ac.uk is an alias, using IP address 130.209.240.1) on port 80. The server is waiting for you to send it a command.

Commands in HTTP comprise an initial request line, followed by one or more header lines containing additional information. To retrieve a page, use the `GET` request, specifying the page and the version of the HTTP protocol used. 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: `Host: www.dcs.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. Try this out by typing the following into the telnet window you previously opened (hit return twice after the Host: line, to get the blank line):

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

The 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. Finally it will close the connection:

```
HTTP/1.1 200 OK
Date: Wed, 21 Nov 2007 12:13:14 GMT
Server: Apache/2.0.46 (Red Hat)
Last-Modified: Mon, 17 Nov 2003 08:06:50 GMT
ETag: "57c0cd-e3e-17901a80"
Accept-Ranges: bytes
Content-Length: 3646
Connection: close
Content-Type: text/html; charset=UTF-8

<HTML>
<HEAD>
<TITLE>Computing Science, University of Glasgow</TITLE>
...
</BODY>
</HTML>
Connection closed by foreign host.
-->
```

Try this with several pages on the departmental web site, comparing what you see in your web browser, with the raw HTML retrieved using telnet.

3 The World's Most Basic Web Browser

Having seen the basic operation of HTTP, your first task is to write a C program to retrieve a page from the main departmental web site (e.g. `http://www.dcs.gla.ac.uk/index.html`). This program will send the commands you have just typed by hand, then retrieve and print the headers and HTML code of the response.

To complete this task, you will use Berkeley Sockets library. This is the standard low-level networking library on Linux and other Unix-like systems (including MacOS X), and is virtually identical to the WinSock networking library used on Microsoft Windows. The following sections describe the sockets functions you will use.

3.1 Header Files

In order to use the sockets library, you must include the following header files in your code:

```
#include <arpa/inet.h>       // For inet_pton()
#include <sys/types.h>       // Supporting types
```

```
#include <sys/socket.h>       // For socket() and connect()
#include <netinet/in.h>       // For struct sockaddr_in
#include <stdio.h>            // For perror(), printf(), etc.
#include <unistd.h>           // For read(), write(), and close()
```

3.2 Creating a Network Socket

The first task in network programming is to create a *socket*. This is a representation of a network connection, which your program accesses via a *file descriptor*. To create a socket, call the `socket()` function, which has a prototype as follows:

```
int socket(int domain, int type, int protocol);
```

The `socket()` function takes three parameters (`domain`, `type`, and `protocol`, each of which is an `int`) and returns an `int` value.

The `domain` parameter indicates whether the socket is an Internet socket, or if some other network is to be used. You should use the value `AF_INET` to indicate that you're using the Internet Protocol, IP (`AF_INET` is an integer constant defined in the `<sys/socket.h>` header). The `type` parameter for sockets in the `AF_INET` domain is used to specify TCP or UDP: use `SOCK_STREAM` to create a TCP socket, and `SOCK_DGRAM` to create a UDP socket. The `protocol` argument is not used for sockets in the `AF_INET` domain, and should be zero.

The return value from the `socket()` call is an `int` file descriptor. This should be assigned to a variable, and is used by all the other calls in the sockets library to indicate which socket they should operate upon. If an error occurs, the value `-1` is returned, otherwise the value will be a small positive integer.

Putting this together, your program should create a socket as in the following code fragment ("`...`" indicates where other code has been omitted):

```
...
int    fd;
...
fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd == -1) {
     perror("unable to create socket\n");
     exit(1);
}
...
```

3.3 Connecting to a Server

Once you have created a socket, you can open a connection to the server. This is done using the `connect()` function, which has the following prototype:

```
int connect(int fd, struct sockaddr *name, socklen_t namelen);
```

The `fd` parameter is the file descriptor returned by the call to the `socket()` function. The `name` and `namelen` parameters give the IP address and port of the server to which you wish to connect. The departmental web server has IP address 130.209.240.1 and runs on port 80. To specify an IP address, create a variable of type `struct sockaddr_in` (a "subclass" of the `struct sockaddr` required by `connect()`) and fill in the appropriate details. The `struct sockaddr_in` is defined in the `<netinet/in.h>` header as:

```
struct sockaddr_in {
     uint8_t          sin_len;
     sa_family_t      sin_family;
     in_port_t        sin_port;
     struct in_addr   sin_addr;
     char             sin_pad[8];
};
```

Of the five fields in `struct sockaddr_in`, you need set only `sin_family`, `sin_port`, and `sin_addr`. This is done as in the following example:

```
...
struct sockaddr_in    addr;
socklen_t             addr_len = sizeof(addr);
...
addr.sin_family = AF_INET;
addr.sin_port   = htons(80);
inet_pton(AF_INET, "130.209.240.1", &(addr.sin_addr));
...
```

The `inet_pton()` function converts an Internet address from textual ("presentation") format to the numeric format used internally by the system (note this is given a pointer to the `sin_addr` field, so it can change the value). The `htons()` function converts the port number from host byte order (which is little-endian on Intel CPUs) to standard network byte order (which is always big-endian).

When the address has been filled in, it can be passed to the `connect()` function (remember that `connect()` takes a `struct sockaddr *`, so you'll have to cast to the appropriate pointer type). The `connect()` function returns 0 if a connection is opened, and -1 if an error occurs.

Lecture 4 will show how to extend this to pass a hostname, instead of an IP address.

3.4 Sending a Request

Once the connection is open, data can be sent to the server using the `write()` function:

```
ssize_t write(int fd, void *buf, size_t buflen);
```

The `fd` parameter is, once again, the file descriptor returned by the `socket()` function. The `buf` parameter is a pointer to the data to send to the server, while `buflen` is the amount of data to send (don't forget to count newlines, spaces, etc.). Since `buf` is a `void *` pointer, any type of data can be sent. For the purpose of this exercise, you should pass a char array containing the HTTP request and any headers needed.

The `write()` function returns the number of bytes successfully sent to the server, or -1 on error.

3.5 Retrieving a Response

The response can be retrieved from the server using the `read()` function:

```
ssize_t read(int fd, void *buf, size_t buflen);
```

The `fd` parameter is, once again, the file descriptor returned by the `socket()` function. The `buf` parameter is a pointer to a buffer in which to place the data returned from the server (usually a `char` array), and `buflen` is the size of that buffer.

The `read()` function returns the number of bytes successfully read from the server, 0 when the connection is closed, or -1 on error.

Since you don't know the amount of data that the server might return, your client should enclose the `read()` call in a loop, reading data until either the connection is closed, or all the expected data has been received.

3.6 Closing the Connection

Finally, the client can close the connection to the server:

```
int close(int fd);
```

Like the other network functions, `close()` returns -1 if an error occurs (e.g. if the server crashes while the connection is being closed).

4 Notes

The initial version of your program should use a hard-coded IP address for the server and the file to retrieve. It should display the entire response from the server, without any formatting. If you have time after completing the initial version of the program, the following extensions should be attempted:

- Extend the program to use a hostname rather than a raw IP address, looking up the hostname in the DNS. Lecture 4 will outline how to do this.
- Extend the program so the hostname and file to be retrieved can be specified on the command line, rather than being hardcoded.
- Extend the program to just display the HTML code, suppressing the headers sent by the server. Parse the headers, to report an error message if the requested file cannot be found.

The Linux manual pages provide detailed information on the sockets library functions in section 2 of the manual (e.g. `man 2 socket`). The book "Unix Network Programming, Volume 1 (3rd Edition)" by W. Richard Stevens, Bill Fenner, and M. Andrew Rudoff is the standard reference reference book for the sockets library and network programming in C.

You are strongly advised to write a simple Makefile to compile your code, to enable all compiler warnings (at minimum, use `gcc -W -Wall`), and to fix your code so it compiles without warnings.

A worked solution to this programming task will be provided in two weeks, when the second task is distributed.