1 Controlling and Monitoring the Server

1.1 Description

Covers techniques to restart mod_perl enabled Apache, SUID scripts, monitoring, and other maintenance chores, as well as some specific setups.

1.2 Restarting Techniques

All of these techniques require that you know the server process id (PID). The easiest way to find the PID is to look it up in the *httpd.pid* file. It's easy to discover where to look, by looking in the *httpd.conf* file. Open the file and locate the entry `PidFile`. Here is the line from one of my own *httpd.conf* files:

```
  PidFile /usr/local/var/httpd_perl/run/httpd.pid
```

As you see, with my configuration the file is */usr/local/var/httpd_perl/run/httpd.pid*.

Another way is to use the `ps` and `grep` utilities. Assuming that the binary is called *httpd_perl*, we would do:

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

or maybe:

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

This will produce a list of all the `httpd_perl` (parent and children) processes. You are looking for the parent process. If you run your server as root, you will easily locate it since it belongs to root. If you run the server as some other user (when you don't have root access, the processes will belong to that user unless defined differently in *httpd.conf*. It's still easy to find which is the parent--usually it's the process with the smallest PID.

You will see several `httpd` processes running on your system, but you should never need to send signals to any of them except the parent, whose pid is in the *PidFile*. There are three signals that you can send to the parent: `SIGTERM`, `SIGHUP`, and `SIGUSR1`.

Some folks prefer to specify signals using numerical values, rather than using symbols. If you are looking for these, check out your `kill(1)` man page. My page points to */usr/include/linux/signal.h*, the relevant entries are:

```
  #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 */
```

Note that to send these signals from the command line the `SIG` prefix must be omitted and under some operating systems they will need to be preceded by a minus sign, e.g. `kill -15` or `kill -TERM` followed by the PID.

1.3 Server Stopping and Restarting

We will concentrate here on the implications of sending `TERM`, `HUP`, and `USR1` signals (as arguments to kill(1)) to a mod_perl enabled server. See http://www.apache.org/docs/stopping.html for documentation on the implications of sending these signals to a plain Apache server.

- **TERM Signal: Stop Now**

  Sending the `TERM` signal to the parent causes it to immediately attempt to kill off all its children. Any requests in progress are terminated, and no further requests are served. This process may take quite a few seconds to complete. To stop a child, the parent sends it a `SIGHUP` signal. If that fails it sends another. If that fails it sends the `SIGTERM` signal, and as a last resort it sends the `SIGKILL` signal. For each failed attempt to kill a child it makes an entry in the *error_log*.

  When all the child processes were terminated, the parent itself exits and any open log files are closed. This is when all the accumulated `END` blocks, apart from the ones located in scripts running under `Apache::Registry` or `Apache::PerlRun` handlers. In the latter case, `END` blocks are executed after each request is served.

- **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, i.e. any requests in progress are terminated; but the parent does not exit. Instead, the parent re-reads 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. How to perform the check will be covered shortly;

  Sometimes using this approach to restart mod_perl enabled Apache may cause the processes memory incremental growth 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're not serving a request. The parent re-reads its configuration files and re-opens 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 it begins serving new requests immediately.

  The only difference between `USR1` and `HUP` is that `USR1` allows the children to complete any current requests prior to killing them off and there is no interruption in the services compared to the killing with `HUP` signal, where it might take a few seconds for a restart to get completed and there is no real service at this time.

By default, if a server is restarted (using `kill -USR1 ‘cat logs/httpd.pid‘` or with the `HUP` signal), Perl scripts and modules are not reloaded. To reload `PerlRequire`s, `PerlModule`s, other `use()`’d modules and flush the `Apache::Registry` cache, use this directive in *httpd.conf*:

```
  PerlFreshRestart On
```

Make sure you read Evil things might happen when using PerlFreshRestart.

1.4 Speeding up the Apache Termination and Restart

We’ve already mentioned that restart or termination can sometimes take quite a long time, (e.g. tens of seconds), for a mod_perl server. The reason for that is a call to the `perl_destruct()` Perl API function during the child exit phase. This will cause proper execution of `END` blocks found during server startup and will invoke the `DESTROY` method on global objects which are still alive.

It is also possible that this operation may take a long time to finish, causing a long delay during a restart. Sometimes this will be followed by a series of messages appearing in the server *error_log* file, warning that certain child processes did not exit as expected. This happens when after a few attempts advising the child process to quit, the child is still in the middle of perl_destruct(), and a lethal `KILL` signal is sent, aborting any operation the child has happened to execute and *brutally* killing it.

If your code does not contain any `END` blocks or `DESTROY` methods which need to be run during child server shutdown, or may have these, but it’s insignificant to execute them, this destruction can be avoided by setting the `PERL_DESTRUCT_LEVEL` environment variable to `-1`. 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 of the current process that would not be handled by the operating system itself. So committing database transactions and removing the lock on some resource are significant operations, but closing an ordinary file isn’t.

1.5 Using apachectl to Control the Server

The Apache distribution comes with a script to control the server. It’s called `apachectl` and it is installed into the same location as the httpd executable. We will assume for the sake of our examples that it’s in `/usr/local/sbin/httpd_perl/apachectl`:

To start httpd_perl:

```
  % /usr/local/sbin/httpd_perl/apachectl start
```

To stop httpd_perl:

```
  % /usr/local/sbin/httpd_perl/apachectl stop
```

To restart httpd_perl (if it is running, send `SIGHUP`; if it is not already running just start it):

```
  % /usr/local/sbin/httpd_perl/apachectl restart
```

Do a graceful restart by sending a `SIGUSR1`, or start if not running:

```
  % /usr/local/sbin/httpd_perl/apachectl graceful
```

To do a configuration test:

```
  % /usr/local/sbin/httpd_perl/apachectl configtest
```

Replace `httpd_perl` with `httpd_docs` in the above calls to control the `httpd_docs` server.

There are other options for `apachectl`, use the `help` option to see them all.

It's important to remember that `apachectl` uses the PID file, which is specified by the `PIDFILE` directive in *httpd.conf*. If you delete the PID file by hand while the server is running, `apachectl` will be unable to stop or restart the server.

1.6 Safe Code Updates on a Live Production Server

You have prepared a new version of code, uploaded it into a production server, restarted it and it doesn't work. What could be worse than that? You also cannot go back, because you have overwritten the good working code.

It's quite easy to prevent it, just don't overwrite the previous working files!

Personally I do all updates on the live server with the following sequence. Assume that the server root directory is */home/httpd/perl/rel*. When I'm about to update the files I create a new directory */home/httpd/perl/beta*, copy the old files from */home/httpd/perl/rel* and update it with the new files. Then I do some last sanity checks (check file permissions are [read+executable], and run `perl -c` on the new modules to make sure there no errors in them). When I think I'm ready I do:

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

Let me explain what this does.

Firstly, note that I put all the commands on one line, separated by &&, and only then press the `Enter` key. As I am working remotely, this ensures that if I suddenly lose my connection (sadly this happens sometimes) I won't leave the server down if only the `stop` command squeezed in. && also ensures that if any command fails, the rest won't be executed. I am using aliases (which I have already defined) to make the typing easier:

```
% alias | grep apachectl
graceful /usr/local/apache/bin/apachectl graceful
rehup   /usr/local/apache/sbin/apachectl restart
restart /usr/local/apache/bin/apachectl restart
start   /usr/local/apache/bin/apachectl start
stop    /usr/local/apache/bin/apachectl stop

% alias err
tail -f /usr/local/apache/logs/error_log
```

Taking the line apart piece by piece:

```
mv rel old &&
```

back up the working directory to *old*

```
mv beta rel &&
```

put the new one in its place

```
stop &&
```

stop the server

```
sleep 3 &&
```

give it a few seconds to shut down (it might take even longer)

```
restart &&
```

`restart` the server

```
err
```

view of the tail of the *error_log* file in order to see that everything is OK

`apachectl` generates the status messages a little too early (e.g. when you issue `apachectl stop` it says the server has been stopped, while in fact it's still running) so don't rely on it, rely on the `error_log` file instead.

Also notice that I use `restart` and not just `start`. I do this because of Apache's potentially long stopping times (it depends on what you do with it of course!). If you use `start` and Apache hasn't yet released the port it's listening to, the start would fail and `error_log` would tell you that the port is in use, e.g.:

```
Address already in use: make_sock: could not bind to port 8080
```

But if you use `restart`, it will wait for the server to quit and then will cleanly restart it.

Now what happens if the new modules are broken? First of all, I see immediately an indication of the problems reported in the `error_log` file, which I `tail -f` immediately after a restart command. If there's a problem, I just put everything back as it was before:

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

Usually everything will be fine, and I have had only about 10 seconds of downtime, which is pretty good!

1.7 An Intentional Disabling of Live Scripts

What happens if you really must take down the server or disable the scripts? This situation might happen when you need to do some maintenance work on your database server. If you have to take your database down then any scripts that use it will fail.

If you do nothing, the user will see either the grey `An Error has happened` message or perhaps a customized error message if you have added code to trap and customize the errors. See Redirecting Errors to the Client instead of to the error_log for the latter case.

A much friendlier approach is to confess to your users that you are doing some maintenance work and plead for patience, promising (keep the promise!) that the service will become fully functional in X minutes. There are a few ways to do this:

The first doesn't require messing with the server. It works when you have to disable a script running under `Apache::Registry` and relies on the fact that it checks whether the file was modified before using the cached version. Obviously it won't work under other handlers because these serve the compiled version of the code and don't check to see if there was a change in the code on the disk.

So if you want to disable an `Apache::Registry` script, prepare a little script like this:

```
/home/http/perl/maintenance.pl
----------------------------
#!/usr/bin/perl -Tw

use strict;
use CGI;
my $q = new CGI;
print $q->header, $q->p(
"Sorry, the service is temporarily down for maintenance.
 It will be back in ten to fifteen minutes.
 Please, bear with us.
 Thank you!");
```

So if you now have to disable a script for example `/home/http/perl/chat.pl`, just do this:

```
% mv /home/http/perl/chat.pl /home/http/perl/chat.pl.orig
% ln -s /home/http/perl/maintenance.pl /home/http/perl/chat.pl
```

Of course you server configuration should allow symbolic links for this trick to work. Make sure you have the directive

```
Options FollowSymLinks
```

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

When you’re done, it’s easy to restore the previous setup. Just do this:

```
  % mv /home/http/perl/chat.pl.orig /home/http/perl/chat.pl
```

which overwrites the symbolic link.

Now make sure that the script will have the current timestamp:

```
  % touch /home/http/perl/chat.pl
```

Apache will automatically detect the change and will use the moved script instead.

The second approach is to change the server configuration and configure a whole directory to be handled by a `My::Maintenance` handler (which you must write). For example if you write something like this:

```
  My/Maintenance.pm
  ------------------
  package My::Maintenance;
  use strict;
  use Apache::Constants qw(:common);
  sub handler {
    my $r = shift;
    print $r->send_http_header("text/plain");
    print qq{
      We apologize, but this service is temporarily stopped for
      maintenance.  It will be back in ten to fifteen minutes.
      Please, bear with us.  Thank you!
    };
    return OK;
  }
  1;
```

and put it in a directory that is in the server’s `@INC`, to disable all the scripts in Location `/perl` you would replace:

```
  <Location /perl>
    SetHandler perl-script
    PerlHandler My::Handler
    [snip]
  </Location>
```

with

```
  <Location /perl>
    SetHandler perl-script
    PerlHandler My::Maintenance
    [snip]
  </Location>
```

Now restart the server. Your users will be happy to go and read http://slashdot.org for ten minutes, knowing that you are working on a much better version of the service.

If you need to disable a location handled by some module, the second approach would work just as well.

1.8 SUID Start-up Scripts

If you want to allow a few people in your team to start and stop the server you will have to give them the root password, which is not a good thing to do. The less people know the password, the less problems are likely to be encountered. But there is an easy solution for this problem available on UNIX platforms. It’s called a setuid executable.

1.8.1 Introduction to SUID Executables

The setuid executable has a setuid permissions bit set. This sets the process’s effective user ID to that of the file upon execution. You perform this setting with the following command:

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

You probably have used setuid executables before without even knowing about it. For example when you change your password you execute the `passwd` utility, which among other things modifies the */etc/passwd* file. In order to change this file you need root permissions, the `passwd` utility has the setuid bit set, therefore when you execute this utility, its effective ID is the same of the root user ID.

You should avoid using setuid executables as a general practice. The less setuid executables you have the less likely that someone will find a way to break into your system, by exploiting some bug you didn’t know about.

When the executable is setuid to root, you have to make sure that it doesn’t have the group and world read and write permissions. If we take a look at the `passwd` utility we will see:

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

You achieve this with the following command:

```
  % chmod 4511 filename
```

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

1.8.2 Apache Startup SUID Script’s Security

In our case, we want to allow setuid access only to a specific group of users, who all belong to the same group. For the sake of our example we will use the group named *apache*. It’s important that users who aren’t root or who don’t belong to the *apache* group will not be able to execute this script. Therefore we perform the following commands:

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

The execution order is important. If you swap the command execution order you will lose the setuid bit.

Now if we look at the file we see:

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

Now we are all set... Almost...

When you start Apache, Apache and Perl modules are being loaded, code can be executed. Since all this happens with root effective ID, any code executed as if the root user was doing that. You should be very careful because while you didn't gave anyone the root password, all the users in the *apache* group have an indirect root access. Which means that if Apache loads some module or executes some code that is writable by some of these users, users can plant code that will allow them to gain a shell access to root account and become a real root.

Of course if you don't trust your team you shouldn't use this solution in first place. You can try to check that all the files Apache loads aren't writable by anyone but root, but there are too many of them, especially in the mod_perl case, where many Perl modules are loaded at the server startup.

By the way, don't let all this setuid stuff to confuse you -- when the parent process is loaded, the children processes are spawned as non-root processes. This section has presented a way to allow non-root users to start the server as root user, the rest is exactly the same as if you were executing the script as root in first place.

1.8.3 Sample Apache Startup SUID Script

Now if you are still with us, here is an example of the setuid Apache startup script.

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

Note that you must be using a version of Perl that recognizes and emulates the suid bits in order for this to work. This script will do different things depending on whether it is named `start_httpd`, `stop_httpd` or `restart_httpd`. You can use symbolic links for this purpose.

```
suid_apache_ctl
---------------
#!/usr/bin/perl -T

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

# These prevent taint warnings while running suid
$ENV{PATH}='/bin:/usr/bin';
$ENV{IFS}='';
```

```
# 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
($name) = $0 =~ m|([^/]+)$|;

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

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

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

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

    exit 0;
}

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

1.9 Preparing for Machine Reboot

When you run your own development box, it's okay to start the webserver by hand when you need to. On a production system it is possible that the machine the server is running on will have to be rebooted. When the reboot is completed, who is going to remember to start the server? It's easy to forget this task, and what happens if you aren't around when the machine is rebooted?

After the server installation is complete, it's important not to forget that you need to put a script to perform the server startup and shutdown into the standard system location, for example */etc/rc.d* under RedHat Linux, or */etc/init.d/apache* under Debian Slink Linux.

This is the directory which contains scripts to start and stop all the other daemons. The directory and file names vary from one Operating System (OS) to another, and even between different distributions of the same OS.

Generally the simplest solution is to copy the `apachectl` script to your startup directory or create a symbolic link from the startup directory to the `apachectl` script. You will find `apachectl` in the same directory as the httpd executable after Apache installation. If you have more than one Apache server you will need a separate script for each one, and of course you will have to rename them so that they can co-exist in the same directories.

For example on a RedHat Linux machine with two servers, I 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 other directories. The names are the same as the script names but they have numerical prefixes, which are used for executing the scripts in a particular order: the lower numbers are executed earlier.

When the system starts (level 3) we want the Apache to be started when almost all of the services are running already, therefore I've used *S91*. For example if the mod_perl enabled Apache issues a `connect_on_init()` the SQL server should be started before Apache.

When the system shuts down (level 6), Apache should be stopped as one of the first processes, therefore I've used `K16`. Again if the server does some cleanup processing during the shutdown event and requires third party services to be running (e.g. SQL server) it should be stopped before these services.

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

Under RedHat Linux and similar systems, when a machine is booted and its runlevel set to 3 (multiuser + network), Linux goes into */etc/rc.d/rc3.d/* and executes the scripts the symbolic links point to with the `start` argument. When it sees *S91httpd_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 RedHat Linux includes the `control-panel` utility, which amongst other things includes the `RunLevel Manager`. (which can be invoked directly as either ntsysv(8) or tksysv(8)). This will help you to create the proper symbolic links. Of course before you use it, you should put `apachectl` or similar scripts into the *init.d* or equivalent directory. Or you can have a symbolic link to some other location instead.

The simplest approach is to use the chkconfig(8) utility which adds and removes the services for you. The following example shows how to add an *httpd_perl* startup script to the system.

First move or copy the file into the directory */etc/rc.d/init.d*:

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

Now open the script in your favorite editor and add the following lines after the main header of the script:

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

So 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 RedHat Linux
# chkconfig: 2345 91 16
# description: mod_perl enabled Apache Server

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

Adjust the line:

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

to your needs. The above setting says to says that 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 have to do is to ask `chkconfig` to configure the startup scripts. Before we do that let's look at what we have:

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

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

Which means that we only have the startup script itself. Now we execute:

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

and see what has changed:

```
% 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
```

As you can see `chkconfig` created all the symbolic links for us, using the startup and shutdown priorities as specified in the line:

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

If for some reason you want to remove the service from the startup scripts, all you have to do is to tell `chkconfig` to remove the links:

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

Now if we look at the files under the directory */etc/rc.d/* we see again only the script itself.

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

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

Of course you may keep the startup script in any other directory as long as you can link to it. For example if you want to keep this file with all the Apache binaries in */usr/local/apache/bin*, all you have to do is to provide a symbolic link to this file:

```
% ln -s /usr/local/apache/bin/apachectl /etc/rc.d/init.d/httpd_perl
```

and then:

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

Note that in case of using symlinks the link name in */etc/rc.d/init.d* is what matters and not the name of the script the link points to.

1.10 Monitoring the Server. A watchdog.

With mod_perl many things can happen to your server. It is possible that the server might die when you are not around. As with any other critical service you need to run some kind of watchdog.

One simple solution is to use a slightly modified `apachectl` script, which I've named *apache.watchdog*. Call it from the crontab every 30 minutes -- or even every minute -- to make sure the server is up all the time.

The crontab entry for 30 minutes intervals:

```
0,30 * * * * /path/to/the/apache.watchdog >/dev/null 2>&1
```

The script:

```
#!/bin/sh

# this script is a watchdog to see whether the server is online
# It tries to restart the server, and if it’s
# down it sends an email alert to admin

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

# the path to your PID file
PIDFILE=/usr/local/var/httpd_perl/run/httpd.pid

# the path to your httpd binary, including options if necessary
HTTPD=/usr/local/sbin/httpd_perl/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 2>&1
  else
    echo "$0 $ARG: httpd could not be started"
    mail $EMAIL -s \
    "$0 $ARG: httpd could not be started" > /dev/null 2>&1

  fi
fi
```

Another approach, probably even more practical, is to use the cool `LWP` Perl package to test the server by trying to fetch some document (script) served by the server. Why is it more practical? Because while the server can be up as a process, it can be stuck and not working. Failing to get the document will trigger restart, and "probably" the problem will go away.

Like before we set a cronjob to call this script every few minutes to fetch some very light script. The best thing of course is to call it every minute. Why so often? If your server starts to spin and trash your disk space with multiple error messages filling the *error_log*, in five minutes you might run out of free disk space which might bring your system to its knees. Chances are that no other child will be able to serve requests, since the system will be too busy writing to the *error_log* file. Think big--if you are running a heavy service (which is very fast since you are running under mod_perl) adding one more request every minute will not be felt by the server at all.

So we end up with a crontab entry like this:

```
  * * * * * /path/to/the/watchdog.pl >/dev/null 2>&1
```

And the watchdog itself:

```
  #!/usr/bin/perl -wT
  
  # untaint
  $ENV{'PATH'} = '/bin:/usr/bin';
  delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};
  
  use strict;
  use diagnostics;
  use URI::URL;
  use LWP::MediaTypes qw(media_suffix);
  
  my $VERSION = '0.01';
  use vars qw($ua $proxy);
  $proxy = '';
  
  require LWP::UserAgent;
  use HTTP::Status;
  
  ###### Config ########
  my $test_script_url = 'http://www.example.com:81/perl/test.pl';
  my $monitor_email   = 'root@localhost';
  my $restart_command = '/usr/local/sbin/httpd_perl/apachectl restart';
  my $mail_program    = '/usr/lib/sendmail -t -n';
  ######################
  
  $ua  = new LWP::UserAgent;
  $ua->agent("$0/watchdog " . $ua->agent);
  # Uncomment the proxy if you access a machine from behind a firewall
  # $proxy = "http://www-proxy.com";
  $ua->proxy('http', $proxy) if $proxy;
  
  # If it returns '1' it means we are alive
  exit 1 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 is_success($res->code);

  # failed
  return 0;
} #  end of sub checkurl

# 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
From: $from
Subject: $subject

$messagebody

__END_OF_MAIL__

  close MAIL;
}
```

1.11 Running a Server in Single Process Mode

Often while developing new code, you will want to run the server in single process mode. See Sometimes it works Sometimes it does Not and Names collisions with Modules and libs. Running in single process mode inhibits the server from "daemonizing", and this allows you to run it under the control of a debugger more easily.

```
% /usr/local/sbin/httpd_perl/httpd_perl -X
```

When you use the `-X` switch the server will run in the foreground of the shell, so you can kill it with *Ctrl-C*.

Note that in `-X` (single-process) mode the server will run very slowly when fetching images.

Note for Netscape users:

If you use Netscape while your server is running in single-process mode, HTTP's `KeepAlive` feature gets in the way. Netscape tries to open multiple connections and keep them open. Because there is only one server process listening, each connection has to time out before the next succeeds. Turn off `KeepAlive` in *httpd.conf* to avoid this effect while developing. If you use the image size parameters, Netscape will be able to render the page without the images so you can press the browser's *STOP* button after a few seconds.

In addition you should know that when running with `-X` you will not see the control messages that the parent server normally writes to the *error_log* (*"server started"*, *"server stopped"* etc). Since `httpd -X` causes the server to handle all requests itself, without forking any children, there is no controlling parent to write the status messages.

1.12 Starting a Personal Server for Each Developer

If you are the only developer working on the specific server:port you have no problems, since you have complete control over the server. However, often you will have a group of developers who need to develop mod_perl scripts and modules concurrently. This means that each developer will want to have control over the server - to kill it, to run it in single server mode, to restart it, etc., as well as having control over the location of the log files, configuration settings like `MaxClients`, and so on.

You *can* work around this problem by preparing a few *httpd.conf* files and forcing each developer to use

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

but I approach it in a different way. I use the `-Dparameter` startup option of the server. I call my version of the server

```
% http_perl -Dstas
```

In *httpd.conf* I write:

```
# Personal development Server for stas
# stas uses the server running on port 8000
<IfDefine stas>
Port 8000
PidFile /usr/local/var/httpd_perl/run/httpd.pid.stas
ErrorLog /usr/local/var/httpd_perl/logs/error_log.stas
Timeout 300
KeepAlive On
MinSpareServers 2
MaxSpareServers 2
```

```
StartServers 1
MaxClients 3
MaxRequestsPerChild 15
</IfDefine>

# Personal development Server for userfoo
# userfoo uses the server running on port 8001
<IfDefine userfoo>
Port 8001
PidFile /usr/local/var/httpd_perl/run/httpd.pid.userfoo
ErrorLog /usr/local/var/httpd_perl/logs/error_log.userfoo
Timeout 300
KeepAlive Off
MinSpareServers 1
MaxSpareServers 2
StartServers 1
MaxClients 5
MaxRequestsPerChild 0
</IfDefine>
```

With this technique we have achieved full control over start/stop, number of children, a separate error log file, and port selection for each server. This saves Stas from getting called every few minutes by Eric: "Stas, I’m going to restart the server".

In the above technique, you need to discover the PID of your parent `httpd_perl` process, which is written in `/usr/local/var/httpd_perl/run/httpd.pid.stas` (and the same for the user *eric*). 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 we change two lines in each script:

```
PIDFILE=/usr/local/var/httpd_perl/run/httpd.pid.username
HTTPD='/usr/local/sbin/httpd_perl/httpd_perl -Dusername'
```

So for the user *stas* we prepare a startup script called *apachectl.stas* and we change these two lines in the standard apachectl script as it comes unmodified from Apache distribution.

```
PIDFILE=/usr/local/var/httpd_perl/run/httpd.pid.stas
HTTPD='/usr/local/sbin/httpd_perl/httpd_perl -Dstas'
```

So now when user *stas* wants to stop the server he will execute:

```
apachectl.stas stop
```

And to start:

```
apachectl.stas start
```

Certainly the rest of the `apachectl` arguments apply as before.

You might think about having only one `apachectl` and know who is calling by checking the UID, but since you have to be root to start the server it is not possible, unless you make the setuid bit on this script, as we’ve explained in the beginning of this chapter. If you do so, you can have a single `apachectl` script for all developers, after you modify it to automatically find out the UID of the user, who executes the script and set the right paths.

The last thing is to provide developers with an option to run in single process mode by:

```
/usr/local/sbin/httpd_perl/httpd_perl -Dstas -X
```

In addition to making life easier, we decided to use relative links everywhere in the static documents, including the calls to CGIs. You may ask how using relative links will get to the right server port. It's very simple, we use `mod_rewrite`.

To use mod_rewrite you have to configure your *httpd_docs* server with `--enable-module=rewrite` and recompile, or use DSO and load the module in *httpd.conf*. In the *httpd.conf* of our `httpd_docs` server we have the following code:

```
RewriteEngine on

# stas's server
# port = 8000
RewriteCond  %{REQUEST_URI} ^/(perl|cgi-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|cgi-perl)
RewriteCond  %{REMOTE_ADDR} 123.34.45.57
RewriteRule ^(.*)           http://example.com:8001/$1 [P,L]

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

The IP addresses are the addresses of the developer desktop machines (where they are running their web browsers). So if an html file includes a relative URI */perl/test.pl* or even *http://www.example.com/perl/test.pl*, clicking on the link will be internally proxied to http://www.example.com:8000/perl/test.pl if the click has been made at the user *stas*'s desktop machine, or to *http://www.example.com:8001/perl/test.pl* for a request generated from the user *eric*'s machine, per our above URI rewrite example.

Another possibility is to use `REMOTE_USER` variable if all the developers are forced to authenticate themselves before they can access the server. If you do, you will have to change the `RewriteRule`s to match `REMOTE_USER` in the above example.

We wish to stress again, that the above setup will work only with relative URIs in the HTML code. If you choose to generate full URIs including non-80 port the requests originated from this HTML code will bypass the light server listening to the default port 80, and go directly to the *server:port* of the full URI.

1.13 Wrapper to Emulate the Server Perl Environment

Often you will start off debugging your script by running it from your favorite shell program. Sometimes you encounter a very weird situation when the script runs from the shell but dies when processed as a CGI script by a web-server. The real problem often lies in the difference between the environment variables that is used by your web-server and the ones used by your shell program.

For example you may have a set of non-standard Perl directories, used for local Perl modules. You have to tell the Perl interpreter where these directories are. If you don't want to modify `@INC` in all scripts and modules, you can use a `PERL5LIB` environment variable, to tell Perl where the directories are. But then you might forget to alter the mod_perl startup script to correct `@INC` there as well. And if you forget this, you can be quite puzzled why the scripts are running from the shell program, but not from the web.

Of course the *error_log* will help as well to find out what the problem is, but there can be other obscure cases, where you do something different at the shell program and your scripts refuse to run under the web-server.

Another example is when you have more than one version of Perl installed. You might call the first version of the Perl executable in the first script's line (the shebang line), but to have the web-server compiled with another Perl version. Since mod_perl ignores the path to the Perl executable at the first line of the script, you can get quite confused the code won't do the same when processed as request, compared to be executed from the command line. it will take a while before you realize that you test the scripts from the shell program using the *wrong* Perl version.

The best debugging approach is to write a wrapper that emulates the exact environment of the server, first deleting environment variables like `PERL5LIB` and then calling the same perl binary that it is being used by the server. Next, set the environment identical to the server's by copying the Perl run directives from the server startup and configuration files or even *require()*'ing the startup file, if it doesn't include `Apache::` modules stuff, unavailable under shell. This will also allow you to remove completely the first line of the script, since mod_perl doesn't need it anyway and the wrapper knows how to call the script.

Here is an example of such a script. Note that we force the use of `-Tw` when we call the real script. Since when debugging we want to make sure that the code is working when the taint mode is on, and we want to see all the warnings, to help Perl help us have a better code.

We have also added the ability to pass parameters, which will not happen when you will issue a request to script, but it can be helpful at times.

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

# This is a wrapper example

# It simulates the web server environment by setting @INC and other
# stuff, so what will run under this wrapper will run under Web and
# vice versa.

#
# Usage: wrap.pl some_cgi.pl
#
BEGIN {
  # we want to make a complete emulation, so we must reset all the
  # paths and add the standard Perl libs
  @INC =
    qw(/usr/lib/perl5/5.00503/i386-linux
   /usr/lib/perl5/5.00503
   /usr/lib/perl5/site_perl/5.005/i386-linux
```

```
     /usr/lib/perl5/site_perl/5.005
     .
    );
  }

  use strict;
  use File::Basename;

    # process the passed params
  my $cgi = shift || '';
  my $params = (@ARGV) ? join(" ", @ARGV) : '';

  die "Usage:\n\t$0 some_cgi.pl\n" unless $cgi;

    # Set the environment
  my $PERL5LIB = join ":", @INC;

    # if the path includes the directory
    # we extract it and chdir there
  if (index($cgi,'/') >= 0) {
    my $dirname = dirname($cgi);
    chdir $dirname or die "Can't chdir to $dirname: $! \n";

    $cgi =~ m|$dirname/(.*)|;
    $cgi = $1;
  }

    # run the cgi from the script's directory
    # Note that we set Warning and Taint modes ON!!!
  system qq{/usr/bin/perl -I$PERL5LIB -Tw $cgi $params};
```

1.14 Server Maintenance Chores

It's not enough to have your server and service up and running. You have to maintain the server even when everything seems to be fine. This includes security auditing, keeping an eye on the size of remaining unused disk space, available RAM, the load of the system, etc.

If you forget about these chores one day (sooner or later) your system will crash either because it has run out of free disk space, all the available CPU has been used and system has started heavily to swap or someone has broken in. Unfortunately the scope of this guide is not covering the latter, since it will take more than one book to profoundly cover this issue, but the rest of the thing are quite easy to prevent if you follow our advices.

Certainly, your particular system might have maintenance chores that aren't covered here, but at least you will be alerted that these chores are real and should be taken care of.

1.14.1 Handling Log Files

There are two issues to solve with log files. First they should be rotated and compressed on the constant basis, since they tend to use big parts of the disk space over time. Second these should be monitored for possible sudden explosive growth rates, when something goes astray in your code running at the mod_perl server and the process starts to log thousands of error messages in second without stopping, until all the disk space is used, and the server cannot work anymore.

1.14.1.1 Log Rotation

The first issue is solved by having a process run by crontab at certain times (usually off hours, if this term is still valid in the Internet era) and rotate the logs. The log rotation includes the current log file renaming, server restart (which creates a fresh new log file), and renamed file compression and/or moving it on a different disk.

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

```
% mv access_log access_log.renamed
% apachectl restart
% sleep 5; # allow all children to complete requests and logging
           # now it’s safe to use access_log.renamed
% mv access_log.renamed /some/directory/on/another/disk
```

This is the script that we run from the crontab to rotate the log files:

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

# This script does log rotation. Called from crontab.

use strict;
$ENV{PATH}=’/bin:/usr/bin’;

### configuration
my @logfiles = qw(access_log error_log);
umask 0;
my $server = "httpd_perl";
my $logs_dir = "/usr/local/var/$server/logs";
my $restart_command = "/usr/local/sbin/$server/apachectl restart";
my $gzip_exec = "/usr/bin/gzip";

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;
$^I = ".$time";

# rename log files
chdir $logs_dir;
@ARGV = @logfiles;
while (<>) {
  close ARGV;
}

# 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 (@logfiles) {
      system "$gzip_exec $_.$time";
  }
```

Note: Setting `$^I` sets the in-place edit flag to a dot followed by the time. We copy the names of the logfiles into `@ARGV`, and open each in turn and immediately close them without doing any changes; but because the in-place edit flag is set they are effectively renamed.

As you see the rotated files will include the date and the time in their filenames.

Here is a more generic set of scripts for log rotation. Cron job fires off setuid script called log-roller that looks like this:

```
  #!/usr/bin/perl -Tw
  use strict;
  use File::Basename;

  $ENV{PATH} = "/usr/ucb:/bin:/usr/bin";

  my $ROOT = "/WWW/apache"; # names are relative to this
  my $CONF = "$ROOT/conf/httpd.conf"; # master conf
  my $MIDNIGHT = "MIDNIGHT";  # name of program in each logdir

  my ($user_id, $group_id, $pidfile); # will be set during parse of conf
  die "not running as root" if $>;

  chdir $ROOT or die "Cannot chdir $ROOT: $!";

  my %midnights;
  open CONF, "<$CONF" or die "Cannot open $CONF: $!";
  while (<CONF>) {
    if (/^User (\w+)/i) {
      $user_id = getpwnam($1);
      next;
    }
    if (/^Group (\w+)/i) {
      $group_id = getgrnam($1);
      next;
    }
    if (/^PidFile (.*)/i) {
      $pidfile = $1;
      next;
    }
   next unless /^ErrorLog (.*)/i;
    my $midnight = (dirname $1)."/$MIDNIGHT";
    next unless -x $midnight;
    $midnights{$midnight}++;
  }
  close CONF;

  die "missing User definition" unless defined $user_id;
  die "missing Group definition" unless defined $group_id;
  die "missing PidFile definition" unless defined $pidfile;
  open PID, $pidfile or die "Cannot open $pidfile: $!";
  <PID> =~ /(\d+)/;
  my $httpd_pid = $1;
  close PID;
  die "missing pid definition" unless defined $httpd_pid and $httpd_pid;
  kill 0, $httpd_pid or die "cannot find pid $httpd_pid: $!";


  for (sort keys %midnights) {
    defined(my $pid = fork) or die "cannot fork: $!";
    if ($pid) {
      ## parent:
      waitpid $pid, 0;
    } else {
      my $dir = dirname $_;
      ($(,$)) = ($group_id,$group_id);
      ($<,$>) = ($user_id,$user_id);
      chdir $dir or die "cannot chdir $dir: $!";
      exec "./$MIDNIGHT";
      die "cannot exec $MIDNIGHT: $!";
    }
  }

  kill 1, $httpd_pid or die "Cannot SIGHUP $httpd_pid: $!";
```

And then individual `MIDNIGHT` scripts can look like this:

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

  die "bad guy" unless getpwuid($<) =~ /^(root|nobody)$/;
  my @LOGFILES = qw(access_log error_log);
  umask 0;
  $^I = ".".time;
  @ARGV = @LOGFILES;
  while (<>) {
    close ARGV;
  }
```

Can you spot the security holes? Take your time... This code shouldn't be used in hostile situations.

1.14.1.2 Non-Scheduled Emergency Log Rotation

As we have mentioned before, there are times when the web server goes wild and starts to log lots of messages to the *error_log* file non-stop. If no one monitors this, it possible that in a few minutes all the free disk spaces will be filled and no process will be able to work normally. When this happens, the I/O the faulty server causes is so heavy that its sibling processes cannot serve requests.

Generally this not the case, but a few people have reported to encounter this problem. If you are one of these people, you should run the monitoring program that checks the log file size and if it notices that the file has grown too large, it should attempt to restart the server and probably trim the log file.

When we have used a quite old mod_perl version, sometimes we have had bursts of an error *Callback called exit* showing up in our *error_log*. The file could grow to 300 Mbytes in a few minutes.

We will show you is an example of the script that should be executed from the crontab, to handle the situations like this. The cron job should run every few minutes or even every minute, since if you experience this problem you know that log files fills up very fast. The example script will rotate when the *error_log* will grow over 100K. Note that this script is useful when you have the normal scheduled log rotation facility working, remember that this one is an emergency solver and not to be used for routine log rotation.

```
emergency_rotate.sh
-------------------
#!/bin/sh
S=‘ls -s /usr/local/apache/logs/error_log | awk ’{print $1}’‘
if [ "$S" -gt 100000 ] ; then
  mv /usr/local/apache/logs/error_log /usr/local/apache/logs/error_log.old
  /etc/rc.d/init.d/httpd restart
  date | /bin/mail -s "error_log $S kB on inx" admin@example.com
fi
```

Of course you could write a more advanced script, using the timestamps and other whistles. This example comes to illustrate how to solve the problem in question.

Another solution is to use an out of box tools that are written for this purpose. The `daemontools` package (ftp://koobera.math.uic.edu/www/daemontools.html) includes a utility called `multilog`. This utility saves 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 doesn’t restart the server, so while it tries to solve the log file handling problem it doesn’t handle the originator of the problem. But since the I/O of the log writing process Apache process will be quite heavy, the rest of the servers will work very slowly if at all, and a normal watchdog should detect this abnormal situation and restart the Apache server.

1.15 Swapping Prevention

Before I delve into swapping process details, let’s refresh our knowledge of memory components and memory management

The computer memory is called RAM, which stands for Random Access Memory. Reading and writing to RAM is, by a few orders, faster than doing the same operations on a hard disk, the former uses non-movable memory cells, while the latter uses rotating magnetic media.

On most operating systems swap memory is used as an extension for RAM and not as a duplication of it. So if your OS is one of those, if you have 128MB of RAM and 256MB swap partition, you have a total of 384MB of memory available. You should never count the extra memory when you decide on the maximum number of processes to be run, and I will show why in the moment.

The swapping memory can be built of a number of hard disk partitions and swap files formatted to be used as swap memory. When you need more swap memory you can always extend it on demand as long as you have some free disk space (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 1KB and 8KB. So if you have 256MB of RAM installed on your machine and the page size is 4KB your system has 64,000 main memory pages to work with and these pages are fast. If you have 256MB swap partition the system can use yet another 64,000 memory pages, but they are much slower.

When the system is started all memory pages are available for use by the programs (processes).

Unless the program is really small, 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 required to be loaded into the memory.

When the process needs an additional program's segment to be loaded into the memory, it asks the system whether the page containing this segment is already loaded in the memory. If the page is not found--an event know as a *page fault* occurs, which requires the system to allocate a free memory page, go to the disk, read and load the requested program's 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 an image or data file and has not been written to then 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 a *swapping out*.

Accesses to the swap file are very long relative to 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.

In order to improve the swapping out process, to decrease the possibility that the page that has just been swapped out, will be needed at the next moment, the LRU (least recently used) or a similar algorithm is used.

To summarize the two swapping scenarios, read-only pages discarding incurs no overhead in contrast with the discarding scenario of the data pages that have been written to, since in the latter case the pages have to be written to a swap partition located on the slow disk. Therefore your machine's overall performance will be much better if there will be less memory pages that can become dirty.

But the problem is, Perl is a language with no strong data types, which means that both the program code and the program data are seen as a data pages by OS since both mapped to the same memory pages. Therefore a big chunk of your Perl code becomes dirty when its variables are modified and when the 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.

- Running your system when there is no free main memory available hinders performance, because processes memory pages should be discarded and then reread from disk again and again.
- Since a majority of the running code is a Perl code, in addition to the overhead of reading the previously discarded pages in, the overhead of saving the dirty pages to the swap partition is occurring.

When the system has to swap memory pages in and out, the system slows down, not serving the processes as fast as before. This leads to an accumulation of processes waiting for their turn to run, which further causes processing demands to go up, which in turn slows down the system even more as more memory is required. This ever worsening spiral will lead the machine to halt, unless the resource demand suddenly drops down and allows the processes to catch up with their tasks and go back to normal memory usage.

In addition it's important to know that for a better performance, most programs, particularly programs written in Perl, on most modern OSs don't return memory pages while they are running. If some of the memory gets freed it's reused when needed by the process, without creating the additional overhead of asking the system to allocate new memory pages. That's why you will observe that Perl programs grow in size as they run and almost never shrink.

When the process quits it returns its memory pages to the pool of freely available pages for other processes to use.

This scenario is certainly educating, and it should be now obvious that your system that runs the web server should never swap. It's absolutely normal for your desktop to start swapping. You will see it immediately since things will slow down and sometimes the system will freeze for a short periods. But as I've just mentioned, you can stop starting new programs and can quit some, thus allowing the system to catch up with the load and come back to use the RAM.

In the case of the web server you have much less control since it's users who load your machine by issuing requests to your server. Therefore you should configure the server, so that the maximum number of possible processes will be small enough using the `MaxClients` directive (For the technique for choosing the right `MaxClients` refer to the section 'Choosing MaxClients'). This will ensure that at peak hours the system won't swap. Remember that swap space is an emergency pool, not a resource to be used routinely. If you are low on memory and you badly need it, buy it or reduce the number of processes to prevent swapping.

However sometimes, due to the faulty code, some process might start spinning in an unconstrained loop, consuming all the available RAM and starting to heavily use swap memory. In such a situation it helps when you have a big emergency pool (i.e. lots of swap memory). But you have to resolve this problem as soon as possible since this pool won't last for a long time. In the meanwhile the `Apache::Resource` module can be handy.

For swapping monitoring techniques see the section 'Apache::VMonitor -- Visual System and Apache Server Monitor'.

1.16 Preventing mod_perl Processes From Going Wild

Sometimes people report that they had a problem with their code running under mod_perl that has caused all the RAM or all the disk to be used. The following tips should help you prevent these problems, before if at all they hit you.

1.16.1 All RAM Consumed

Sometimes calling an undefined subroutine in a module can cause a tight loop that consumes all the available memory. Here is a way to catch such errors. Define an `UNIVERSAL::AUTOLOAD` subroutine in your *startup.pl*, or in a <Perl></Perl> section in your *httpd.conf* file:

```
sub UNIVERSAL::AUTOLOAD {
  my $class = shift;
  warn "$class can't \$UNIVERSAL::AUTOLOAD=$UNIVERSAL::AUTOLOAD!\n";
}
```

You can either put it in your startup.pl, or in a `<Perl></Perl>` section in your httpd.conf file. I do the latter. Putting it in all your mod_perl modules would be redundant (and might give you compile-time errors).

This will produce a nice error in *error_log*, giving the line number of the call and the name of the undefined subroutine.

1.17 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman <stas (at) stason.org>

1.18 Authors

- Stas Bekman <stas (at) stason.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents:

1 Controlling and Monitoring the Server 1
1.1 Description 2
1.2 Restarting Techniques 2
1.3 Server Stopping and Restarting 3
1.4 Speeding up the Apache Termination and Restart 4
1.5 Using apachectl to Control the Server 4
1.6 Safe Code Updates on a Live Production Server 5
1.7 An Intentional Disabling of Live Scripts 7
1.8 SUID Start-up Scripts 9
1.8.1 Introduction to SUID Executables 9
1.8.2 Apache Startup SUID Script's Security 9
1.8.3 Sample Apache Startup SUID Script 10
1.9 Preparing for Machine Reboot 11
1.10 Monitoring the Server. A watchdog. 14
1.11 Running a Server in Single Process Mode 17
1.12 Starting a Personal Server for Each Developer 18
1.13 Wrapper to Emulate the Server Perl Environment 20
1.14 Server Maintenance Chores 22
1.14.1 Handling Log Files 22
1.14.1.1 Log Rotation 23
1.14.1.2 Non-Scheduled Emergency Log Rotation 25
1.15 Swapping Prevention 26
1.16 Preventing mod_perl Processes From Going Wild 29
1.16.1 All RAM Consumed 29
1.17 Maintainers 29
1.18 Authors 29