CHAPTER 5

Web Server Control, Monitoring, Upgrade, and Maintenance

This chapter covers everything about administering a running mod_perl server. First, we will explain techniques for starting, restarting, and shutting down the server. As with Perl, there's more than one way to do it, and each technique has different implications for the server itself and the code it runs. A few widely used techniques for operating a server are presented. You may choose to use one of the suggested techniques or develop your own.

Later in the chapter, we give instructions on upgrading and disabling scripts on a live server, using a three-tier scheme, and monitoring and maintaining a web server.

Starting the Server in Multi-Process Mode

To start Apache manually, just run its executable. For example, on our machine, a mod_perl-enabled Apache executable is located at */home/httpd/httpd_perl/httpd_perl*. So to start it, we simply execute:

```
panic% /home/httpd/httpd_perl/bin/httpd_perl
```

This executable accepts a number of optional arguments. To find out what they are (without starting the server), use the *-h* argument:

```
panic% /home/httpd/httpd_perl/bin/httpd_perl -h
```

The most interesting arguments will be covered in the following sections. Any other arguments will be introduced as needed.

Starting the Server in Single-Process Mode

When developing new code, it is often helpful to run the server in single-process mode. This is most often used to find bugs in code that seems to work fine when the server starts, but refuses to work correctly after a few requests have been made. It also helps to uncover problems related to collisions between module names.

Running in single-process mode inhibits the server from automatically running in the background. This allows it to more easily be run under the control of a debugger. The *-X* switch is used to enable this mode:

```
panic% /home/httpd/httpd_perl/bin/httpd_perl -X
```

With the *-X* switch, the server runs in the foreground of the shell, so it can be killed by typing Ctrl-C. You can run it in the background by appending an ampersand:

```
panic% /home/httpd/httpd_perl/bin/httpd_perl -X &
```

Note that in *-X* (single-process) mode, the server will run very slowly when fetching images. Because only one request can be served at a time, requests for images normally done in parallel by the browser will now be serialized, making the page display slower.

Note for Netscape Users

If Netscape is being used as the test browser while the server is running in single-process mode, the HTTP protocol's `KeepAlive` feature gets in the way. Netscape tries to open multiple connections and keep them all open, as this should be faster for browsing. But because there is only one server process listening, each connection has to time out before the next one succeeds. Turn off `KeepAlive` in *httpd.conf* to avoid this effect while testing. Assuming you use `width` and `height` image size parameters in your HTML files, Netscape will be able to render the page without the images, so you can press the browser's Stop button after a few seconds to speed up page display. It's always good practice to specify `width` and `height` image size parameters.

Also note that when running with *-X*, the control messages that the parent server normally writes to *error_log* (e.g., "server started", "server stopped", etc.) will not be written anywhere. *httpd -X* causes the server to handle all requests itself without forking any children, so there is no controlling parent to write the status messages.

Usually Ctrl-C is used to kill a server running in single process mode, but Ctrl-C doesn't constitute a clean shutdown. *httpd.pid* doesn't get removed, so the next time the server is started, the message:

```
[warn] pid file /home/httpd/httpd_perl/logs/httpd.pid
overwritten -- Unclean shutdown of previous Apache run?
```

will appear in *error_log*. You can ignore this warning; there's nothing to worry about.

Using kill to Control Processes

Linux and other Unix-like operating systems support a form of interprocess communication called *signals*. The *kill* command is used to send a signal to a running process. How a process responds to a signal, if it responds at all, depends on the specific signal sent and on the handler set by the process. If you are familiar with Unix signal handling, you will find that Apache adheres to the usual conventions, and you can probably skip this section. This section describes the use of *kill* in relation to Apache for readers who aren't accustomed to working with signals.

The name "kill" is a misnomer; it sounds as if the command is inherently destructive, but *kill* simply sends signals to programs. Only a few signals will actually kill the process by default. Most signals can be caught by the process, which may choose to either perform a specific action or ignore the signal. When a process is in a zombie or uninterruptible `sleep( )` state, it might ignore any signals.

The following example will help dispel any fear of using this command. Most people who are familiar with the command line know that pressing Ctrl-C will usually terminate a process running in a console. For example, it is common to execute:

```
panic% tail -f /home/httpd/httpd_perl/logs/error_log
```

to monitor the Apache server's *error_log* file. The only way to stop *tail* is by pressing Ctrl-C in the console in which the process is running. The same result can be achieved by sending the INT (interrupt) signal to this process. For example:

```
panic% kill -INT 17084
```

When this command is run, the *tail* process is aborted, assuming that the process identifier (PID) of the *tail* process is 17084.

Every process running in the system has its own PID. *kill* identifies processes by their PIDs. If *kill* were to use process names and there were two *tail* processes running, it might send the signal to the wrong process. The most common way to determine the PID of a process is to use *ps* to display information about the current processes on the machine. The arguments to this utility vary depending on the operating system. For example, on BSD-family systems, the following command works:

```
panic% ps auxc | grep tail
```

On a System V Unix flavor such as Solaris, the following command may be used instead:

```
panic% ps -eaf | grep tail
```

In the first part of the command, *ps* prints information about all the current processes. This is then piped to a *grep* command that prints lines containing the text "tail". Assuming only one such *tail* process is running, we get the following output:

```
root  17084  0.1  0.1  1112  408  pts/8  S  17:28  0:00  tail
```

The first column shows the username of the account running the process, the second column shows the PID, and the last column shows the name of the command. The other columns vary between operating systems.

Processes are free to ignore almost all signals they receive, and there are cases when they will. Let's run the *less* command on the same *error_log* file:

```
panic% less /home/httpd/httpd_perl/logs/error_log
```

Neither pressing Ctrl-C nor sending the INT signal will kill the process, because the implementers of this utility chose to ignore that signal. The way to kill the process is to type *q*.

Sometimes numerical signal values are used instead of their symbolic names. For example, 2 is normally the numeric equivalent of the symbolic name INT. Hence, these two commands are equivalent on Linux:

```
panic% kill -2 17084
panic% kill -INT 17084
```

On Solaris, the *-s* option is used when working with symbolic signal names:

```
panic% kill -s INT 17084
```

To find the numerical equivalents, either refer to the *signal(7)* manpage, or ask Perl to help you:

```
panic% perl -MConfig -e 'printf "%6s %2d\n", $_, $sig++ \
                    for split / /, $Config{sig_name}'
```

If you want to send a signal to all processes with the same name, you can use *pkill* on Solaris or *killall* on Linux.

kill Signals for Stopping and Restarting Apache

Apache performs certain actions in response to the KILL, TERM, HUP, and USR1 signals (as arguments to *kill*). All Apache system administrators should be familiar with the use of these signals to control the Apache web server.

By referring to the *signal.h* file, we learn the numerical equivalents of these signals:

```
#define SIGHUP     1    /* hangup, generated when terminal disconnects */
#define SIGKILL    9    /* last resort */
#define SIGTERM   15    /* software termination signal */
#define SIGUSR1   30    /* user defined signal 1 */
```

The four types of signal are:

KILL signal: forcefully shutdown

The KILL (9) signal should *never* be used unless absolutely necessary, because it will unconditionally kill Apache, without allowing it to clean up properly. For example, the *httpd.pid* file will not be deleted, and any existing requests will simply be terminated halfway through. Although failure to delete *httpd.pid* is harmless, if code was registered to run upon child exit but was not executed because Apache was sent the KILL signal, you may have problems. For example, a database connection may be closed incorrectly, leaving the database in an inconsistent state.

The three other signals have safe and legitimate uses, and the next sections will explain what happens when each of them is sent to an Apache server process.

It should be noted that these signals should be sent only to the *parent* process, not to any of the child processes. The parent process PID may be found either by using *ps auxc | grep apache* (where it will usually be the lowest-numbered Apache process) or by executing *cat* on the *httpd.pid* file. See "Finding the Right Apache PID," later in this chapter, for more information.

TERM signal: stop now

Sending the TERM signal to the parent causes it to attempt to kill off all its children immediately. Any requests in progress are terminated, and no further requests are accepted. This operation may take tens of seconds to complete. To stop a child, the parent sends it an HUP signal. If the child does not die before a predetermined amount of time, the parent sends a second HUP signal. If the child fails to respond to the second HUP, the parent then sends a TERM signal, and if the child still does not die, the parent sends the KILL signal as a last resort. Each failed attempt to kill a child generates an entry in the *error_log* file.

Before each process is terminated, the Perl cleanup stage happens, in which Perl `END` blocks and global objects' `DESTROY` methods are run.

When all child processes have been terminated, all open log files are closed and the parent itself exits.

Unless an explicit signal name is provided, *kill* sends the TERM signal by default. Therefore:

```
panic# kill -TERM 1640
```

and:

```
panic# kill 1640
```

will do the same thing.

HUP signal: restart now

Sending the HUP signal to the parent causes it to kill off its children as if the TERM signal had been sent. That is, any requests in progress are terminated, but the parent does not exit. Instead, the parent rereads its configuration files, spawns a new set of child processes, and continues to serve requests. It is almost equivalent to stopping and then restarting the server.

If the configuration files contain errors when restart is signaled, the parent will exit, so it is important to check the configuration files for errors before issuing a restart. We'll cover how to check for errors shortly.

Using this approach to restart mod_perl-enabled Apache may cause the processes' memory consumption to grow after each restart. This happens when Perl code loaded in memory is not completely torn down, leading to a memory leak.

USR1 signal: gracefully restart now

The USR1 signal causes the parent process to advise the children to exit after serving their current requests, or to exit immediately if they are not serving a request. The parent rereads its configuration files and reopens its log files. As each child dies off, the parent replaces it with a child from the new generation (the new children use the new configuration) and the new child processes begin serving new requests immediately.

The only difference between USR1 and HUP is that USR1 allows the children to complete any current requests prior to terminating. There is no interruption in the service, unlike with the HUP signal, where service is interrupted for the few (and sometimes more) seconds it takes for a restart to complete.

By default, if a server is restarted using the USR1 or the HUP signal and mod_perl is not compiled as a DSO, Perl scripts and modules are not reloaded. To reload modules pulled in via `PerlRequire`, `PerlModule`, or `use`, and to flush the `Apache::Registry` cache, either completely stop the server and then start it again, or use this directive in *httpd.conf*:

```
PerlFreshRestart On
```

(This directive is not always recommended. See Chapter 22 for further details.)

Speeding Up Apache's Termination and Restart

Restart or termination of a mod_perl server may sometimes take quite a long time, perhaps even tens of seconds. The reason for this is a call to the `perl_destruct()` function during the child exit phase, which is also known as the cleanup phase. In this phase, the Perl `END` blocks are run and the `DESTROY` method is called on any global objects that are still around.

Sometimes this will produce a series of messages in the *error_log* file, warning that certain child processes did not exit as expected. This happens when a child process, after a few attempts have been made to terminate it, is still in the middle of `perl_destruct()`. So when you shut down the server, you might see something like this:

```
[warn]   child process 7269 still did not exit,
         sending a SIGTERM
[error]  child process 7269 still did not exit,
         sending a SIGKILL
[notice] caught SIGTERM, shutting down
```

First, the parent process sends the TERM signal to all of its children, without logging a thing. If any of the processes still doesn't quit after a short period, it sends a second TERM, logs the PID of the process, and marks the event as a warning. Finally, if the process still hasn't terminated, it sends the KILL signal, which unconditionaly terminates the process, aborting any operation in progress in the child. This event is logged as an error.

If the mod_perl scripts do not contain any END blocks or DESTROY methods that need to be run during shutdown, or if the ones they have are nonessential, this step can be avoided by setting the PERL_DESTRUCT_LEVEL environment variable to -1. (The -1 value for PERL_DESTRUCT_LEVEL is special to mod_perl.) For example, add this setting to the *httpd.conf* file:

```
PerlSetEnv PERL_DESTRUCT_LEVEL -1
```

What constitutes a significant cleanup? Any change of state outside the current process that cannot be handled by the operating system itself. Committing database transactions and removing the lock on a resource are significant operations, but closing an ordinary file is not. For example, if DBI is used for persistent database connections, Perl's destructors should *not* be switched off.

Finding the Right Apache PID

In order to send a signal to a process, its PID must be known. But in the case of Apache, there are many *httpd* processes running. Which one should be used? The parent process is the one that must be signaled, so it is the parent's PID that must be identified.

The easiest way to find the Apache parent PID is to read the *httpd.pid* file. To find this file, look in the *httpd.conf* file. Open *httpd.conf* and look for the PidFile directive. Here is the line from our *httpd.conf* file:

```
PidFile /home/httpd/httpd_perl/logs/httpd.pid
```

When Apache starts up, it writes its own process ID in *httpd.pid* in a human-readable format. When the server is stopped, *httpd.pid* should be deleted, but if Apache is killed abnormally, *httpd.pid* may still exist even if the process is not running any more.

Of course, the PID of the running Apache can also be found using the *ps(1)* and *grep(1)* utilities (as shown previously). Assuming that the binary is called *httpd_perl*, the command would be:

```
panic% ps auxc | grep httpd_perl
```

or, on System V:

```
panic% ps -ef | grep httpd_perl
```

This will produce a list of all the *httpd_perl* (parent and child) processes. If the server was started by the *root* user account, it will be easy to locate, since it will belong to *root*. Here is an example of the sort of output produced by one of the *ps* command lines given above:

```
root   17309 0.9 2.7 8344 7096 ?  S 18:22 0:00 httpd_perl
nobody 17310 0.1 2.7 8440 7164 ?  S 18:22 0:00 httpd_perl
nobody 17311 0.0 2.7 8440 7164 ?  S 18:22 0:00 httpd_perl
nobody 17312 0.0 2.7 8440 7164 ?  S 18:22 0:00 httpd_perl
```

In this example, it can be seen that all the child processes are running as user *nobody* whereas the parent process runs as user *root*. There is only one *root* process, and this must be the parent process. Any *kill* signals should be sent to this parent process.

If the server is started under some other user account (e.g., when the user does not have *root* access), the processes will belong to that user. The only truly foolproof way to identify the parent process is to look for the process whose parent process ID (PPID) is 1 (use *ps* to find out the PPID of the process).

If you have the GNU tools installed on your system, there is a nifty utility that makes it even easier to discover the parent process. The tool is called *pstree*, and it is very simple to use. It lists all the processes showing the *family* hierarchy, so if we *grep* the output for the wanted process's family, we can see the parent process right away. Running this utility and *grep*ing for *httpd_perl*, we get:

```
panic% pstree -p | grep httpd_perl
  |-httpd_perl(17309)-+-httpd_perl(17310)
  |                   |-httpd_perl(17311)
  |                   |-httpd_perl(17312)
```

And this one is even simpler:

```
panic% pstree -p | grep 'httpd_perl.*httpd_perl'
  |-httpd_perl(17309)-+-httpd_perl(17310)
```

In both cases, we can see that the parent process has the PID 17309.

ps's *f* option, available on many Unix platforms, produces a tree-like report of the processes as well. For example, you can run *ps axfwwww* to get a tree of all processes.

Using apachectl to Control the Server

The Apache distribution comes with a script to control the server called *apachectl*, installed into the same location as the *httpd* executable. For the sake of the examples, let's assume that it is in */home/httpd/httpd_perl/bin/apachectl*.

All the operations that can be performed by using signals can also be performed on the server by using *apachectl*. You don't need to know the PID of the process, as *apachectl* will find this out for itself.

To start *httpd_perl*:

```
panic% /home/httpd/httpd_perl/bin/apachectl start
```

To stop *httpd_perl*:

```
panic% /home/httpd/httpd_perl/bin/apachectl stop
```

To restart *httpd_perl* (if it is running, send HUP; if it is not, just start it):

```
panic% /home/httpd/httpd_perl/bin/apachectl restart
```

Do a graceful restart by sending a USR1 signal, or start it if it's not running:

```
panic% /home/httpd/httpd_perl/bin/apachectl graceful
```

To perform a configuration test:

```
panic% /home/httpd/httpd_perl/bin/apachectl configtest
```

There are other options for *apachectl*. Use the *help* option to see them all.

```
panic% /home/httpd/httpd_perl/bin/apachectl help
```

It is important to remember that *apachectl* uses the PID file, which is specified by the `PidFile` directive in *httpd.conf*. If the PID file is deleted by hand while the server is running, or if the `PidFile` directive is missing or in error, *apachectl* will be unable to stop or restart the server.

Validating Server Configuration

If the configuration file has syntax errors, attempting to restart the server will fail and the server will die. However, if a graceful restart is attempted using *apachectl* and the configuration file contains errors, the server will issue an error message and continue running with the existing configuration. This is because *apachectl* validates the configuration file before issuing the actual restart command when a graceful restart is requested.

Apache provides a method to check the configuration's syntax without actually starting the server. You can run this check at any time, whether or not a server is currently running. The check has two forms, using the *-t* or *-T* options. For example:

```
panic% /home/httpd/httpd_perl/bin/httpd_perl -t
```

-t will verify that the `DocumentRoot` directory exists, whereas *-T* will not. *-T* is most useful when using a configuration file containing a large number of virtual hosts, where verifying the existence of each `DocumentRoot` directory can take a substantial amount of time.

Note that when running this test with a mod_perl server, the Perl code will be executed just as it would be at server startup—that is, from within the *httpd.conf* `<Perl>` sections or a startup file.

Setuid root Startup Scripts

If a group of developers need to be able to start and stop the server, there may be a temptation to give them the *root* password, which is probably not a wise thing to do. The fewer people that know the *root* password, the less likely you will encounter problems. Fortunately, an easy solution to this problem is available on Unix platforms. It is called a *setuid executable* (setuid *root* in this case).

Before continuing, we must stress that this technique should not be used unless it is absolutely necessary. If an improperly written setuid script is used, it may compromise the system by giving *root* privileges to system breakers (crackers).

To be on the safe side, do not deploy the techniques explained in this section. However, if this approach is necessary in a particular situation, this section will address the possible problems and provide solutions to reduce the risks to a minimum.

Introduction to setuid Executables

A setuid executable has the setuid permissions bit set, with the following command:

```
panic% chmod u+s filename
```

This sets the process's effective user ID to that of the file upon execution. Most users have used setuid executables even if they have not realized it. For example, when a user changes his password he executes the *passwd* command, which, among other things, modifies the */etc/passwd* file. In order to change this file, the *passwd* program needs *root* permissions. The *passwd* command has the setuid bit set, so when someone executes this utility, its effective ID becomes the *root* user ID.

Using setuid executables should be avoided as a general practice. The less setuid executables there are in a system, the less likely it is that someone will find a way to break in. One approach that crackers use is to find and exploit unanticipated bugs in setuid executables.

When the executable is setuid to *root*, it is vital to ensure that it does not extend read and write permissions to its group or to the world. Let's take the *passwd* utility as an example. Its permissions are:

```
panic% ls -l /usr/bin/passwd
-r-s--x--x 1 root root 12244 Feb 8 00:20 /usr/bin/passwd
```

The program is group- and world-executable but cannot be read or written by group or world. This is achieved with the following command:

```
panic% chmod 4511 filename
```

The first digit (4) stands for the setuid bit, the second digit (5) is a bitwise-OR of read (4) and executable (1) permissions for the user, and the third and fourth digits set the executable (1) permissions for group and world.

Apache Startup Script's setuid Security

In the situation where several developers need to be able to start and stop an Apache server that is run by the *root* account, setuid access must be available only to this specific group of users. For the sake of this example, let's assume that these developers belong to a group named *apache*. It is important that users who are not *root* or are not part of the *apache* group are unable to execute this script. Therefore, the following commands must be applied to the *apachectl* program:

```
panic% chgrp apache apachectl
panic% chmod  4510  apachectl
```

The execution order is important. If the commands are executed in reverse order, the setuid bit is lost.

The file's permissions now look like this:

```
panic% ls -l apachectl
-r-s--x--- 1 root apache 32 May 13 21:52 apachectl
```

Everything is set. Well, almost...

When Apache is started, Apache and Perl modules are loaded, so code may be executed. Since all this happens with the *root* effective ID, any code is executed as if run by the root user. This means that there is a risk, even though none of the developers has the *root* password—all users in the *apache* group now have an indirect *root* access. For example, if Apache loads some module or executes some code that is writable by any of these users, they can plant code that will allow them to gain shell access to the *root* account.

Of course, if the developers are not trusted, this setuid solution is not the right approach. Although it is possible to try to check that all the files Apache loads are not writable by anyone but *root*, there are so many of them (especially with mod_perl, where many Perl modules are loaded at server startup) that this is a risky approach.

If the developers are trusted, this approach suits the situation. Although there are security concerns regarding Apache startup, once the parent process is loaded, the child processes are spawned as non-*root* processes.

This section has presented a way to allow non-*root* users to start and stop the server. The rest is exactly the same as if they were executing the script as *root* in the first place.

Sample setuid Apache Startup Script

Example 5-1 shows a sample setuid Apache startup script.

Note the line marked *WORKAROUND*, which fixes an obscure error when starting a mod_perl-enabled Apache, by setting the real UID to the effective UID. Without this workaround, a mismatch between the real and the effective UIDs causes Perl to croak on the *-e* switch.

This script depends on using a version of Perl that recognizes and emulates the setuid bits. This script will do different things depending on whether it is named *start_httpd*, *stop_httpd*, or *restart_httpd*; use symbolic links to create the names in the filesystem.

Example 5-1. suid_apache_ctl

```
#!/usr/bin/perl -T
use strict;

# These constants will need to be adjusted.
my $PID_FILE = '/home/httpd/httpd_perl/logs/httpd.pid';
my $HTTPD = '/home/httpd/httpd_perl/bin/httpd_perl ';
$HTTPD   .= '-d /home/httpd/httpd_perl';

# These prevent taint checking failures
$ENV{PATH} = '/bin:/usr/bin';
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};

# This sets the real to the effective ID, and prevents
# an obscure error when starting apache/mod_perl
$< = $>; # WORKAROUND
$( = $) = 0; # set the group to root too

# Do different things depending on our name
my $name = $0;
$name =~ m|([^/]+)$|;

if ($name eq 'start_httpd') {
    system $HTTPD and die "Unable to start HTTPD";
    print "HTTP started.\n";
    exit 0;
}

# extract the process id and confirm that it is numeric
my $pid = `cat $PID_FILE`;
$pid =~ /^(\d+)$/ or die "PID $pid not numeric or not found";
$pid = $1;

if ($name eq 'stop_httpd') {
    kill 'TERM', $pid or die "Unable to signal HTTPD";
    print "HTTP stopped.\n";
    exit 0;
}

if ($name eq 'restart_httpd') {
    kill 'HUP', $pid or die "Unable to signal HTTPD";
    print "HTTP restarted.\n";
    exit 0;
}

# script is named differently
die "Script must be named start_httpd, stop_httpd, or restart_httpd.\n";
```

Preparing for Machine Reboot

When using a non-production development box, it is OK to start and stop the web server by hand when necessary. On a production system, however, it is possible that the machine on which the server is running will have to be rebooted. When the reboot is completed, who is going to remember to start the server? It is easy to forget this task, and what happens if no one is around when the machine is rebooted? (Some OSs will reboot themselves without human intervention in certain situations.)

After the server installation is complete, it is important to remember that a script to perform the server startup and shutdown should be put in a standard system location—for example, */etc/rc.d* under Red Hat Linux, or */etc/init.d/apache* under Debian GNU/Linux.

This book uses Red Hat-compatible Linux distributions in its examples. Let's step aside for a brief introduction to the System V (SysV) *init* system that many Linux and other Unix flavors use to manage starting and stopping daemons. (A *daemon* is a process that normally starts at system startup and runs in the background until the system goes down.)

The SysV *init* system keeps all its files in the */etc/rc.d/* directory. This directory contains a number of subdirectories:

```
panic% find /etc/rc.d -type d
/etc/rc.d
/etc/rc.d/init.d
/etc/rc.d/rc0.d
/etc/rc.d/rc1.d
/etc/rc.d/rc2.d
/etc/rc.d/rc3.d
/etc/rc.d/rc4.d
/etc/rc.d/rc5.d
/etc/rc.d/rc6.d
```

/etc/rc.d/init.d contains many scripts, one for each service that needs to be started at boot time or when entering a specific runlevel. Common services include networking, file sharing, mail servers, web servers, FTP servers, etc.

When the system boots, the special *init* script runs all scripts for the default runlevel. The default runlevel is specified in the */etc/inittab* file. This file contains a line similar to this one:

```
id:3:initdefault:
```

The second column indicates that the default runlevel is 3, which is the default for most server systems. (5 is the default for desktop machines.)

Let's now see how the scripts are run. We'll first look at the contents of the */etc/rc.d/rc3.d* directory:

```
panic% ls -l /etc/rc.d/rc3.d
lrwxrwxrwx 1 root root 13 Jul  1 01:08 K20nfs -> ../init.d/nfs
lrwxrwxrwx 1 root root 18 Jul  1 00:54 K92ipchains -> ../init.d
lrwxrwxrwx 1 root root 17 Jul  1 00:51 S10network -> ../init.d/network
lrwxrwxrwx 1 root root 16 Jul  1 00:51 S30syslog -> ../init.d/syslog
lrwxrwxrwx 1 root root 13 Jul  1 00:52 S40atd -> ../init.d/atd
lrwxrwxrwx 1 root root 15 Jul  1 00:51 S40crond -> ../init.d/crond
lrwxrwxrwx 1 root root 15 Jul  1 01:13 S91httpd_docs -> ../init.d/httpd_docs
lrwxrwxrwx 1 root root 15 Jul  1 01:13 S91httpd_perl -> ../init.d/httpd_perl
lrwxrwxrwx 1 root root 17 Jul  1 00:51 S95kheader -> ../init.d/kheader
lrwxrwxrwx 1 root root 11 Jul  1 00:51 S99local -> ../rc.local
```

(Only part of the output is shown here, since many services are started and stopped at runlevel 3.)

There are no real files in the directory. Instead, each file is a symbolic link to one of the scripts in the *init.d* directory. The links' names start with a letter (*S* or *K*) and a two-digit number. *S* specifies that the script should be run when the service is started and *K* specifies that the script should be run when the service is stopped. The number following *S* or *K* is there for ordering purposes: *init* will start services in the order in which they appear.

init runs each script with an argument that is either *start* or *stop*, depending on whether the link's name starts with *S* or *K*. Scripts can be executed from the command line; the following command line will stop the *httpd* server:

```
panic# /etc/rc.d/init.d/httpd_perl stop
```

Unfortunately, different Unix flavors implement different *init* systems. Refer to your system's documentation.

Now that we're familiar with how the *init* system works, let's return to our discussion of *apachectl* scripts.

Generally, the simplest solution is to copy the *apachectl* script to the startup directory or, better still, create a symbolic link from the startup directory to the *apachectl* script. The *apachectl* utility is in the same directory as the Apache executable after Apache installation (e.g., */home/httpd/httpd_perl/bin*). If there is more than one Apache server, there will need to be a separate script for each one, and of course they will have to have different names so that they can coexist in the same directory.

On one of our Red Hat Linux machines with two servers, we have the following setup:

```
/etc/rc.d/init.d/httpd_docs
/etc/rc.d/init.d/httpd_perl
/etc/rc.d/rc3.d/S91httpd_docs -> ../init.d/httpd_docs
/etc/rc.d/rc3.d/S91httpd_perl -> ../init.d/httpd_perl
/etc/rc.d/rc6.d/K16httpd_docs -> ../init.d/httpd_docs
/etc/rc.d/rc6.d/K16httpd_perl -> ../init.d/httpd_perl
```

The scripts themselves reside in the */etc/rc.d/init.d* directory. There are symbolic links to these scripts in */etc/rc.d/rc*.d* directories.

When the system starts (runlevel 3), we want Apache to be started when all the services on which it might depend are already running. Therefore, we have used *S91*. If, for example, the mod_perl-enabled Apache issues a `connect_on_init()`, the SQL server should be started before Apache.

When the system shuts down (runlevel 6), Apache should be one of the first processes to be stopped—therefore, we have used *K16*. Again, if the server does some cleanup processing during the shutdown event and requires third-party services (e.g., a MySQL server) to be running at the time, it should be stopped before these services.

Notice that it is normal for more than one symbolic link to have the same sequence number.

Under Red Hat Linux and similar systems, when a machine is booted and its runlevel is set to 3 (multiuser plus network), Linux goes into */etc/rc.d/rc3.d/* and executes the scripts to which the symbolic links point with the *start* argument. When it sees *S87httpd_perl*, it executes:

```
/etc/rc.d/init.d/httpd_perl start
```

When the machine is shut down, the scripts are executed through links from the */etc/rc.d/rc6.d/* directory. This time the scripts are called with the *stop* argument, like this:

```
/etc/rc.d/init.d/httpd_perl stop
```

Most systems have GUI utilities to automate the creation of symbolic links. For example, Red Hat Linux includes the *ntsysv* and *tksysv* utilities. These can be used to create the proper symbolic links. Before it is used, the *apachectl* or similar scripts should be put into the *init.d* directory or an equivalent directory. Alternatively, a symbolic link to some other location can be created.

However, it's been reported that sometimes these tools mess up and break things. Therefore, the robust *chkconfig* utility should be used instead. The following example shows how to add an *httpd_perl* startup script to the system using *chkconfig*.

The *apachectl* script may be kept in any directory, as long as it can be the target of a symbolic link. For example, it might be desirable to keep all Apache executables in the same directory (e.g., */home/httpd/httpd_perl/bin*), in which case all that needs to be done is to provide a symbolic link to this file:

```
panic% ln -s /home/httpd/httpd_perl/bin/apachectl /etc/rc.d/init.d/httpd_perl
```

Edit the *apachectl* script to add the following lines after the script's main header:

```
# Comments to support chkconfig on RedHat Linux
# chkconfig: 2345 91 16
# description: mod_perl enabled Apache Server
```

Now the beginning of the script looks like:

```
#!/bin/sh
#
# Apache control script designed to allow an easy command line
# interface to controlling Apache.  Written by Marc Slemko,
# 1997/08/23

# Comments to support chkconfig on Red Hat Linux
# chkconfig: 2345 91 16
# description: mod_perl-enabled Apache Server
```

```
#
# The exit codes returned are:
# ...
```

Adjust the line:

```
# chkconfig: 2345 91 16
```

to suit your situation. For example, the setting used above says the script should be started in levels 2, 3, 4, and 5, that its start priority should be 91, and that its stop priority should be 16.

Now all you need to do is ask *chkconfig* to configure the startup scripts. Before doing so, it is best to check what files and links are in place:

```
panic% find /etc/rc.d | grep httpd_perl

/etc/rc.d/init.d/httpd_perl
```

This response means that only the startup script itself exists. Now execute:

```
panic% chkconfig --add httpd_perl
```

and repeat the *find* command to see what has changed:

```
panic% find /etc/rc.d | grep httpd_perl

/etc/rc.d/init.d/httpd_perl
/etc/rc.d/rc0.d/K16httpd_perl
/etc/rc.d/rc1.d/K16httpd_perl
/etc/rc.d/rc2.d/S91httpd_perl
/etc/rc.d/rc3.d/S91httpd_perl
/etc/rc.d/rc4.d/S91httpd_perl
/etc/rc.d/rc5.d/S91httpd_perl
/etc/rc.d/rc6.d/K16httpd_perl
```

The *chkconfig* program has created all the required symbolic links using the startup and shutdown priorities as specified in the line:

```
# chkconfig: 2345 91 16
```

If for some reason it becomes necessary to remove the service from the startup scripts, *chkconfig* can perform the removal of the links automatically:

```
panic% chkconfig --del httpd_perl
```

By running the *find* command once more, you can see that the symbolic links have been removed and only the original file remains:

```
panic% find /etc/rc.d | grep httpd_perl

/etc/rc.d/init.d/httpd_perl
```

Again, execute:

```
panic% chkconfig --add httpd_perl
```

Note that when using symbolic links, the link name in */etc/rc.d/init.d* is what matters, not the name of the script to which the link points.

Upgrading a Live Server

When you're developing code on a development server, anything goes: modifying the configuration, adding or upgrading Perl modules without checking that they are syntactically correct, not checking that Perl modules don't collide with other modules, adding experimental new modules from CPAN, etc. If something goes wrong, configuration changes can be rolled back (assuming you're using some form of version control), modules can be uninstalled or reinstalled, and the server can be started and stopped as many times as required to get it working.

Of course, if there is more than one developer working on a development server, things can't be quite so carefree. Possible solutions for the problems that can arise when multiple developers share a development server will be discussed shortly.

The most difficult situation is transitioning changes to a live server. However much the changes have been tested on a development server, there is always the risk of breaking something when a change is made to the live server. Ideally, any changes should be made in a way that will go unnoticed by the users, except as new or improved functionality or better performance. No users should be exposed to even a single error message from the upgraded service—especially not the "database busy" or "database error" messages that some high-profile sites seem to consider acceptable.

Live services can be divided into two categories: servers that must be up 24 hours a day and 7 days a week, and servers that can be stopped during non-working hours. The latter generally applies to Intranets of companies with offices located more or less in the same time zone and not scattered around the world. Since the Intranet category is the easier case, let's talk about it first.

Upgrading Intranet Servers

An Intranet server generally serves the company's internal staff by allowing them to share and distribute internal information, read internal email, and perform other similar tasks. When all the staff is located in the same time zone, or when the time difference between sites does not exceed a few hours, there is often no need for the server to be up all the time. This doesn't necessarily mean that no one will need to access the Intranet server from home in the evenings, but it does mean that the server can probably be stopped for a few minutes when it is necessary to perform some maintenance work.

Even if the update of a live server occurs during working hours and goes wrong, the staff will generally tolerate the inconvenience unless the Intranet has become a really mission-critical tool. For servers that *are* mission critical, the following section will describe the least disruptive and safest upgrade approach.

If possible, any administration or upgrades of the company's Intranet server should be undertaken during non-working hours, or, if this is not possible, during the times of least activity (e.g., lunch time). Upgrades that are carried out while users are using the service should be done with a great deal of care.

In very large organizations, upgrades are often scheduled events and employees are notified ahead of time that the service might not be available. Some organizations deem these periods "at-risk" times, when employees are expected to use the service as little as possible and then only for noncritical work. Again, these major updates are generally scheduled during the weekends and late evening hours.

The next section deals with this issue for services that need to be available all the time.

Upgrading 24 × 7 Internet Servers

Internet servers are normally expected to be available 24 hours a day, 7 days a week. E-commerce sites, global B2B (business-to-business) sites, and any other revenue-producing sites may be critical to the companies that run them, and their unavailability could prove to be very expensive. The approach taken to ensure that servers remain in service even when they are being upgraded depends on the type of server in use. There are two categories to consider: *server clusters* and *single servers*.

The server cluster

When a service is very popular, a single machine probably will not be able to keep up with the number of requests the service has to handle. In this situation, the solution is to add more machines and to distribute the load amongst them. From the user's point of view, the use of multiple servers must be completely transparent; users must still have a single access point to the service (i.e., the same single URL) even though there may be many machines with different server names actually delivering the service. The requests must also be properly distributed across the machines: not simply by giving equal numbers of requests to each machine, but rather by giving each machine a load that reflects its actual capabilities, given that not all machines are built with identical hardware. This leads to the need for some smart load-balancing techniques.

All current load-balancing techniques are based on a central machine that dispatches all incoming requests to machines that do the real processing. Think of it as the only entrance into a building with a doorkeeper directing people into different rooms, all of which have identical contents but possibly a different number of clerks. Regardless of what room they're directed to, all people use the entrance door to enter and exit the building, and an observer located outside the building cannot tell what room people are visiting. The same thing happens with the cluster of servers—users send their browsers to URLs, and back come the pages they requested. They remain unaware of the particular machines from which their browsers collected their pages.

No matter what load-balancing technique is used, it should always be straightforward to be able to tell the central machine that a new machine is available or that some machine is not available any more.

How does this long introduction relate to the upgrade problem? Simple. When a particular machine requires upgrading, the dispatching server is told to stop sending requests to that machine. All the requests currently being executed must be left to complete, at which point whatever maintenance and upgrade work is to be done can be carried out. Once the work is complete and has been tested to ensure that everything works correctly, the central machine can be told that it can again send requests to the newly upgraded machine. At no point has there been any interruption of service or any indication to users that anything has occurred. Note that for some services, particularly ones to which users must log in, the wait for all the users to either log out or time out may be considerable. Thus, some sites stop requests to a machine at the end of the working day, in the hope that all requests will have completed or timed out by the morning.

How do we talk to the central machine? This depends on the load-balancing technology that is implemented and is beyond the scope of this book. The references section at the end of this chapter gives a list of relevant online resources.

The single server

It's not uncommon for a popular web site to run on a single machine. It's also common for a web site to run on multiple machines, with one machine dedicated to serving static objects (such as images and static HTML files), another serving dynamically generated responses, and perhaps even a third machine that acts as a dedicated database server.

Therefore, the situation that must be addressed is where just one machine runs the service or where the service is spread over a few machines, with each performing a unique task, such that no machine can be shut down even for a single minute, and leaving the service unavailable for more than five seconds is unacceptable. In this case, two different tasks may be required: upgrading the software on the server (including the Apache server), and upgrading the code of the service itself (i.e., custom modules and scripts).

Upgrading live server components by swapping machines. There are many things that you might need to update on a server, ranging from a major upgrade of the operating system to just an update of a single piece of software (such as the Apache server itself).

One simple approach to performing an upgrade painlessly is to have a backup machine, of similar capacity and identical configuration, that can replace the production machine while the upgrade is happening. It is a good idea to have such a machine handy and to use it whenever major upgrades are required. The two machines must be kept synchronized, of course. (For Unix/Linux users, tools such as *rsync* and *mirror* can be used for synchronization.)

However, it may not be necessary to have a special machine on standby as a backup. Unless the service is hosted elsewhere and you can't switch the machines easily, the development machine is probably the best choice for a backup—all the software and scripts are tested on the development machine as a matter of course, and it probably has a software setup identical to that of the production machine. The development machine might not be as powerful as the live server, but this may well be acceptable for a short period, especially if the upgrade is timed to happen when the site's traffic is fairly quiet. It's much better to have a slightly slower service than to close the doors completely. A web log analysis tool such as *analog* can be used to determine the hour of the day when the server is under the least load.

Switching between the two machines is very simple:

1. Shut down the network on the backup machine.
2. Configure the backup machine to use the same IP address and domain name as the live machine.
3. Shut down the network on the live machine (do not shut down the machine itself!).
4. Start up the network on the backup machine.

When you are certain that the backup server has successfully replaced the live server (that is, requests are being serviced, as revealed by the backup machine's *access_log*), it is safe to switch off the master machine or do any necessary upgrades.

Why bother waiting to check that everything is working correctly with the backup machine? If something goes wrong, the change can immediately be rolled back by putting the known working machine back online. With the service restored, there is time to analyze and fix the problem with the replacement machine before trying it again. Without the ability to roll back, the service may be out of operation for some time before the problem is solved, and users may become frustrated.

We recommend that you practice this technique with two unused machines before using the production boxes.

After the backup machine has been put into service and the original machine has been upgraded, test the original machine. Once the original machine has been passed as ready for service, the server replacement technique described above should be repeated in reverse. If the original machine does not work correctly once returned to service, the backup machine can immediately be brought online while the problems with the original are fixed.

You cannot have two machines configured to use the same IP address, so the first machine must release the IP address by shutting down the link using this IP before the second machine can enable its own link with the same IP address. This leads to a short downtime during the switch. You can use the *heartbeat* utility to automate this process and thus possibly shorten the downtime period. See the references section at the end of this chapter for more information about *heartbeat*.

Upgrading a live server with port forwarding. Using more than one machine to perform an update may not be convenient, or even possible. An alternative solution is to use the port-forwarding capabilities of the host's operating system.

One approach is to configure the web server to listen on an unprivileged port, such as 8000, instead of 80. Then, using a firewalling tool such as *iptables*, *ipchains*, or *ipfwadm*, redirect all traffic coming for port 80 to port 8000. Keeping a rule like this enabled at all times on a production machine will not noticeably affect performance.

Once this rule is in place, it's a matter of getting the new code in place, adjusting the web server configuration to point to the new location, and picking a new unused port, such as 8001. This way, you can start the "new" server listening on that port and not affect the current setup.

To check that everything is working, you could test the server by accessing it directly by port number. However, this might break links and redirections. Instead, add another port forwarding rule before the first one, redirecting traffic for port 80 from your test machine or network to port 8001.

Once satisfied with the new server, publishing the change is just a matter of changing the port-forwarding rules one last time. You can then stop the now old server and everything is done.

Now you have your primary server listening on port 8001, answering requests coming in through port 80, and nobody will have noticed the change.

Upgrading a live server with prepackaged components. Assuming that the testbed machine and the live server have an identical software installation, consider preparing an upgrade package with the components that must be upgraded. Test this package on the testbed machine, and when it is evident that the package gets installed flawlessly, install it on the live server. Do not build the software from scratch on the live server, because if a mistake is made, it could cause the live server to misbehave or even to fail.

For example, many Linux distributions use the Red Hat Package Manager (RPM) utility, *rpm*, to distribute source and binary packages. It is not necessary for a binary package to include any compiled code (for example, it can include Perl scripts, but it is still called a binary). A binary package allows the new or upgraded software to be used the moment you install it. The *rpm* utility is smart enough to make upgrades (i. e., remove previous installation files, preserve configuration files, and execute appropriate installation scripts).

If, for example, the mod_perl server needs to be upgraded, one approach is to prepare a package on a similarly configured machine. Once the package has been built, tested, and proved satisfactory, it can then be transferred to the live machine. The *rpm* utility can then be used to upgrade the mod_perl server. For example, if the package file is called *mod_perl-1.26-10.i386.rpm*, this command:

```
panic% rpm -Uvh mod_perl-1.26-10.i386.rpm
```

will remove the previous server (if any) and install the new one.

There's no problem upgrading software that doesn't break any dependencies in other packages, as in the above example. But what would happen if, for example, the Perl interpreter needs to be upgraded on the live machine?

If the mod_perl package described earlier was properly prepared, it would specify the packages on which it depends and their versions. So if Perl was upgraded using an RPM package, the *rpm* utility would detect that the upgrade would break a dependency, since the mod_perl package is supposed to work with the previous version of Perl. *rpm* will not allow the upgrade unless forced to.

This is a very important feature of RPM. Of course, it relies on the fact that the person who created the package has set all the dependencies correctly. Do not trust packages downloaded from the Web. If you have to use an RPM package prepared by someone else, get its source, read its specification file, and make doubly sure that it's what you want.

The Perl upgrade task is in fact a very easy problem to solve. Have two packages ready on the development machine: one for Perl and the other for mod_perl, the latter built using the Perl version that is going to be installed. Upload both of them to the live server and install them together. For example:

```
panic% rpm -Uvh mod_perl-1.26-10.i386.rpm perl-5.6.1-5.i386.rpm
```

This should be done as an *atomic* operation—i.e., as a single execution of the *rpm* program. If the installation of the packages is attempted with separate commands, they will both fail, because each of them will break some dependency.

If a mistake is made and checks reveal that a faulty package has been installed, it is easy to roll back. Just make sure that the previous version of the properly packaged software is available. The packages can be downgraded by using the *--force* option—and voilà, the previously working system is restored. For example:

```
panic% rpm -Uvh --force mod_perl-1.26-9.i386.rpm perl-5.6.1-4.i386.rpm
```

Although this example uses the *rpm* utility, other similar utilities exist for various operating systems and distributions. Creating packages provides a simple way of upgrading live systems (and downgrading them if need be). The packages used for any successful upgrade should be kept, because they will become the packages to downgrade to if a subsequent upgrade with a new package fails.

When using a cluster of machines with identical setups, there is another important benefit of prepackaged upgrades. Instead of doing all the upgrades by hand, which could potentially involve dozens or even hundreds of files, preparing a package can save lots of time and will minimize the possibility of error. If the packages are properly written and have been tested thoroughly, it is perfectly possible to make updates to machines that are running live services. (Note that not all operating systems permit the upgrading of running software. For example, Windows does not permit DLLs that are in active use to be updated.)

It should be noted that the packages referred to in this discussion are ones made locally, specifically for the systems to be upgraded, not generic packages downloaded from the Internet. Making local packages provides complete control over what is installed and upgraded and makes upgrades into atomic actions that can be rolled back if necessary. We do not recommend using third-party packaged binaries, as they will almost certainly have been built for a different environment and will not have been fine-tuned for your system.

Upgrading a live server using symbolic links. Yet another alternative is to use symbolic links for upgrades. This concept is quite simple: install a package into some directory and symlink to it. So, if some software was expected in the directory */usr/local/foo*, you could simply install the first version of the software in the directory */usr/local/foo-1.0* and point to it from the expected directory:

```
panic# ln -sf /usr/local/foo-1.0 /usr/local/foo
```

If later you want to install a second version of the software, install it into the directory */usr/local/foo-2.0* and change the symbolic link to this new directory:

```
panic# ln -sf /usr/local/foo-2.0 /usr/local/foo
```

Now if something goes wrong, you can always switch back with:

```
panic# ln -sf /usr/local/foo-1.0 /usr/local/foo
```

In reality, things aren't as simple as in this example. It works if you can place all the software components under a single directory, as with the default Apache installation. Everything is installed under a single directory, so you can have:

```
/usr/local/apache-1.3.17
/usr/local/apache-1.3.19
```

and use the symlink */usr/local/apache* to switch between the two versions.

However, if you use a default installation of Perl, files are spread across multiple directories. In this case, it's not easy to use symlinks—you need several of them, and they're hard to keep track of. Unless you automate the symlinks with a script, it might take a while to do a switch, which might mean some downtime. Of course, you can install all the Perl components under a single root, just like the default Apache installation, which simplifies things.

,ch05.22279 Page 169 Thursday, November 18, 2004 12:36 PM

Another complication with upgrading Perl is that you may need to recompile mod_perl and other Perl third-party modules that use XS extensions. Therefore, you probably want to build everything on some other machine, test it, and when ready, just un*tar* everything at once on the production machine and adjust the symbolic links.

Upgrading Perl code. Although new versions of mod_perl and Apache may not be released for months at a time and the need to upgrade them may not be pressing, the handlers and scripts being used at a site may need regular tweaks and changes, and new ones may be added quite frequently.

Of course, the safest and best option is to prepare an RPM (or equivalent) package that can be used to automatically upgrade the system, as explained in the previous section. Once an RPM specification file has been written (a task that might take some effort), future upgrades will be much less time consuming and have the advantage of being very easy to roll back.

But if the policy is to just overwrite files by hand, this section will explain how to do so as safely as possible.

All code should be thoroughly tested on a development machine before it is put on the live server, and both machines must have an identical software base (i.e., the same versions of the operating system, Apache, any software that Apache and mod_perl depend on, mod_perl itself, and all Perl modules). If the versions do not match, code that works perfectly on the development machine might not work on the live server.

For example, we have encountered a problem when the live and development servers were using different versions of the MySQL database server. The new code took advantage of new features added in the version installed on the development machine. The code was tested and shown to work correctly on the development machine, and when it was copied to the live server it seemed to work fine. Only by chance did we discover that scripts did not work correctly when the new features were used.

If the code hadn't worked at all, the problem would have been obvious and been detected and solved immediately, but the problem was subtle. Only after a thorough analysis did we understand that the problem was that we had an older version of the MySQL server on the live machine. This example reminded us that all modifications on the development machine should be logged and the live server updated with all of the modifications, not just the new version of the Perl code for a project.

We solved this particular problem by immediately reverting to the old code, upgrading the MySQL server on the live machine, and then successfully reapplying the new code.

Moving files and restarting the server. Now let's discuss the techniques used to upgrade live server scripts and handlers.

The most common scenario is a live running service that needs to be upgraded with a new version of the code. The new code has been prepared and uploaded to the production server, and the server has been restarted. Unfortunately, the service does not work anymore. What could be worse than that? There is no way back, because the original code has been overwritten with the new but non-working code.

Another scenario is where a whole set of files is being transferred to the live server but some network problem has occurred in the middle, which has slowed things down or totally aborted the transfer. With some of the files old and some new, the service is most likely broken. Since some files were overwritten, you can't roll back to the previously working version of the service.

No matter what file transfer technique is used, be it FTP, NFS, or anything else, live running code should never be directly overwritten during file transfer. Instead, files should be transferred to a temporary directory on the live machine, ready to be moved when necessary. If the transfer fails, it can then be restarted safely.

Both scenarios can be made safer with two approaches. First, do not overwrite working files. Second, use a revision control system such as CVS so that changes to working code can easily be undone if the working code is accidentally overwritten. Revision control will be covered later in this chapter.

We recommend performing all updates on the live server in the following sequence. Assume for this example that the project's code directory is */home/httpd/perl/rel*. When we're about to update the files, we create a new directory, */home/httpd/perl/test*, into which we copy the new files. Then we do some final sanity checks: check that file permissions are readable and executable for the user the server is running under, and run *perl -Tcw* on the new modules to make sure there are no syntax errors in them.

To save some typing, we set up some aliases for some of the *apachectl* commands and for *tail*ing the *error_log* file:

```
panic% alias graceful /home/httpd/httpd_perl/bin/apachectl graceful
panic% alias restart  /home/httpd/httpd_perl/bin/apachectl restart
panic% alias start    /home/httpd/httpd_perl/bin/apachectl start
panic% alias stop     /home/httpd/httpd_perl/bin/apachectl stop
panic% alias err      tail -f /home/httpd/httpd_perl/logs/error_log
```

Finally, when we think we are ready, we do:

```
panic% cd /home/httpd/perl
panic% mv rel old && mv test rel && stop && sleep 3 && restart && err
```

Note that all the commands are typed as a single line, joined by `&&`, and only at the end should the Enter key be pressed. The `&&` ensures that if any command fails, the following commands will not be executed.

The elements of this command line are:

`mv rel old &&`
: Backs up the working directory to *old*, so none of the original code is deleted or overwritten

`mv test rel &&`
: Puts the new code in place of the original

`stop &&`
: Stops the server

`sleep 3 &&`
: Allows the server a few seconds to shut down (it might need a longer sleep)

`restart &&`
: Restarts the server

`err`
: *tail*s the *error_log* file to make sure that everything is OK

If *mv* is overriden by a global alias *mv -i*, which requires confirming every action, you will need to call *mv -f* to override the *-i* option.

When updating code on a remote machine, it's a good idea to prepend *nohup* to the beginning of the command line:

```
panic% nohup mv rel old && mv test rel && stop && sleep 3 && restart && err
```

This approach ensures that if the connection is suddenly dropped, the server will not stay down if the last command that executes is *stop*.

apachectl generates its status messages a little too early. For example, when we execute *apachectl stop*, a message saying that the server has been stopped is displayed, when in fact the server is still running. Similarly, when we execute *apachectl start*, a message is displayed saying that the server has been started, while it is possible that it hasn't yet. In both cases, this happens because these status messages are not generated by Apache itself. Do not rely on them. Rely on the *error_log* file instead, where the running Apache server indicates its real status.

Also note that we use *restart* and not just *start*. This is because of Apache's potentially long stopping times if it has to run lots of destruction and cleanup code on exit. If *start* is used and Apache has not yet released the port it is listening to, the start will fail and the *error_log* will report that the port is in use. For example:

```
Address already in use: make_sock: could not bind to port 8000
```

However, if *restart* is used, *apachectl* will wait for the server to quit and unbind the port and will then cleanly restart it.

Now, what happens if the new modules are broken and the newly restarted server reports problems or refuses to start at all?

The aliased *err* command executes *tail -f* on the *error_log*, so that the failed restart or any other problems will be immediately apparent. The situation can quickly and easily be rectified by returning the system to its pre-upgrade state with this command:

```
panic% mv rel bad && mv old rel && stop && sleep 3 && restart && err
```

This command line moves the new code to the directory *bad*, moves the original code back into the runtime directory *rel*, then stops and restarts the server. Once the server is back up and running, you can analyze the cause of the problem, fix it, and repeat the upgrade again. Usually everything will be fine if the code has been extensively tested on the development server. When upgrades go smoothly, the downtime should be only about 5–10 seconds, and most users will not even notice anything has happened.

Using CVS for code upgrades. The *Concurrent Versions System* (CVS) is an open source version-control system that allows multiple developers to work on code or configuration in a central repository while tracking any changes made. We use it because it's the dominant open source tool, but it's not the only possibility: commercial tools such as Perforce would also work for these purposes.

If you aren't familiar with CVS, you can learn about it from the resources provided at the end of this chapter. CVS is too broad a topic to be covered in this book. Instead, we will concentrate on the CVS techniques that are relevant to our purpose.

Things are much simpler when using CVS for server updates, especially since it allows you to tag each production release. By *tagging* files, we mean having a group of files under CVS control share a common label. Like RCS and other revision-control systems, CVS gives each file its own version number, which allows us to manipulate different versions of this file. But if we want to operate on a group of many files, chances are that they will have different version numbers. Suppose we want to take snapshots of the whole project so we can refer to these snapshots some time in the future, after the files have been modified and their respective version numbers have been changed. We can do this using tags.

To tag the project whose module name is *myproject*, execute the following from any directory on any machine:

```
panic% cvs -rtag PRODUCTION_1_20 myproject
```

Now when the time comes to update the online version, go to the directory on the live machine that needs to be updated and execute:

```
panic% cvs update -dP -r PRODUCTION_1_20
```

The *-P* option to *cvs* prunes empty directories and deleted files, the *-d* option brings in new directories and files (like *cvs checkout* does), and *-r PRODUCTION_1_20* tells CVS to update the current directory recursively to the `PRODUCTION_1_20` CVS version of the project.

Suppose that after a while, we have more code updated and we need to make a new release. The currently running version has the tag `PRODUCTION_1_20`, and the new version has the tag `PRODUCTION_1_21`. First we tag the files in the current state with a new tag:

```
panic% cvs -rtag PRODUCTION_1_21 myproject
```

and update the live machine:

```
panic% cvs update -dP -r PRODUCTION_1_21
```

Now if there is a problem, we can go back to the previous working version very easily. If we want to get back to version `PRODUCTION_1_20`, we can run the command:

```
panic% cvs update -dP -r PRODUCTION_1_20
```

As before, the update brings in new files and directories not already present in the local directory (because of the *-dP* options).

Remember that when you use CVS to update the live server, you should avoid making any minor changes to the code on this server. That's because of potential collisions that might happen during the CVS update. If you modify a single line in a single file and then do *cvs update*, and someone else modifies the same line at the same time and commits it just before you do, CVS will try to merge the changes. If they are different, it will see a conflict and insert both versions into the file. CVS leaves it to you to resolve the conflict. If this file is Perl code, it won't compile and it will cause temporal troubles until the conflict is resolved. Therefore, the best approach is to think of live server files as being read-only.

Updating the live code directory should be done only if the update is atomic—i.e., if all files are updated in a very short period of time, and when no network problems can occur that might delay the completion of the file update.

The safest approach is to use CVS in conjunction with the safe code update technique presented previously, by working with CVS in a separate directory. When all files are extracted, move them to the directory the live server uses. Better yet, use symbolic links, as described earlier in this chapter: when you update the code, prepare everything in a new directory and, when you're ready, just change the symlink to point to this new directory. This approach will prevent cases where only a partial update happens because of a network or other problem.

The use of CVS needn't apply exclusively to code. It can be of great benefit for configuration management, too. Just as you want your mod_perl programs to be identical between the development and production servers, you probably also want to keep your *httpd.conf* files in sync. CVS is well suited for this task too, and the same methods apply.

Disabling Scripts and Handlers on a Live Server

Perl programs running on the mod_perl server may be dependent on resources that can become temporarily unavailable when they are being upgraded or maintained. For example, once in a while a database server (and possibly its corresponding DBD module) may need to be upgraded, rendering it unusable for a short period of time.

Using the development server as a temporary replacement is probably the best way to continue to provide service during the upgrade. But if you can't, the service will be unavailable for a while.

Since none of the code that relies on the temporarily unavailable resource will work, users trying to access the mod_perl server will see either the ugly gray "An Error has occurred" message or a customized error message (if code has been added to trap errors and customize the error-reporting facility). In any case, it's not a good idea to let users see these errors, as they will make your web site seem amateurish.

A friendlier approach is to confess to the users that some maintenance work is being undertaken and plead for patience, promising that the service will become fully functional in a few minutes (or however long the scheduled downtime is expected to be).

It is a good idea to be honest and report the real duration of the maintenance operation, not just "we will be back in 10 minutes." Think of a user (or journalist) coming back 20 minutes later and still seeing the same message! Make sure that if the time of resumption of service is given, it is not the system's local time, since users will be visiting the site from different time zones. Instead, we suggest using Greenwich Mean Time (GMT). Most users have some idea of the time difference between their location and GMT, or can find out easily enough. Although GMT is known by programmers as Universal Coordinated Time (UTC), end users may not know what UTC is, so using the older acronym is probably best.

Disabling code running under Apache::Registry

If just a few scripts need to be disabled temporarily, and if they are running under the `Apache::Registry` handler, a maintenance message can be displayed without messing with the server. Prepare a little script in */home/httpd/perl/down4maintenance.pl*:

```
#!/usr/bin/perl -Tw

use strict;
print "Content-type: text/plain\n\n",
  qq{We regret that the service is temporarily
     unavailable while essential maintenance is undertaken.
     It is expected to be back online from 12:20 GMT.
     Please, bear with us. Thank you!};
```

Let's say you now want to disable the */home/httpd/perl/chat.pl* script. Just do this:

```
panic% mv /home/httpd/perl/chat.pl /home/httpd/perl/chat.pl.orig
panic% ln -s /home/httpd/perl/down4maintenance.pl /home/httpd/perl/chat.pl
```

Of course, the server configuration must allow symbolic links for this trick to work. Make sure that the directive:

```
Options FollowSymLinks
```

is in the `<Location>` or `<Directory>` section of *httpd.conf*.

Alternatively, you can just back up the real script and then copy the file over it:

```
panic% cp /home/httpd/perl/chat.pl /home/httpd/perl/chat.pl.orig
panic% cp /home/httpd/perl/down4maintenance.pl /home/httpd/perl/chat.pl
```

Once the maintenance work has been completed, restoring the previous setup is easy. Simply overwrite the symbolic link or the file:

```
panic% mv /home/httpd/perl/chat.pl.orig /home/httpd/perl/chat.pl
```

Now make sure that the script has the current timestamp:

```
panic% touch /home/httpd/perl/chat.pl
```

`Apache::Registry` will automatically detect the change and use the new script from now on.

This scenario is possible because `Apache::Registry` checks the modification time of the script before each invocation. If the script's file is more recent than the version already loaded in memory, `Apache::Registry` reloads the script from disk.

Disabling code running under other handlers

Under non-`Apache::Registry` handlers, you need to modify the configuration. You must either point all requests to a new location or replace the handler with one that will serve the requests during the maintenance period.

Example 5-2 illustrates a maintenance handler.

Example 5-2. Book/Maintenance.pm

```
package Book::Maintenance;

use strict;
use Apache::Constants qw(:common);

sub handler {
  my $r = shift;
  $r->send_http_header("text/plain");
  print qq{We regret that the service is temporarily
           unavailable while essential maintenance is undertaken.
           It is expected to be back online from 12:20 GMT.
           Please be patient. Thank you!};
  return OK;
}
1;
```

In practice, the maintenance script may well read the "back online" time from a variable set with a `PerlSetVar` directive in *httpd.conf*, so the script itself need never be changed.

Edit *httpd.conf* and change the handler line from:

```
<Location /perl>
    SetHandler perl-script
```

```
        PerlHandler Book::Handler
        ...
    </Location>
```

to:

```
    <Location /perl>
        SetHandler perl-script
        #PerlHandler Book::Handler
        PerlHandler Book::Maintenance
        ...
    </Location>
```

Now restart the server. Users will be happy to read their email for 10 minutes, knowing that they will return to a much improved service.

Disabling services with help from the frontend server

Many sites use a more complicated setup in which a "light" Apache frontend server serves static content but proxies all requests for dynamic content to the "heavy" mod_perl backend server (see Chapter 12). Those sites can use a third solution to temporarily disable scripts.

Since the frontend machine rewrites all incoming requests to appropriate requests for the backend machine, a change to the `RewriteRule` is sufficient to take handlers out of service. Just change the directives to rewrite all incoming requests (or a subgroup of them) to a single URI. This URI simply tells users that the service is not available during the maintenance period.

For example, the following `RewriteRule` rewrites all URIs starting with */perl* to the maintenance URI */control/maintain* on the mod_perl server:

```
RewriteRule ^/perl/(.*)$ http://localhost:8000/control/maintain [P,L]
```

The `Book::Maintenance` handler from the previous section can be used to generate the response to the URI */control/maintain*.

Make sure that this rule is placed before all the other `RewriteRules` so that none of the other rules need to be commented out. Once the change has been made, check that the configuration is not broken and restart the server so that the new configuration takes effect. Now the database server can be shut down, the upgrade can be performed, and the database server can be restarted. The `RewriteRule` that has just been added can be commented out and the Apache server stopped and restarted. If the changes lead to any problems, the maintenance `RewriteRule` can simply be uncommented while you sort them out.

Of course, all this is error-prone, especially when the maintenance is urgent. Therefore, it can be a good idea to prepare all the required configurations ahead of time, by having different configuration sections and enabling the right one with the help of the `IfDefine` directive during server startup.

The following configuration will make this approach clear:

```
RewriteEngine On

<IfDefine maintain>
    RewriteRule /perl/  http://localhost:8000/control/maintain [P,L]
</IfDefine>

<IfDefine !maintain>
   RewriteRule ^/perl/(.*)$ http://localhost:8000/$1 [P,L]
   # more directives
</IfDefine>
```

Now enable the maintenance section by starting the server with:

```
panic% httpd -Dmaintain
```

Request URIs starting with */perl/* will be processed by the */control/maintain* handler or script on the mod_perl side.

If the *-Dmaintain* option is not passed, the "normal" configuration will take effect and each URI will be remapped to the mod_perl server as usual.

Of course, if *apachectl* or any other script is used for server control, this new mode should be added so that it will be easy to make the correct change without making any mistakes. When you're in a rush, the less typing you have to do, the better. Ideally, all you'd have to type is:

```
panic% apachectl maintain
```

Which will both shut down the server (if it is running) and start it with the *-Dmaintain* option. Alternatively, you could use:

```
panic% apachectl start_maintain
```

to start the server in maintenance mode. *apachectl graceful* will stop the server and restart it in normal mode.

Scheduled Routine Maintenance

If maintenance tasks can be scheduled when no one is using the server, you can write a simple `PerlAccessHandler` that will automatically disable the server and return a page stating that the server is under maintenance and will be back online at a specified time. When using this approach, you don't need to worry about fiddling with the server configuration when the maintenance hour comes. However, all maintenance must be completed within the given time frame, because once the time is up, the service will resume.

The `Apache::DayLimit` module from *http://www.modperl.com/* is a good example of such a module. It provides options for specifying which day server maintenance occurs. For example, if Sundays are used for maintenance, the configuration for `Apache::DayLimit` is as follows:

```
<Location /perl>
    PerlSetVar ReqDay Sunday
    PerlAccessHandler Apache::DayLimit
</Location>
```

It is very easy to adapt this module to do more advanced filtering. For example, to specify both a day and a time, use a configuration similar to this:

```
<Location /perl>
    PerlSetVar ReqDay    Sunday
    PerlSetVar StartHour 09:00
    PerlSetVar EndHour   11:00
    PerlAccessHandler Apache::DayTimeLimit
</Location>
```

Three-Tier Server Scheme: Development, Staging, and Production

To facilitate transfer from the development server to the production server, the code should be free of any server-dependent variables. This will ensure that modules and scripts can be moved from one directory on the development machine to another directory (possibly in a different path) on the production machine without problems.

If two simple rules are followed, server dependencies can be safely isolated and, as far as the code goes, effectively ignored. First, never use the server name (since development and production machines have different names), and second, never use explicit base directory names in the code. Of course, the code will often need to refer to the server name and directory names, but we can centralize them in server-wide configuration files (as seen in a moment).

By trial and error, we have found that a three-tier (development, staging, and production) scheme works best:

Development
: The development tier might include a single machine or several machines (for example, if there are many developers and each one prefers to develop on his own machine).

Staging
: The staging tier is generally a single machine that is basically identical to the production machine and serves as a backup machine in case the production machine needs an immediate replacement (for example, during maintenance). This is the last station where the code is staged before it is uploaded to the production machine.

  The staging machine does not have to be anywhere near as powerful as the production server if finances are stretched. The staging machine is generally used only for staging; it does not require much processor power or memory since only a few developers are likely to use it simultaneously. Even if several developers are using it at the same time, the load will be very low, unless of course benchmarks are being run on it along with programs that create a load similar to that on the production server (in which case the staging machine should have hardware identical to that of the production machine).

Production
The production tier might include a single machine or a huge cluster comprising many machines.

You can also have the staging and production servers running on the same machine. This is not ideal, especially if the production server needs every megabyte of memory and every CPU cycle so that it can cope with high request rates. But when a dedicated machine just for staging purposes is prohibitively expensive, using the production server for staging purposes is better than having no staging area at all.

Another possibility is to have the staging environment on the development machine.

So how does this three-tier scheme work?

- Developers write the code on their machines (development tier) and test that it works as expected. These machines should be set up with an environment as similar to the production server as possible. A manageable and simple approach is to have each developer running his own local Apache server on his own machine. If the code relies on a database, the ideal scenario is for each developer to have access to a development database account and server, possibly even on their own machines.
- The pre-release manager installs the code on the staging tier machine and stages it. Whereas developers can change their own *httpd.conf* files on their own machines, the pre-release manager will make the necessary changes on the staging machine in accordance with the instructions provided by the developers.
- The release manager installs the code on the production tier machine(s), tests it, and monitors for a while to ensure that things work as expected.

Of course, on some projects, the developers, the pre-release managers, and the release managers can actually be the same person. On larger projects, where different people perform these roles and many machines are involved, preparing upgrade packages with a packaging tool such as RPM becomes even more important, since it makes it far easier to keep every machine's configuration and software in sync.

Now that we have described the theory behind the three-tier approach, let us see how to have all the code independent of the machine and base directory names.

Although the example shown below is simple, the real configuration may be far more complex; however, the principles apply regardless of complexity, and it is straightforward to build a simple initial configuration into a configuration that is sufficient for more complex environments.

Basically, what we need is the name of the machine, the port on which the server is running (assuming that the port number is not hidden with the help of a proxy server), the root directory of the web server–specific files, the base directories of static objects and Perl scripts, the appropriate relative and full URIs for these base directories, and a support email address. This amounts to 10 variables.

We prepare a minimum of three `Local::Config` packages, one per tier, each suited to a particular tier's environment. As mentioned earlier, there can be more than one machine per tier and even more than one web server running on the same machine. In those cases, each web server will have its own `Local::Config` package. The total number of `Local::Config` packages will be equal to the number of web servers.

For example, for the development tier, the configuration package might look like Example 5-3.

Example 5-3. Local/Config.pm

```
package Local::Config;
use strict;
use constant SERVER_NAME      => 'dev.example.com';
use constant SERVER_PORT      => 8000;
use constant ROOT_DIR         => '/home/userfoo/www';
use constant CGI_BASE_DIR     => '/home/userfoo/www/perl';
use constant DOC_BASE_DIR     => '/home/userfoo/www/docs';
use constant CGI_BASE_URI     => 'http://dev.example.com:8000/perl';
use constant DOC_BASE_URI     => 'http://dev.example.com:8000';
use constant CGI_RELATIVE_URI => '/perl';
use constant DOC_RELATIVE_URI => '';
use constant SUPPORT_EMAIL    => 'stas@example.com';
1;
```

The constants have uppercase names, in accordance with Perl convention.

The configuration shows that the name of the development machine is *dev.example.com*, listening to port 8000. Web server–specific files reside under the */home/userfoo/www* directory. Think of this as a directory *www* that resides under user *userfoo*'s home directory, */home/userfoo*. A developer whose username is *userbar* might use */home/userbar/www* as the development root directory.

If there is another web server running on the same machine, create another `Local::Config` with a different port number and probably a different root directory.

To avoid duplication of identical parts of the configuration, the package can be rewritten as shown in Example 5-4.

Example 5-4. Local/Config.pm

```
package Local::Config;
use strict;
use constant DOMAIN_NAME      => 'example.com';
use constant SERVER_NAME      => 'dev.' . DOMAIN_NAME;
```

Example 5-4. Local/Config.pm (continued)

```
use constant SERVER_PORT      => 8000;
use constant ROOT_DIR         => '/home/userfoo/www';
use constant CGI_BASE_DIR     => ROOT_DIR . '/perl';
use constant DOC_BASE_DIR     => ROOT_DIR . '/docs';
use constant CGI_BASE_URI     => 'http://' . SERVER_NAME . ':' . SERVER_PORT
                                 . '/perl';
use constant DOC_BASE_URI     => 'http://' . SERVER_NAME . ':' . SERVER_PORT;
use constant CGI_RELATIVE_URI => '/perl';
use constant DOC_RELATIVE_URI => '';
use constant SUPPORT_EMAIL    => 'stas@' . DOMAIN_NAME;
1;
```

Reusing constants that were previously defined reduces the risk of making a mistake. In the original file, several lines need to be edited if the server name is changed, but in this new version only one line requires editing, eliminating the risk of your forgetting to change a line further down the file. All the `use constant` statements are executed at compile time, in the order in which they are specified. The `constant` pragma ensures that any attempt to change these variables in the code leads to an error, so they can be relied on to be correct. (Note that in certain contexts—e.g., when they're used as hash keys—Perl can misunderstand the use of constants. The solution is to either prepend `&` or append `()`, so `ROOT_DIR` would become either `&ROOT_DIR` or `ROOT_DIR()`.)

Now, when the code needs to access the server's global configuration, it needs to refer only to the variables in this module. For example, in an application's configuration file, you can create a dynamically generated configuration, which will change from machine to machine without your needing to touch any code (see Example 5-5).

Example 5-5. App/Foo/Config.pm

```
package App::Foo::Config;

use Local::Config ();
use strict;
use vars qw($CGI_URI $CGI_DIR);

# directories and URIs of the App::Foo CGI project
$CGI_URI = $Local::Config::CGI_BASE_URI . '/App/Foo';
$CGI_DIR = $Local::Config::CGI_BASE_DIR . '/App/Foo';
1;
```

Notice that we used fully qualified variable names instead of importing these global configuration variables into the caller's namespace. This saves a few bytes of memory, and since `Local::Config` will be loaded by many modules, these savings will quickly add up. Programmers used to programming Perl outside the mod_perl environment might be tempted to add Perl's exporting mechanism to `Local::Config` and thereby save themselves some typing. We prefer not to use `Exporter.pm` under mod_perl, because we want to save as much memory as possible. (Even though the amount of memory overhead for using an exported name is small, this must be multiplied by the number of concurrent users of the code, which could be hundreds or even thousands on a busy site and could turn a small memory overhead into a large one.)

For the staging tier, a similar `Local::Config` module with just a few changes (as shown in Example 5-6) is necessary.

Example 5-6. Local/Config.pm

```
package Local::Config;
use strict;
use constant DOMAIN_NAME      => 'example.com';
use constant SERVER_NAME      => 'stage.' . DOMAIN_NAME;
use constant SERVER_PORT      => 8000;
use constant ROOT_DIR         => '/home';
use constant CGI_BASE_DIR     => ROOT_DIR . '/perl';
use constant DOC_BASE_DIR     => ROOT_DIR . '/docs';
use constant CGI_BASE_URI     => 'http://' . SERVER_NAME . ':' . SERVER_PORT
                                 . '/perl';
use constant DOC_BASE_URI     => 'http://' . SERVER_NAME . ':' . SERVER_PORT;
use constant CGI_RELATIVE_URI => '/perl';
use constant DOC_RELATIVE_URI => '';
use constant SUPPORT_EMAIL    => 'stage@' . DOMAIN_NAME;
1;
```

We have named our staging tier machine *stage.example.com*. Its root directory is */home*.

The production tier version of `Local/Config.pm` is shown in Example 5-7.

Example 5-7. Local/Config.pm

```
package Local::Config;
use strict;
use constant DOMAIN_NAME      => 'example.com';
use constant SERVER_NAME      => 'www.' . DOMAIN_NAME;
use constant SERVER_PORT      => 8000;
use constant ROOT_DIR         => '/home/';
use constant CGI_BASE_DIR     => ROOT_DIR . '/perl';
use constant DOC_BASE_DIR     => ROOT_DIR . '/docs';
use constant CGI_BASE_URI     => 'http://' . SERVER_NAME . ':' . SERVER_PORT
                                 . '/perl';
use constant DOC_BASE_URI     => 'http://' . SERVER_NAME . ':' . SERVER_PORT;
use constant CGI_RELATIVE_URI => '/perl';
use constant DOC_RELATIVE_URI => '';
use constant SUPPORT_EMAIL    => 'support@' . DOMAIN_NAME;
```

You can see that the setups of the staging and production machines are almost identical. This is only in our example; in reality, they can be very different.

The most important point is that the `Local::Config` module from a machine on one tier must *never* be moved to a machine on another tier, since it will break the code. If locally built packages are used, the `Local::Config` file can simply be excluded—this will help to reduce the risk of inadvertently copying it.

From now on, when modules and scripts are moved between machines, you shouldn't need to worry about having to change variables to accomodate the different machines' server names and directory layouts. All this is accounted for by the `Local::Config` files.

Some developers prefer to run conversion scripts on the moved code that adjust all variables to the local machine. This approach is error-prone, since variables can be written in different ways, and it may result in incomplete adjustment and broken code. Therefore, the conversion approach is not recommended.

Starting a Personal Server for Each Developer

When just one developer is working on a specific server, there are fewer problems, because she can have complete control over the server. However, often a group of developers need to develop mod_perl scripts and modules concurrently on the same machine. Each developer wants to have control over the server: to be able to stop it, run it in single-server mode, restart it, etc. They also want control over the location of log files, configuration settings such as `MaxClients`, and so on.

Each developer might have her own desktop machine, but all development and staging might be done on a single central development machine (e.g., if the developers' personal desktop machines run a different operating system from the one running on the development and production machines).

One workaround for this problem involves having a few versions of the *httpd.conf* file (each having different `Port`, `ErrorLog`, etc. directives) and forcing each developer's server to be started with:

```
panic% httpd_perl -f /path/to/httpd.conf
```

However, this means that these files must be kept synchronized when there are global changes affecting all developers. This can be quite difficult to manage. The solution we use is to have a single *httpd.conf* file and use the *-Dparameter* server startup option to enable a specific section of *httpd.conf* for each developer. Each developer starts the server with his or her username as an argument. As a result, a server uses both the global settings and the developer's private settings.

For example, user *stas* would start his server with:

```
panic% httpd_perl -Dstas
```

In *httpd.conf*, we write:

```
# Personal development server for stas
# stas uses the server running on port 8000
<IfDefine stas>
    Port 8000
```

```
        PidFile /home/httpd/httpd_perl/logs/httpd.pid.stas
        ErrorLog /home/httpd/httpd_perl/logs/error_log.stas
        Timeout 300
        KeepAlive On
        MinSpareServers 2
        MaxSpareServers 2
        StartServers 1
        MaxClients 3
        MaxRequestsPerChild 15
        # let developers to add their own configuration
        # so they can override the defaults
        Include /home/httpd/httpd_perl/conf/stas.conf
    </IfDefine>

    # Personal development server for eric
    # eric uses the server running on port 8001
    <IfDefine eric>
        Port 8001
        PidFile /home/httpd/httpd_perl/logs/httpd.pid.eric
        ErrorLog /home/httpd/httpd_perl/logs/error_log.eric
        Timeout 300
        KeepAlive Off
        MinSpareServers 1
        MaxSpareServers 2
        StartServers 1
        MaxClients 5
        MaxRequestsPerChild 0
        Include /home/httpd/httpd_perl/conf/eric.conf
    </IfDefine>
```

With this technique, we have separate *error_log* files and full control over server starting and stopping, the number of child processes, and port selection for each server. This saves Eric from having to call Stas several times a day just to warn, "Stas, I'm restarting the server" (a ritual in development shops where all developers are using the same mod_perl server).

With this technique, developers will need to learn the PIDs of their parent *httpd_perl* processes. For user *stas*, this can be found in */home/httpd/httpd_perl/logs/httpd.pid.stas*. To make things even easier, we change the *apachectl* script to do the work for us. We make a copy for each developer, called *apachectl.username*, and change two lines in each script:

```
PIDFILE=/home/httpd/httpd_perl/logs/httpd.pid.username
HTTPD='/home/httpd/httpd_perl/bin/httpd_perl -Dusername'
```

For user *stas*, we prepare a startup script called *apachectl.stas* and change these two lines in the standard *apachectl* script:

```
PIDFILE=/home/httpd/httpd_perl/logs/httpd.pid.stas
HTTPD='/home/httpd/httpd_perl/bin/httpd_perl -Dstas'
```

Now when user *stas* wants to stop the server, he executes:

```
panic% apachectl.stas stop
```

And to start the server, he executes:

```
panic% apachectl.stas start
```

And so on, for all other *apachectl* arguments.

It might seem that we could have used just one *apachectl* and have it determine for itself who executed it by checking the UID. But the setuid bit must be enabled on this script, because starting the server requires *root* privileges. With the setuid bit set, a single *apachectl* script can be used for all developers, but it will have to be modified to include code to read the UID of the user executing it and to use this value when setting developer-specific paths and variables.

The last thing you need to do is to provide developers with an option to run in single-process mode. For example:

```
panic% /home/httpd/httpd_perl/bin/httpd_perl -Dstas -X
```

In addition to making the development process easier, we decided to use relative links in all static documents, including calls to dynamically generated documents. Since each developer's server is running on a different port, we have to make it possible for these relative links to reach the correct port number.

When typing the URI by hand, it's easy. For example, when user *stas*, whose server is running on port 8000, wants to access the relative URI */test/example*, he types *http://www.example.com:8000/test/example* to get the generated HTML page. Now if this document includes a link to the relative URI */test/example2* and *stas* clicks on it, the browser will automatically generate a full request (*http://www.example.com:8000/test/example2*) by reusing the *server:port* combination from the previous request.

Note that all static objects will be served from the same server as well. This may be an acceptable situation for the development environment, but if it is not, a slightly more complicated solution involving the mod_rewrite Apache module will have to be devised.

To use mod_rewrite, we have to configure our *httpd_docs* (light) server with *--enable-module=rewrite* and recompile, or use DSOs and load and enable the module in *httpd.conf*. In the *httpd.conf* file of our *httpd_docs* server, we have the following code:

```
RewriteEngine on

# stas's server
# port = 8000
RewriteCond  %{REQUEST_URI} ^/perl
RewriteCond  %{REMOTE_ADDR} 123.34.45.56
RewriteRule ^(.*)           http://example.com:8000/$1 [P,L]

# eric's server
# port = 8001
RewriteCond  %{REQUEST_URI} ^/perl
RewriteCond  %{REMOTE_ADDR} 123.34.45.57
RewriteRule ^(.*)           http://example.com:8001/$1 [P,L]
```

```
# all the rest
RewriteCond  %{REQUEST_URI} ^/perl
RewriteRule ^(.*)          http://example.com:81/$1 [P]
```

The IP addresses are those of the developer desktop machines (i.e., where they run their web browsers). If an HTML file includes a relative URI such as */perl/test.pl* or even *http://www.example.com/perl/test.pl*, requests for those URIs from user *stas*'s machine will be internally proxied to *http://www.example.com:8000/perl/test.pl*, and requests generated from user *eric*'s machine will be proxied to *http://www.example.com:8001/perl/test.pl*.

Another possibility is to use the `REMOTE_USER` variable. This requires that all developers be authenticated when they access the server. To do so, change the `RewriteRules` to match `REMOTE_USER` in the above example.

Remember, the above setup will work only with relative URIs in the HTML code. If the HTML output by the code uses full URIs including a port other than 80, the requests originating from this HTML code will bypass the light server listening on the default port 80 and go directly to the server and port of the full URI.

Web Server Monitoring

Once the production system is working, you may think that the job is done and the developers can switch to a new project. Unfortunately, in most cases the server will still need to be maintained to make sure that everything is working as expected, to ensure that the web server is always up, and much more. A large part of this job can be automated, which will save time. It will also increase the uptime of the server, since automated processes generally work faster than manual ones. If created properly, automated processes also will always work correctly, whereas human operators are likely to make occassional mistakes.

Interactive Monitoring

When you're getting started, it usually helps to monitor the server interactively. Many different tools are available to do this. We will discuss a few of them now.

When writing automated monitoring tools, you should start by monitoring the tools themselves until they are reliable and stable enough to be left to work by themselves.

Even when everything is automated, you should check at regular intervals that everything is working OK, since a minor change in a single component can silently break the whole monitoring system. A good example is a silent failure of the mail system—if all alerts from the monitoring tools are delivered through email, having no messages from the system does not necessarily mean that everything is OK. If emails alerting about a problem cannot reach the webmaster because of a broken email system, the webmaster will not realize that a problem exists. (Of course, the mailing system should be monitored as well, but then problems must be reported by means other than email. One common solution is to send messages by both email and to a mobile phone's short message service.)

Another very important (albeit often-forgotten) risk time is the post-upgrade period. Even after a minor upgrade, the whole service should be monitored closely for a while.

The first and simplest check is to visit a few pages from the service to make sure that things are working. Of course, this might not suffice, since different pages might use different resources—while code that does not use the database system might work properly, code that does use it might not work if the database server is down.

The second thing to check is the web server's *error_log* file. If there are any problems, they will probably be reported here. However, only obvious syntactic or malfunction bugs will appear here—the subtle bugs that are a result of bad program logic will be revealed only through careful testing (which should have been completed before upgrading the live server).

Periodic system health checking can be done using the *top* utility, which shows free memory and swap space, the machine's CPU load, etc.

Apache::VMonitor—The Visual System and Apache Server Monitor

The `Apache::VMonitor` module provides even better monitoring functionality than *top*. It supplies all the relevant information that *top* does, plus all the Apache-specific information provided by Apache's mod_status module (request processing time, last request's URI, number of requests served by each child, etc.) In addition, `Apache::VMonitor` emulates the reporting functions of the *top*, *mount*, and *df* utilities.

`Apache::VMonitor` has a special mode for mod_perl processes. It also has visual alerting capabilities and a configurable "automatic refresh" mode. A web interface can be used to show or hide all sections dynamically.

The module provides two main viewing modes:

- Multi-processes and overall system status
- Single-process extensive reporting

Prerequisites and configuration

To run `Apache::VMonitor`, you need to have `Apache::Scoreboard` installed and configured in *httpd.conf*. `Apache::Scoreboard`, in turn, requires mod_status to be installed with `ExtendedStatus` enabled. In *httpd.conf*, add:

```
ExtendedStatus On
```

Turning on extended mode will add a certain overhead to each request's response time. If every millisecond counts, you may not want to use it in production.

,ch05.22279 Page 188 Thursday, November 18, 2004 12:36 PM

You also need `Time::HiRes` and `GTop` to be installed. And, of course, you need a running mod_perl-enabled Apache server.

To enable `Apache::VMonitor`, add the following configuration to *httpd.conf*:

```
<Location /system/vmonitor>
    SetHandler perl-script
    PerlHandler Apache::VMonitor
</Location>
```

The monitor will be displayed when you request *http://localhost/system/vmonitor/*.

You probably want to protect this location from unwanted visitors. If you are accessing this location from the same IP address, you can use a simple host-based authentication:

```
<Location /system/vmonitor>
    SetHandler perl-script
    PerlHandler Apache::VMonitor
    order deny,allow
    deny  from all
    allow from 132.123.123.3
</Location>
```

Alternatively, you may use Basic or other authentication schemes provided by Apache and its extensions.

You should load the module in *httpd.conf*:

```
PerlModule Apache::VMonitor
```

or from the the startup file:

```
use Apache::VMonitor();
```

You can control the behavior of `Apache::VMonitor` by configuring variables in the startup file or inside the `<Perl>` section. To alter the monitor reporting behavior, tweak the following configuration arguments from within the startup file:

```
$Apache::VMonitor::Config{BLINKING} = 1;
$Apache::VMonitor::Config{REFRESH}  = 0;
$Apache::VMonitor::Config{VERBOSE}  = 0;
```

To control what sections are to be displayed when the tool is first accessed, configure the following variables:

```
$Apache::VMonitor::Config{SYSTEM}   = 1;
$Apache::VMonitor::Config{APACHE}   = 1;
$Apache::VMonitor::Config{PROCS}    = 1;
$Apache::VMonitor::Config{MOUNT}    = 1;
$Apache::VMonitor::Config{FS_USAGE} = 1;
```

You can control the sorting of the mod_perl processes report by sorting them by one of the following columns: `pid`, `mode`, `elapsed`, `lastreq`, `served`, `size`, `share`, `vsize`, `rss`, `client`, or `request`. For example, to sort by the process size, use the following setting:

```
$Apache::VMonitor::Config{SORT_BY}  = "size";
```

As the application provides an option to monitor processes other than mod_perl processes, you can define a regular expression to match the relevant processes. For example, to match the process names that include "httpd_docs", "mysql", and "squid", the following regular expression could be used:

```
$Apache::VMonitor::PROC_REGEX = 'httpd_docs|mysql|squid';
```

We will discuss all these configuration options and their influence on the application shortly.

Multi-processes and system overall status reporting mode

The first mode is the one that's used most often, since it allows you to monitor almost all important system resources from one location. For your convenience, you can turn different sections on and off on the report, to make it possible for reports to fit into one screen.

This mode comes with the following features:

Automatic Refresh Mode

You can tell the application to refresh the report every few seconds. You can preset this value at server startup. For example, to set the refresh to 60 seconds, add the following configuration setting:

```
$Apache::VMonitor::Config{REFRESH} = 60;
```

A `0` (zero) value turns off automatic refresh.

When the server is started, you can always adjust the refresh rate through the user interface.

top Emulation: System Health Report

Like *top*, this shows the current date/time, machine uptime, average load, and all the system CPU and memory usage levels (CPU load, real memory, and swap partition usage).

The *top* section includes a swap space usage visual alert capability. As we will explain later in this chapter, swapping is very undesirable on production systems. This tool helps to detect abnormal swapping situations by changing the swap report row's color according to the following rules:

```
swap usage                  report color
---------------------------------------------------------
5Mb < swap < 10 MB          light red
20% < swap (swapping is bad!)  red
70% < swap (almost all used!)  red + blinking (if enabled)
```

Note that you can turn on the blinking mode with:

```
$Apache::VMonitor::Config{BLINKING} = 1;
```

The module doesn't alert when swap is being used just a little (< 5 Mb), since swapping is common on many Unix systems, even when there is plenty of free RAM.

If you don't want the system section to be displayed, set:

```
$Apache::VMonitor::Config{SYSTEM} = 0;
```

The default is to display this section.

top Emulation: Apache/mod_perl Processes Status

Like *top*, this emulation gives a report of the processes, but it shows only information relevant to mod_perl processes. The report includes the status of the process (Starting, Reading, Sending, Waiting, etc.), process ID, time since the current request was started, last request processing time, size, and shared, virtual, and resident size. It shows the last client's IP address and the first 64 characters of the request URI.

This report can be sorted by any column by clicking on the name of the column while running the application. The sorting can also be preset with the following setting:

```
$Apache::VMonitor::Config{SORT_BY}  = "size";
```

The valid choices are `pid`, `mode`, `elapsed`, `lastreq`, `served`, `size`, `share`, `vsize`, `rss`, `client`, and `request`.

The section is concluded with a report about the total memory being used by all mod_perl processes as reported by the kernel, plus an extra number approximating the real memory usage when memory sharing is taking place. We discuss this in more detail in Chapter 10.

If you don't want the mod_perl processes section to be displayed, set:

```
$Apache::VMonitor::Config{APACHE} = 0;
```

The default is to display this section.

top Emulation: Any Processes

This section, just like the mod_perl processes section, displays the information as the *top* program would. To enable this section, set:

```
$Apache::VMonitor::Config{PROCS} = 1;
```

The default is not to display this section.

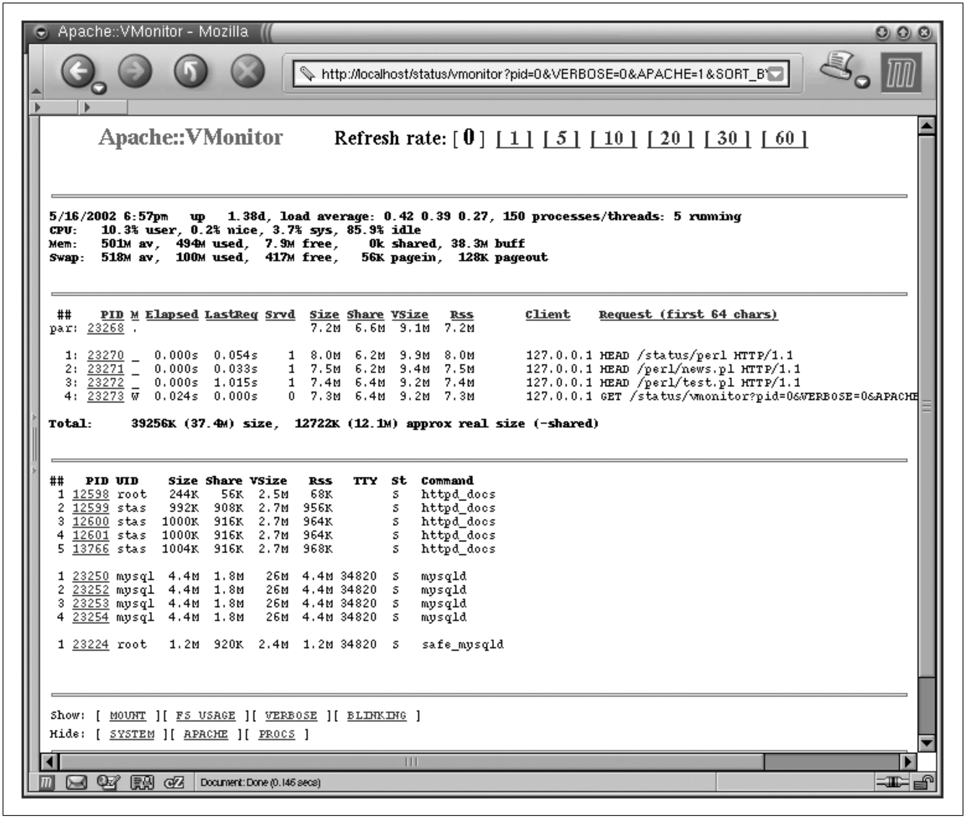
You need to specify which processes are to be monitored. The regular expression that will match the desired processes is required for this section to work. For example, if you want to see all the processes whose names include any of the strings "http", "mysql", or "squid", use the following regular expression:

```
$Apache::VMonitor::PROC_REGEX = 'httpd|mysql|squid';
```

Figure 5-1 visualizes the sections that have been discussed so far. As you can see, the swap memory is heavily used. Although you can't see it here, the swap memory report is colored red.

mount Emulation

This section provides information about mounted filesystems, as if you had called *mount* with no parameters.


Apache::VMonitor - Mozilla
http://localhost/status/vmonitor?pid=0&VERBOSE=0&APACHE=1&SORT_B
Apache::VMonitor
Refresh rate: [0] [1] [5] [10] [20] [30] [60]
Document: Done (0.146 secs)

```
5/16/2002 6:57pm   up   1.38d, load average: 0.42 0.39 0.27, 150 processes/threads: 5 running
CPU:   10.3% user, 0.2% nice, 3.7% sys, 85.9% idle
Mem:   501M av,  494M used,  7.9M free,    0k shared, 38.3M buff
Swap:  518M av,  100M used,  417M free,   56K pagein,  128K pageout

 ##    PID M Elapsed LastReq Srvd  Size Share VSize   Rss       Client    Request (first 64 chars)
par: 23268 .                       7.2M  6.6M  9.1M  7.2M

  1: 23270 _   0.000s  0.054s    1  8.0M  6.2M  9.9M  8.0M       127.0.0.1 HEAD /status/perl HTTP/1.1
  2: 23271 _   0.000s  0.033s    1  7.5M  6.2M  9.4M  7.5M       127.0.0.1 HEAD /perl/news.pl HTTP/1.1
  3: 23272 _   0.000s  1.015s    1  7.4M  6.4M  9.2M  7.4M       127.0.0.1 HEAD /perl/test.pl HTTP/1.1
  4: 23273 W   0.024s  0.000s    0  7.3M  6.4M  9.2M  7.3M       127.0.0.1 GET /status/vmonitor?pid=0&VERBOSE=0&APACHE

Total:     39256K (37.4M) size,  12722K (12.1M) approx real size (-shared)

##   PID UID     Size Share VSize   Rss   TTY  St  Command
 1 12598 root    244K   56K  2.5M   68K        S   httpd_docs
 2 12599 stas    992K  908K  2.7M  956K        S   httpd_docs
 3 12600 stas   1000K  916K  2.7M  964K        S   httpd_docs
 4 12601 stas   1000K  916K  2.7M  964K        S   httpd_docs
 5 13766 stas   1004K  916K  2.7M  968K        S   httpd_docs

 1 23250 mysql  4.4M  1.8M   26M  4.4M 34820  S   mysqld
 2 23252 mysql  4.4M  1.8M   26M  4.4M 34820  S   mysqld
 3 23253 mysql  4.4M  1.8M   26M  4.4M 34820  S   mysqld
 4 23254 mysql  4.4M  1.8M   26M  4.4M 34820  S   mysqld

 1 23224 root    1.2M  920K  2.4M  1.2M 34820  S   safe_mysqld

Show: [ MOUNT ][ FS USAGE ][ VERBOSE ][ BLINKING ]
Hide: [ SYSTEM ][ APACHE ][ PROCS ]
```

Figure 5-1. Emulation of top, centralized information about mod_perl and selected processes

If you want the *mount* section to be displayed, set:

```
$Apache::VMonitor::Config{MOUNT} = 1;
```

The default is not to display this section.

df Emulation

This section completely reproduces the *df* utility. For each mounted filesystem, it reports the number of total and available blocks for both superuser and user, and usage in percentages. In addition, it reports available and used file inodes in numbers and percentages.

This section can give you a visual alert when a filesystem becomes more than 90% full or when there are less than 10% of free file inodes left. The relevant filesystem row will be displayed in red and in a bold font. A mount point directory will blink if blinking is turned on. You can turn the blinking on with:

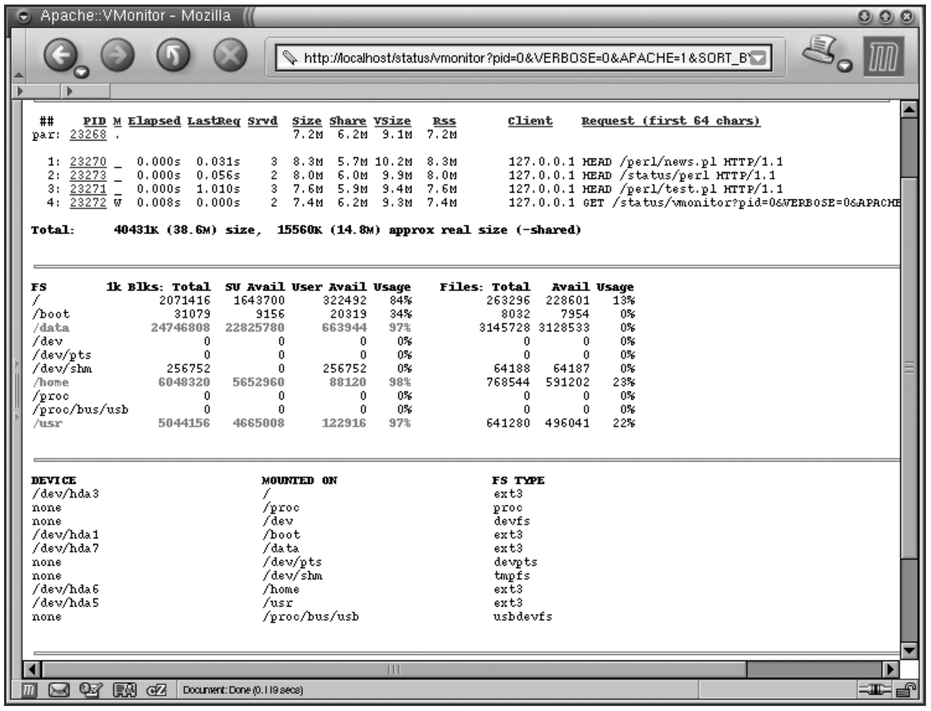
```
$Apache::VMonitor::Config{BLINKING} = 1;
```

If you don't want the *df* section to be displayed, set:

```
$Apache::VMonitor::Config{FS_USAGE} = 0;
```

The default is to display this section.

Figure 5-2 presents an example of the report consisting of the last two sections that were discussed (*df* and *mount* emulation), plus the ever-important mod_perl processes report.


Apache::VMonitor - Mozilla
http://localhost/status/vmonitor?pid=0&VERBOSE=0&APACHE=1&SORT_BY
Document: Done (0.119 secs)

```
 ##    PID M Elapsed LastReq Srvd  Size Share VSize   Rss       Client     Request (first 64 chars)
par: 23268 .                       7.2M  6.2M  9.1M  7.2M

  1: 23270 _  0.000s  0.031s    3  8.3M  5.7M 10.2M  8.3M       127.0.0.1 HEAD /perl/news.pl HTTP/1.1
  2: 23273 _  0.000s  0.056s    2  8.0M  6.0M  9.9M  8.0M       127.0.0.1 HEAD /status/perl HTTP/1.1
  3: 23271 _  0.000s  1.010s    3  7.6M  5.9M  9.4M  7.6M       127.0.0.1 HEAD /perl/test.pl HTTP/1.1
  4: 23272 W  0.008s  0.000s    2  7.4M  6.2M  9.3M  7.4M       127.0.0.1 GET /status/vmonitor?pid=0&VERBOSE=0&APACHE

Total:     40431K (38.6M) size,  15560K (14.8M) approx real size (-shared)


FS          1k Blks: Total  SU Avail User Avail Usage    Files: Total   Avail Usage
/                  2071416   1643700     322492   84%          263296  228601   13%
/boot                31079      9156      20319   34%            8032    7954    0%
/data             24746808  22825780     663944   97%         3145728 3128533    0%
/dev                     0         0          0    0%               0       0    0%
/dev/pts                 0         0          0    0%               0       0    0%
/dev/shm            256752         0     256752    0%           64188   64187    0%
/home              6048320   5652960      88120   98%          768544  591202   23%
/proc                    0         0          0    0%               0       0    0%
/proc/bus/usb            0         0          0    0%               0       0    0%
/usr               5044156   4665008     122916   97%          641280  496041   22%


DEVICE                        MOUNTED ON                    FS TYPE
/dev/hda3                     /                             ext3
none                          /proc                         proc
none                          /dev                          devfs
/dev/hda1                     /boot                         ext3
/dev/hda7                     /data                         ext3
none                          /dev/pts                      devpts
none                          /dev/shm                      tmpfs
/dev/hda6                     /home                         ext3
/dev/hda5                     /usr                          ext3
none                          /proc/bus/usb                 usbdevfs
```

Figure 5-2. Emulation of df, both inodes and blocks

In Figure 5-2, the */mnt/cdrom* and */usr* filesystems are more than 90% full and therefore are colored red. This is normal for */mnt/cdrom*, which is a mounted CD-ROM, but might be critical for the */usr* filesystem, which should be cleaned up or enlarged.

Abbreviations and hints

The report uses many abbreviations that might be new for you. If you enable the `VERBOSE` mode with:

```
$Apache::VMonitor::Config{VERBOSE} = 1;
```

this section will reveal the full names of the abbreviations at the bottom of the report.

The default is not to display this section.

Single-process extensive reporting system

If you need to get in-depth information about a single process, just click on its PID. If the chosen process is a mod_perl process, the following information is displayed:

- Process type (child or parent), status of the process (Starting, Reading, Sending, Waiting, etc.), and how long the current request has been being processed (or how long the previous request was processed for, if the process is inactive at the moment the report was made).
- How many bytes have been transferred so far, and how many requests have been served per child and per slot. (When the child process quits, it is replaced by a new process running in the same slot.)
- CPU times used by the process: `total`, `utime`, `stime`, `cutime`, `cstime`.

For all processes (mod_perl and non-mod_perl), the following information is reported:

- General process information: UID, GID, state, TTY, and command-line arguments
- Memory usage: size, share, VSize, and RSS
- Memory segments usage: text, shared lib, date, and stack
- Memory maps: start-end, offset, device_major:device_minor, inode, perm, and library path
- Sizes of loaded libraries

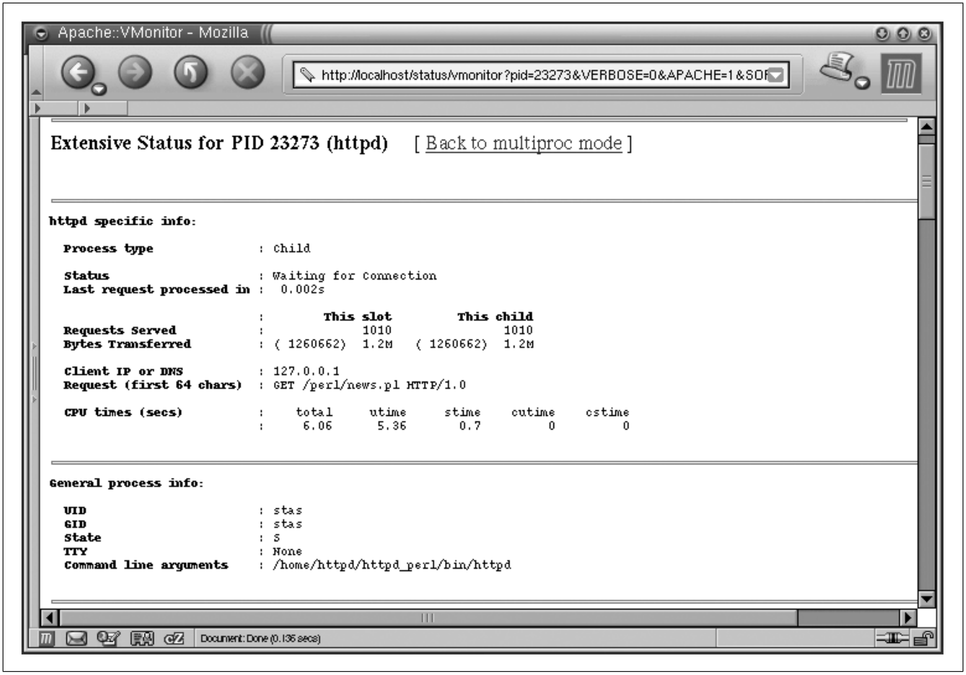
Just as with the multi-process mode, this mode allows you to automatically refresh the page at the desired intervals.

Figures 5-3, 5-4, and 5-5 show an example report for one mod_perl process.

Automated Monitoring

As we mentioned earlier, the more things are automated, the more stable the server will be. In general, there are three things that we want to ensure:

1. Apache is up and properly serving requests. Remember that it can be running but unable to serve requests (for example, if there is a stale lock and all processes are waiting to acquire it).
2. All the resources that mod_perl relies on are available and working. This might include database engines, SMTP services, NIS or LDAP services, etc.
3. The system is healthy. Make sure that there is no system resource contention, such as a small amount of free RAM, a heavily swapping system, or low disk space.



Apache::VMonitor - Mozilla

http://localhost/status/vmonitor?pid=23273&VERBOSE=0&APACHE=1&SOF

Extensive Status for PID 23273 (httpd) [Back to multiproc mode]

```
httpd specific info:

  Process type              : Child

  Status                    : Waiting for Connection
  Last request processed in :  0.002s

                            :         This slot         This child
  Requests Served           :              1010               1010
  Bytes Transferred         : ( 1260662)  1.2M   ( 1260662)  1.2M

  Client IP or DNS          : 127.0.0.1
  Request (first 64 chars)  : GET /perl/news.pl HTTP/1.0

  CPU times (secs)          :    total     utime     stime    cutime    cstime
                            :     6.06      5.36       0.7         0         0

General process info:

  UID                       : stas
  GID                       : stas
  State                     : S
  TTY                       : None
  Command line arguments    : /home/httpd/httpd_perl/bin/httpd
```

Document: Done (0.136 secs)

Figure 5-3. Extended information about processes: general process information

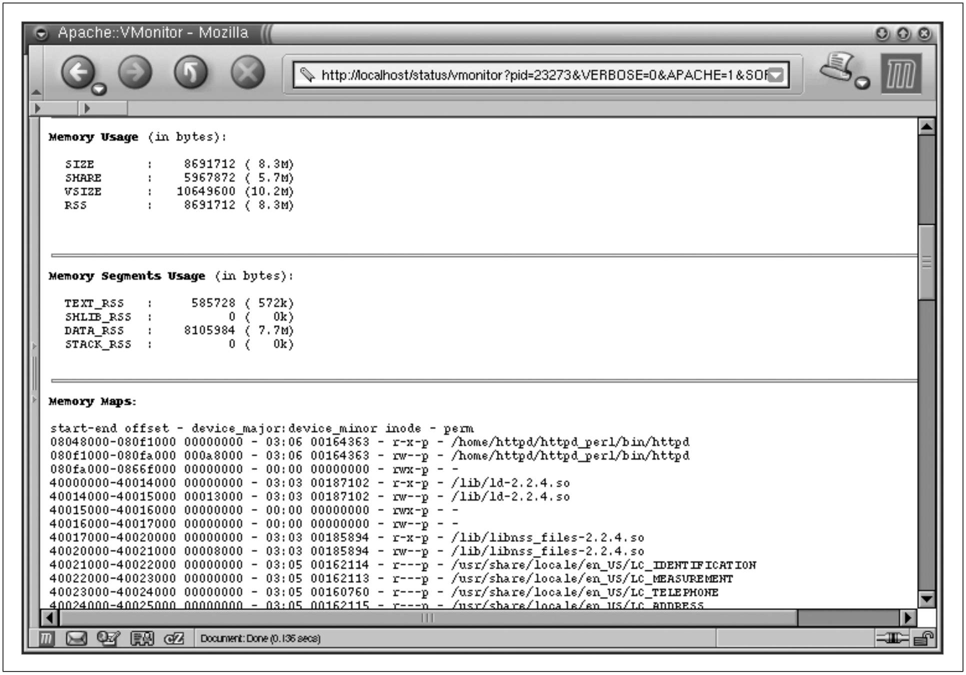
None of these categories has a higher priority than the others. A system administrator's role includes the proper functioning of the whole system. Even if the administrator is responsible for just part of the system, she must still ensure that her part does not cause problems for the system as a whole. If any of the above categories is not monitored, the system is not safe.

A specific setup might certainly have additional concerns that are not covered here, but it is most likely that they will fall into one of the above categories.

Before we delve into details, we should mention that all automated tools can be divided into two categories: tools that know how to detect problems and notify the owner, and tools that not only detect problems but also try to solve them, notifying the owner about both the problems and the results of the attempt to solve them.

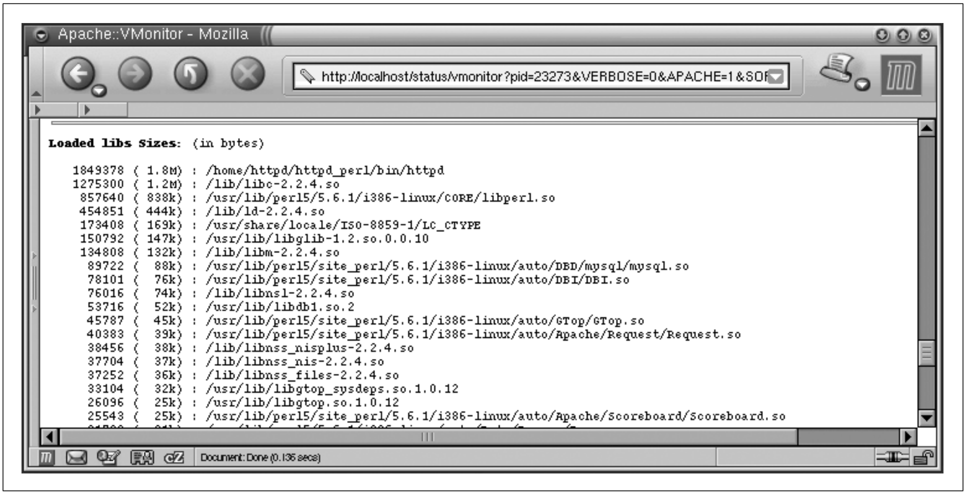
Automatic tools are generally called *watchdogs*. They can alert the owner when there is a problem, just as a watchdog will bark when something is wrong. They will also try to solve problems themselves when the owner is not around, just as watchdogs will bite thieves when their owners are asleep.

Although some tools can perform corrective actions when something goes wrong without human intervention (e.g., during the night or on weekends), for some problems it may be that only human intervention can resolve the situation. In such cases,


Apache::VMonitor - Mozilla
http://localhost/status/vmonitor?pid=23273&VERBOSE=0&APACHE=1&SOF
Document: Done (0.136 secs)

```
Memory Usage (in bytes):

  SIZE       :    8691712 ( 8.3M)
  SHARE      :    5967872 ( 5.7M)
  VSIZE      :   10649600 (10.2M)
  RSS        :    8691712 ( 8.3M)


Memory Segments Usage (in bytes):

  TEXT_RSS   :     585728 ( 572k)
  SHLIB_RSS  :          0 (   0k)
  DATA_RSS   :    8105984 ( 7.7M)
  STACK_RSS  :          0 (   0k)


Memory Maps:

start-end offset - device_major:device_minor inode - perm
08048000-080f1000 00000000 - 03:06 00164363 - r-x-p - /home/httpd/httpd_perl/bin/httpd
080f1000-080fa000 000a8000 - 03:06 00164363 - rw--p - /home/httpd/httpd_perl/bin/httpd
080fa000-0866f000 00000000 - 00:00 00000000 - rwx-p - -
40000000-40014000 00000000 - 03:03 00187102 - r-x-p - /lib/ld-2.2.4.so
40014000-40015000 00013000 - 03:03 00187102 - rw--p - /lib/ld-2.2.4.so
40015000-40016000 00000000 - 00:00 00000000 - rwx-p - -
40016000-40017000 00000000 - 00:00 00000000 - rw--p - -
40017000-40020000 00000000 - 03:03 00185894 - r-x-p - /lib/libnss_files-2.2.4.so
40020000-40021000 00008000 - 03:03 00185894 - rw--p - /lib/libnss_files-2.2.4.so
40021000-40022000 00000000 - 03:05 00162114 - r---p - /usr/share/locale/en_US/LC_IDENTIFICATION
40022000-40023000 00000000 - 03:05 00162113 - r---p - /usr/share/locale/en_US/LC_MEASUREMENT
40023000-40024000 00000000 - 03:05 00160760 - r---p - /usr/share/locale/en_US/LC_TELEPHONE
40024000-40025000 00000000 - 03:05 00162115 - r---p - /usr/share/locale/en_US/LC_ADDRESS
```

Figure 5-4. Extended information about processes: memory usage and maps


Apache::VMonitor - Mozilla
http://localhost/status/vmonitor?pid=23273&VERBOSE=0&APACHE=1&SOF
Document: Done (0.136 secs)

```
Loaded libs Sizes: (in bytes)

   1849378 ( 1.8M) : /home/httpd/httpd_perl/bin/httpd
   1275300 ( 1.2M) : /lib/libc-2.2.4.so
    857640 ( 838k) : /usr/lib/perl5/5.6.1/i386-linux/CORE/libperl.so
    454851 ( 444k) : /lib/ld-2.2.4.so
    173408 ( 169k) : /usr/share/locale/ISO-8859-1/LC_CTYPE
    150792 ( 147k) : /usr/lib/libglib-1.2.so.0.0.10
    134808 ( 132k) : /lib/libm-2.2.4.so
     89722 (  88k) : /usr/lib/perl5/site_perl/5.6.1/i386-linux/auto/DBD/mysql/mysql.so
     78101 (  76k) : /usr/lib/perl5/site_perl/5.6.1/i386-linux/auto/DBI/DBI.so
     76016 (  74k) : /lib/libnsl-2.2.4.so
     53716 (  52k) : /usr/lib/libdb1.so.2
     45787 (  45k) : /usr/lib/perl5/site_perl/5.6.1/i386-linux/auto/GTop/GTop.so
     40383 (  39k) : /usr/lib/perl5/site_perl/5.6.1/i386-linux/auto/Apache/Request/Request.so
     38456 (  38k) : /lib/libnss_nisplus-2.2.4.so
     37704 (  37k) : /lib/libnss_nis-2.2.4.so
     37252 (  36k) : /lib/libnss_files-2.2.4.so
     33104 (  32k) : /usr/lib/libgtop_sysdeps.so.1.0.12
     26096 (  25k) : /usr/lib/libgtop.so.1.0.12
     25543 (  25k) : /usr/lib/perl5/site_perl/5.6.1/i386-linux/auto/Apache/Scoreboard/Scoreboard.so
```

Figure 5-5. Extended information about processes: loaded libraries

the tool should not attempt to do anything at all. For example, if a hardware failure occurs, it is almost certain that a human will have to intervene.

Below are some techniques and tools that apply to each category.

mod_perl server watchdogs

One simple watchdog solution is to use a slightly modified *apachectl* script, which we have called *apache.watchdog*. Call it from *cron* every 30 minutes—or even every minute—to make sure that the server is always up.

The *crontab* entry for 30-minute intervals would read:

```
5,35 * * * * /path/to/the/apache.watchdog >/dev/null 2>&1
```

The script is shown in Example 5-8.

Example 5-8. apache.watchdog

```
  --------------------
#!/bin/sh

# This script is a watchdog checking whether
# the server is online.
# It tries to restart the server, and if it is
# down it sends an email alert to the admin.

# admin's email
EMAIL=webmaster@example.com

# the path to the PID file
PIDFILE=/home/httpd/httpd_perl/logs/httpd.pid

# the path to the httpd binary, including any options if necessary
HTTPD=/home/httpd/httpd_perl/bin/httpd_perl

# check for pidfile
if [ -f $PIDFILE ] ; then
    PID=`cat $PIDFILE`

    if kill -0 $PID; then
        STATUS="httpd (pid $PID) running"
        RUNNING=1
    else
        STATUS="httpd (pid $PID?) not running"
        RUNNING=0
    fi
else
    STATUS="httpd (no pid file) not running"
    RUNNING=0
fi

if [ $RUNNING -eq 0 ]; then
    echo "$0 $ARG: httpd not running, trying to start"
    if $HTTPD ; then
        echo "$0 $ARG: httpd started"
        mail $EMAIL -s "$0 $ARG: httpd started" \
             < /dev/null > /dev/null 2>&1
    else
        echo "$0 $ARG: httpd could not be started"
```

Example 5-8. apache.watchdog (continued)

```
        mail $EMAIL -s "$0 $ARG: httpd could not be started" \
             < /dev/null > /dev/null 2>&1
    fi
fi
```

Another approach is to use the Perl LWP module to test the server by trying to fetch a URI served by the server. This is more practical because although the server may be running as a process, it may be stuck and not actually serving any requests—for example, when there is a stale lock that all the processes are waiting to acquire. Failing to get the document will trigger a restart, and the problem will probably go away.

We set a *cron* job to call this LWP script every few minutes to fetch a document generated by a very light script. The best thing, of course, is to call it every minute (the finest resolution *cron* provides). Why so often? If the server gets confused and starts to fill the disk with lots of error messages written to the *error_log*, the system could run out of free disk space in just a few minutes, which in turn might bring the whole system to its knees. In these circumstances, it is unlikely that any other child will be able to serve requests, since the system will be too busy writing to the *error_log* file. Think big—if running a heavy service, adding one more request every minute will have no appreciable impact on the server's load.

So we end up with a *crontab* entry like this:

```
* * * * * /path/to/the/watchdog.pl > /dev/null
```

The watchdog itself is shown in Example 5-9.

Example 5-9. watchdog.pl

```
#!/usr/bin/perl -Tw

# These prevent taint checking failures
$ENV{PATH} = '/bin:/usr/bin';
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};

use strict;
use diagnostics;

use vars qw($VERSION $ua);
$VERSION = '0.01';

require LWP::UserAgent;

###### Config ########
my $test_script_url = 'http://www.example.com:81/perl/test.pl';
my $monitor_email   = 'root@localhost';
my $restart_command = '/home/httpd/httpd_perl/bin/apachectl restart';
my $mail_program    = '/usr/lib/sendmail -t -n';
######################

$ua  = LWP::UserAgent->new;
```

Example 5-9. watchdog.pl (continued)

```
$ua->agent("$0/watchdog " . $ua->agent);
# Uncomment the following two lines if running behind a firewall
# my $proxy = "http://www-proxy";
# $ua->proxy('http', $proxy) if $proxy;

# If it returns '1' it means that the service is alive, no need to
# continue
exit if checkurl($test_script_url);

# Houston, we have a problem.
# The server seems to be down, try to restart it.
my $status = system $restart_command;

my $message = ($status == 0)
            ? "Server was down and successfully restarted!"
            : "Server is down. Can't restart.";

my $subject = ($status == 0)
            ? "Attention! Webserver restarted"
            : "Attention! Webserver is down. can't restart";

# email the monitoring person
my $to = $monitor_email;
my $from = $monitor_email;
send_mail($from, $to, $subject, $message);

# input:  URL to check
# output: 1 for success, 0 for failure
#######################
sub checkurl {
    my($url) = @_;

    # Fetch document
    my $res = $ua->request(HTTP::Request->new(GET => $url));

    # Check the result status
    return 1 if $res->is_success;

    # failed
    return 0;
}

# send email about the problem
#######################
sub send_mail {
    my($from, $to, $subject, $messagebody) = @_;

    open MAIL, "|$mail_program"
        or die "Can't open a pipe to a $mail_program :$!\n";

    print MAIL <<__END_OF_MAIL__;
To: $to
```

Example 5-9. watchdog.pl (continued)

```
From: $from
Subject: $subject

$messagebody

--
Your faithful watchdog

__END_OF_MAIL__

    close MAIL or die "failed to close |$mail_program: $!";
}
```

Of course, you may want to replace a call to *sendmail* with `Mail::Send`, `Net::SMTP` code, or some other preferred email-sending technique.

Server Maintenance Chores

It is not enough to have your server and service up and running. The server must be maintained and monitored even when everything seems to be fine. This includes security auditing as well as keeping an eye on the amount of remaining unused disk space, available RAM, the system's load, etc.

If these chores are forgotten, sooner or later the system will crash, either because it has run out of free disk space, all available RAM has been used and the system has started to swap heavily, or it has been broken into. The last issue is much too broad for this book's scope, but the others are quite easily addressed if you follow our advice.

Particular systems might require maintenance chores that are not covered here, but this section highlights some of the most important general tasks.

Handling Log Files

Apache generally logs all the web server access events in the *access_log* file, whereas errors and warnings go into the *error_log* file. The *access_log* file can later be analyzed to report server usage statistics, such as the number of requests made in different time spans, who issued these requests, and much more. The *error_log* file is used to monitor the server for errors and warnings and to prompt actions based on those reports. Some systems do additional logging, such as storing the referrers of incoming requests to find out how users have learned about the site.

The simplest logging technique is to dump the logs into a file opened for appending. With Apache, this is as simple as specifying the logging format and the file to which to log. For example, to log all accesses, use the default directive supplied in *httpd.conf*:

```
LogFormat "%h %l %u %t \"%r\" %>s %b" common
CustomLog /home/httpd/httpd_perl/logs/access_log common
```

This setting will log all server accesses to a file named */home/httpd/httpd_perl/logs/access_log* using the format specified by the `LogFormat` directive—in this case, `common`. Please refer to the Apache documentation for a complete explanation of the various tokens that you can use when specifying log formats. If you're tempted to change the format of the log file, bear in mind that some log analysis tools may expect that only the default or one of a small subset of logging formats is used.

The only risk with log files is their size. It is important to keep log files trimmed. If they are needed for later analysis, they should be rotated and the rotation files should be moved somewhere else so they do not consume disk space. You can usually compress them for storage offline.

The most important thing is to monitor log files for possible sudden explosive growth rates. For example, if a developer makes a mistake in his code running on the mod_perl server and the child processes executing the code start to log thousands of error messages a second, all disk space can quickly be consumed, and the server will cease to function.

Scheduled log file rotation

The first issue is solved by having a process that rotates the logs run by *cron* at certain times (usually off-peak hours, if this term is still valid in the 24-hour global Internet era). Usually, log rotation includes renaming the current log file, restarting the server (which creates a fresh new log file), and compressing and/or moving the rotated log file to a different disk.

For example, if we want to rotate the *access_log* file, we could do:

```
panic% mv access_log access_log.renamed
panic% apachectl graceful
panic% sleep 5
panic% mv access_log.renamed /some/directory/on/another/disk
```

The *sleep* delay is added to make sure that all children complete requests and logging. It's possible that a longer delay is needed. Once the restart is completed, it is safe to use *access_log.renamed*.

There are several popular utilities, such as *rotatelogs* and *cronolog*, that can perform the rotation, although it is also easy to create a basic rotation script. Example 5-10 shows a script that we run from *cron* to rotate our log files.

Example 5-10. logrotate

```
#!/usr/local/bin/perl -Tw

# This script does log rotation. Called from crontab.

use strict;
$ENV{PATH}='/bin:/usr/bin';
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

Example 5-10. logrotate (continued)

```
### configuration
my @logfiles = qw(access_log error_log);
umask 0;
my $server = "httpd_perl";
my $logs_dir = "/home/httpd/$server/logs";
my $restart_command = "/home/httpd/$server/bin/apachectl restart";
my $gzip_exec = "/usr/bin/gzip -9"; # -9 is maximum compression

my ($sec, $min, $hour, $mday, $mon, $year) = localtime(time);
my $time = sprintf "%0.4d.%0.2d.%0.2d-%0.2d.%0.2d.%0.2d",
                   $year+1900, ++$mon, $mday, $hour, $min, $sec;

chdir $logs_dir;

# rename log files
foreach my $file (@logfiles) {
    rename $file, "$file.$time";
}

# now restart the server so the logs will be restarted
system $restart_command;

# allow all children to complete requests and logging
sleep 5;

# compress log files
foreach my $file (@logfiles) {
    system "$gzip_exec $file.$time";
}
```

As can be seen from the code, the rotated files will include the date and time in their filenames.

Non-scheduled emergency log rotation

As we mentioned earlier, there are times when the web server goes wild and starts to rapidly log lots of messages to the *error_log* file. If no one monitors this, it is possible that in a few minutes all free disk space will be consumed and no process will be able to work normally. When this happens, the faulty server process may cause so much I/O that its sibling processes cannot serve requests.

Although this rarely happens, you should try to reduce the risk of it occurring on your server. Run a monitoring program that checks the log file size and, if it detects that the file has grown too large, attempts to restart the server and trim the log file.

Back when we were using quite an old version of mod_perl, we sometimes had bursts of “Callback called exit” errors showing up in our *error_log*. The file could grow to 300 MB in a few minutes.

Example 5-11 shows a script that should be executed from *crontab* to handle situations like this. This is an emergency solution, not to be used for routine log rotation.

The *cron* job should run every few minutes or even every minute, because if the site experiences this problem, the log files will grow very rapidly. The example script will rotate when *error_log* grows over 100K. Note that this script is still useful when the normal scheduled log-rotation facility is working.

Example 5-11. emergency_rotate.sh

```
#!/bin/sh
S=`perl -e 'print -s "/home/httpd/httpd_perl/logs/error_log"'`;
if [ "$S" -gt 100000 ] ; then
    mv /home/httpd/httpd_perl/logs/error_log \
       /home/httpd/httpd_perl/logs/error_log.old
    /etc/rc.d/init.d/httpd restart
    date | /bin/mail -s "error_log $S kB" admin@example.com
fi
```

Of course, a more advanced script could be written using timestamps and other bells and whistles. This example is just a start, to illustrate a basic solution to the problem in question.

Another solution is to use ready-made tools that are written for this purpose. The *daemontools* package includes a utility called *multilog* that saves the STDIN stream to one or more log files. It optionally timestamps each line and, for each log, includes or excludes lines matching specified patterns. It automatically rotates logs to limit the amount of disk space used. If the disk fills up, it pauses and tries again, without losing any data.

The obvious caveat is that it does not restart the server, so while it tries to solve the log file–handling issue, it does not deal with the problem's real cause. However, because of the heavy I/O induced by the log writing, other server processes will work very slowly if at all. A normal watchdog is still needed to detect this situation and restart the Apache server.

Centralized logging

If you are running more than one server on the same machine, Apache offers the choice of either having a separate set of log files for each server, or using a central set of log files for all servers. If you are running servers on more than one machine, having them share a single log file is harder to achieve, but it is possible, provided that a filesharing system is used (logging into a database, or a special purpose application like *syslog*).

There are a few file-sharing systems that are widely used:

Network File System (NFS)
: NFS is a network file-sharing system. It's a very useful system, when it works. Unfortunately, it breaks too often, which makes it unreliable to use on production systems. NFS is available on most Unix flavors.

Andrew File System (AFS)
: AFS is a distributed filesystem that enables cooperating hosts (clients and servers) to efficiently share filesystem resources across both local area and wide area networks. This filesystem is reliable, but it costs money and is available only on the HP, Next, DEC, IBM, SUN, and SGI operating systems. For more information, see *http://www.transarc.com/* and *http://www.angelfire.com/hi/plutonic/afs-faq.html*.

Coda
: Coda is a distributed filesystem with its origin in AFS2. It has many features that are very desirable for network filesystems. Coda is platform-independent: you can mix and match servers and clients on any supported platform. As of this writing, it's not clear how stable the system is; some people have reported success using it, but others have had some problems with it. For more information, see *http://www.coda.cs.cmu.edu/*.

Apache permits the location of the file used for logging purposes to be specified, but it also allows you to specify a program to which all logs should be piped. To log to a program, modify the log handler directive (for example, `CustomLog`) to use the logging program instead of specifying an explicit filename:

```
LogFormat "%h %l %u %t \"%r\" %>s %b" common
CustomLog "| /home/httpd/httpd_perl/bin/sqllogger.pl" common
```

Logging into a database is a common solution, because you can do insertions from different machines into a single database. Unless the logger is programmed to send logs to a few databases at once, this solution is not reliable, since a single database constitutes a single failure point. If the database goes down, the logs will be lost. Sending information to one target is called *unicast* (see Figure 5-6), and sending to more than one target is called *multicast* (see Figure 5-7). In the latter case, if one database goes down, the others will still collect the data.

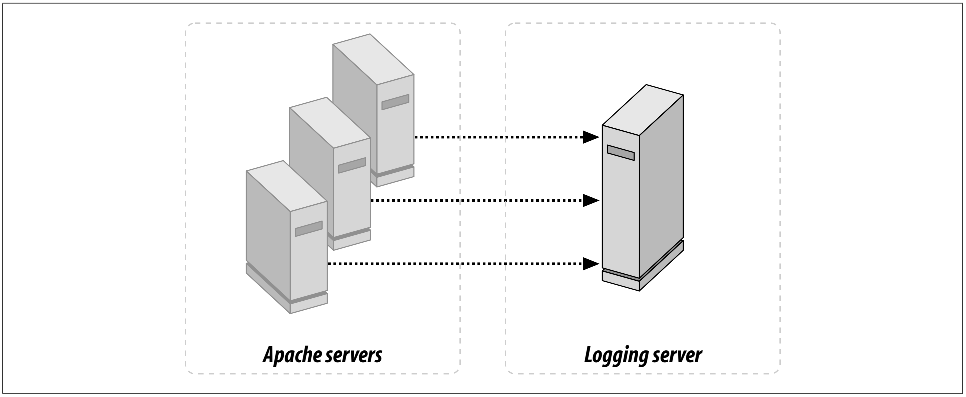


Figure 5-6. Unicast solution

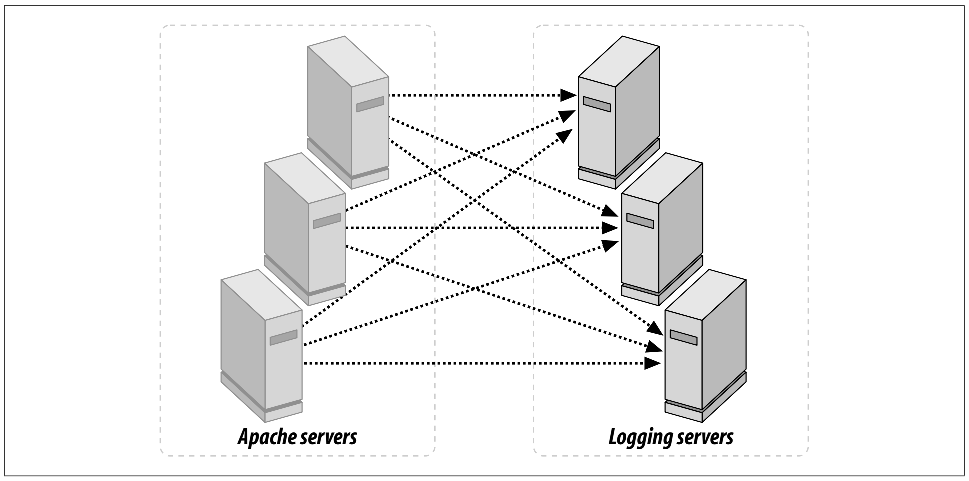


Figure 5-7. Multicast solution

Another solution is to use a centralized logger program based on *syslog(3)* to send all logs to a central location on a master host. *syslog(3)* is not a very scalable solution, because it's slow. It's also unreliable—it uses UDP to send the data, which doesn't ensure that the data will reach its destination. This solution is also unicast: if the master host goes down, the logs will be lost.

One advanced system that provides consolidated logging is `mod_log_spread`. Based on the group communications toolkit Spread, using IP multicast, `mod_log_spread` provides reliable, scalable centralized logging whith minimal performance impact on the web servers. For more information, see *http://www.backhand.org/mod_log_spread/*.

Swapping Prevention

Before we delve into swapping process details, let's look briefly at memory components and memory management.

Computer memory is called RAM (Random Access Memory). Reading and writing to RAM is faster than doing the same operations on a hard disk, by around five orders of magnitude (and growing). RAM uses electronic memory cells (transistors) with no moving parts, while hard disks use a rotating magnetic medium. It takes about one tenth of a microsecond to write to RAM but something like ten thousand microseconds to write to hard disk. It is possible to write just one byte (well, maybe one word) to RAM, whereas the minimum that can be written to a disk is often four thousand or eight thousand bytes (a single block). We often refer to RAM as *physical memory*.

A program may take up many thousands of bytes on disk. However, when it is executed normally, only the parts of the code actually needed at the time are loaded into memory. We call these parts *segments*.

Using syslog

The *syslog* solution can be implemented using the following configuration:

```
LogFormat "%h %l %u %t \"%r\" %>s %b" common
CustomLog "| /home/httpd/httpd_perl/bin/syslogger.pl hostnameX" common
```

where a simple *syslogger.pl* can look like this:

```
#!/usr/bin/perl
use Sys::Syslog qw(:DEFAULT setlogsock);

my $hostname = shift || 'localhost';
my $options  = 'ndelay'; # open the connection immediately
my $facility = 'local0'; # one of local0..local7
my $priority = 'info';   # debug|info|notice|warning|err...

setlogsock 'unix';
openlog $hostname, $options, $facility;
while (<>) {
    chomp;
    syslog $priority, $_;
}
closelog;
```

The *syslog* utility needs to know the facility to work with and the logging level. We will use *local0*, one of the special logging facilities reserved for local usage (eight local facilities are available: *local0* through *local7*). We will use the *info* priority level (again, one of eight possible levels: *debug*, *info*, *notice*, *warning*, *err*, *crit*, *alert*, and *emerg*).

Now make the *syslog* utility on the master machine (where all logs are to be stored) log all messages coming from facility *local0* with logging level `info` to a file of your choice. This is achieved by editing the */etc/syslog.conf* file. For example:

```
local0.info /var/log/web/access_log
```

All other machines forward their logs from facility *local0* to the central machine. Therefore, on all but the master machine, we add the forwarding directive to the */etc/syslog.conf* file (assuming that the master machine's hostname is *masterhost*):

```
local0.info @masterhost
```

We must restart the *syslogd* daemon or send it the HUP kill signal for the changes to take effect before the logger can be used.

On most operating systems, swap memory is used as an extension for RAM and not as a duplication of it. Assuming the operating system you use is one of those, if there is 128 MB of RAM and a 256 MB swap partition, there is a total of 384 MB of memory available. However, the extra (swap) memory should never be taken into consideration when deciding on the maximum number of processes to be run (we will show you why in a moment). The swap partition is also known as *swap space* or *virtual memory*.

The swapping memory can be built from a number of hard disk partitions and swap files formatted to be used as swap memory. When more swap memory is required, as long as there is some free disk space, it can always be extended on demand. (For more information, see the *mkswap* and *swapon* manpages.)

System memory is quantified in units called *memory pages*. Usually the size of a memory page is between 1 KB and 8 KB. So if there is 256 MB of RAM installed on the machine, and the page size is 4 KB, the system has 64,000 main memory pages to work with, and these pages are fast. If there is a 256-MB swap partition, the system can use yet another 64,000 memory pages, but they will be much slower.

When the system is started, all memory pages are available for use by the programs (processes). Unless a program is really small (in which case at any one time the entire program will be in memory), the process running this program uses only a few segments of the program, each segment mapped onto its own memory page. Therefore, only a few memory pages are needed—generally fewer than the program's size might imply.

When a process needs an additional program segment to be loaded into memory, it asks the system whether the page containing this segment is already loaded. If the page is not found, an event known as a "page fault" occurs. This requires the system to allocate a free memory page, go to the disk, and finally read and load the requested segment into the allocated memory page.

If a process needs to bring a new page into physical memory and there are no free physical pages available, the operating system must make room for this page by discarding another page from physical memory.

If the page to be discarded from physical memory came from a binary image or data file and has not been modified, the page does not need to be saved. Instead, it can be discarded, and if the process needs that page again it can be brought back into memory from the image or data file.

However, if the page has been modified, the operating system must preserve the contents of that page so that it can be accessed at a later time. This type of page is known as a *dirty page*, and when it is removed from memory it is saved in a special sort of file called the *swap file*. This process is referred to as *swapping out*.

Accesses to the swap file are very slow compared with the speed of the processor and physical memory, and the operating system must juggle the need to write pages to disk with the need to retain them in memory to be used again.

To try to reduce the probability that a page will be needed just after it has been swapped out, the system may use the LRU (least recently used) algorithm or some similar algorithm.

To summarize the two swapping scenarios, discarding read-only pages incurs little overhead compared with discarding data pages that have been modified, since in the

,ch05.22279 Page 207 Thursday, November 18, 2004 12:36 PM

latter case the pages have to be written to a swap partition located on the (very slow) disk. Thus, the fewer memory pages there are that can become dirty, the better will be the machine's overall performance.

But in Perl, both the program code and the program data are seen as data pages by the OS. Both are mapped to the same memory pages. Therefore, a big chunk of Perl code can become dirty when its variables are modified, and when those pages need to be discarded they have to be written to the swap partition.

This leads us to two important conclusions about swapping and Perl:

1. Running the system when there is no free physical memory available hinders performance, because processes' memory pages will be discarded and then reread from disk again and again.
2. Since the majority of the running code is Perl code, in addition to the overhead of reading in the previously discarded pages, there is the additional overhead of saving the dirty pages to the swap partition.

When the system has to swap memory pages in and out, it slows down. This can lead to an accumulation of processes waiting for their turn to run, which further increases processing demands, which in turn slows down the system even more as more memory is required. Unless the resource demand drops and allows the processes to catch up with their tasks and go back to normal memory usage, this ever-worsening spiral can cause the machine to thrash the disk and ultimately to halt.

In addition, it is important to be aware that for better performance, many programs (particularly programs written in Perl) do not return memory pages to the operating system even when they are no longer needed. If some of the memory is freed, it is reused when needed by the process itself, without creating the additional overhead of asking the system to allocate new memory pages. That is why Perl programs tend to grow in size as they run and almost never shrink.

When the process quits, it returns all the memory pages it used to the pool of available pages for other processes to use.

It should now be obvious that a system that runs a web server should never swap. Of course, it is quite normal for a desktop machine to swap, and this is often apparent because everything slows down and sometimes the system starts freezing for short periods. On a personal machine, the solution to swapping is simple: do not start up any new programs for a minute, and try to close down any that are running unnecessarily. This will allow the system to catch up with the load and go back to using just RAM. Unfortunately, this solution cannot be applied to a web server.

In the case of a web server, we have much less control, since it is the remote users who load the machine by issuing requests to the server. Therefore, the server should be configured such that the maximum number of possible processes will be small enough for the system to handle. This is achieved with the `MaxClients` directive, discussed in Chapter 11. This will ensure that at peak times, the system will not swap.

Remember that for a web server, swap space is an emergency pool, not a resource to be used routinely. If the system is low on memory, either buy more memory or reduce the number of processes to prevent swapping, as discussed in Chapter 14.

However, due to faulty code, sometimes a process might start running in an infinite loop, consuming all the available RAM and using lots of swap memory. In such a situation, it helps if there is a big emergency pool (i.e., lots of swap memory). But the problem must still be resolved as soon as possible, since the pool will not last for long. One solution is to use the `Apache::Resource` module, described in the next section.

Limiting Resources Used by Apache Child Processes

There are times when we need to prevent processes from excessive consumption of system resources. This includes limiting CPU or memory usage, the number of files that can be opened, and more.

The `Apache::Resource` module uses the `BSD::Resource` module, which in turn uses the C function `setrlimit()` to set limits on system resources.

A resource limit is specified in terms of a soft limit and a hard limit. When a soft limit (for example, CPU time or file size) is exceeded, the process may receive a signal, but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The `rlimit` structure is used to specify the hard and soft limits on a resource. (See the *setrlimit* manpage for OS-specific information.)

If the value of variable in `rlimit` is of the form `S:H`, `S` is treated as the soft limit, and `H` is the hard limit. If the value is a single number, it is used for both soft and hard limits. So if the value is `10:20`, the soft limit is 10 and the hard limit is 20, whereas if the value is just `20`, both the soft and the hard limits are set to 20.

The most common use of this module is to limit CPU usage. The environment variable `PERL_RLIMIT_CPU` defines the maximum amount of CPU time the process can use. If it attempts to run longer than this amount, it is killed, no matter what it is doing at the time, be it processing a request or just waiting. This is very useful when there is a bug in the code and a process starts to spin in an infinite loop, using a lot of CPU resources and never completing the request.

The value is measured in seconds. The following example sets the soft limit for CPU usage to 120 seconds (the default is 360):

```
PerlModule Apache::Resource
PerlSetEnv PERL_RLIMIT_CPU 120
```

Although 120 seconds does not sound like a long time, it represents a great deal of work for a modern processor capable of millions of instructions per second. Furthermore, because the child process shares the CPU with other processes, it may be quite some time before it uses all its allotted CPU time, and in all probability it will die from other causes (for example, it may have served all the requests it is permitted to serve before this hard limit is reached).

Of course, we should tell mod_perl to use this module, which is done by adding the following directive to *httpd.conf*:

```
PerlChildInitHandler Apache::Resource
```

There are other resources that we might want to limit. For example, we can limit the data and bstack memory segment sizes (`PERL_RLIMIT_DATA` and `PERL_RLIMIT_STACK`), the maximum process file size (`PERL_RLIMIT_FSIZE`), the core file size (`PERL_RLIMIT_CORE`), the address space (virtual memory) limit (`PERL_RLIMIT_AS`), etc. Refer to the *setrlimit* manpage for other possible resources. Remember to prepend `PERL_` to the resource types that are listed in the manpage.

If `Apache::Status` is configured, it can display the resources set in this way. Remember that `Apache::Status` must be loaded before `Apache::Resource`, in order to enable the resources display menu.

To turn on debug mode, set the `$Apache::Resource::Debug` variable before loading the module. This can be done using a Perl section in *httpd.conf*.

```
<Perl>
    $Apache::Resource::Debug = 1;
    require Apache::Resource;
</Perl>
PerlChildInitHandler Apache::Resource
```

Now view the *error_log* file using *tail -f* and watch the debug messages show up when requests are served.

OS-specific notes

Under certain Linux setups, `malloc( )` uses `mmap( )` instead of `brk( )`. This is done to conserve virtual memory—that is, when a program `malloc( )`s a large block of memory, the block is not actually returned to the program until it is initialized. The old-style `brk( )` system call obeyed resource limits on data segment sizes as set in `setrlimit( )`. `mmap( )` does not.

`Apache::Resource`'s defaults put limits on data size and stack size. Linux's current memory-allocation scheme does not honor these limits, so if we just do:

```
PerlSetEnv PERL_RLIMIT_DEFAULTS On
PerlModule Apache::Resource
PerlChildInitHandler Apache::Resource
```

our Apache processes are still free to use as much memory as they like.

However, `BSD::Resource` also has a limit called `RLIMIT_AS` (Address Space), which limits the total number of bytes of virtual memory assigned to a process. Fortunately, Linux's memory manager *does* honor this limit.

Therefore, we *can* limit memory usage under Linux with `Apache::Resource`. Simply add a line to *httpd.conf*:

```
PerlSetEnv PERL_RLIMIT_AS  67108864
```

This example sets hard and soft limits of 64 MB of total address space.

Refer to the `Apache::Resource` and *setrlimit(2)* manpages for more information.

Tracking and Terminating Hanging Processes

Generally, limits should be imposed on mod_perl processes to prevent mayhem if something goes wrong. There is no need to limit processes if the code does not have any bugs, or at least if there is sufficient confidence that the program will never over-consume resources. When there is a risk that a process might hang or start consuming a lot of memory, CPU, or other resources, it is wise to use the `Apache::Resource` module.

But what happens if a process is stuck waiting for some event to occur? Consider a process trying to acquire a lock on a file that can never be satisfied because there is a deadlock. The process just hangs waiting, which means that neither extra CPU nor extra memory is used. We cannot detect and terminate this process using the resource-limiting techniques we just discussed. If there is such a process, it is likely that very soon there will be many more processes stuck waiting for the same or a different event to occur. Within a short time, all processes will be stuck and no new processes will be spawned because the maximum number, as specified by the `MaxClients` directive, has been reached. The service enters a state where it is up but not serving clients.

If a watchdog is run that does not just check that the process is up, but actually issues requests to make sure that the service responds, then there is some protection against a complete service outage. This is because the watchdog will restart the server if the testing request it issues times out. This is a last-resort solution; the ideal is to be able to detect and terminate hanging processes that do not consume many resources (and therefore cannot be detected by the `Apache::Resource` module) as soon as possible, not when the service stops responding to requests, since by that point the quality of service to the users will have been severely degraded.

This is where the `Apache::Watchdog::RunAway` module comes in handy. This module samples all live child processes every `$Apache::Watchdog::RunAway::POLLTIME` seconds. If a process has been serving the same request for more than `$Apache::Watchdog::RunAway::TIMEOUT` seconds, it is killed.

To perform accounting, the `Apache::Watchdog::RunAway` module uses the `Apache::Scoreboard` module, which in turn delivers various items of information about live child processes. Therefore, the following configuration must be added to *httpd.conf*:

```
<Location /scoreboard>
    SetHandler perl-script
    PerlHandler Apache::Scoreboard::send
    order deny,allow
    deny from all
    allow from localhost
</Location>
```

Make sure to adapt the access permission to the local environment. The above configuration allows access to this handler only from the *localhost* server. This setting can be tested by issuing a request for *http://localhost/scoreboard*. However, the returned data cannot be read directly, since it uses a binary format.

We are now ready to configure `Apache::Watchdog::RunAway`. The module should be able to retrieve the information provided by `Apache::Scoreboard`, so we will tell it the URL to use:

```
$Apache::Watchdog::RunAway::SCOREBOARD_URL = "http://localhost/scoreboard";
```

We must decide how many seconds the process is allowed to be busy serving the same request before it is considered a runaway. Consider the slowest clients. Scripts that do file uploading and downloading might take a significantly longer time than normal mod_perl code.

```
$Apache::Watchdog::RunAway::TIMEOUT = 180; # 3 minutes
```

Setting the timeout to `0` will disable the `Apache::Watchdog::RunAway` module entirely.

The rate at which the module polls the server should be chosen carefully. Because of the overhead of fetching the scoreboard data, this is not a module that should be executed too frequently. If the timeout is set to a few minutes, sampling every one or two minutes is a good choice. The following directive specifies the polling interval:

```
$Apache::Watchdog::RunAway::POLLTIME = 60; # 1 minute
```

Just like the timeout value, polling time is measured in seconds.

To see what the module does, enable debug mode:

```
$Apache::Watchdog::RunAway::DEBUG = 1;
```

and watch its log file using the *tail* command.

The following statement allows us to specify the log file's location:

```
$Apache::Watchdog::RunAway::LOG_FILE = "/tmp/safehang.log";
```

This log file is also used for logging information about killed processes, regardless of the value of the `$DEBUG` variable.

The module uses a lock file in order to prevent starting more than one instance of itself. The default location of this file may be changed using the `$LOCK_FILE` variable.

```
$Apache::Watchdog::RunAway::LOCK_FILE = "/tmp/safehang.lock";
```

There are two ways to invoke this process: using the Perl functions, or using the bundled utility called *amprapmon* (mnemonic: *ApacheModPerlRunAwayProcessMonitor*).

The following functions are available:

`stop_monitor()`
: Stops the monitor based on the PID contained in the lock file. Removes the lock file.

`start_monitor()`
: Starts the monitor in the current process. Creates the lock file.

`start_detached_monitor()`
: Starts the monitor as a forked process (used by *amprapmon*). Creates the lock file.

In order for mod_perl to invoke this process, all that is needed is the `start_detached_monitor()` function. Add the following code to *startup.pl*:

```
use Apache::Watchdog::RunAway();
Apache::Watchdog::RunAway::start_detached_monitor();
```

Another approach is to use the *amprapmon* utility. This can be started from the *startup.pl* file:

```
system "amprapmon start";
```

This will fork a new process. If the process is already running, it will just continue to run.

The *amprapmon* utility could instead be started from *cron* or from the command line.

No matter which approach is used, the process will fork itself and run as a daemon process. To stop the daemon, use the following command:

```
panic% amprapmon stop
```

If we want to test this module but have no code that makes processes hang (or we do, but the behavior is not reproducible on demand), the following code can be used to make the process hang in an infinite loop when executed as a script or handler. The code writes `"\0"` characters to the browser every second, so the request will never time out. The code is shown in Example 5-12.

Example 5-12. hangnow.pl

```
my $r = shift;
$r->send_http_header('text/plain');
print "PID = $$\n";
$r->rflush;
while(1) {
    $r->print("\0");
    $r->rflush;
    sleep 1;
}
```

The code prints the PID of the process running it before it goes into an infinite loop, so that we know which process hangs and whether it gets killed by the `Apache::Watchdog::RunAway` daemon as it should.

Of course, the watchdog is used only for prevention. If you have a serious problem with hanging processes, you have to debug your code, find the reason for the problem, and resolve it, as discussed in Chapter 21.

Limiting the Number of Processes Serving the Same Resource

To limit the number of Apache children that can simultaneously serve a specific resource, take a look at the Apache `mod_throttle_access` module.

Throttling access is useful, for example, when a handler uses a resource that places a limitation on concurrent access or that is very CPU-intensive. `mod_throttle_access` limits the number of concurrent requests to a given URI.

Consider a service providing the following three URIs:

```
/perl/news/
/perl/webmail/
/perl/morphing/
```

The response times of the first two URIs are critical, since people want to read the news and their email interactively. The third URI is a very CPU- and RAM-intensive image-morphing service, provided as a bonus to the users. Since we do not want users to abuse this service, we have to set some limit on the number of concurrent requests for this resource. If we do not, the other two critical resources may have their performance degraded.

When compiled or loaded into Apache and enabled, `mod_throttle_access` makes the `MaxConcurrentReqs` directive available. For example, the following setting:

```
<Location "/perl/morphing">
    <Limit PUT GET POST>
        MaxConcurrentReqs 10
    </Limit>
</Location>
```

will allow only 10 concurrent `PUT`, `GET`, `HEAD` (as implied by `GET`), or `POST` requests for the URI */perl/morphing* to be processed at any given time. The other two URIs in our example remain unlimited.

Limiting the Request-Rate Speed (Robot Blocking)

Web services generally welcome search engine robots, also called *spiders*. Search engine robots are programs that query the site and index its documents for a search engine.

Most indexing robots are polite and pause between requests. However, some search engine robots behave very badly, issuing too many requests too often, thus slowing down the service for human users. While everybody wants their sites to be indexed by search engines, it is really annoying when an initially welcomed spider gives the server a hard time, eventually becoming an unwanted spider.

A common remedy for keeping impolite robots off a site is based on an `AccessHandler` that checks the name of the robot and disallows access to the server if it is listed in the robot blacklist. For an example of such an `AccessHandler`, see the `Apache::BlockAgent` module, available from *http://www.modperl.com/*.

Unfortunately, some robots have learned to work around this blocking technique, masquerading as human users by using user agent strings identifying them as conventional browsers. This prevents us from blocking just by looking at the robot's name—we have to be more sophisticated and beat the robots by turning their own behavior against them. Robots work much faster than humans, so we can gather statistics over a period of time, and when we detect too many requests issued too fast from a specific IP, this IP can be blocked.

The `Apache::SpeedLimit` module, also available from *http://www.modperl.com/*, provides this advanced filtering technique.

There might be a problem with proxy servers, however, where many users browse the Web via a single proxy. These users are seen from the outside world (and from our sites) as coming from the proxy's single IP address or from one of a small set of IP addresses. In this case, `Apache::SpeedLimit` cannot be used, since it might block legitimate users and not just robots. However, we could modify the module to ignore specific IP addresses that we designate as acceptable.

Stonehenge::Throttle

Randal Schwartz wrote `Stonehenge::Throttle` for one of his *Linux Magazine* columns. This module does CPU percentage–based throttling. The module looks at the recent CPU usage over a given window for a given IP. If the percentage exceeds a threshold, a 503 error and a correct `Retry-After:` header are sent, telling for how long access from this IP is banned. The documentation can be found at *http://www.stonehenge.com/merlyn/LinuxMag/col17.html*, and the source code is available at *http://www.stonehenge.com/merlyn/LinuxMag/col17.listing.txt*.

Spambot Trap

Neil Gunton has developed a Spambot Trap (*http://www.neilgunton.com/spambot_trap/*) that keeps robots harvesting email addresses away from your web content. One of the important components of the trap is the *robots.txt* file, which is a standard mechanism for controlling which agents can reach your site and which areas can be browsed. This is an advisory mechanism, so if the agent doesn't follow the standard it will simply ignore the rules of the house listed in this file. For more information, refer to the W3C specification at *http://www.w3.org/TR/html401/appendix/notes.html#h-B.4.1.1*.

References

- "Stopping and Restarting Apache," from the Apache documentation: *http://httpd.apache.org/docs/stopping.html*.
- RPM resources:
  - The Red Hat Package Manager web site: *http://www.rpm.org/*.
  - *Maximum RPM*, by Ed Bailey (Red Hat Press).
  - "RPM-HOWTO," by Donnie Barnes: *http://www.rpm.org/support/RPM-HOWTO.html*.
- CVS (Concurrent Versions System) resources:
  - *http://www.cvshome.org/* is the home of the CVS project and includes a plethora of documentation. Of special interest is the *Cederqvist*, the official CVS manual, available at *http://www.cvshome.org/docs/manual/*.
  - *Open Source Development with CVS*, by Karl Fogel (Coriolis, Inc.). Most of the book is available online at *http://cvsbook.red-bean.com/*.
  - CVS Quick Reference Card: *http://www.refcards.com/about/cvs.html*.
- daemontools, a collection of tools for managing Unix services: *http://cr.yp.to/daemontools.html*.
- Log collecting and processing tools: *http://www.apache-tools.com/search.jsp?keys=log*.
- *cronolog*, a log file–rotation program for the Apache web server: *http://www.cronolog.org/*.
- mod_log_spread, which provides reliable distributed logging for Apache *http://www.backhand.org/mod_log_spread/*.
- Spread, a wide area group communication system: *http://www.spread.org/*.
- Recall, an open source library for writing distributed, fault-tolerant, replicated storage servers. A Recall-based server will allow you to save and access data even in the presence of machine failures. See *http://www.fault-tolerant.org/recall/*.
- Chapters 2, 4, 9, 11, and 28 in *UNIX System Administration Handbook*, by Evi Nemeth, Garth Snyder, Scott Seebass, and Trent H. Hein (Prentice Hall).
- Chapters 4 and 5 in *Optimizing UNIX for Performance*, by Amir H. Majidimehr (Prentice Hall).
- To learn more about memory management, refer to a book that deals with operating system theory, and especially with the operating systems used on web server machines.

A good starting point is one of the classic textbooks used in operating system courses. For example:

— *Operating System Concepts*, by Abraham Silberschatz and Peter Baer Galvin (John Wiley & Sons, Inc.).

— *Applied Operating System Concepts*, by Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (John Wiley & Sons, Inc.).

— *Design of the Unix Operating System*, by Maurice Bach (Prentice Hall).

The Memory Management Reference at *http://www.xanalys.com/software_tools/mm/* is also very helpful.

- mod_throttle_access: *http://www.fremen.org/apache/mod_throttle_access.html*.
- mod_backhand, which provides load balancing for Apache: *http://www.backhand.org/mod_backhand/*.
- The High-Availability Linux Project, the definitive guide to load-balancing techniques: *http://www.linux-ha.org/*.

  The Heartbeat project is a part of the HA Linux project.
- *lbnamed*, a load-balancing name server written in Perl: *http://www.stanford.edu/~riepel/lbnamed/* or *http://www.stanford.edu/~schemers/docs/lbnamed/lbnamed.html*.
- "Network Address Translation and Networks: Virtual Servers (Load Balancing)": *http://www.suse.de/~mha/linux-ip-nat/diplom/node4.html#SECTION00043100000000000000*.
- Linux Virtual Server Project: *http://www.linuxvirtualserver.org/*.
- Linux and port forwarding: *http://www.netfilter.org/ipchains/* or *http://www.netfilter.org/*.
- "Efficient Support for P-HTTP in Cluster-Based Web Servers," by Mohit Aron and Willy Zwaenepoel, in Proceedings of the USENIX 1999 Annual Technical Conference, Monterey, CA, June 1999: *http://www.cs.rice.edu/~druschel/usenix99lard.ps.gz* or *http://www.usenix.org/publications/library/proceedings/usenix99/full_papers/aron/aron_html/*.
- IP filter: *http://coombs.anu.edu.au/~avalon/*. The latest IP filter includes some simple load-balancing code that allows a round-robin distribution onto several machines via *ipnat*.
- Perl modules available from *http://www.modperl.com/book/source* (not on CPAN):

  — `Apache::BlockAgent`, which allows you to block impolite web agents.

  — `Apache::SpeedLimit`, which allows you to limit indexing robots' speed.