developerWorks®

nweb: a tiny, safe Web server (static pages only)

Nigel Griffiths

August 08, 2012
(First published September 25, 2006)

Have you ever wondered how a Web server actually works? Experiment with nweb -- a simple Web server with only 200 lines of C source code. In this article, Nigel Griffiths provides a copy of this Web server and includes the source code as well. You can see exactly what it can and can't do.

Introduction

Have you ever wanted to run a tiny, safe web server without worrying about using a fully blown web server that could be complex to install and configure? Do you wonder how to write a program that accepts incoming messages with a network socket? Have you ever just wanted your own Web server to experiment and learn with? **Further updates in 2012 to support recent web-server and browser standards and a code refresh.**

Well, look no further -- nweb is what you need. This is a simple Web server that has only 200 lines of C source code. It runs as a regular user and can't run any server-side scripts or programs, so it can't open up any special privileges or security holes.

This article covers:

- What the nweb server program offers
- Summary of C functions features in the program
- Pseudo code to aid understanding of the flow of the code
- Network socket system calls used and other system calls
- How the client side operates
- C source code

nweb only transmits the following types of files to the browser :

- Static Web pages with extensions .html or .htm
- Graphical images such as .gif, .png, .jgp, or .jpeg
- Compressed binary files and archives such as .zip, .gz, and .tar

If your favorite static file type is not in this list, you can simply add it in the source code and recompile to allow it.

The example file supplied includes the UNIX® source code in C and a precompiled one for AIX®. The source will compile with the IBM VisualAge® C compiler or the GNU C compiler, and should run unchanged on AIX, Linux®, or any other UNIX version.

Compile with the following command. The program is in C and requires no additional libraries or services. The **-O2** option is needed only if you want the code to be optimized:

```
cc –O2  nweb.c –o nweb
```

Functions within nweb

There are only a few functions in the source code, explained below.

`log()`

Logs messages to a log file. If the user requests an operation from the Web server that is not allowed or can't be completed, then nweb tries to inform the user directly. This is done by returning a fake Web page to the user that includes the error message. Since this function is only called from the child Web server process, the function can (once completed) exit and the main Web server process continues to allow further browser connection requests. If this is not a recoverable error, then the process is stopped.

`web()`

Deals with the HTTP browser request and returns the data to the browser. This function is called in a child process -- one for each Web request. It also allows the main Web server process to continue waiting for more connections. Checks are made to ensure the request is safe and can be completed. After the checks have been made, it transmits the requested static file to the browser and exits.

`main()`

This is the main Web server process function. After checking the command parameters, it creates a socket for incoming browser requests, sits in a loop accepting requests, and starts child processes to handle them. It should never end.

Pseudo code

Listing 1 below is the pseudo code for the approximate 200 lines of source code. It should help you understand the flow of the program.

Listing 1. Pseudo code

```
logger()

{

outputs error, sorry or log messages to the nweb.log file

if a sorry message, transmit it to the browser as a fake HTML response

if error or sorry message the program is stopped

}
```

```
web()  - this function returns the request back to the browser
{
read from the socket the HTTP request
check it’s a simple GET command
check no parent directory requested to escape the web servers home directory
if no file name given assume index.html
check the file extension is valid and supported
check the file is readable by opening it
transmit the HTTP header to the browser
transmit the file contents to the browser
if LINUX sleep for one second to ensure the data arrives at the browser
stop
}

main()
{
if option is "-?", output the hints and stop
check the directory supplied is sensible and not a security risk
become a daemon process
ignore child programs (to avoid zombies when child processes stop)

create a socket, bind it to a port number and start listening to the socket
forever {
   wait and accept incoming socket connection
   fork a child process
   if the child process then call the web function
   else close new connection
 }
}
```

Socket system calls

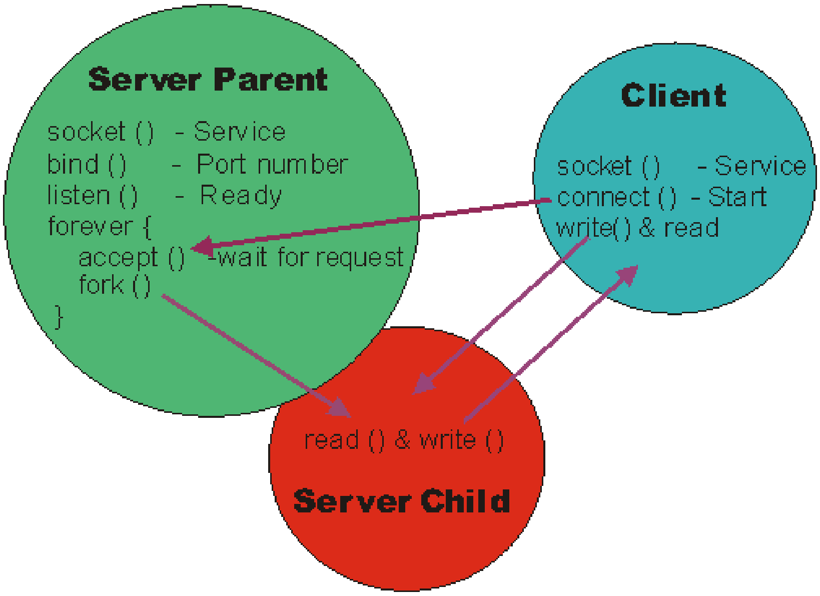
In case you haven't come across these network socket system calls before, they are explained below -- particularly how they all fit together. You can also look them up in the manual or on the Web -- though it can be hard to see what they do and why they make up a Web server from just the source code and manual.

The `socket()`, `bind()`, `listen()`, and `accept()` network system calls all work together to create a server process. They set up a socket ready for use to communicate over a network when combined. A socket is:

- An input and output stream -- regular pipes and files, for example.
- Remote access to or from a server -- when used over a network.
- Bidirectional -- when you read and write the same socket at both ends.
- Regular read and write functions -- used to send and receive data.
- A stream (no natural structure) -- you have to decide the protocol.
- For HTTP -- the request message and response message are finished with a carriage return (CR or /r in C code) and a line feed (LF and /n in C code). The URL is terminated by a space character and the end of the requested file is highlighted by closing the socket. There are more complex alternatives in HTTP and many optional features, but this is the simplest way to do it.

Figure 1 below explains how they fit together:

Figure 1.



The `socket()` function creates the socket and returns the file descriptor, which can be used with any function that uses file descriptors, such as read, write, and close. The arguments tell the operating system what type of socket and communication you need.

On the parameters to the `socket()` and `bind()`, there are dozens of permutations and combinations. The arguments used in the program are very typical for a regular general purpose socket using IP; in my experience, other more complex options are rare.

The `bind()` function attaches a particular port number to the socket. When a client is trying to contact your server, it will use the IP address (often found by using a DNS service to convert a hostname to an IP address) and the port number. The port number tells the operating system which service you want on the server. The details of the common port numbers on most UNIX machines are listed in the **/etc/services** files. These files include standard port numbers for services such as FTP (port 21), Telnet (port 23), and Web servers (usually port 80). You should check the port numbers in the **/etc/services** file to make sure you don't try one that is already in use. Although, if you try, you should get an error message in the log file, as it is normally impossible for two services to use the same port number.

The `listen()` function call tells the operating system you are now ready to accept incoming requests. This is the final switch that makes the socket available to local and remote programs.

The `accept()` function does not return to your program until there is a socket connection request to this IP address and port number. The `accept()` function uses the file descriptor of the socket, but a new structure for the incoming connection (`struct sockaddr_in`) allows a server to check who is connecting to it. In nweb, you don't care who is connecting.

Once the `accept()` function returns, it means that the client socket file descriptor is live. If you read bytes from the socket, you get characters from the client, and if you write bytes to the socket, they get transmitted to the client. But it is not typical to read or write the socket from the main program. Normally, you want to allow multiple clients to have access to the services of your server, and if this main program does a read or write operation, it could block until there are characters to be read or the written characters transmitted. This main program should be running the `accept()` function again to allow a new connection to start. The main program should start a child process that does the "talking" to the client and any work it needs performed. For the main program to close the socket, rerun the accept function and await the next client connection. When the child process is started, it inherits the parents open sockets in order to keep the socket alive.

Other system calls

`getenv()`

A simple function for returning the shell variable values. If you set a Korn shell variable `$ export ABC=123`, then the getenv("ABC") function would return a pointer to a null terminated string containing 123.

`chdir()`

Changes directory.

`fork()`

Starts a child process. In the parent, it returns the process id (PID) of the child, and in the child, it returns zero.

`setpgrp()`

Sets the process group. The effect is for this process to break away from the other processes started by this user so it will not be affected by what happens to the users (like logging off).

`signal()`

Decides what happens when software interrupts arrive for the process. Within the nweb.c code, the main server wants to ignore the death of a child signal. Without this, the main server process would have to run the wait system call for each child process, or they would be forever stuck in the "zombie" state waiting for the parent to call the `wait()` function. Eventually, there would be too many zombie processes and the user's environment would hang, as they could not create further processes.

`open(), read(), write()` and `close()`

These are regular C library functions, but they are used for reading the sockets and files sent to the client Web browsers as the files and sockets are accessed by the file descriptor. The socket is opened with the `accept()` function; the `open()` in the code is for opening the requested file.

The protocol used between the browser and the web server

For complete details about the protocol, refer to the World Wide Web Consortium (W3C).

Browser sends: GET URL and lots of other stuff can be ignored but some browsers or application might check it for the browser type, OS, and so on.

Example**: GET /index.html**

Example**: GET /mydirector/mypicture.jpg**

The "/" refers to the directory at the top of the web server supported files—this is the directory specified when you started nweb as the second parameter. The web server should never send any files outside this directory. Some web servers allow specifying other directories for special file types, such as CGI scripts, but nweb does not handle such fancy services.

The Mozilla Firefox 10 browser makes a request to nweb as shown in the following code:

```
GET /index.html HTTP/1.1**Host: 192.168.0.2:8181**User-Ag
ent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:10.0.5) Gecko/20100101 Firefox/10.0.
5**Accept: image/png,image/*;q=0.8,*/*;q=0.5**Accept-Language: en-us,en;q=0.5**A
ccept-Encoding: gzip, deflate**DNT: 1**Connection: keep-alive**Referer: http://w
w.abcncc.uk.abc.com:8181/index.html**Cookie: UnicaNIODID=tVJWsYnAGTS-Xf5TqZJ; IB
M_W3ABC_ACCESS=wwww3.abc.abc.com; IBMISP=70fdfc95d93011d783e4de784ea97766-70fdfc
95d93011d783e4de784ea97766-f67749a8b899e8ceed7e940b8c4bf189; returnURL=http%3A%2
F%2Fwww-933.abc.com%2Fsupport%2Ffixcentral%2Faix%2FdownloadOptions; PD-FROMPAGE=
e%3D7.1%26function%3Drelease; PD-REFPAGE=/support/customercare/common/login%3Frt
n%3Dfixcentral; PD-ERR=; PD-HOST=www-304.abc.com; PD-SGNPAGE=%2Fsupport%2Fcustom
ercare%2Fcommon%2Flogin%3Frtn%3Dfixcentral; PD-REFERER=http%3A%2F%2Fwww-933.abc.
com%2Fsupport%2Ffixcentral%2Faix%2FselectFixes%3Frelease%3D7.1%26function%3Drele
ase; IBMISS=70fdfc95d93011d783e4de784ea97766-70fdfc95d93011d783e4de784ea97766-00
012ZSL0GV7c$TERIaw13er_zZ:15afkdghj-1340791897187-781004b2f37ec973cc8136eb5ddb53
67; rlang=en_US; ASESESSION2=EkSpHzOnrtcT90+EMdCXhRrFB3U+LxgKvOgc2ig+py0+Zq4GFgy
UHQB35BGnKy4i3pb6pyO0DkVv+6S/RizqpusAst5sz+xESyBQv4dsfWVm
```

In total, this is about 1300 bytes in size.

A lot of this is in the web standard but it is largely meaningless to me! You need to read the information in The World Wide Web Consortium (W3C) for all the details. Only the first part, that is, the GET command and the file name need to be used and the remaining information can be ignored. From the HTTP/1.1, there are extra details that can be optionally used by the web server. Note that it is sending information, such as the PC operating system (here Microsoft® Windows® 7) and browser and version (here Firefox 10.0.4). This is how web servers can report details of the users who are using the site.

The nweb web server checks whether:

1.This is a GET request - only **GET** or **get** is supported.

2.There are no ".." in the file name. This means that the browser or user is attempting to get a file that is not in the web server directory and is not allowed.

3.This supports file extensions such as .hmtl, .jpg, .gif and so on; only limited files types are supported.

If it is a valid GET request, the nweb web server responds:

```
HTTP/1.1 200 OK[Newline]
Server: nweb/22.0[Newline]
Content-Length: 12345 [Newline]
Connection: close[Newline]
Content-Type: text/html[Newline][Newline]
```

Notes about this header response and the file returned:

The **[Newline]** in the C language is **\n** and means two characters are send = Carriage-Return and Line-Feed. These **Newline** characters are not optional.

HTTP/1.1 200 OK

nweb does not support fancy web services, so it allows the browser know that it supports just the HTTP standard 1.1, which is the lowest used these days, and 200 is a special number informing the browser that the web server is going to respond.

Server: nweb/22.0

This informs the browser about the type of web server it is connected to and the version number. In this case, the browser will do nothing as it will not have heard of nweb, but the browser can tell the user what sort of web server is at the other end, build statistics, start using web server specific features, or work around the known web server errors. It is up to the browser developer, as at least they know what is at the other end of the socket.

Content-Length: 12345

Is the exact length of the file in bytes that is about to be sent. In this example, I just made up a number. The character count starts after **[Newline][Newline]**

Connection: close

This tells the browser that the web server is not going to keep the connection open for the next request, so it needs to make another socket connection for the other items of the web pages. Typically, a web page is now made up of the basic page content and hundreds of other graphics, Javascripts, and so on. Closing the link immediately is not good for performance but it makes the nmon web server code simple. Otherwise, we would have to start time out to close the sockets at some point later on.

Content-Type: text/html

This tells the browser about the format / content of the file that it is about to get as a response. nmon only returns text or image files and the acceptable list is in the C code in the **extensions** data structure. The second part is the detailed file type. In our case, based on the file extension to make life simple, I think the idea is the file might not have an extension at all or it might be wrong or meaningless. So the file extension might not tell the browser the file format. And, this gives the browser a hint about what to do with the contents.

There are two,**Newline** characters next and this is not optional. It tells the browser that there are no more header information lines and the file content comes next.

Then, the **file content is sent to the browser.** The browser knows when it is finished because of the length in line 3 that is counted after the two **Newline** characters.

After the last byte of the file is sent, the nweb web server web() function stops for one second. This is to enable the file contents to be sent down the socket. If it immediately closes the socket, some operating systems do not wait for the socket to finish sending the data but drops the connection very abruptly. This would mean that some of the file content would not get to the browser, and this confuses the browser by waiting forever for the last bit of the file and often results in a blank web page being displayed.

Finally, the socket is closed and the child process exits (stops).

Client-side socket system calls

The **nweb.c** code only shows you the server side of the socket. Listing 2 below illustrates the code that is required for the client side. Fortunately, the client side is much simpler, since you only need to supply the IP address and port number to make the connection to the server. In the code below, the servers IP address is assumed to be 192.168.0.42 and the server port is assumed to be 8181. In practice, hard coding these numbers in the code is not a good idea.

Listing 2. Client-side socket system calls

```
int sockfd;

static struct sockaddr_in serv_addr;



if((sockfd = socket(AF_INET, SOCK_STREAM,0)) <0)

 pexit("Network socket system call");

serv_addr.sin_family = AF_INET;

serv_addr.sin_addr.s_addr = inet_addr("196.168.0.42");

serv_addr.sin_port = htons(8181);

if(connect(connectfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) <0){

 perror("Network connect system call");
 exit(3);
/* now the sockfd can be used to communicate to the server */
/* with read() and write() */
```

Beware: the `bind()` and `connect()` functions take a structure as the argument that must be zero filled before setting particular options. In **nweb.c**, the structure is made a static C variable to ensure it is zero filled when the program is started. If this structure was on the stack as a variable within the function or in the heap of the process, it could be filled with old data and so its content will not be zero filled. In such a case, you need to ensure it is zero filled; the `bzero()` function can be used to do this.

Server source code

See Listing 4 for the server source code.

Listing 4 README.txt – **Lots of hints and tips on how the code works in practice, compiling, running, and testing the nweb code**

These 200 lines of source code is offered as a sample worked example. This is all basic UNIX systems programming and most programmers would come up with something similar, so it cannot be copyrighted as original work.

You are free to use it for any purpose with no limitations and no warrantee given or implied.
If you do use it for some project or product, then a link to this web page or reference to it is recommended, but it is completely optional.

The download includes:

- nweb.c -- the source code
- client.c -- sample client end source code
- nweb executable files for:
  - AIX 6.1 TL7 for POWER
  - Ubuntu 12.4 for x86_64
  - Fedora 17 Linux for x86_64
  - OpenSUSE 12.1 for x86_64
  - Debian Squeeze for ARM on Raspberry Pi
- Minimum test website
  - index.html -- test web page
  - nigel.jpg -- test image file

Downloadable resources

Description	Name	Size
nweb.c ,client.c, nweb executable files	es-nweb.zip	36 KB

Related topics

- Want more? The developerWorks AIX zone hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials.
- Check out *UNIX Network Programming* (W. Richard Stevens -- Prentice Hall) at Amazon.