Lab 2 Webserver in C under UNIX – DV1305: Programming in the UNIX environment

Stefan Axelsson 2008-09-16

Introduction

Your task is to develop a web server in C. The program shall implement the HTTP 1.0 protocol specification with a few exceptions. Apache (http://www.apache.org) is currently by far the most popular1 web server and also one of the larger ones as it consists of approximately 70000 lines of code2. While Apache is rich of options, the THTTPD (http://www.acme.com/software/thttpd/) is developed to be "simple, small, portable, fast, and secure". Moreover, THTTPD is much smaller than Apache — it consists of approximately 7000 lines of code and has, of course, fewer features. A more modern small, fast webserver is lighttpd (http://www.lighttpd.net) These servers might be useful to look at when performing the lab, though of course you have to be wary of actual copying of code, that's not allowed. There are many more webservers listed at wikipedia:
http://en.wikipedia.org/wiki/Comparison_of_web_servers

Requirements

The server must be written in C, compile using GCC and run in Ubuntu Linux latest release (or the release that is installed in the security lab). There shall be no software limitation in the number of served clients or concurrent requests. General requirements:

Simple:	Handle only minimum of required functionality HTTP 1.0
Small:	Small run-time size and careful of memory allocation
Portable:	No, not this time, only on Ubuntu Linux (latest version).
Fast:	Handles request in a speedy manner, even during heavy load.
Secure:	Protect the web server machine from hacks and intrusions and Denial-of-Service

The server should be able to efficiently handle requests in parallel, i.e. if one request takes (relatively speaking) a lot of time to process it should not block requests coming later that could proceed from beginning to being processed. See the section at the end of the document for various strategies. In order to have the possibility of achieving top marks for the course you have to implement at least two different strategies. This is a necessary but not sufficient condition in order to be considered for a grade higher than just "pass."

Process management

The server may not create zombie processes, regardless of the server load (see the manual page for ps). See the course literature for information and tips on how to deal with zombie processes. The benchmark tool siege is usable when testing if the process management works correctly. Any dead processes will be visible using *ps*.

Jail

The web server should correctly use the chroot system call (similar to *jail*(*2*) in BSD) to imprison the process and its descendants. Read the jail manual page for more information on how to implement this requirement.

URL validation

The server should validate URLs to have a rudimentary protection against hacking or DoS-attacks. Examples of URLs that should not be served are:

http://localhost:80/../../../etc/passwd
http://localhost:80/<very_long_string>.html

The suggested way of validating is by using the library function realpath (see the realpath manual, section 3).

HTTP 1.0 methods

The server shall implement both the simple and full versions of the following HTTP 1.0 methods:

- HEAD
- GET

HTTP requests for all other types of methods should result in the HTTP status code 501.

HTTP 1.0 status codes

The following status codes should be implemented. All status responses should be properly HTML 4.0 formatted.

Code	**Reason phrase**
200	OK
400	Bad Request
403	Forbidden
404	Not Found
500	Internal Server Error
501	Not Implemented

MIME

The server should use Apache's MIME file, called mime.types. A copy of the file can be found on the Internet (Google for it), note that mine might not be the same as yours. The server should use the file in its text format, you may not make the file a part of the server source code (e.g. by statically defining the MIME types as variables). The server must be able to correctly handle files that do not match a MIME type. See RFC 1521 for more information on MIME.

Logging

The server should log standard activity and errors in CLF (Common Logfile Format), a log format used by several web servers and log analysers, see the References section. Logging shall be possible to output both to syslog and to a file (however, you may choose to output to a separate files, e.g. <filename>.log and <filename>.err). When output is made to syslog, which should be the default, the server must perform string length checks to ensure that possible overflow attacks do not succeed. See the command line section below for more information on how to handle logging output. IP addresses should be printed in IPv4 format.

To check that your logs conform to CLF, an analysing tool is recommended. A list of suggested tools is available in the References section. If you have no preferences, the Webalizer tool is recommended.

Command line options

The server must support at least the following command line options:

-*p port* Listen to port number port. This overrides the default
port that is specified in the configuration file.

-*d* Run as a daemon instead of a normal program. After starting, the server should print its process ID.

-*l logfile* Log to logfile. If this option is not specified, logging will be output to syslog, which is the default.

Configuration file settings

The server shall read a configuration file called .lab3-config, which shall be user-editable (e.g. in ASCII text format). The following configurations should be specified:

• full path of the server's document root directory; This should not be the same as the server's execution directory

• default listen port

HTML 4.0

All HTML formatted output from the server should follow the HTML 4.0 specification. Validation tools are available on the Internet, e.g. The W3C Markup Validation Service at http://validator.w3.org/ or WDG HTML Validator at http://www.htmlhelp.com/tools/validator/. HTML Tidy might be a useful tool to check your server output. The tool is available at http://tidy.sourceforge.net/.

Client compatibility

The server should be able to serve Internet Explorer, Mozilla and Opera on Windows, and Konqueror, Mozilla, Opera and Lynx on Linux.

Tools

Some tools that may be useful during development are:
• sendfile - Not really a tool but a Linux system call to send a file to a socket.
• tcpdump - Network sniffer without GUI.
• Ethereal - GUI network protocol analyser.
• Hammerhead – A web testing tool, installed in the security lab
• netcat - Networking utility which reads and writes data across network connections, using the TCP/IP protocol.
• wget - Utility for non-interactive download of files from the Web.

Laboratory examination

You should present your lab server in the lab as usual. Be prepared to answer questions about the web server, i.e. how you handled parallelisation etc.

Strategies for handling multiple requests

Your webserver has to implement at least one of these. If you aim for the highest mark on the course you are required to implement two or more strategies, if you don't you are only eligible for a "pass" mark.

The strategies are: fork/exec, threads, preforking/thread pool, or I/O multiplexing/single process. For a description of the various strategies, read the following historical account. In almost all of these the question of how to communicate between the controlling process and the worker processes, several ways are possible; using pipes, a FIFO, mmap, or another technique, research the available techniques and argue and document your choice.

Short webserver performance history

select() / poll() / epoll() / kqueue() are Unix system calls used to multiplex between a bunch of file descriptors. To understand why this is important we have to go back through the history of web servers.

The basic operation of a web server is to accept a request and send back a response. The first web servers were probably written to do exactly that. Their users no doubt noticed very quickly that while the server was sending a response to someone else, they couldn't get their own requests serviced. There would have been long annoying pauses.

The second generation of web servers addressed this problem by forking off a child process for each request. This is very straightforward to do under Unix, only a few extra lines of code. CERN and NCSA 1.3 are examples of type of server. Unfortunately, forking a process is a fairly expensive operation, so performance of this type of server is still pretty poor. The long random pauses are gone, but instead every request has a short constant pause at startup. Because of this, the server can't handle a high rate of connections.

A slight variant of this type of server uses "lightweight processes" or "threads" instead of full-blown UNIX processes. This is better, but there is no standard LWP/threads interface so this approach is inherently non-portable. Examples of these servers: MDMA and phttpd, both of which run only under Solaris 2.x.

The third generation of servers is called "pre-forking". Instead of starting a new subprocess for each request, they have a pool of subprocesses or threads that they keep around and re-use. NCSA 1.4, Apache, and Netscape Netsite are examples of this type. Performance of these servers is excellent; they can handle from two to ten times as many connections per second as the forking servers. One problem, however, is that implementing this simple-to-state idea turns out to be fairly complicated and non-portable. The method used by NCSA involves transferring a file descriptor from the parent process to an already-existing child process; you can hardly use the same code on any two different OS's, and some OS's (e.g. Linux) don't support it at all. Apache uses a different method, with all the child processes doing their own round-robin work queue via lock files, which brings in issues of portability/speed/deadlock. Besides, you still have multiple processes hanging around using up memory and context-switch CPU cycles. This brings us to...

The fourth generation. One process only. No non-portable threads/LWPs. Sends multiple files concurrently using non-blocking I/O, calling select()/poll()/kqueue() to tell which ones are ready for more data. Speed is excellent. Memory use is excellent. Portability is excellent. Examples of this generation: Spinner, Open Market, and thttpd. Perhaps Apache will switch

to this method at some point. I really can't understand why they went with that complicated pre-forking stuff. Using non-blocking I/O is just not that hard.

References

- RFC 1945 — Hypertext Transfer Protocol – HTTP/1.0
- RFC 1521 — MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies RFCs are available at http://www.ietf.org/rfc.html http://www.rfc-editor.org/
- CLF (Common Logfile Format) http://httpd.apache.org/docs/mod/mod log common.html
- A list of log analyzers: http://www.mela.de/Unix/log.html
- The Webalizer, "a fast, free web server log file analysis program" http://www.mrunix.net/webalizer/
- The Apache HTTP Server Project: http://httpd.apache.org/
- thttpd, tiny/turbo/throttling HTTP server: http://www.acme.com/software/thttpd/
- PHP — a widely-used general-purpose scripting language: http://www.php.net/
- Hammerhead 2, a web testing tool: http://hammerhead.sourceforge.net
- Nessus, a remote security scanner: http://www.nessus.org