5.3 Web Servers

5.3.1 Introduction to Web Servers

One of the most popular ways Linux is used is as a platform for running a Web server. Most people these days are familiar with the World-Wide Web (WWW). Much like many other distributed systems described in this book, the WWW is built on the client-server model. In the Web, clients are the people who "surf," using browsers such as Internet Explorer or Mozilla, generating Web requests that are sent to Web Servers, which respond to these requests. The server is responsible for receiving the request, taking the appropriate actions to find and process the request and then sending the proper response to the client. Web servers thus implement the server side functionality in the WWW, and communicate with clients using the HyperText Transfer Protocol (HTTP). HTTP is the standard by which clients and servers communicate, allowing interoperability between different vendors and different software. In this chapter we provide an overview of the following:

1. What Web servers do;
2. How Web servers use the network;
3. What steps Web servers take to service requests;
4. What concurrency models are used;
5. Common tuning options for Web servers, and
6. How Web server performance is evaluated.

In this chapter we focus on how Web servers deal with static content, such as HTML files and GIF images. By static content, we mean that the HTTP *responses* that are provided by the server change relatively infrequently, say, through human intervention. In this context, Web servers are similar to file servers in that their main function is to distribute files, albeit files that have special meaning and interpretation to HTTP clients. Web servers are also capable of producing content that is generated more *dynamically*, namely, through a parameter-driven program such as CGI and PHP. However, dynamic content generation has evolved considerably beyond simple HTTP, and thus is the subject of the next chapter. In this chapter we stick to relatively simple HTTP requests.

5.3.2 HTTP Requests and Responses

HTTP requests and responses are unusual compared to other client-server exchanges in that they are ASCII text based, rather than binary encoded as is done, say, in NFS. This is an artifact of how the Web was developed, yet it is convenient in that it allows humans to easily read the generated requests and responses. The following is an example of a Web request generated by a browser (in this case, Mozilla):

```
GET /index.html HTTP/1.1
Host: www.kernel.org
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.0.2)
Accept:text/xml,application/xml,application/xhtml+xml,text/html;
```

```
q=0.9,text/plain;q=0.8,video/x-mng,image/png,image/jpeg,
image/gif;q=0.2,text/css,*/*;q=0.1
Accept-Language: en-us, en;q=0.50
Accept-Encoding: identity;q=1.0, *;q=0
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Cache-Control: max-age=0
Connection: close
<cr><lf>
```

Note that the first line contains the request for the file desired, and that each following line contains headers with appropriate values. The GET request both specifies the file requested and the protocol version used by the client. The headers communicate information to the server about what kinds of features this particular client supports. In this case, the client is Mozilla, accepts various formats such as HTML text, GIF and JPEG and XML, and uses the English language. This negotiation allows clients and servers to dynamically learn each other's capabilities so that they can communicate most effectively. We continue the example with the server's ASCII HTTP response:

```
HTTP/1.1 200 OK
Date: Wed, 17 Mar 2004 21:38:55 GMT
Server: Apache/2.0.40 (Red Hat Linux)
Accept-Ranges: bytes
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html
<cr><lf>
50c
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<!-- $Id: index.shtml,v 1.222 2004/02/24 02:05:15 hpa Exp $ -->
<HTML>
<HEAD>
   <META HTTP-EQUIV="Content-Type" CONTENT="text/html;
    charset=utf-8">
   <TITLE>The Linux Kernel Archives</TITLE>
   <LINK REL="icon" TYPE="image/png" HREF="images/tux16-16.png">
</HEAD>
<BODY TEXT="#000000" BGCOLOR="#FFFFFF" LINK="#0000E0"
VLINK="#8A1A49" ALINK="#ff0000" BACKGROUND="images/splash.png">

<CENTER><P><H1>The Linux Kernel Archives</H1></CENTER>
<P>
<CENTER>
Welcome to the Linux Kernel Archives.  This is the primary site
for the Linux kernel source, but it has much more than just
kernels.
</CENTER>
```

This response shows that the server understood the client request and is providing the response (indicated by the 200 OK message). In addition, the server uses the headers to tell the client that it is using chunked encoding and understands byte range requests. Finally, the HTML content is returned, which is parsed and displayed by the browser.

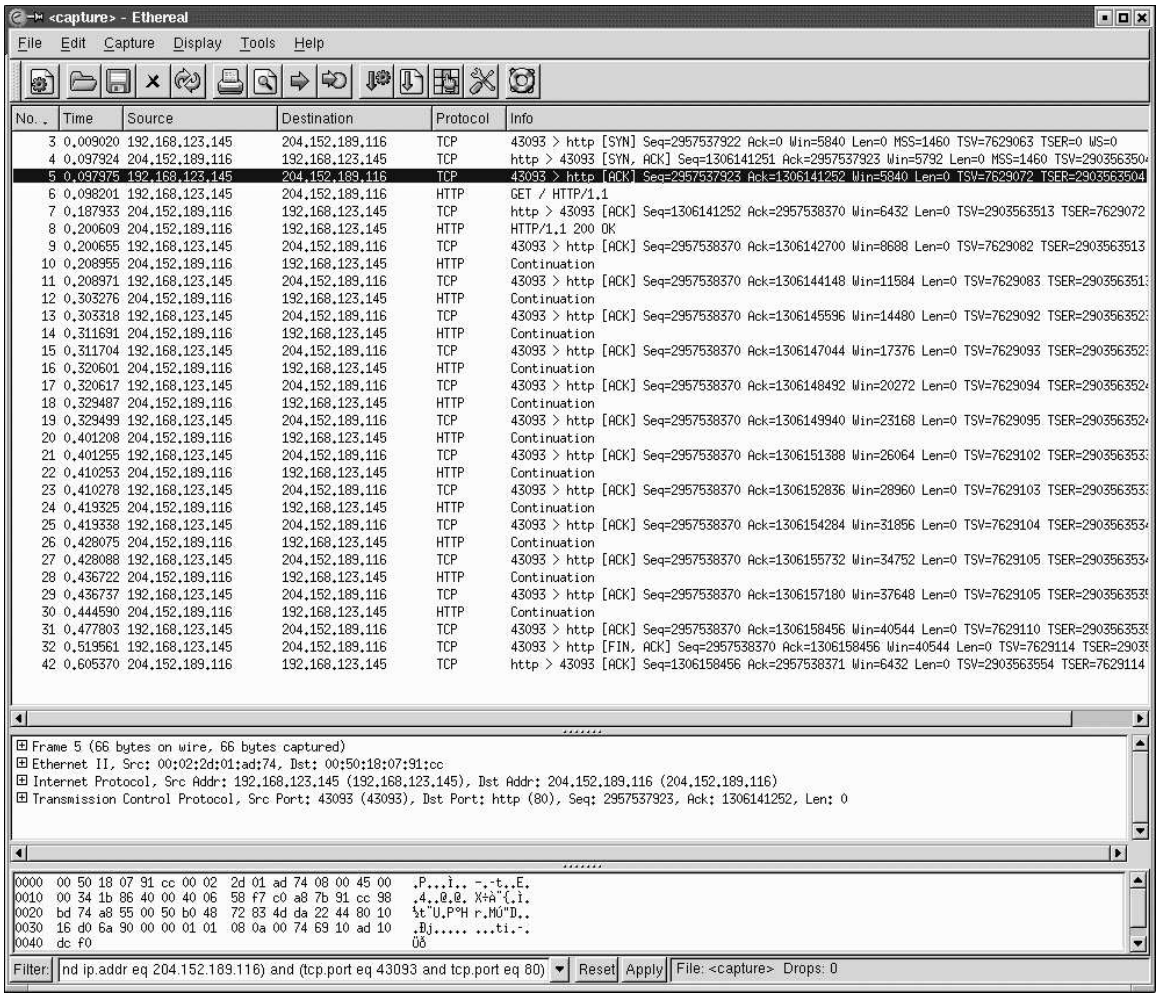
This exchange is just an example of *one* of the ways in which HTTP works, albeit perhaps the most common example. HTTP is a large, complex protocol and elaborating its many intricacies is beyond the scope of this book. We refer the reader to [Krishnamurthy and Rexford 2001] for an excellent overview of HTTP.

5.3.3 Network Behavior of a Web Server

Like other protocols, HTTP is layered above the TCP/IP stack. The following diagram provides an illustration of this layering, using the same ISO model from Chapter 5.2:
:

Application	Browser (e.g., Mozilla)
Presentation	
Session	HTTP
Transport	TCP
Network	IP
Data Link	IEEE 802.3
Physical	Ethernet

Using the same packet capture tool Ethereal used in Chapter 5.2, we can see how the individual requests and responses from Section 5.3.2 use the network. The following diagram shows the packet exchange used in that example:



The first three packets illustrate the TCP 3-way handshake used to establish a connection to the Web server. The fourth packet contains the HTTP request from the client., and the sixth contains the response header with the 200 OK message. The remaining packets are mostly either data packets containing the body of the HTTP response (packets from the server to the client) or the TCP acknowledgments for that data (packets from the client to the server). The final four packets are the 4-way handshake used to shut down the connection (the first FIN from the server is packet number 30; the FIN bit is not visible in the diagram but it is there if you expand the view of the packet).

The next diagram illustrates one packet in detail:

Zooming in on packet 8, which contains the HTTP response headers from the server, we see that this is a HTTP packet encapsulated on top of TCP, which in turn is embedded in an IP packet, which itself is the payload of an Ethernet packet.

5.3.4 Anatomy of a Web Server Transaction

In this section we provide an overview of the steps a Web server takes in response to a client request. For this purpose, we provide the following pseudo-code:

```
s = socket(); /* allocate listen socket */
bind(s, 80);  /* bind to TCP port 80    */
listen(s);    /* indicate willingness to accept */
while (1) {
       newconn = accept(s);                          /* accept new connection */
       remoteIP = getsockname(newconn);              /* get remote IP addr */
       remoteHost = gethostbyname(remoteIP);         /* get remote IP DNS name */
       gettimeofday(currentTime);                    /* determine time of day */
       read(newconn, reqBuffer, sizeof(reqBuffer)); /* read client request */
       reqInfo = serverParse(reqBuffer);             /* parse client request */
       fileName = parseOutFileName(requestBuffer);  /* determine file name */
       fileAttr = stat(fileName);                    /* get file attributes */
       serverCheckFileStuff(fileName, fileAttr);     /* check permissions */
       open(fileName);                               /* open file */
       read(fileName, fileBuffer);                   /* read file into buffer */
       headerBuffer = serverFigureHeaders(fileName, /* determine headers */
                      reqInfo);
       write(newSock, headerBuffer);                 /* write headers to socket */
       write(newSock, fileBuffer);                   /* write file to socket */
       close(newSock);                               /* close socket */
       close(fileName);                              /* close file */
       write(logFile, requestInfo);                  /* write log info to disk */
}
```

The above is a relatively simple implementation of a server, which does not perform any possible optimizations and can handle only one request at a time. The example only hints at the more complex functionality that is required by the server, such as how to parse the HTTP request and determine whether the client has the appropriate permissions to view the file. In addition, it has no error handling, for example if the client requests a file that does not exist. However, the example gives a good idea of what steps are required by a server.

5.3.5 Different Models of Web Servers

Like many other servers, Web servers have certain performance requirements that affect how the server is implemented. Several possibilities are available for the architectural model that the server is implemented with. One of the major issues is how the server will handle *concurrency*. Web servers are required to handle many clients simultaneously, sometimes up to tens of thousands of clients. In the example above, the server only deals with one client at a time. For example, if the file that the client requests is not in the servers' file cache, the server will *block* waiting for the file to be loaded from disk. During this time, the server could be handling other requests that may be less expensive to serve, but the above example will instead wait, wasting cycles. In addition, since the example above is using a single process, if the server is an SMP, other processors will be completely under-utilized. The typical approach to dealing with this problem is through some form of concurrency mechanism; we describe several approaches below.

The most common form of concurrency is using processes. The most popular Web server, Apache, was originally implemented using processes. Each process has a separate address space and is fully protected and isolated from other processes through a virtual machine abstraction. By assigning each request to a separate process, one process can make forward progress while another is blocked on another activity (such as waiting for a

client or a disk). In addition, on an SMP, multiple processes can run in parallel. The disadvantage of processes is that they are relatively expensive abstractions to use, requiring resources such as memory to be allocated to them. If a server has thousands of clients, it may have thousands of processes, which can tax a system's resources. Typically, any system is limited in the number of processes it can have active.

The next most common approach is using threads. Apache 2.0 provides the option of using threads rather than processes. Threads are similar to processes but are "lighter-weight", namely, require fewer system resources. Threads typically share address spaces but have separate program counters and stacks. Threads are cheaper than processes but trade off protection for speed. In addition, systems can run low on resources even using threads, given a large enough number of clients.

Many research Web servers, such as the Flash Web server from Rice University (not to be confused with MacroMedia's Flash browser plug-in), use something called the event-driven model. In this model, a single process is used, without threads, and connections are managed using a multiplexing system call such as select() or poll(). The server queries the readiness of all connections using the system call, and determines which connections require service (and conversely, which are idle). The server then reads requests or writes responses as appropriate for that particular connection. The advantage of the event-driven model is that it is typically faster than process or thread-based servers, since it maximizes locality and has many performance optimizations not available to these other models. The disadvantage is that it is more difficult to program, since the developer must explicitly manage state for many requests, each of which can be in a different stage of progress. In addition, operating system support for interacting with the disk in this model (namely, asychronously) has been historically absent in Linux. However, with the introduction of asynchronous I/O in Linux 2.6 this limitation is being remedied.

The final architectural model we consider here is the in-kernel model. Examples include RedHat's Tux server and IBM's AFPA server. In this approach, the entire Web server is run as a set of kernel threads, rather than in user space. The advantage to this approach is that performance is maximized, since sharing is easy and no expensive user-kernel crossings are incurred. The disadvantage is that kernel programming is more difficult, less portable, and more dangerous than user-space programming. For example, if the server has a programming error and crashes, an in-kernel server will take down the whole machine with it.

5.3.6 Tuning Web Servers

Many of the approaches to tuning other servers given in other chapters are appropriate for Web servers as well. For example, increasing the size of the send and receive socket buffers via sysctl or setsockopt, as described in Section 4.6, are useful. Similarly, increasing the size of the accept queue helps prevent requests from being dropped by the operating system before the Web server even sees them. In this section we focus on

tuning that is done for Web servers in particular. This in addition to tuning that is done on the operating system.

5.3.6.1 Tuning for all Web servers

These changes are useful for all Web servers, regardless of the architectural model.

```
echo 30000 > /proc/sys/net/ipv4/tcp_max_syn_backlog
```

This parameter increases the number of TCP SYN packets that the server can queue before SYNs are dropped.

```
echo 2000000 > /proc/sys/net/ipv4/tcp_max_tw_buckets
```

Web servers typically have a large number of TCP connections in the TIME-WAIT state. This parameter increases the number connections that are allowed in that state.

```
echo 50000 > /proc/sys/net/core/netdev_max_backlog
```

This parameter sets the length for the number of packets that can be queued in the network core (below the IP layer). This allows more memory to be used for incoming packets, which would otherwise be dropped.

5.3.6.2 Apache

Apache's main configuration file is httpd.conf. Several parameters can be modified in that file to improve performance:

```
MaxClients 150
```

This parameter sets the upper bound on the number of processes that Apache can have running concurrently. Larger values allow larger numbers of clients to be served simultaneously. Very large values may require Apache to be re-compiled.

```
MaxKeepAliveRequests 100
```

This parameter indicates the number of requests that a single Apache process will perform on a connection for a client before it closes the connection. Opening and closing connections consumes CPU cycles, and thus it is better for the server to provide as many responses on a single connection as possible, so larger numbers are better. In fact, setting this value to 0 indicates that the server should never close the connection if possible.

```
MaxRequestsPerChild 0
```

This parameter determines the number of requests an individual Apache process will serve before it dies (and is re-forked()). 0 implies the number is unlimited, but on some

systems with memory leaks in the operating system, setting this to a non-zero value keeps the memory leak under control.

```
MinSpareServers 5
MaxSpareServers 10
```

These parameters determine the minimum and maximum number of idle processes that Apache keeps around in anticipation of new requests coming in. The idea is that it is cheaper to keep an live process idle than to fork a new process in response to a new request arrival. For highly loaded sites one may wish to increase these values.

5.3.6.3 Flash and other Event-Driven Servers

A common problem event-driven servers have is that they use a large number of file descriptors, and thus can run out of these descriptors if the maximum is not increased.

```
ulimit -n 16384
```

The above is a sh (shell) command that increases the number of open files a process may have. The process (in this case, the Web server) must also be modified to take advantage of large numbers of descriptors:

```
#include <bits/types.h>
#undef  __FD_SETSIZE
#define __FD_SETSIZE 16384
```

The default FD_SETSIZE on Linux 2.4 is only 1024.

5.3.6.3 Tux

```
echo 20000 > /proc/sys/net/tux/max_connect
```

This parameter determines the number of active simultaneous connections.

```
echo 8192 > /proc/sys/net/tux/max_backlog
```

This parameter sets the max number of connections waiting in Tux's accept queue.

```
echo 0 > /proc/sys/net/tux/logging
```

This parameter disables logging of requests to disk.

5.6.5 Performance Tools for Evaluating Web Servers

Many tools are available for evaluating the performance of Web servers, also known as *workload generators*. These are programs that run on client machines, emulating the behavior of a client, constructing HTTP requests, and sending them to the server. The

workload generator can typically vary the volume of requests that it generates, called *load*, and measures how the server behaves in response to that load. Performance metrics include items such as request latency (how long it took for an individual response to come back from the server) and throughput (how many responses a server can generate per second).

Perhaps the most commonly used tool is SPECWeb99. This tool is distributed by the Systems Performance Evaluaton Cooperative (SPEC) non-profit organization, whose Web site is www.spec.org. This tool is probably the most-cited benchmark, and is used for marketing purposes by server vendors such as IBM, Sun, and Microsoft. Unfortunately, the tool costs money, although it is available freely to member institutions such as IBM. The benchmark is intended to capture the main performance characteristics that have been observed in Web servers, such as the size distribution and popularity of files requested. The tool is considered a macro-benchmark in that it is meant to measure whole system performance.

Another tool frequently used is httperf from HP Labs, which is available freely under an open-source license. This tool is highly configurable, allowing one to stress isolated components of a Web server, for example how well a server handles many idle connections. It is thus used more as a micro-benchmark.

Many other tools exist for evaluating Web server performance, including SURGE, WebBench, and WaspClient. However, describing them all is outside the scope of this chapter. Nevertheless, many options are available for stressing, testing, and measuring servers, and many of these are freely available.