Networked Systems Architecture 3: Laboratory Task 2

Dr. Colin Perkins

30 January 2008

1 Introduction

The aim of the second laboratory task is to write a simple web server. This will extend your knowledge of network programming, illustrating how basic server applications are implemented.

Your web server should bind to port 8080, and listen for HTTP GET requests. It should parse the GET request and any HTTP headers of interest 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`). The 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). The server will then respond to each request with a response containing appropriate HTTP headers, followed by the data (the contents of the requested file).

If the hostname matches, and the requested file exists, a success ("200 OK") response should be sent. 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). If the hostname of the server doesn't match the `Host:` header, or the requested file doesn't exist, a "404 Not Found" response should be generated. An example "404 Not Found" response would be as follows:

```
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>
```

In all cases, the first line of the response indicates the version of HTTP used (`HTTP/1.1`) and the status of the response (`200 OK` or `404 Not Found`). This is followed by several header lines giving information about the response, a blank line, and then the actual data requested. Once all the data is sent, the server closes the connection.

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, if you have time later you might want to choose an appropriate content type based on the file extension. Finally, the `Connection: close` header line signals that the server will close the connection after sending the data.

2 Implementation

You should implement your web server in C using the Berkeley Sockets library, running on Linux. You will use many of the same sockets functions you used to write your basic web browser in the last task, with the addition to functions to bind a port, then listen for connections, then accept a connection.

```
int bind(int fd, struct sockaddr *addr, socklen_t addrlen);
```

The `bind()` function binds a socket to a network address (`addr` is a pointer to a `struct sockaddr_in`, but is cast to be a pointer to the "superclass" `struct sockaddr`). You should use port 8080, with an address of `INADDR_ANY` (indicating that the server should bind to any network interface available). As usual, -1 is returned on error.

Once the socket is bound to a port, it can be told to listen for connections:

```
int listen(int fd, int backlog);
```

The `backlog` parameter specifies the number of connections that can be outstanding (connected, but not yet accepted) on the socket. For the purpose of this task, your server may exit after servicing a single connection, so you may set the backlog to one. As usual, -1 is returned on error.

After a socket has been bound to a port and told to listen for connections, the `accept()` function is used to wait for, then accept, a connection from a client.

```
int accept(int fd, struct sockaddr *addr, socklen_t *addrlen);
```

The `accept()` call fills in `addr` and `addrlen` with the address of the client. It returns -1 if an error occurs. If successful, it returns a new file descriptor, representing the new connection. To send or receive data from the connection, use the returned file descriptor (the original `fd` remains open, and can be used to accept new connections later).

The `close()` function should be used to close both the original file descriptor returned from `socket()`, and the per-connection file descriptor returned from the `accept()` function.

3 Notes

You should initially test your server using the web browser program you wrote in the last task (you'll need to write a simple HTML file to retrieve). Once that works, consider testing with a standard web browser (e.g. Firefox). When connecting to your server using a standard browser, don't forget to specify the port number on the URL. For example, if your server runs on port 8080 of host bo720-1-01, then use a URL of the form `http://bo720-1-01.dcs.gla.ac.uk:8080/index.html` in your browser.

When parsing the HTTP request and headers, consider using the string functions provided in the standard C library (for example, the `sscanf()` function can be used to simply parse the GET request line).

You are again 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 third task is distributed.