LibHTTPD Article written for Linux – May 2002

Introduction

It has often been stated that the success of the Internet is due to the global acceptance of the "World Wide Web". Although many of us were quite happy using the services provided by the 'Net in the days before HTTP and HTML, I'm sure we can all appreciate the impact they have had. The average network user now has at his or her fingers a tool that can be used to retrieve vast amounts of data with relative ease. This empowerment of John Citizen is very exciting to many people around the globe.

Odd as it may be, but there is another group of people out there that are similarly excited but for totally different reasons. The people who develop applications, devices and network services are overjoyed by the acceptance of the WWW. Not because it puts information into the hands of millions of people, but because millions of people now have a common "platform" that can be used to display a user interface. With the vast proliferation of the web browser, virtually every computer currently in use is capable of displaying the configuration screen of a router, the GUI of a software application, or the usage statistics of any number of network appliances.

Although HTML isn't as powerful as a windowing system based GUI, it can quite readily be used to provide access to a vast array of applications and devices. A greater challenge is finding suitable software for providing the HTTP server. There are many web servers out there, but I can't imagine many people being happy about trying to shoe-horn something like the Apache server into an embedded device. Similarly, if you want to provide a web interface to an application, having to ship another complete and complex application (i.e. a fully blown web server) is inconvenient and often problematic. If all that is required is a simple web based user interface then distributing a web server that is capable of serving data for hundreds of virtual web sites must be viewed as overkill.

Necessity is the mother of invention, and it was the realisation that a "normal" web server just wasn't suitable that resulted in the development of LibHTTPD. We wanted to fix this problem once and for all. We had already discovered some of the things that are wrong with the "high-end" web servers : they are both complex and enormous. Those points alone make such packages unsuitable for most embedded devices and application interfaces. An efficient web server implemented within a library that could be linked to an application looked like the perfect solution.

By using a linkable library to provide the web server solution we could achieve two of the major functional requirements. Firstly, it could be small and efficient enough to be suitable for embedded devices. Without all the high-end features associated with fully-blown web servers the memory footprint could be kept to a minimum. Similarly, because the web server could be linked directly into a software application, distributing and installing a web-accessible application would be no more complex than a normal application.

Careful consideration had to be given to exactly what was implemented within the library. If the entire HTTP specification was implemented then the applications linked against the library would be bloated. However, if certain features of the protocol were not implemented, vast amounts of network bandwidth would be wasted. It was decided to implement a subset of the protocol that included the basic data request mechanisms as well as the conditional fetch features used by browsers and web caches to reduce bandwidth consumption.

To be well suited to a wide range of potential applications, the library would need to provide a convenient and flexible way to define both static and dynamic web content. The content module would also need to ensure that only the defined content was available via the server. There are many well known tricks that people use to try to access "secure" data on a web server. The library should ensure that any request for data that hasn't been specifically defined by the application is rejected.

As the main purpose of this library is to aid in the development of web based interfaces to applications, the library should simplify the creation of dynamic content. Extraction and manipulation of "form data" must be primarily handled by the library without the intervention of the programmer. The programmer should have complete access to all data and cookies provided with a request. The API must also provide a simple yet complete interface to the headers and response codes used by HTTP to signify certain events or situations.

In short, the library must be small, fast, efficient, secure, and provide routines that will greatly simplify the creation of web based application. That's quite a task for a small library.

Implementation

The library implements an opaque view of the capabilities of the web server via a "handle" mechanism. An applications first task is to create a new handle using the httpdCreate function. Once the handle has been created the web server can be configured and the content can be added.

The only configuration items currently available are the locations of the access log and the error log. By default, the server will not log any information about the requests it is serving. An application can use the httpdSetAccessLog and httpdSetAccessLog functions to specify a destination for the log information. The log formats have been designed to match the format used by the Apache server. An example of creating and configuring a server instance is shown below.

```
server = httpdCreate(host,port);
if (server == NULL)
{
        perror("Can't create server");
        exit(1);
}
fp = fopen("access.log","a");
httpdSetAccessLog(server, fp);
httpdSetErrorLog(server, stderr);
```

The web server's content module supports content contained in external files, content contained in static buffers, and content dynamically created by issuing a C function call-back. Each content item is defined based on its location in the "content tree" using a combination of a URL path and an item name. This is conceptually the same as a file system's directory / filename combination. The fact that the item's URL path is specified allows the item to be tagged with special features. Special tagging includes identifying the item as the "index" item for the URL path or to allow wildcard requests within that URL path to be processed by a C call-back.

The content module also allows the programmer to define a C function that is called before the request for the specified content item is processed. The pre-load function, as it is called, can be used for authentication or authorization. Authorization tests such as the time of day or the IP Address of the requesting machine are all possible via the pre-load function.

Some sample content definitions are shown below. The first shows a dynamic content item (i.e. a C function call-back) that is made available via the /intro/index.html URL. The boolian flag indicates that it is an index item (i.e. a request for the directory will result in a fetch of this item). The 5th argument is a C function pointer to the pre-load function for the item (or NULL if there is no pre-load function). The final parameter is a pointer to the C function to be called.

The second example shows a static content item. The function parameters are similar to those outlined above for the dynamic content item. The parameters are the server handle, the URL path, the item name, the index item flag, the pre-load function pointer, and the static text buffer itself.

```
httpdAddCContent(server,"/intro", "index.html", HTTP_TRUE, NULL, index_html);
#define buffer_1 "This is a <b>test</b> <br>\n"
httpdAddStaticContent(server, "/intro", "test1.html", HTTP_FALSE, NULL, buffer_1);
```

Once the content has been defined, the server is ready to handle requests from clients. The basic process is simply to accept a connection, read the request, process the request and then clean up. It is implemented using the libraries API as follows

```
while(1 == 1)
{
        if (httpdGetConnection(server) < 0)
                continue;
        if(httpdReadRequest(server) < 0)
        {
                httpdEndRequest(server);
                continue;
        }
        httpdProcessRequest(server);
        httpdEndRequest(server);
}
```

When httpProcessRequest is called, the library will review the request and return the requested content to the client. If the content is defined as a dynamic item, the associated C function will be called and passed a pointer to the server handle. If there is no content matching the request then a standard HTTP error code (Error 404) is sent.

The remainder of the functionality provided by LibHTTPD is aimed at simplifying the creation of dynamic web content. One of the key features is the library's symbol table module. When a request is received, the library will identify any accompanying data, such as the results of an HTML form. If there is any data it is extracted and stored in a symbol table. The library provides routines that allow you to access the symbol table entries using the names of the data items. It also provides routines for performing automatic expansion of variables embedded within HTML. A sample C function call-back routine that demonstrates accessing the symbol table is shown below.

```
void test2_html(server)
        httpd    *server;
{
        httpVar *variable;

        /* Grab the symbol table entry to see if the variable exists */
        variable = httpdGetVariableByName(server, "name");
        if (variable == NULL)
        {
                httpdPrintf(server,"Missing form data!");
                return;
        }

        /*
        ** Use httpdOutput() rather than httpdPrintf() so that the embedded variable
        ** is expanded automatically
        */
        httpdOutput(server,"Hello $name");
}
```

Sending result data to the client is achieved using one of two functions provided by the library: httpdPrintf and httpdOutput. httpdPrintf provides the same semantics as the C library's printf routine although it also takes a server handle as an argument and directs the output to the client. httpdOutput however is more like the echo function provided by UNIX shell languages. It accepts just two arguments, the server handle and an output string. Before the string is sent to the client it is scanned for embedded variables. Any references to variables included in the output string are replaced by the variable's current value in the symbol table.

To further aid in the development of dynamic content, LibHTTPD also provides a range of routines to simplify access to features offered by the HTTP protocol. Routines are provided that set the response code, set the response content type, add headers to the response, set "cookies", and force HTTP authentication. The features available should provide a developer with the convenience and flexibility to tackle even the most sophisticated HTTP requirements.

Included in the software distribution is a test application that implements a very simple HTTP server providing access to 6 pages. Some of the pages are static, some dynamic, and some authenticated. The sample server consists of only 67 lines of code and consumes just 14k of memory when running! The example code has proven that our goals of providing a convenient interface and limiting the memory consumption of the host application have been achieved.

License and Availability

LibHTTPD is being made available by its authors, Hughes Technologies in Australia, under a dual license scheme. It is provided under the terms of the GNU Public License which ensures that it will always be available as a tool for the open source community. It is also available under an OEM license for commercial uses that are not compatible with the GNU GPL. It can be downloaded from the Hughes Technologies web site at www.Hughes.com.au