

Project 1: Simple WWW client and server

Assigned: Thu Sep 19, 2002

Due: Mon Sep 30, 2002 (by midnight)

1 Introduction

In this assignment you will use Unix sockets to implement a WWW client and WWW server that use a restricted subset of the HTTP protocol. The main objective of the assignment is to give you some hands-on experience with sockets, while ignoring some of the many complications that arise in high performance client-server systems.

Your assignment is to write two programs: a server program called `webserver` and a client program called `webclient`. The two processes will communicate with one another using sockets in a manner very similar to how real web browsers (e.g., Netscape or IE) talk to real web servers (e.g., Apache or IIS).

2 Problem Description

2.1 Client operation

Your web client must read and parse a series of commands from `stdin`. For this assignment, only two commands need to be handled:¹

1. GET *filename hostname (port-number)*
2. KILLSERVER *hostname (port-number)*

Each command will appear on a separate line of input from `stdin`. Note that the *port-number* is optional. If it is not specified, use a default HTTP port number: 8000.

In response to a GET operation, the client must open a connection to an HTTP server on the specified host listening on the specified (or default) port number. It must then transmit a legal HTTP GET request, as described below, to request the file specified. It must then store this file in the local directory (i.e., the directory from which the client program was run) *and* display it to `stdout`.

In response to a KILLSERVER operation, the client should establish a connection to the specified server, as above, and send it a KILLSERVER command. Note that KILLSERVER is *not* a legal HTTP request, and thus will generate an error if sent to a real HTTP server. It is used to cleanly shutdown your server.

After each GET or KILLSERVER operation, the client should close the connection that was used. If you want to play around with more fancy HTTP options, consider implementing the HTTP 1.1 “persistent connection” option for use with real web servers.

The client should shut down gracefully when reaching the end of file.

2.2 Overview of command syntax

Performing a GET operation with a server is done by sending two lines across a socket connection that has been established with the server. The first line is a string containing the key word GET, followed by a file name, followed by the version of HTTP supported by this client. The second line is empty, consisting only of a newline character. For example, if the client sends the two strings:

```
"GET /index.html HTTP/1.0\n"
"\n"
```

to an http server, the http server will respond with the file `/index.html`.

The syntax of the KILLSERVER command will consist of two lines of the form:

```
"KILLSERVER\n"
"\n"
```

¹In command synopses in this handout, optional arguments are indicated by parentheses instead of the usual square brackets.

2.3 Server operation

Your server process should accept one optional argument, a port number on which to listen. In other words, the syntax of the `webserver` command is:

```
webserver (port-number)
```

The server process should `accept()` incoming connection requests on the specified port number, or, if no port number is specified, port number 8000. It should then look for GET or KILLSERVER requests and handle them appropriately. Note that a GET request from a real WWW client may have several lines of optional information following the GET. These optional lines, though, will be terminated by a blank line (i.e., a line containing zero or more spaces, terminated by a newline character. Your server should first print out the received GET command as well as any optional lines following the GET (and preceding the empty line). The server should then respond with the line:

```
"HTTP/1.0 200 Document follows\n"
```

followed by a blank line (i.e., a string with only blanks, terminated by newline). The server should then send the contents of the requested document, and then close the socket created by the `accept()` and wait to accept a new connection request. The file names specified will be *relative to the directory in which the server is started*. That means that a request for `html` should look for a file named `/index.html` in the directory where the `webserver` was started, *not* a file called `index.html` in the root of the local filesystem². If the document is not found, the server should respond with:

```
"HTTP/1.0 404 Not Found"
```

as would a real http server.

In response to a KILLSERVER command, the server should shut down cleanly.

3 Additional Notes

- The RFC that defines the HTTP 1.0 protocol is available at <http://www.rfc-editor.org/rfc/rfc1945.txt>, May 1996. However, you do not need to print it (or even read it) for this assignment. Note that the RFC is 150 pages long! You can find other relevant RFCs on the www.rfc-editor.org site. In particular, HTTP 1.1, the current draft standard, is defined by RFC 2616, June 1999 (there are a few later RFCs that update portions of it).
- Read the man pages for `socket()`, `bind()`, `listen()`, `write()` and the other related operations to learn the syntax of establishing and using TCP socket connections.
- Read the man pages for `gethostbyname()` and related functions to see how to convert a hostname to an IP address. We have extracted the example that is provided in the man page into files that you can see in `/home/cs/handin/cs5480/public/examples`. Note that you must link your program using the `-lnsl` command to link against the appropriate nameserver library.
- We will link in to the class Web page some tutorial material on socket programming, which can be useful. However, the man pages are the definitive reference.
- I would suggest that everyone begin by writing a client, and getting that client to work with a normal HTTP server. Then write a server and test it with a WWW browser.
- In writing your code, make sure to check for an error return from all system calls and library routines other than `(f)printf` and `exit`. If there is an error, the system declared global variable, `errno`, will give you information about the type of error that occurred.

²Doing this wrong will introduce a serious security hole to your server that would allow web clients to download *any file* that is accessible to the person who runs the web server!

- Note that `read()` and `write()` might return without having read or written the full number of bytes requested. In such a case, these calls will return a positive return value indicating the number of bytes actually read or written. If the number does not correspond with the number requested, be sure to loop back until all the data have been transferred.
- Your processes should communicate using the reliable stream protocol (`SOCK_STREAM`) and the Internet domain protocols (`AF_INET`).
- Make sure you close every socket that you use in your program. A call to `exit` will insure that these sockets are closed gracefully before the server terminates. If you abort your program, e.g., by typing control-C, the socket may still hang around and the next time you try and bind a new socket to the port number you previously used (but never closed), you may get an error. Please be aware that port numbers, when bound to sockets, are system-wide values and thus other students may be using the port number you are trying to use.

To ensure that your programs close all sockets even when aborted via a control-C, use the `signal` library function to “catch” control-C and call `exit()`. `signal()` can be used to call a function that you designate to be executed as soon as a control-C or a kill signal is received. In the code fragment below, the user-supplied function `clean_exit()` will be called when a control-C or kill signal are received. The two calls to `signal` in the main routine let the OS know that `clean_exit` should be called on a control-C or a kill signal:

```
#include <signal.h>
.....
main()
{
.....
    signal(SIGTERM, clean_exit);
    signal(SIGINT, clean_exit);
.....
} /* end of main */

/* Exit cleanly: called on ctrl-C or terminate */
void
clean_exit()
{
    exit(); /* this closes open sockets before terminating the process */
}
```

4 Testing your client and server

Your client can be tested by connecting to an arbitrary http server (e.g., connected to port 80 on host `www.cs.utah.edu`) and doing a GET on a file there. Note that the http server will return several lines of optional information following the “... Document follows” line. Your server can be tested by using a Netscape (or other) browser to be the client for your server. For example, if your server is on a machine `lab3-3.eng.utah.edu`, listening on port 6090, and you want to GET a file called `junk.html` from this server, you would enter “`http://lab3-3.eng.utah.edu:6090/junk.html`” as the URL to open in your Netscape/IE browser. This would cause Netscape/IE to contact your server at port 6090 and do a GET on `junk.html`

5 Logistics and Grading

This is an *individual* project, not a group effort. As such, make sure that your directories are protected appropriately (not world readable). Sharing code is strictly forbidden and could result in you failing the course outright.

While you may develop your code anywhere, we will be grading it in the CADE lab on the Sun machines, so make sure that your code compiles and runs there before handing it in! **Repeat: your program must**

compile and run properly on the CADE Sun machines. Not Linux! Not Windows! We specify a single platform for fairness, uniformity, and practicality of evaluation and assistance. We specify the CADE Suns because they are the most numerous, best-run, and are accessible to all.

Be sure to use the *handin* program to hand in the source code files, your makefile, and the executables. Be sure to **handin** *every* file that we will need to recreate your executable, including all header files, *.c* files, and **makefile**'s. Your executables should be called **webserver** and **webclient**. Also be sure to *handin* any external documentation that you have written (e.g., a **README** file) and comment your code thoroughly and clearly. Your documentation should describe the overall organization of your programs, the major functions and data structures, any assumptions that you make about the execution environment, and (important) a discussion of your testing strategy (i.e., how did you test your programs).

The C programming language is required; C++ will not be accepted.

Don't wait until the last minute, especially if you don't have ample experience with Unix or C. Things will go wrong. We're here to help you; ask for help early.

This project is worth **8%** of your final course grade. Your grade on the project will be based 80% on the correctness of your program and 20% on the quality of your documentation (both internal and external). If we cannot compile your program, you will receive **NO CREDIT**, so make sure you hand in everything needed, including the Makefile!