

The Apache Conference
August 22, 1999
Monterey, California

***Tutorial:
Getting Started with mod_perl***

***By Stas Bekman
Internet and Intranet programmer
<http://singleheaven.com/stas/>
<sbekman@iname.com>***

This document is originally written in **POD**, converted to **HTML** by **pod2html** utility and then to **PostScript** by **html2ps** utility.

Copyright © 1998, 1999 Stas Bekman. All rights reserved.

1 Tutorial's Agenda

1.1 Tutorial's Agenda

- A short overview of `mod_perl`.
- We start with **`mod_perl` Coding Guidelines**.
 - Explain what are the differences between scripts running under `mod_cgi` and `mod_perl`
 - Go through the changes that should be applied in order to make the existing scripts run under `mod_perl`
 - Provide guidelines for a proper `mod_perl` programming
- Then we will proceed to the biggest and the most important part -- improving performance.

- Explain the details of the mod_perl-enabled server performance tuning
- Suggest how to improve the performance of the scripts,
- Present some utilities, helping to benchmark and then make a fine tune of the server.
- Picking the right strategy. I'll present a few widely used mod_perl deployments scenarios, discuss the pros and cons of each one.
- Real World Scenarios Implementation. Will complete the presented scenarios with detailed building, installation and configuration details.

- More advanced installation techniques will be covered afterwards.
- Then I'll talk about advanced extended configurations and present various configuration examples.
- Once you have a running server, you will want:
 - To start and stop it,
 - Prevent server's failures with help of watchdogs,
 - Learn how to run a personal webserver for each developer.
 - and more

I'll show you how

- Databases? We will learn how to turn your database connections to persistent.
- Running an ISP business and planning to extend your services, by providing mod_perl services -- a fantasy or reality.
- Finally, I'll give you a list of related information resources, like learning perl programming and SQL, understanding security, building databases and more.

And the most important -- how to get helped if you are in trouble and need help.

1.2 Q & A and Breaks

Every 45 minutes or so I'll stop for a short Q&A session followed by a 5 minute break.

1.3 What prior knowledge is required.

It is assumed that you know at least a basic perl and installed once the apache webserver.

But if you don't -- you will still be able to understand 99% of the presentation :)

;o)

2 mod_perl Technology Overview

2.1 What is mod_perl

The Apache/Perl integration project brings together the full power of the Perl programming language and the Apache HTTP server.

With mod_perl it is possible to write Apache modules entirely in Perl, this lets you easily do things that are more difficult or impossible in regular CGI programs, such as running sub requests for example.

In addition, the persistent interpreter embedded in the server saves the overhead of starting an external perl interpreter, the penalty of Perl start-up time.

A not least important feature is code caching, the modules and scripts are being loaded and compiled only once, then for the rest of the server's life the scripts are being served from the cache,

thus server spends its time only to run the already loaded and compiled code, which is very fast.

The primary advantages of mod_perl are power and speed.

You have full access to the inner-workings of the web server and can intervene at any stage of request-processing.

This allows for customized processing of (to name just a few of the phases) URI to filename translation, authentication, response generation and logging.

There is very little run-time overhead. In particular, it is not necessary to start a separate process, as is often done with web-server extensions.

The most wide-spread such extension mechanism, the Common Gateway Interface (CGI), can be replaced entirely with perl-code that handles the response generation phase of request processing.

Mod_perl includes 2 general purpose modules for this purpose: **Apache::Registry**, which can transparently run existing perl CGI scripts and **Apache::PerlRun**, which does a similar job but allows you to run “dirtier” (to some extent) scripts.

You can configure your httpd server and handlers in Perl. You can even define your own configuration directives.

Many people wonder and ask “How much of a performance improvement does mod_perl give?”.

Well, it all depends on what you are doing with mod_perl and possibly who you ask.

Developers report speed boosts from 200% to 2000%. The best way to measure is to try it and see for yourself! (see <http://perl.apache.org/tidbits.html> and <http://perl.apache.org/stories/> for the facts)

;o)

3 mod_perl Coding Guidelines

3.1 Exposing Apache::Registry secrets

Let's start with some simple code

- and see what can go wrong with it,
- detect bugs and debug them,
- discuss possible caveats and how to avoid them.

I will use a simple CGI script, that initializes a **\$counter** to 0, and prints its value to the screen while incrementing it.

```
counter.pl:
-----
#!/usr/bin/perl -w
use strict;
print "Content-type: text/html\r\n\r\n";
my $counter = 0;
for (1..5) {
    increment_counter();
}
sub increment_counter{
    $counter++;
    print "Counter is equal to $counter !<BR>\n";
}
-----
```

You would expect to see an output:

```
Counter is equal to 1 !  
Counter is equal to 2 !  
Counter is equal to 3 !  
Counter is equal to 4 !  
Counter is equal to 5 !
```

And that's what you see when you execute this script at first time.

But let's reload it a few times... See, suddenly after a few reloads the counter doesn't start its count from 5 anymore.

We continue to reload and see that it keeps on growing, but not steadily 10, 10, 10, 15, 20... Weird...

```
Counter is equal to 6 !  
Counter is equal to 7 !  
Counter is equal to 8 !  
Counter is equal to 9 !  
Counter is equal to 10 !
```

We saw two anomalies in this very simple script:

- Unexpected growth of counter over 5
- Inconsistent growth over reloads

Let's investigate this script.

First let's peek into an **error_log** file... what we see is:

Variable "\$counter" will not stay shared
at /home/httpd/perl/conference/counter.pl line 13.

What kind of error is this? We should ask perl to help us.

```
use diagnostics;
```

Reloading again... **error_log** now includes an explanation of the warning we just saw.

(W) An inner (nested) named subroutine is referencing a lexical variable defined in an outer subroutine.

[more explanations snipped]

Actually, perl detected a **closure**, which is sometimes a wanted effect, but not in our case.

While **diagnostics.pm** sometimes is handy for debugging purpose - it drastically slows down your CGI script. Make sure you remove it in your production server.

Do you see a **nested named subroutine** in my script? I do not!!!

May be a perl interpreter sees the script in a different way, may be the code goes through some changes before it actually gets executed?

The easiest way to check what's actually happening is to run the script with debugger.

```
PerlsetEnv PERLDB_OPTS "Nonstop=1 LineInfo=/tmp/db.out
    AutoTrace=1 frame=2"
PerlModule Apache::DB
<Location /perl>
    PerlFixupHandler Apache::DB
    SetHandler perl-script
    PerlHandler Apache::Registry::handler
    Options ExecCGI
    PerlSendHeader On
</Location>
```

Restarting the server and running the script again does 2 things:

1. the **/tmp/db.out** was written, with a complete trace of the code that was executed,

2. **error_log** file showed us the whole code that was executed as a side effect of reporting the warning we saw before.

```
package Apache::ROOT::perl::conference::counter_2ep1;
use Apache qw(exit);
sub handler {
    BEGIN {$^W = 1;}; $^W = 1;

    use strict;
    print "Content-type: text/html\r\n\r\n";
    my $counter = 0;
    for (1..5) {
        increment_counter();
    }
    sub increment_counter{
        $count++;
        print "Counter is equal to $counter !<BR>\n";
    }
}
```

What do we learn from this discovering?

- every cgi script is being cached under a package whose name is compounded from **Apache::ROOT::** prefix and the relative part of the script's URL (**perl::conference::counter_2epl**) by replacing all occurrences of / with ::.
- That's how mod_perl knows what script should be fetched from cache - each script is just a package with a single subroutine named **handler()**.
- Now you understand why **diagnostics** pragma talked about inner (nested) subroutine - **increment_counter()** is actually a nested sub. In every script each subroutine is nested inside the **handler()** subroutine.

Solving the problem.

The workaround is to use global declared variables, with **vars** pragma.

```
# !/usr/bin/perl -w
use strict;
use vars qw($counter);
print "Content-type: text/html\r\n\r\n";
$counter = 0;
for (1..5) {
    increment_counter();
}
sub increment_counter{
    $counter++;
    print "Counter is equal to $counter !<BR>\n";
}
```

There is no more **closure** effect, since there is no **my()** (lexically) defined variable being used in the nested subroutine.

Another approach is to use fully qualified variables, which is even better, since less memory will be used, but it adds an overhead of extra typing:

```
#!/usr/bin/perl -w
use strict;
print "Content-type: text/html\r\n\r\n";
$main::counter = 0;
for (1..5) {
    increment_counter();
}
sub increment_counter{
    $main::counter++;
    print "Counter is equal to $main::counter !<BR>\n";
}
```

Now let's proceed to the second mystery. Why did we see inconsistent results over numerous reloads.

That's very simple. Every time a server gets a request to process, it handles it over one of the children, generally in a round robin fashion.

So if you have 10 httpd children alive, first 10 reloads might seem to be correct.

Since the closure starts to effect from the second re-invocation, consequent reloads return unexpected results.

Moreover children don't serve the same request always consequently, at any given moment one of the children could serve more times the same script than any other.

That's why we saw that strange behavior.

Enter the magic **single server mode**. It helps to immediately isolate possible problems.

httpd -X invokes apache in a single mode.

We detect our problem on the second reload (since there is no other servers (children) running)

Don't ignore warnings, like in our case -- these are potential errors!

You better make sure to check every warning that is being detected by perl, and correct the code in a way, that none of the warnings will show up in the **error_log**.

If your **error_log** file is being filled up with hundreds of lines on every script invocation - you will have a problem to locate and notice real problems.

Of course none of the warnings will be reported if the warning mechanism will not be turned ON.

With mod_perl it is also possible to turn on warnings globally via the PerlWarn directive, just add into a **httpd.conf**:

PerlWarn On

You can turn it off within your code with **local \$^W=0**. on the local basis (or inside the block).

Note that if you write **\$^W=0** -- you disable the warning mode everywhere inside the child, **\$^W=1** enables it back. So if perl warns you somewhere you sure it's not a problem, you can locally disable the warning, e.g.:

```
[snip]
    # we want perl to be quiet here -
    # we don't care whether $a was initialized
    local $^W = 0;
    print $a;
    local $^W = 1;
[snip]
```

or even better using a block:

```
{  
    local $^W = 0;  
    print $a;  
}
```

Of course this is not a way to fix initialization and other problems, but sometimes it helps.

While having a warning mode turned **On** is a must in a development server, you better turn it globally **Off** in a production server:

PerlWarn Off

3.2 Sometimes it Works Sometimes it Does Not

Generally the problem you have -- is of using global variables.

Since global variables don't change from one script invocation to another unless you change them, you can find your scripts do "fancy" things.

The first example is amazing -- Web Services. Imagine that you enter some site you have your account on (Free Email Account?). Now you want to see what other users read.

You type in a username you want to peek at and a dummy password and try to enter the account. On some services it does works!!

You say, why in the world does this happen? The answer is simple: **Global Variables**.

You have entered the account of someone who happened to be served by the same server child as you.

Because of sloppy programming, a global variable was not reset at the beginning of the program and voila, you can easily peek into other people's emails!

```
use vars ($authenticated);
my $q = new CGI;
my $username = $q->param('username');
my $passwd = $q->param('passwd');
authenticate($username, $passwd);
# failed, break out
die "Wrong passwd" unless $authenticated == 1;
# user is OK, fetch user's data
show_user($username);

sub authenticate{
    my ($username, $passwd) = @_;
    # some checking
    $authenticated = 1 if (SOMETHING);
}
```

Since **\$authenticated** is global - if it becomes **1** once it'll be **1** for the remainder of the child's life!!!

The solution is trivial -- reset **\$authenticated** to 0 at the beginning of the program.

Just another little one-liner that can spoil your day, assuming you forgot to reset the **\$allowed** variable. It works perfectly OK in plain mod_cgi:

```
$allowed = 1 if $username eq 'admin' ;
```

But you will let any user to admin your system with the line above.

Another good example is usage of the `/o` regular expression qualifier, which compiles a regular expression once, on its first execution and never recompiles it again.

This problem can be difficult to detect, as after restarting the server each request you make will be served by a different child process, and thus the regex pattern for that child will be compiled fresh.

Only when you make a request that happens to be served by a child which has already cached the regexp will you see the problem.

Generally you miss that and when you press reload, you see that it works (with a new, fresh child) and then it doesn't (with a child that already cached the regexp and wouldn't recompile because of `/o`.)

The example of such a case would be:

```
my $pat = $q->param( "keyword" );  
foreach( @list ) {  
    print if /$pat/o;  
}
```

To make sure you catch these bugs, always test your CGI scripts under a server running in a single mode.

I'll talk in more extension about Compiled Regular Expressions at the end of this section.

3.3 What's different about modperl

There are a few things that behave differently under mod_perl. It's good to know what they are.

3.3.1 *Script's name space*

Scripts under **Apache::Registry** do not run in package **main**, they run in a unique name space based on the requested URI.

For example, if your URI is **/perl/test.pl** the package will be called **Apache::ROOT::perl::test_2epl**.

3.3.2 *Name collisions with Modules and libs*

To make things clear before we go into details: each child process has its own **%INC** hash which is used to store information about its compiled modules.

The keys of the hash are the names of the modules or parameters passed to **require()** (**use()**). The values are the full or relative paths to these modules/files. Let's say we have **my-lib.pl** and **MyModule.pm** both located at **/home/httpd/perl/my/**.

- **/home/httpd/perl/my/** is in the **@INC** path at the server startup.

```
require "my-lib.pl";  
use MyModule.pm;  
print $INC{"my-lib.pl"}, "\n";  
print $INC{"MyModule.pm"}, "\n";
```

prints:

```
/home/httpd/perl/my/my-lib.pl  
/home/httpd/perl/my/MyModule.pm
```

Adding use lib:

```
use lib qw(.);  
require "my-lib.pl";  
use MyModule.pm;  
print $INC{"my-lib.pl"}, "\n";  
print $INC{"MyModule.pm"}, "\n";
```

prints:

```
my-lib.pl  
MyModule.pm
```

- **/home/httpd/perl/my/** isn't in the **@INC** path at the server startup.

```
require "my-lib.pl";  
use MyModule.pm;  
print $INC{"my-lib.pl"}, "\n";  
print $INC{"MyModule.pm"}, "\n";
```

Wouldn't work, since perl cannot find the modules.

Adding use lib:

```
use lib qw(.);  
require "my-lib.pl";  
use MyModule.pm;  
print $INC{"my-lib.pl"}, "\n";  
print $INC{"MyModule.pm"}, "\n";
```

prints:

```
my-lib.pl  
MyModule.pm
```

I'm talking about single server child below!

Let's look at 3 faulty script's name space related scenarios:

Scenario 1

You can't have 2 identical module names running under the same server!

Only the first one **use()**d or **require()**d will be compiled into the package, the request to the other identical module will be skipped since server will think that it's already compiled.

It's already in the child's **%INC**.

(Having **/perl-status** mode enabled will allow you to find out what is loaded and where)

So if you have two different **Foo** modules in two different directories and two scripts **script1.pl** and **script2.pl**, placed like:

```
./perl/tool1/Foo.pm
./perl/tool1/tool1.pl
./perl/tool2/Foo.pm
./perl/tool2/tool2.pl
```

Where a sample code could be:

```
./perl/tool1/tool1.pl
-----
use Foo;
print "Content-type: text/html\n\n";
print "I'm script number One<BR>\n";
foo();
-----
```

```
./perl/tool1/Foo.pm
```

```
-----
```

```
sub foo{  
    print "<B>I'm Tool Number One!</B><BR>\n";  
}  
1;  
-----
```

```
./perl/tool2/tool2.pl
```

```
-----
```

```
use Foo;  
print "Content-type: text/html\n\n";  
print "I'm script number Two<BR>\n";  
foo();  
-----
```

```
./perl/tool2/Foo.pm
```

```
-----
```

```
sub foo{  
    print "<B>I'm Tool Number Two!</B><BR>\n";  
}  
1;  
-----
```

And both scripts call: **use Foo**; -- only the first one called will know about **Foo**.

Run the server in a single server mode to detect this kind of bug immediately.

You will see the following in the error_log file:

Undefined subroutine

**&Apache::ROOT::perl::tool2::tool2_2ep1::foo called
at /home/httpd/perl/tool2/tool2.pl line 4.**

Scenario 2

The above is true for the files you **require()** as well (assuming that the required files do not declare a package). If you have:

```
./perl/tool1/config.pl  
./perl/tool1/tool1.pl  
./perl/tool2/config.pl  
./perl/tool2/tool2.pl
```

And both scripts do:

```
use lib qw(.);  
require "config.pl";
```

The second scenario is not different from the first one, since there is no difference between **use()** and **require()** if you don't have to import some symbols into a calling script.

Scenario 3

What's interesting that the following scenario wouldn't work too!

```
./perl/tool/config.pl  
./perl/tool/tool1.pl  
./perl/tool/tool2.pl
```

where **tool1.pl** and **tool2.pl** both **require()** the **same config.pl**.

There are 3 solutions for that:

Solution 1

The first two faulty scenarios can be solved by placing your library modules in a subdirectory structure so that they have different path prefixes.

The file system layout will be something like:

- `./perl/tool1/Tool1/Foo.pm`
- `./perl/tool1/tool1.pl`
- `./perl/tool2/Tool2/Foo.pm`
- `./perl/tool2/tool2.pl`

And modify the scripts:

```
use Tool1::Foo;  
use Tool2::Foo;
```

For **require()** (scenario number 2) use the following:

```
./perl/tool1/tool1-lib/config.pl  
./perl/tool1/tool1.pl  
./perl/tool2/tool2-lib/config.pl  
./perl/tool2/tool2.pl
```

And each script does respectively:

```
use lib qw(.);  
require "tool1-lib/config.pl";  
  
use lib qw(.);  
require "tool2-lib/config.pl";
```

But this solution isn't good, since while it might work for you now, if you add another script that wants to use the same module or **config.pl** file, it still wouldn't work as we saw in the third scenario.

So let's see better solutions.

Solution 2

Another option is to use a full path to the script, so it'll be compiled into the name of the key in the **%INC**;

```
require "/full/path/to/the/config.pl";
```

This solution solves the first two scenarios. I was surprised but it worked for the third scenario as well!

But with this solution you lose portability! (If you move the tool around in the file system you will have to change the base dir)

Solution 3

Declare a package in the required files! (It should be unique to the rest of the package names you use!)

The **%INC** will use the package name for the key!

It's a good idea to build at least 2 level package names for your private modules. e.g. **MyProject::Carp** and not **Carp**, since it will collide with existent standard package.

Even if as of the time of your coding it doesn't exist yet - it might enter the next perl distribution as a standard module and your code will become broken.

What are the implications of package declaration?

Before:

- All the (global and lexical) variables and subroutines were part of the **main::** package, so any of them could be used as if they were part of the main script

After:

- Now you should use **Package::function()** method to call a subroutine from a package **Package** and **\$Package::some_variable**.
- You will be unable to access lexically defined variables inside **Package** (declared with **my()**).
- You still can leave your scripts unchanged if you import the names of the global variables and subs into a **main::** package's namespace, like:

```
use Module qw( :mysubs sub_b $var1 :myvars );
```

You can export both -- subroutines and global variables. This is not a good approach since it'll consume more memory for the current process.

(See **perldoc Exporter** for information about exporting variables)

This solution completely covers the third scenario. By using different module names in package declarations, as explained above you solve the first two as well.

See also **perlmodlib** and **perlmod** manpages.

From the above discussion it should be clear that you cannot run a development and a production versions of the tools using the same apache server!

You have to run a separate server for each (it still can be the same machine, but the server will use a different port).

3.3.3 `__END__` or `__DATA__` tokens

Apache::Registry scripts cannot contain `__END__` or `__DATA__` tokens.

3.3.4 *Output from system calls*

Output of `system()`, `exec()`, and `open(PIPE, "lprogram")` calls will not be sent to the browser unless your Perl was configured with **sfio**.

3.3.5 *Using format()*

Currently possible only if you have perl compiled with **sfio**.

3.3.6 *Using exit()*

Perl's `exit()` built-in function cannot be used in `mod_perl` scripts.

Calling it causes the server child to exit (which makes the whole idea of using `mod_perl` irrelevant.)

The **`Apache::exit()`** function should be used instead.

You might start your scripts by overriding the `exit` sub (if you use **`Apache::exit()`** directly, you will have a problem testing the script from the shell, unless you stuff **`use Apache ();`** into your code.) I use the following code:

```

BEGIN {
    $USE_MOD_PERL = ( (exists $ENV{'GATEWAY_INTERFACE'} }
        and $ENV{'GATEWAY_INTERFACE'} =~ /CGI-Perl/)
        or exists $ENV{'MOD_PERL'} ) ? 1 : 0;
}
use subs (exit);
# select the correct exit way
sub exit{
    # Apache::exit(-2) will cause the server to exit
    # gracefully, once logging happens and protocol,
    # etc (-2 == Apache::Constants::DONE)
    $USE_MOD_PERL ? Apache::exit(0) : CORE::exit(0);
}

```

Now the correct **exit()** will be always chosen, whether you run the script as a CGI or from the shell.

Note that if you run the script under **Apache::Registry**, **The Apache function exit()** overrides the **Perl core built-in function**.

Note that if you still use **CORE::exit()** in your scripts running under modperl, the child will exit, but neither proper exit nor logging will happen on the way. **CORE::exit()** cuts off the server's legs...

If you need to properly shutdown the child , use **\$r->child_terminate**.

3.3.7 *Running from shell*

Your scripts **will not** run from the command line (yet) unless you use **CGI::Switch** or **CGI.pm** and perl 5.004+ and do not make any direct calls to **Apache->methods**.

3.3.8 I/O is different

If you are using Perl 5.004 or better, most CGI scripts can run under `mod_perl` untouched. If you're using 5.003, Perl's built-in `read()` and `print()` functions do not work as they do under CGI. If you're using **CGI.pm**, use `$query->print` instead of plain `print()`.

3.3.9 HTTP + MIME Headers (*PerlSendHeader*)

By default, `mod_perl` does not send any headers by itself, however, you may wish to change this (in `httpd.conf`):

`PerlSendHeader On`

Now the response line and common headers will be sent as they are by `mod_cgi`. And, just as with `mod_cgi`, **PerlSendHeader** will not send the MIME type and a terminating newline.

Your script must send that itself, e.g.:

```
print "Content-type: text/html\r\n\r\n";
```

If you are using CGI.pm or CGI::Switch and print \$q->header you do `_not_` need PerlSendHeader On.

3.3.10 NPH (Non-parsed Headers) scripts

To run a NPH CGI script under `mod_perl`, simply add to your code:

```
local $| = 1;
```

And if you normally set **PerlSendHeader On**, add this to your **httpd.conf**:

```
<Files */nph-*>  
    PerlSendHeader Off  
</Files>
```

3.3.11 *BEGIN* blocks

Perl executes **BEGIN** blocks during the compile time of code as soon as possible. The same is true under `mod_perl`.

As **perlmod manpage** explains, once a **BEGIN** has run, it is immediately undefined.

BEGIN blocks in modules and files pulled in via **require()** or **use()** will be executed:

- Only once, if pulled in by the parent process.
- Once per-child process if not pulled in by the parent process.

- An additional time, once per-child process if the module is pulled in off a disk again via **Apache::StatINC**.
- An additional time, in the parent process on each restart if **PerIFreshRestart** is **On**.
- Unpredictable if you fiddle with **%INC** yourself.

BEGIN blocks in **Apache::Registry** scripts will be executed, as above plus:

- Only once, if pulled in by the parent process via **Apache::RegistryLoader** - once per-child process if not pulled in by the parent process.
- An additional time, once per-child process if the script file has changed on disk.

- An additional time, in the parent process on each restart if pulled in by the parent process via **Apache::RegistryLoader** and **PerlFreshRestart** is **On**.

3.3.12 *END* blocks

As `perlmod` explains, an **END** subroutine is executed as late as possible, that is, when the interpreter exits.

In the `mod_perl` environment, the interpreter does not exit until the server shutdown.

Any **END** blocks encountered during main server startup, i.e. those pulled in by the **PerlRequire** or by any **PerlModule**, are suspended and run at server shutdown, aka **child_exit()** (requires `apache 1.3b3+`).

However, `mod_perl` does make a special case for **Apache::Registry** scripts.

Any **END** blocks that are encountered during compilation of **Apache::Registry** scripts are called **after the script has completed** (not during the cleanup phase though) including subsequent invocations when the script is cached in memory.

All other **END** blocks encountered during other **Perl*Handler** callbacks, e.g. **PerlChildInithandler**, will be suspended while the process is running and called during **child_exit()** when the process is shutting down.

3.3.13 *strict pragma*

It's `_absolutely_` mandatory (at least for development) to start all your scripts with:

```
use strict;
```

If needed, you can always turn off the 'strict' pragma or a part of it inside the block, e.g:

```
{  
    no strict 'refs';  
    ... some code  
}
```

It's more important to have **strict** pragma enabled under mod_perl than anywhere else.

While it's not required, it is strongly recommended, it will save you more time in the long run.

And, of course, clean scripts will still run under mod_cgi (plain CGI)!

3.3.14 Turning warnings ON

Have a **local \$^W=1** in the script or **PerlWarn ON** at the server configuration file.

Turning the warning on will save you a lot of troubles with debugging your code.

Note that all perl switches, but **-w** in the first magic (shebang) line of the script **#!/perl -switches** are being ignored by `mod_perl`.

If you write **-T** you will be warned to set **PerlTaintCheck ON** in the config file.

If you need **--** you can always turn off the warnings with **local \$^W=0** in your code if you have some section you don't want the perl compiler to warn in. The correct way to do this is:

```
{  
  local $^W=0;  
  # some code  
}
```

It preserves the previous value of **`$^W`** when you quit the block (so if it was set before, it will return to be set at the leaving of the block).

In production code, it can be a good idea to turn warnings off:

- If your code isn't very clean and spits a few lines of warnings here and there, you will end up with a huge **`error_log`** file in a short time on the heavily loaded server.
- Enabling runtime warning checking has a small performance impact -- in any script, not just under `mod_perl` -- so your approach should be to enable warnings during development,

and then disable them when your code is production-ready.

Controlling the warnings mode through the **httpd.conf** is much better, since you can control the behavior of all of the scripts from a central place.

I have **PerlWarn On** on my development server and **PerlWarn Off** on the production machine.

diagnostics pragma can shed more light on the errors and warnings you see, but again, it's better not to use it in production, since otherwise you incur a huge overhead of the diagnostics pragma examining every bit of the code `mod_perl` executes.

(You can run your script with **-dDprof** to check the overhead. See **Devel::Dprof** for more info).

3.3.15 *Passing ENV variables to CGI*

To pass an environment variable from a configuration file, add to it:

```
PerlSetEnv key val  
PerlPassEnv key
```

e.g.:

```
PerlSetEnv PERLDB_OPTS "LineInfo=/tmp/out AutoTrace=1"
```

will set **`$ENV{PERLDB_OPTS}`**, and it'll be accessible in every child.

3.3.16 *Global Variables*

It's always a good idea to stay away from global variables when possible.

Some variables must be global so Perl can see them, such as a module's **@ISA** or **\$VERSION** variables (or fully qualified **@MyModule::ISA**).

In common practice, a combination of **strict** and **vars** pragmas keeps modules clean and reduces a bit of noise. However, **vars** pragma also creates aliases as the **Exporter** does, which eat up more memory. When possible, try to use fully qualified names instead of use vars. Example:

```
package MyPackage;
use strict;
@MyPackage::ISA = qw(...);
$MyPackage::VERSION = "1.00";
```

VS.

```
package MyPackage;
use strict;
use vars qw(@ISA $VERSION);
@ISA = qw(...);
$VERSION = "1.00";
```

3.3.17 *Memory leakage*

Scripts under `mod_perl` can very easily leak memory! Global variables stay around indefinitely, lexical variables (declared with `my()`) are destroyed when they go out of scope, provided there are no references to them from outside of that scope.

Perl doesn't return the memory it acquired from the kernel. It does reuse it though!

First example demonstrates reading in a whole file:

```
open IN, $file or die $!;  
$/ = undef;  
$content = <IN>;  
close IN;
```

If your file is 5Mb, the child who served that script will grow exactly by that size.

Now if you have 20 children and all of them will serve this CGI, all of them will consume additional $20 * 5M = 100M$ of RAM!

If that's the case, try to use other approaches of processing the file, if possible of course.

Try to process a line at a time and print it back to the file. (If you need to modify the file itself, use a temporary file. When finished, overwrite the source file, make sure to provide a locking mechanism!)

Second example demonstrates copying variables between functions (passing variables by value).

Let's use the example above, assuming we have no choice but to read the whole file before any data processing takes place.

Now you have some imagine **process()** subroutine that processes the data and returns it back.

What happens if you pass the **\$content** by value? You have just copied another 5M and the child has grown by another 5M in size (watch your swap space!) now multiply it again by factor of 20 you have 200M of wasted RAM, which will be apparently reused but it's a waste!

Whenever you think the variable can grow bigger than few Kb, pass it by reference!

It's better to plan ahead and pass the variables by reference, if a variable you are going to pass might be bigger than you think at the time of your coding process.

There are a few approaches you can use to pass and use variables passed by reference.

For example:

```
my $content = qq{foobarfoobar};
process(\$content);
sub process{
    my $r_var = shift;
    $$r_var =~ s/foo/bar/g;
    # nothing returned - the variable $content
    # outside has been already modified
}

@{$var_lr} -- dereferences an array
%{$var_hr} -- dereferences a hash
```

For more info see **perldoc perlref**.

Another approach would be to directly use a `@_array`. Using directly the `@_array` serves the job of passing by reference!

```
process($content);  
sub process{  
    $_[0] =~ s/foo/bar/g;  
}
```

From **perldoc perlsub**:

The array `@_` is a local array, but its elements are aliases for the actual scalar parameters. In particular, if an element `$_[0]` is updated, the corresponding argument is updated (or an error occurs if it is not possible to update)...

Be careful when you write this kind of subroutines, since it can confuse a potential user. It's not obvious that call like **process(\$content)**; modifies the passed variable -- programmers

(which are the users of your library in this case) are used to subs that either modify variables passed by reference or return the processed variable (e.g. **\$content=process(\$content);**).

Third example demonstrates a work with DataBases.

If you do some DB processing, many times you encounter the need to read lots of records into your program, and then print them to the browser after they are formatted.

(I don't even mention the horrible case where programmers read in the whole DB and then use perl to process it!!! Use a relational DB and let the SQL do the job, so you get only the records you need!!!).

We will use **DBI** for this (assume that we are already connected to the DB).

(refer to **perldoc DBI** for a complete manual of the **DBI** module):

```
$sth->execute;
while(@row_ary = $sth->fetchrow_array) {
    <do DB accumulation into some variable>
}
<print the output using the the data returned from the DB>
```

In the example above the `httpd_process` will grow up by the size of the variables that have been allocated for the records that matched the query. (Again remember to multiply it by the number of the children your server runs!).

A better approach is to not accumulate the records, but rather print them as they are fetched from the DB. Moreover, we will use the **`bind_col()`** and **`$sth->fetchrow_arrayref()`** (aliased to **`$sth->fetch()`**) methods, to fetch the data in the fastest possible way.

The example below prints a HTML TABLE with matched data, the only memory that is being used is a **@cols** array to hold temporary row values:

```
my @select_fields = qw(a b c);
    # create a list of cols values
my @cols = ();
@cols[0..$#select_fields] = ();
$sth = $dbh->prepare($do_sql);
$sth->execute;
    # Bind perl variables to columns.
$sth->bind_columns(undef, \@cols);
print "<TABLE>";
while($sth->fetch) {
    print "<TR>",
        map("<TD>$_</TD>", @cols),
        "</TR>";
}
print "</TABLE>";
```

Note: the above method doesn't allow you to know how many records have been matched. The workaround is to run an identical query before the code above where you use **SELECT count(*)** ... instead of **'SELECT *** ... to get the number of matched records.

3.4 Reloading Modules and Required Files

When you develop plain CGI scripts, you can just change the code, and rerun the CGI from your browser.

Since the script isn't cached in memory, the next time you call it the server starts up a new perl process, which recompiles it from scratch.

The effects of any modifications you've applied are immediately present.

The situation is different with **Apache::Registry**, since the whole idea is to get maximum performance from the server. By default, the server won't spend the time to check whether any included

library modules have been changed.

It assumes that they weren't, thus saving a few milliseconds to **stat()** the source file (multiplied by however many modules/libraries you are **use()-ing** and/or **require()-ing** in your script.)

The only check that is being done is whether your main script has been changed.

There are a couple of techniques, to reload modified files:

3.4.1 Restarting the server

The simplest approach is to restart the server each time you apply some change to your code.

3.4.2 Using Apache::StatINC

After restarting the server about 100 times, you will be tired and will look for another solutions. Help comes from the **Apache::StatINC** module.

Apache::StatINC reloads **%INC** files when updated on disk. When Perl pulls a file via require, it stores the filename in the global hash **%INC**.

The next time Perl tries to require the same file, it sees the file in **%INC** and does not reload from disk. This module's handler iterates over **%INC** and reloads the file if it has changed on disk.

To enable this module just add two lines to **httpd.conf** file.

```
PerlModule Apache::StatINC  
PerlInitHandler Apache::StatINC
```

To be sure it really works, turn on the debug mode on your development box with **PerlSetVar StatINCDebug On**. You end up with something like:

```
PerlModule Apache::StatINC
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry::handler
  Options ExecCGI
  PerlSendHeader On
  PerlInitHandler Apache::StatINC
  PerlSetVar StatINCDebug On
</Location>
```

Beware that only the modules located in **@INC** are being reloaded on change, and you can change the **@INC** only before the server has been started.

Whatever you do in your scripts/modules which are being **required()** after the server startup will not have any effect on **@INC**.

When you do **use lib qw(foo/bar);**, the **@INC** is being changed only for the time the code is being parsed and compiled.

When it's over the **@INC** is being reset to its original value.

To make sure that you have set a correct **@INC** fetch <http://www.nowhere.com/perl-status?inc> and watch the bottom of the page.

(I assume you have configured the `<Location /perl-status>` section in **httpd.conf** as the `mod_perl` docs show.)

3.4.3 *Reloading only specific files*

Checking all the modules in **%INC** every time can add a large overhead to server response times, and you certainly would not want **Apache::StatINC** module to be enabled in your production site's configuration.

But sometimes you want to have some Configuration module to be reloaded without restarting the whole server. To accomplish this, one of the solutions is to use a code that I describe below.

Assuming that you start your script with loading **Foo::Bar** and importing some tags:

```
use Lib "some/private/path";  
use Foo::Bar qw(:tags_group tag1 tag2);
```

Now to make a modification testing and reload at runtime you have to use something like this:

```
# child's private global variable to keep the timestamps
use vars qw(%MODIFIED);

my $module = "Foo::Bar";

(my $inc_key = $module) =~ s|:|/|g;
$inc_key .= ".pm";

my $path = $INC{$inc_key} or warn "Can't find $inc_key in %INC\n";

# set modification time if it wasn't set before (first
# time) Note: Use (stat $path)[9] instead of -M test, if
# you reset time with $^M=time
```

```
$MODIFIED{$module} ||= -M $path;

# now check whether it was changed
if ($MODIFIED{$module} != -M $path) {

    delete $INC{$inc_key};
    require $path;

# now reimport the symbols (if you need them back :)
import $module qw(:tags_group tag1 tag2);

# Update the MODIFICATION times
$MODIFIED{$module} = -M $path;
}
```

You may want to add debug print statements to debug this code in your application.

3.5 Filehandlers and locks leakages

When you write a script running under mod_cgi, you can get away with sloppy programming, like opening a file and letting the interpreter to close it for you when the script had finished his run:

```
open IN, "in.txt"  
or die "Cannot open in.txt for reading : $!\n";
```

For mod_perl you **must close()** the files you opened!

```
close IN;
```

somewhere before the end of the script, since if you forget to **close()**, you might get a file descriptor leakage and unlock problem (if you **flock()**ed on this file descriptor).

Even if you do have it, but for some reason the interpreter was stopped before the cleanup call, because of various reasons, such as user aborted script the leakage is still there.

In a long run your machine might get run out of file descriptors, and even worse - file might be left locked and unusable by other invocations of the same and other scripts.

What can you do? Use **IO::File** (and other **IO::*** modules), which allows you to assign the file handler to variable, which can be **my()** (lexically) scoped.

And when this variable goes out of scope the file or other file system entity will be properly closed and unlocked (if it was locked).

Lexically scoped variable will always go out of scope at the end of the script's run even if it was aborted in the middle or before the end if it was defined inside some internal block. For example:

```
{
    my $fh = new IO::File("filename") or die $!;
    # read from $fh
} # ...$fh is closed automatically at end of block, without leaks.
```

As I have just mentioned, you don't have to create a special block for this purpose, for a file the code is written in is a virtual block as well, so you can simply write:

```
my $fh = new IO::File("filename") or die $!;  
# read from $fh  
# ...$fh is closed automatically at end of block, without leaks.
```

What the first technique (using **{ BLOCK }**) makes sure is that the file will be closed the moment, the block is finished.

But even faster and lighter technique is to use **Symbol.pm**:

```
my $fh = Symbol::gensym( );  
open $fh, "filename" or die $!
```

Use these approaches to ensure you have no leakages, but don't be lazy to write **close()** statements, make it a habit.

3.6 The Script is too dirty, But It does the job and I can't afford rewriting it.

You still can win from using `mod_perl`.

One approach is to replace the **Apache::Registry** handler with **Apache::PerlRun** and define a new location (the script can reside in the same directory on the disk.

```
# srm.conf
Alias /cgi-perl/ /home/httpd/cgi/

# httpd.conf
<Location /cgi-perl>
    SetHandler perl-script
    PerlHandler Apache::PerlRun
    Options ExecCGI
    allow from all
    PerlSendHeader On
</Location>
```

Another “bad”, but working method is to set

MaxRequestsPerChild to 1, which will force each child to exit after serving only one request, so you’ll get the preloaded modules, etc., the script will be compiled each request, then killed off.

This isn't good for "high-traffic" sites though, as the parent server will need to fork a new child each time one is killed, but you can fiddle with **MaxStartServers**, **MinSpareServers**, to make the parent spawn more servers ahead so the killed one will be immediately replaced with the fresh one. Again, probably that's not what you want.

3.7 Apache::**PerlRun** - a closer look

Apache::PerlRun**** gives you a benefit of preloaded perl and its modules.

This module's handler emulates the CGI environment, allowing programmers to write scripts that run under CGI or mod_perl without any change:

- the **Apache::**PerlRun**** handler does not cache the script inside of a subroutine.
- Scripts will be “compiled” on each request.

- After the script has run, its name space is flushed of all variables and subroutines.

Still, you don't have the overhead of loading the perl and compilation time of the standard modules.

If your script is very light, but uses lots of standard modules - you will see no difference between **Apache::PerlRun** and **Apache::Registry** !

Be aware though, that if you use packages that use internal variables that have circular references, they will be not flushed!!!

Apache::PerlRun only flushes your script's name space, which does not include any other required packages' name spaces.

If there's a reference to a **my()** scoped variable that's keeping it from being destroyed after leaving the eval scope (of **Apache::PerlRun**), that cleanup might not be taken care of until the server is shutdown and **perl_destruct()** is run, which always happens after running command line scripts. Consider this example:

```
package Foo;
sub new { bless {} }
sub DESTROY {
    warn "Foo->DESTROY\n";
}
eval <<'EOF';
package my_script;
my $self = Foo->new;
$self->{circle} = $self;
```

EOF

```
print $@ if $@;  
print "Done with script\n";
```

First you'll see:

```
Foo->DESTROY  
Done with script
```

Then, uncomment the line where **\$self** makes a circular reference, and you'll see:

```
Done with script  
Foo->DESTROY
```

In this case, under `mod_perl` you wouldn't see **Foo->DESTROY** until the server shutdown, or your module properly took care of things.

3.8 Selecting the right porting/working mode

If your project schedule is tight, I would suggest converting to `mod_perl` in the following steps: Initially, run all the scripts in the **Apache::PerlRun** mode. Then as time allows, move them into **Apache::Registry** mode.

3.9 Compiled Regular Expressions

When using a regular expression that contains an interpolated Perl variable, if it is known that the variable (or variables) will not vary during the execution of the program, a standard optimization technique consists of adding the `/o` modifier to the regex pattern.

This directs the compiler to build the internal table once, for the entire lifetime of the script, rather than every time the pattern is executed. Consider:

```
# likely to be input from an HTML form field
```

```
my $pat = '^foo$';  
foreach( @list ) {  
    print if /$pat/o;  
}
```

This is usually a big win in loops over lists, or when using **grep()** or **map()** operators.

In long-lived mod_perl scripts, however, this can pose a problem if the variable changes according to the invocation.

The first invocation of a fresh httpd child will compile the regex and perform the search correctly.

However, all subsequent uses by the httpd child will continue to match the original pattern, regardless of the current contents of the Perl variables the pattern is dependent on.

Your script will appear broken.

There are two solutions to this problem:

The first `--` is to use **eval `q//`**, to force the code to be evaluated each time. Just make sure that the eval block covers the entire loop of processing, and not just the pattern match itself.

The above code fragment would be rewritten as:

```
my $pat = '^foo$';
eval q{
    foreach( @list ) {
        print if /$pat/o;
    }
}
```

Just saying:

```
foreach( @list ) {
    eval q{ print if /$pat/o; };
}
```

is going to be a horribly expensive proposition.

You can use this approach if you require more than one pattern match operator in a given section of code.

If the section contains only one operator (be it an **m//** or **s///**), you can rely on the property of the null pattern, that reuses the last pattern seen. This leads to the second solution, which also eliminates the use of `eval`.

The above code fragment becomes:

```
my $pat = '^foo$';  
"something" =~ /$pat/; # dummy match (MUST NOT FAIL!)  
foreach( @list ) {  
    print if //;  
}
```

The only gotcha is that the dummy match that boots the regular expression engine must absolutely, positively succeed, otherwise the pattern will not be cached, and the **//** will match everything. If you can't count on fixed text to ensure the match succeeds, you have two possibilities.

If you can guarantee that the pattern variable contains no meta-characters (things like `*`, `+`, `^`, `$...`), you can use the dummy match:

```
# guaranteed if no meta-characters present
"$pat" =~ /\Q$pat\E/;
```

If there is a possibility that the pattern can contain meta-characters, you should search for the pattern or the unsearchable `\377` character as follows:

```
# guaranteed if meta-characters present
"\377" =~ /$pat|^\[\377]$/;
```

Another approach:

It depends on the complexity of the regexp you apply this technique to. One common usage where compiled regexp is usually more efficient is to “match any one of a group of patterns” over and over again.

Maybe with some helper routine, it's easier to remember. Here is one slightly modified from Jeffery Friedl's example in his book “Mastering Regex.” .

```

# Build_MatchMany_Function
# -- Input: list of patterns
# -- Output: A code ref which matches its $_[0]
#          against ANY of the patterns given in the
#          "Input", efficiently.
sub Build_MatchMany_Function {
    my @R = @_;
    my $expr = join '||',
        map { "\$_[0] =~ m/\$R[\$_]/o" } ( 0..$#R );
    my $matchsub = eval "sub { $expr }";
    die "Failed in building regex @R: \$@" if $@;
    $matchsub;
}

```

Example usage:

```
@some_browsers = qw(Mozilla Lynx MSIE LWP);
$Known_Browser =
    Build_MatchMany_Function(@some_browsers);

while (<ACCESS_LOG> ) {
    # ...
    $browser = get_browser_field($_);
    if ( ! &$Known_Browser($browser) ) {
        print STDERR "Unknown Browser: $browser\n";
    }
    # ...
}
```

3.10 Finding the line number the error/warning has been triggered at

Apache::Registry, Apache::PerlRun and modules that compile-via-eval confuse the line numbering. Other files that are read normally by Perl from disk have no problem with file name/line number.

If you compile with the experimental **PERL_MARK_WHERE=1**, it shows you almost the exact line number, where this is happening.

Generally a compiler makes a shift in its line counter. You can always stuff your code with special compiler directives, to reset its counter to the value you will tell.

At the beginning of the line you should write (the '#' in column 1):

```
#line 298 myscript.pl
```

or

```
#line 890 some_label_to_be_used_in_the_error_message
```

(You will find a script to add these tags in your handouts)

Also notice, that another solution is to move most of the code into a separate modules, which ensures that the line number will be reported correctly.

To have a complete trace of calls add:

```
use Carp ();  
local $SIG{__WARN__} = \&Carp::cluck;
```

3.11 Forking subprocesses from `mod_perl`

Generally you should not fork from your `mod_perl` scripts, since when you do -- you are forking the entire apache web server, lock, stock and barrel.

Not only is your perl code being duplicated, but so is `mod_ssl`, `mod_rewrite`, `mod_log`, `mod_proxy`, `mod_spelling` or whatever modules you have used in your server, all the core routines and so on.

A much wiser approach would be to spawn a sub-process, hand it the information it needs to do the task, and have it detach (close `x3 + setsid()`). This is wise only if the parent who spawns this process, immediately continue, you do not wait for the sub

process to complete.

This approach is suitable for a situation when you want to trigger a long time taking process through the web interface, like processing some data, sending email to thousands of subscribed users and etc.

Otherwise, you should convert the code into a module, and use its functions or methods to call from CGI script.

Just making a **system()** call defeats the whole idea behind `mod_perl`, perl interpreter and modules should be loaded again for this external program to run.

Basically, you would do:

```
$params=FreezeThaw::freeze(  
    [all data to pass to the other process]  
    );  
system("program.pl $params");
```

and in **program.pl** :

```
@params=FreezeThaw::thaw(shift @ARGV);  
# check that @params is ok  
close STDIN;  
close STDOUT;  
open STDERR, ">/dev/null";  
setsid(); # to detach
```

At this point, **program.pl** is running in the “background” while the **system()** returns and permits apache to get on with life.

This has obvious problems. Not the least of which is that **@params** must not be bigger than whatever your architecture's limit is (could depend on your shell).

Also, the communication is only one way.

However, you might want be trying to do the “wrong thing”. If what you want is to send information to the browser and then do some post-processing, look into **PerlCleanupHandler**.

3.12 Debugging your code in Single Server Mode

Running in `httpd -X` mode. (good only for testing during development phase). We have been talking about this mode already... you will find more notes in your handouts.

3.13 -M and other time() file tests under mod_perl

Under mod_perl, files that have been created after the server's (child?) startup are being reported with negative age with **-M (-C -A)** test.

This is obvious if you remember that you will get the negative result if the server was started before the file was created and it's a normal behavior with any perl.

If you want to have **-M** test to count the time relative to the current request, you should reset the **\$^T** variable as with any other perl script. Just add **\$^T=time;** at the beginning of the scripts.

3.14 Handling the 'User pressed Stop button' case

When a user presses the **STOP** button, Apache will detect that via **\$SIG{PIPE}** and will cease the script execution.

When we are talking about `mod_cgi`, there is generally no problem, since all opened files will be closed and all the resources will be freed (almost all -- if you happened to use external lock files, most likely the resources that are being locked by these will be left blocked and non-usable by any others who use the same advisory locking scheme.)

It's important to notice that when the user hits the browser's **STOP** button, the `mod_perl` script is blissfully unaware until it tries to send some data to the browser.

At that point, Apache realizes that the browser is gone, and all the good cleanup stuff happens.

Starting from apache 1.3.6 apache will not catch SIGPIPE anymore and modperl will do it much better.

What happens if your mod_perl script has some global variables, that are being used for resource locking?

It's possible not to notice the pitfall if the critical code section between lock and unlock is very short and finishes fast, so you never see this happens (you aren't fast enough to stop the code in the middle).

But look at the following scenario:

1. lock resource
 <critical section starts>
2. sleep 20 (== do some time consuming processing)
 <critical section ends>
3. unlock resource

If user presses **STOP** and Apache sends **SIGPIPE** before step 3, since we are in the mod_perl mode and we want the lock variable to be cached, it will be not unlocked. A kind of **deadlock** exists.

Here is the working example:

```
use vars qw(%CACHE);
use CGI;
$|=1;
my $q = new CGI;
print $q->header,$q->start_html;

print $q->p("$$ Going to lock!\n");

# actually the while loop below is not needed
```

```
# (since it's an internal lock and accessible only
# by the same process and it if it's locked... it's locked for the
# whole child's life
while (1) {
  unless (defined $CACHE{LOCK} and $CACHE{LOCK} == 1) {
    $CACHE{LOCK} = 1;
    print $q->p("Got the lock!\n");
    last;
  }
}
print $q->p("Going to sleep (I mean working)!");
my $c=0;
foreach (1..10) {
  sleep 1;
  print $c++,"\n<BR>";
}

print $q->p("Going to unlock!");
$CACHE{LOCK} = 0;
print $q->p("Unlock!\n");
```

Run the server with **-X**, Press **STOP** before the count-up to 10 has been finished.

Then rerun the script, it'll hang in **while(1)**! The resource is not available anymore to this child.

You may ask, what is the solution for this problem? All **END** blocks that are encountered during compilation of **Apache::Registry** scripts are called after the script done is running, including subsequent invocations when the script is cached in memory.

So if you are running in **Apache::Registry** mode, the following is your remedy:

```
END {  
    $CACHE{LOCK} = 0;  
}
```

Notice that the **END** block will be run after the **Apache::Registry::handler** is finished (not during the cleanup phase though).

If you are into a perl API, use the **register_cleanup()** method of Apache.

```
$r->register_cleanup( sub { $CACHE{LOCK} = 0; } );
```

If you are into Apache API **Apache->request->connection->aborted()** construct can be used to test for the aborted connection.

I hope you noticed, that this example is very misleading, since there is a different instance of **%CACHE** in every child, so if you modify it -- it is known only inside the same child, none of global **%CACHE** variables in other children is getting affected.

But if you are going to work with code that allows you to control variables that are being visible to every child (some external shared memory or other approach) -- the hazard this example still applies.

Make sure you unlock the resources either when you stop using them or when the script is being aborted in the middle, before the actual unlocking is being happening.

3.15 Handling the server timeout cases and working with `SIG{ALRM}`

A similar situation to **Pressed Stop button disease** happens when client (browser) timeouts the connection (is it about 2 minutes?).

There are cases when your script is about to perform a very long operation and there is a chance that its duration will be longer than the client's timeout.

One case I can think about is the DataBase interaction, where the DB engine hangs or needs a lot of time to return results. If this is the case, use `SIG{ALRM}` to prevent the timeouts:

```
$timeout = 10; # seconds
eval {
  local $SIG{ALRM} =
    sub { die "Sorry timed out. Please try again\n" };
  alarm $timeout;
  ... db stuff ...
  alarm 0;
};
die $@ if $@;
```

But, as lately it was discovered **local \$SIG{ALRM}** does not restore the original underlying C handler. It was fixed in the `mod_perl 1.19_01` (CVS version).

```
;0)
```

4 Performance. Benchmarks.

4.1 Performance: The Overall picture

All the efforts are made to make user's web browsing experience a swift.

Among other web site usability factors, speed is one of the most crucial ones.

What is a correct speed measurement?

Since user is the one that interacts with web site, speed measurement is a time passed from the moment user follows a link or presses a submit button till the resulting page is being rendered by her browser.

So if we trace the data packet's movement as it leaves user's machine (request sent) till the reply arrives, the packet travels through many entities on its way:

- It has to make its way through the network, passing many interconnection nodes,
- before it enters the target machine it might go through proxy (accelerator) servers,
- then it's being served by your server,
- and finally it has to make the whole way back.

A webserver is only one of the elements the packet sees on its way.

You could work hard to fine tune your webserver for the best performance, but a slow NIC (Network Interface Card) or slow network connection from your server might defeat it all.

That's why it's important to think big and to be aware of possible bottlenecks between the server and the web.

Of course there is nothing you can do if user has a slow connection on its behalf.

Moreover you might tune your scripts and webserver to process incoming requests ultra fast, so you will need a little number of working servers, but you might find out that server processes are busy waiting for slow clients to complete the download.

You will see more examples in this chapter.

My point is that a web service is like car, if one of the details or mechanisms is broken the car will not drive smoothly and it can even stop dead if pushed further without first fixing it.

4.2 Sharing Memory

A very important point is the sharing of memory. If your OS supports this (and most sane systems do), you might save more memory by sharing it between child processes.

This is only possible when you preload code at server startup.

However during a child process' life, its memory pages becomes unshared and there is no way we can control perl to make it allocate memory so (dynamic) variables land on different memory pages than constants, that's why the **copy-on-write** effect will hit almost at random.

If you are pre-loading many modules you might be able to balance the memory that stays shared against the time for an occasional fork by tuning the **MaxRequestsPerChild** to a point where you

restart before too much becomes unshared.

In this case the **MaxRequestsPerChild** is very specific to your scenario.

You should do some measurements and you might see if this really makes a difference and what a reasonable number might be.

Each time a child reaches this upper limit and restarts it should release the unshared copies and the new child will inherit pages that are shared until it scribbles on them.

It is very important to understand that your goal is not to have **MaxRequestsPerChild** to be 10000.

Having a child serving 300 requests on precompiled code is already a huge speedup, so if it is 100 or 10000 it does not really matter if it saves you the RAM by sharing.

Do not forget that if you preload most of your code at the server startup, the fork to spawn a new child will be very very fast, because it inherits most of the preloaded code and the perl interpreter from the parent process.

But than, during the work of the child, its memory pages (which aren't really its yet, it uses the parent's pages) are getting dirty (originally inherited and shared variables are getting updated/modified) and the **copy-on-write** happens, which reduces the number of shared memory pages - thus enlarging the memory demands.

Killing the child and respawning a new one, allows to get the pristine shared memory from the parent process again.

The conclusion is that **MaxRequestsPerChild** should not be too big, otherwise you loose the benefits of the memory sharing.

4.3 Preload Perl modules at server startup

Use the **PerlRequire** and **PerlModule** directives to load commonly used modules such as **CGI.pm**, **DBI** and etc., when the server is started.

On most systems, server children will be able to share the code space used by these modules.

Just add the following directives into **httpd.conf**:

```
PerlModule CGI;  
PerlModule DBI;
```

But even a better approach is to create a separate startup file (where you code in plain perl) and put there things like:

```
use DBI;  
use Carp;
```

Then you **require()** this startup file with help of **PerlRequire** directive from **httpd.conf**, by placing it before the rest of the **mod_perl** configuration directives:

```
PerlRequire /path/to/start-up.pl
```

CGI.pm is a special case. Ordinarily **CGI.pm** autoloads most of its functions on an as-needed basis.

This speeds up the loading time by deferring the compilation phase.

However, if you are using `mod_perl`, `FastCGI` or another system that uses a persistent Perl interpreter, you will want to precompile the methods at initialization time.

To accomplish this, call the package function `compile()` like this:

```
use CGI ();  
CGI->compile(':all');
```

The arguments to **compile()** are a list of method names or sets, and are identical to those accepted by the **use()** and **import()** operators.

Note that in most cases you will want to replace **':all'** with tag names you really use in your code, since generally only a subset of subs is actually being used.

4.4 Preload Registry Scripts

`Apache::RegistryLoader` compiles `Apache::Registry` scripts at server startup.

It can be a good idea to preload the scripts you are going to use as well. So the code will be shared among the children.

Here is an example of the use of this technique. This code is included in a `PerlRequire`'d file, and walks the directory tree under which all registry scripts are installed.

For each `.pl` file encountered, it calls the `Apache::RegistryLoader::handler()` method to preload the script in the parent server (before pre-forking the child processes):

```
use File::Find 'finddepth';
use Apache::RegistryLoader ();
{
    my $perl_dir = "perl/";
    my $url = Apache::RegistryLoader->new;
    finddepth(sub {
        return unless /\.pl$/;
        my $url = "$File::Find::dir/$_";
        print "pre-loading $url\n";

        my $status = $url->handler($url);
        unless($status == 200) {
            warn "pre-load of '$url' failed,
                status=$status\n";
        }
    }, $perl_dir);
}
```

4.5 Avoid Importing Functions

When possible, avoid importing a module's functions into your name space.

The aliases which are created can take up quite a bit of space.

Try to use method interfaces and fully qualified **Package::function** or **\$Package::variable** like names instead.

4.6 How can I find if my mod_perl scripts have memory leaks

`Apache::Leak` (derived from `Devel::Leak`) should help you with this task. Example:

```
use Apache::Leak;

my $global = "FOOAAA";

leak_test {
    $global = 1;
    ++$global;
};
```

The argument to **leak_test()** is an anonymous sub, so you can just throw it around any code you suspect might be leaking.

Beware, it will run the code twice, because the first time in, new **SVs** are created, but does not mean you are leaking, the second pass will give better evidence.

You do not need to be inside `mod_perl` to use it, from the command line, the above script outputs:

```
ENTER: 1482 SVs  
new c28b8 : new c2918 :  
LEAVE: 1484 SVs  
ENTER: 1484 SVs  
new db690 : new db6a8 :  
LEAVE: 1486 SVs  
!!! 2 SVs leaked !!!
```

Build a debuggable perl to see dumps of the **SVs**.

The simple way to have both a normal perl and debuggable perl, is to follow hints in the **SUPPORT** doc for building **libperl.a**, when that is built copy the **perl** from that directory to your perl bin directory, but name it **dperl**.

4.7 Limiting the size, resource, speed of the processes

The following modules allow you to restrict and limit a usage of your resources:

- **Apache::SizeLimit** -- Controll the size of the process. See **perldoc Apache::SizeLimit**.
- **Apache::Resource** -- Limit resources used by httpd children. See **perldoc Apache::Resource**.
- **Apache::SpeedLimit** -- Limiting the request rate speed (robots blocking). See http://www.modperl.com/chapters/ch6.html#Blocking_Greedy_Clients.

4.8 Persistent DB Connections

Another popular use of mod_perl is to take advantage of its ability to maintain persistent open database connections.

The basic approach is as follows:

```
# Apache::Registry script
-----
use strict;
use vars qw( $dbh );

$dbh ||= SomeDbPackage->connect(...);
```

Since **\$dbh** is a global variable for the child, once the child has opened the connection it will use it over and over again, unless you perform **disconnect()**.

Be careful to use different names for handlers if you open connection to different databases!

Apache::DBI allows you to make a persistent database connection.

With this module enabled, every **connect()** request to the plain **DBI** module will be forwarded to the **Apache::DBI** module.

This looks to see whether a database handle from a previous **connect()** request has already been opened, and if this handle is still valid using the **ping()** method.

If these two conditions are fulfilled it just returns the database handle.

If there is no appropriate database handle or if the ping method fails, a new connection is established and the handle is stored for later re-use.

There is no need to delete the `disconnect()` statements from your code. They will not do a thing, as the `Apache::DBI` module overloads the `disconnect()` method with a NOP.

On child's exit there is no explicit disconnect, the child dies and so does the database connection.

You may leave the **use DBI**; statement inside the scripts as well.

The usage is simple -- add to **httpd.conf**:

```
PerlModule Apache::DBI
```

It is important, to load this module before any other **ApacheDBI*** module!

```
db.pl
-----
use DBI;
use strict;
my $dbh =
    DBI->connect(
        'DBI:mysql:database',
        'user', 'password',
        { autocommit => 0 }
    ) || die $DBI::errstr;
...rest of the program
```

If you use **DBI** for DB connections, and you use **Apache::DBI** to make them persistent, it also allows you to preopen connections to DB for each child with **connect_on_init()** method, thus saving up a connection overhead on the very first request of every child.

```
use Apache::DBI ( );
Apache::DBI->connect_on_init( "DBI:mysql:test",
    "login",
    "passwd",
    {
        RaiseError => 1,
        PrintError => 0,
        AutoCommit => 1,
    }
);
```

This can be used as a simple way to have apache children establish connections on server startup. This call should be in a startup file **require()** by **PerlRequire** or inside `<Perl>` section.

Another problem is with timeouts: some databases disconnect the client after a certain time of inactivity. This problem is known as **morning bug**. Remedies:

- The **ping()** method ensures that this will not happen. Some **DBD** drivers don't have this method, check the **Apache::DBI** manpage to see how to write a **ping()** method.
- Another approach is to change the client's connection timeout. For mysql users, starting from mysql-3.22.x you can set a **wait_timeout** option at mysqld server startup to change the default value. Setting it to 36 hours probably would fix the timeout problem.

4.9 Benchmarks. Impressing your Boss and Colleagues.

How much faster is `mod_perl` than `mod_cgi` (aka plain `perl/CGI`)? There are many ways to benchmark the two.

I'll present a few examples and numbers below. Checkout the **benchmark** directory of `mod_perl` distribution for more examples.

If you are going to write your own benchmarking utility -- use **Benchmark** module for heavy scripts and **Time::HiRes** module for very fast scripts (faster than 1 sec) where you need better time precision.

There is no need to write a special benchmark though. If you want to impress your boss or colleagues, just take some heavy CGI script you have (e.g. a script that crunches some data and prints the results to STDOUT), open 2 xterms and call the same script in mod_perl mode in one xterm and in mod_cgi mode in the other. You can use **lwp-get** from **LWP** package to emulate the web agent (browser). (**benchmark** directory of mod_perl distribution includes such an example)

4.9.1 Benchmarking scripts with execution times below 1 second :)

As noted before, for very fast scripts you will have to use the **Time::HiRes** module, its usage is similar to the **Benchmark**'s.

```
use Time::HiRes qw(gettimeofday
                   tv_interval);

my $start_time = [ gettimeofday ];
&sub_that_takes_a_teeny_bit_of_time()
my $end_time = [ gettimeofday ];
my $elapsed = tv_interval($start_time,
                          $end_time);

print "the sub took $elapsed secs."
```

4.9.2 *PerlHandler's Benchmarking*

At <http://perl.apache.org/dist/contrib/> you will find **Apache::Timeit** package which does **PerlHandler's Benchmarking**.

4.9.3 More benchmarking techniques

I'll cover more benchmarking packages as I proceed...

4.10 Tuning the Apache's configuration variables for the best performance

It's very important to make a correct configuration of the **MinSpareServers**, **MaxSpareServers**, **StartServers**, **MaxClients**, and **MaxRequestsPerChild** parameters.

There are no defaults, the values of these variable are very important, as if too "low" you will under-use the system's capabilities, and if too "high" chances that the server will bring the machine to its knees.

All the above parameters should be specified on the basis of the resources you have.

While with a plain apache server, there is no big deal if you run too many servers (not too many of course) since the processes are of ~1Mb and aren't eating a lot of your RAM.

Generally the numbers are even smaller if memory sharing is taking place.

The situation is different with mod_perl.

I have seen mod_perl processes of 20Mb and more.

Now if you have **MaxClients** set to 50: 50x20Mb = 1Gb

Do you have 1Gb of RAM? Probably not...

So how do you tune these parameters? Generally by trying different combinations and benchmarking the server.

Again mod_perl processes can be of much smaller size if sharing is in place.

Before you start this task you should be armed with a proper weapon:

- You need a **crashme** utility, which will load your server with mod_perl scripts you possess.
- You need it to have an ability to emulate a multiuser environment and to emulate multiple clients behavior which will call the mod_perl scripts at your server simultaneously.

While there are commercial solutions, you can get away with free ones which do the same job. You can use an **ab (ApacheBench)** utility that comes within apache distribution or use a tool based on **LWP::Parallel::UserAgent**.

Another important issue is to make sure to run testing client (load generator) on a system that is more powerful than the system being tested.

After all we are trying to simulate the Internet users, where many users are trying to reach your service at once -- since a number of concurrent users can be quite large, your testing machine much be very powerful and capable to generate a heavy load.

Of course you should not run the clients and the server on the same machine. If you do -- your testing results would be incorrect, since clients will eat a CPU and a memory that have to be dedicated to the server, and vice versa.

4.10.1 *Tuning with ab - ApacheBench*

ab is a tool for benchmarking your Apache HTTP server.

It is designed to give you an impression on how much performance your current Apache installation can give.

In particular, it shows you how many requests per secs your Apache server is capable of serving.

The **ab** tool comes bundled with apache source distribution (and it's free :).

Let's try it. We will simulate 10 users concurrently requesting a very light script at **www.nowhere.com:81/test/test.pl**. Each "user" makes 10 requests.

```
% ./ab -n 100 -c 10 www.nowhere.com:81/test/test.pl
```

The results are:

Concurrency Level: 10
Time taken for tests: 0.715 seconds
Completed requests: 100
Failed requests: 0
Non-2xx responses: 100
Total transferred: 60700 bytes
HTML transferred: 31900 bytes
Requests per second: 139.86
Transfer rate: 84.90 kb/s received

Connection Times (ms)	min	avg	max
Connect:	0	0	3
Processing:	13	67	71
Total:	13	67	74

The only numbers we really care about are:

Complete requests: 100

Failed requests: 0

Requests per second: 139.86

Let's raise the load of requests to 100 x 10 (10 users, each makes 100 requests)

```
% ./ab -n 1000 -c 10 www.nowhere.com:81/perl/access/access.cgi
Concurrency Level:      10
Complete requests:     1000
Failed requests:        0
Requests per second:   139.76
```

As expected nothing changes -- we have the same 10 concurrent users.

Now let's raise the number of concurrent users to 50:

```
% ./ab -n 1000 -c 50 www.nowhere.com:81/perl/access/access.cgi
Complete requests:    1000
Failed requests:      0
Requests per second:  133.01
```

qWe see that the server is capable of serving 50 concurrent users at an amazing 133 req/sec! Let's find the upper boundary.

Using **-n 10000 -c 1000** failed to get results (Broken Pipe?).

Using **-n 10000 -c 500** derived 94.82 req/sec.

The server's performance went down with the high load.

The above tests were performed with the following configuration:

```
MinSpareServers 8  
MaxSpareServers 6  
StartServers 10  
MaxClients 50  
MaxRequestsPerChild 1500
```

Now let's kill a child after a single request, we will use the following configuration:

```
MinSpareServers 8  
MaxSpareServers 6  
StartServers 10  
MaxClients 100  
MaxRequestsPerChild 1
```

Simulate 50 users each generating a total of 20 requests:

```
% ./ab -n 1000 -c 50 www.nowhere.com:81/perl/access/access.cgi
```

The benchmark timed out with the above configuration....

I watched the output of **ps** as I ran it, the parent process just wasn't capable of respawning the killed children at that rate...

When I raised the **MaxRequestsPerChild** to 10 I've got 8.34 req/sec - very bad (18 times slower!)

(You can't benchmark the importance of the **MinSpareServers**, **MaxSpareServers** and **StartServers** with this kind of test).

Now let's try to return **MaxRequestsPerChild** to 1500, but to lower the **MaxClients** to 10 and run the same test:

```
MinSpareServers 8  
MaxSpareServers 6  
StartServers 10  
MaxClients 10  
MaxRequestsPerChild 1500
```

I've got 27.12 req/sec, which is better but still 4-5 times slower (133 with **MaxClients** of 50)

Summary:

I have tested a few combinations of server configuration variables (**MinSpareServers**, **MaxSpareServers**, **StartServers**, **MaxClients**, **MaxRequestsPerChild**). And the results we have received are as follows:

MinSpareServers, **MaxSpareServers** and **StartServers** are only important for user response times (sometimes user will have to wait a bit).

The important parameters are **MaxClients** and **MaxRequestsPerChild**.

- **MaxClients** should be not too big so it will not abuse your machine's memory resources and not too small, when users will be forced to wait for the children to become free to come serve them.

- **MaxRequestsPerChild** should be as big as possible, to take the full benefit of mod_perl, but watch your server at the beginning to make sure your scripts are not leaking memory, thereby causing your server (and your service) to die very fast.

Also it is important to understand that we didn't test the response times in the tests above, but the ability of the server to respond under a heavy load of requests. If the script that was used to test was heavier, the numbers would be different but the conclusions are very similar.

The benchmarks were run with:

HW: RS6000, 1Gb RAM

SW: AIX 4.1.5 . mod_perl 1.16, apache 1.3.3

**Machine running only mysql, httpd docs and mod_perl servers.
Machine was _completely_ unloaded during the benchmarking.**

After each server restart when I did changes to the server's configurations, I made sure the scripts were preloaded by fetching a script at least once by every child.

It is important to notice that none of requests timed out, even if was kept in server's queue for more than 1 minute! (That is the way **ab** works, which is OK for the testing purposes but will be unacceptable in the real world - users will not wait for more than 5-10 secs for a request to complete, and the client (browser) will timeout in a few minutes.)

Now let's take a look at some real code whose execution time is more than a few milliseconds. We will do real testing and collect the data in tables for easier viewing.

I will use the following abbreviations:

NR = Total Number of Request
NC = Concurrency
MC = MaxClients
MRPC = MaxRequestsPerChild
RPS = Requests per second

Running a mod_perl script with lots of mysql queries (the script under test is mysqlid bounded)
(http://www.nowhere.com:81/perl/access/access.cgi?do_sub=query_form),
with configuration:

```
MinSpareServers      8  
MaxSpareServers     16  
StartServers        10  
MaxClients           50  
MaxRequestsPerChild 5000
```

gives us:

```
NR      NC      RPS      comment
-----
 10     10     3.33    # not a reliable statistics
100     10     3.94
1000    10     4.62
1000    50     4.09
```

Conclusions: Here I wanted to show that when the application is slow -- not due to perl loading, code compilation and execution, but bounded to some external operation like `mysqld` querying which made the bottleneck -- it almost does not matter what load we place on the server.

The RPS (Requests per second) is almost the same (given that all the requests have been served, you have an ability to queue the clients, but be aware that something that goes to queue means a

waiting client and a client (browser) that might time out!

Now we will benchmark the same script without using the mysql (perl only bounded code) (<http://www.nowhere.com:81/perl/access/access.cgi>), it's the same script that just returns a HTML form, without making any SQL queries.

MinSpareServers	8
MaxSpareServers	16
StartServers	10
MaxClients	50
MaxRequestsPerChild	5000

NR	NC	RPS	comment
10	10	26.95	# not a reliable statistics
100	10	30.88	
1000	10	29.31	
1000	50	28.01	
1000	100	29.74	
10000	200	24.92	
100000	400	24.95	

Conclusions: This time the script we executed was pure perl (not bounded to I/O or mysql), so we see that the server serves the requests much faster. You can see the **RequestPerSecond** (RPS) is almost the same for any load, but goes lower when the number of concurrent clients goes beyond the **MaxClients**. With 25 RPS, the client supplying a load of 400 concurrent clients will be served in 16 secs. But to get more realistic and assume the max concurrency of 100, with 30 RPS, the client will be served in

3.5 secs, which is pretty good for a highly loaded server.

Now we will use the server for its full capacity, by keeping all **MaxClients** alive all the time and having a big **MaxRequestsPerChild**, so no server will be killed during the benchmarking.

```
MinSpareServers      50
MaxSpareServers      50
StartServers         50
MaxClients           50
MaxRequestsPerChild 5000
```

```
NR      NC      RPS      comment
```

```
-----
```

```
100     10     32.05
1000    10     33.14
1000    50     33.17
1000    100    31.72
10000   200    31.60
```

Conclusion: In this scenario there is no overhead involving the parent server loading new children, all the servers are available, and the only bottleneck is contention for the CPU.

Now we will try to change the **MaxClients** and to watch the results: Let's reduce MC to 10.

```
MinSpareServers      8
MaxSpareServers     10
StartServers        10
MaxClients           10
MaxRequestsPerChild 5000
```

```
NR      NC      RPS      comment
```

```
-----
 10     10     23.87    # not a reliable statistics
 100    10     32.64
1000    10     32.82
1000    50     30.43
1000    100    25.68
1000    500    26.95
2000    500    32.53
```

Conclusions: A very little difference! Almost no change! 10 servers were able to serve almost with the same throughput as 50 servers.

Why? My guess it's because of CPU throttling. It seems that 10 servers were serving requests 5 times faster than when in the test above we worked with 50 servers.

In the case above each child received its CPU time slice 5 times less frequently. So having a big value for **MaxClients**, doesn't mean that the performance will be better. You have just seen the numbers!

Now we will start to drastically reduce the **MaxRequestsPerChild**:

MinSpareServers	8
MaxSpareServers	16
StartServers	10
MaxClients	50

NR	NC	MRPC	RPS	comment
100	10	10	5.77	
100	10	5	3.32	
1000	50	20	8.92	
1000	50	10	5.47	
1000	50	5	2.83	
1000	100	10	6.51	

Conclusions: When we drastically reduce the **MaxRequestsPerChild**, the performance starts to become closer to the plain `mod_cgi`.

Just for comparison with mod_cgi, here are the numbers of this run with mod_cgi:

MinSpareServers	8
MaxSpareServers	16
StartServers	10
MaxClients	50

NR	NC	RPS	comment
-----------	-----------	------------	----------------

100	10	1.12	
1000	50	1.14	
1000	100	1.13	

Conclusion: mod_cgi is much slower :) in test NReq/NClients 100/10 the RPS in mod_cgi was of 1.12 and in mod_perl of 32, which is 30 times faster!!!

In the first test each child waited about 100 secs to be served. In the second and third 1000 secs!

4.10.2 *Tuning with crashme script*

This is another crashme suite originally written by Michael Schilli and located at <http://www.linux-magazin.de/ausgabe.1998.08/Pounder/pounder.html>. I did a few modifications (mostly adding `my ()` operands).

I also allowed it to accept more than one url to test, since sometimes you want to test an overall and not just one script.

The tool provides the same results as **ab** above but it also allows you to set the timeout value, so requests will fail if not served within the time out period.

You also get **Latency** (secs/Request) and **Throughput** (Requests/sec) numbers.

It can give you a better picture and make a complete simulation of your favorite Netscape browser :).

I have noticed while running these 2 benchmarking suites - **ab** gave me results 2.5-3.0 times better.

Both suites run on the same machine with the same load with the same parameters. But the implementations are different.

Sample output:

```
URL(s) : http://www.nowhere.com:81/perl/access/access.cgi
Total Requests: 100
Parallel Agents: 10
Succeeded: 100 (100.00%)
Errors: NONE
Total Time: 9.39 secs
Throughput: 10.65 Requests/sec
Latency: 0.85 secs/Request
```

And the code is in your handouts...

4.10.3 Choosing *MaxClients*

The **MaxClients** directive sets the limit on the number of simultaneous requests that can be supported; no more than this number of child server processes will be created.

To configure more than 256 clients, you must edit the **HARD_SERVER_LIMIT** entry in **httpd.h** and recompile.

In our case we want this variable to be as small as possible, this way we can virtually bound the resources used by the server children.

Since we can restrict each child's process size (with **Apache::SizeLimit**) -- the calculation of **MaxClients** is pretty straightforward :

```
Total RAM Dedicated to the Webserver
MaxClients = -----
                MAX child's process size
```

So if I have 400Mb left for the webserver to run with, I can set the **MaxClients** to be of 40 if I know that each child is bounded to the 10Mb of memory (e.g. with **Apache::SizeLimit**).

Certainly you will wonder what happens to your server if there are more than **MaxClients** concurrent users at some moment.

This situation is accompanied by the following warning message into the **error.log** file:

```
[sun Jan 24 12:05:32 1999] [error] server
reached MaxClients setting, consider
raising the MaxClients setting
```

There is no problem -- any connection attempts over the **MaxClients** limit will normally be queued, up to a number based on the **ListenBacklog** directive.

Once a child process is freed at the end of a different request, the connection will then be served.

But it **is an error** because clients are being put in the queue rather than getting served at once, despite the fact that they do not get an error response.

The error can be allowed to persist to balance available system resources and response time, but sooner or later you will need to get more RAM so you can start more children.

The best approach is to try not to have this condition reached at all, and if reach it often you should start to worry about it.

It's important to understand how much real memory a child occupies.

Your children can share the memory between them (when OS supports that and you take action to allow the sharing happen.

If this is the case, chances are that your **MaxClients** can be even higher.

But it seems that it's not so simple to calculate the absolute number. (If you come up with solution please let us know!).

If the shared memory was of the same size through the child's life, we could derive a much better formula:

$$\text{MaxClients} = \frac{\text{Total_RAM} + \text{Shared_RAM_per_Child} * \text{MaxClients}}{\text{Max_Process_Size} - 1}$$

which is:

$$\text{MaxClients} = \frac{\text{Total_RAM} - \text{Max_Process_Size}}{\text{Max_Process_size} - \text{shared_RAM_per_Child}}$$

Let's roll some calculations:

$$\begin{aligned}\text{Total_RAM} &= 500\text{Mb} \\ \text{Max_Process_Size} &= 10\text{Mb} \\ \text{shared_RAM_per_Child} &= 4\text{Mb}\end{aligned}$$

$$\begin{aligned}\text{MaxClients} &= \frac{500 - 10}{10 - 4} = 81\end{aligned}$$

With no sharing in place

500

MaxClients = ----- = 50

10

With sharing in place you can have 60% more servers without purchasing more RAM, if you improve and keep a higher sharing memory size, let's say:

Total_RAM = 500Mb

Max_Process_size = 10Mb

shared_RAM_per_Child = 8Mb

we get:

500 - 10

MaxClients = ----- = 245

10 - 8

390% more servers!!! You've got the point :)

4.10.4 *Choosing* **MaxRequestsPerChild**

The **MaxRequestsPerChild** directive sets the limit on the number of requests that an individual child server process will handle.

After **MaxRequestsPerChild** requests, the child process will die.

If **MaxRequestsPerChild** is 0, then the process will live forever.

Setting **MaxRequestsPerChild** to a non-zero limit has two beneficial effects: it solves memory leakages and helps reduce the number of processes when the server load reduces.

The first reason is the most crucial for `mod_perl`, since sloppy programming will cause a child process to consume more memory after each request.

If left unbounded, then after a certain number of requests the children will use up all the available memory and leave the server to die from memory starvation.

Note, that sometimes standard system libraries leak memory too, especially on OSes with bad memory management (e.g. Solaris 2.5 on x86 arch).

If this is your case you can set **MaxRequestsPerChild** to a small number, which will allow the system to reclaim the memory, greedy child process consumed, when it exits after **MaxRequestsPerChild** requests.

But beware -- if you set this number too low, you will loose a fracture of the speed bonus you receive with mod_perl. Consider using **Apache::PerlRun** if this is the case.

Also, setting **MaxSpareServers** to a number close to **MaxClients**, will improve the response time (but your parent process will be busy respawning new children all the time!)

Another approach is to use **Apache::SizeLimit** module. By using it, you should be able to discontinue using the **MaxRequestsPerChild**, although for some folks, using both in combination does the job.

4.10.5 *Choosing MinSpareServers, MaxSpareServers and StartServers*

With `mod_perl` enabled, it might take as much as 30 seconds from the time you start the server until it is ready to serve incoming requests.

This delay depends on the OS, the number of preloaded modules and the process load of the machine.

So it's best to set **StartServers** and **MinSpareServers** to high numbers, so that if you get a high load just after the server has been restarted, the fresh servers will be ready to serve requests immediately.

With `mod_perl`, it's usually a good idea to raise all 3 variables higher than normal.

In order to maximize the benefits of `mod_perl`, you don't want to kill servers when they are idle, rather you want them to stay up and available to immediately handle new requests.

I think an ideal configuration is to set **MinSpareServers** and **MaxSpareServers** to similar values, maybe even the same.

Having the **MaxSpareServers** close to **MaxClients** will completely use all of your resources (if **MaxClients** has been chosen to take the full advantage of the resources), but it'll make sure that at any given moment your system will be capable of responding to requests with the maximum speed (given that number of concurrent requests is not higher than **MaxClients**.)

Let's try some numbers. For a heavily loaded web site and a dedicated machine I would think of (note 400Mb is just for example):

```
Available to webserver RAM: 400Mb
Child's memory size bounded: 10Mb
MaxClients: 400/10 = 40 (larger with mem sharing)
StartServers: 20
MinSpareServers: 20
MaxSpareServers: 35
```

However if I want to use the server for many other tasks, but make it capable of handling a high load, I'd think of:

```
Available to webserver RAM: 400Mb
Child's memory size bounded: 10Mb
MaxClients: 400/10 = 40
StartServers: 5
MinSpareServers: 5
MaxSpareServers: 10
```

(These numbers are taken off the top of my head, and it shouldn't be used as a rule, but rather as examples to show you some possible scenarios. Use this information wisely! Also no sharing was counted, which makes things much better)

4.10.6 Summary of Benchmarking to tune all 5 parameters

OK, we've run various benchmarks -- let's summarize the conclusions:

- **MaxRequestsPerChild**

If your scripts are clean and don't leak memory, set this variable to a number as large as possible (10000?).

If you use **Apache::SizeLimit**, you can set this parameter to 0 (equal to infinity).

You will want this parameter to be smaller if your code becomes unshared over the process' life.

- **StartServers**

If you keep a small number of servers active most of the time, keep this number low.

Especially if **MaxSpareServers** is low as it'll kill the just loaded servers before they were utilized at all (if there is no load).

If your service is heavily loaded, make this number close to **MaxClients** (and keep **MaxSpareServers** equal to **MaxClients** as well.)

- **MinSpareServers**

If your server performs other work besides web serving, make this low so the memory of unused children will be freed when there is no big load.

If your server's load varies (you get loads in bursts) and you want fast response for all clients at any time, you will want to make it high, so that new children will be respawned in advance and be waiting to handle bursts of requests.

- **MaxSpareServers**

The logic is the same as of **MinSpareServers** - low if you need the machine for other tasks, high if it's a dedicated web host and you want a minimal response delay.

- **MaxClients**

Not too low, so you don't get into a situation where clients are waiting for the server to start serving them (they might wait, but not for too long).

Do not set it too high, since if you get a high load and all requests will be immediately granted and served, your CPU will have a hard time keeping up, and if the child's size * number of running children is larger than the total available RAM, your server will start swapping (which will slow down everything, which in turn will make things even more slower, until eventually your machine will die).

It's important that you take pains to ensure that swapping does not normally happen.

Swap space is an emergency pool, not a resource to be used on a consistent basis.

If you are low on memory and you badly need it - buy it, memory is amazingly cheap these days.

But based on the test I conducted above, even if you have plenty of memory like I have (1Gb), increasing **MaxClients** sometimes will give you no speedup.

The more clients are running, the more CPU time will be required, the less CPU time slices each process will receive.

The response latency (the time to respond to a request) will grow, so you won't see the expected improvement.

The best approach is to find the minimum requirement for your kind of service and the maximum capability of your machine.

Then start at the minimum and test like I did, successively raising this parameter until you find the point on the curve of the graph of the latency or/and throughput where the improvement becomes smaller.

Stop there and use it. Of course when you use these parameters in production server, you will have the ability to tune them more precisely, since then you will see the real numbers.

Also don't forget that if you add more scripts, or just modify the running ones -- most probably that the parameters need to be recalculated, since the processes will grow in size as you compile in more code.

4.11 Using `$|=1` under `mod_perl` and better `print()` techniques.

As you know local `$|=1`; disables the buffering of the currently selected file handle (default is **STDOUT**).

If you enable it, `ap_rflush()` is called after each `print()`, unbuffering Apache's IO.

If you are using a `_bad_` style in generating output, which consist of multiple `print()` calls, or you just have too many of them, you will experience a degradation in performance.

The severity depends on the number of the calls you make.

Many old CGIs were written in the style of:

```
print "<BODY BGCOLOR=\"black\" TEXT=\"white\">";
print "<H1>";
print "Hello";
print "</H1>";
print "<A HREF=\"foo.html\"> foo </A>";
print "</BODY>";
```

which reveals the following drawbacks: multiple **print()** calls - performance degradation with **\$|=1**, backslashism which makes the code less readable and more difficult to format the HTML to be easily readable as CGI's output.

The code below solves them all:

```
print qq{
  <BODY BGCOLOR="black" TEXT="white">
    <H1>
      Hello
    </H1>
    <A HREF="foo.html"> foo </A>
  </BODY>
};
```

Now let's go back to the `$|=1` topic.

I still disable buffering, for 2 reasons:

- I use few `print()` calls by printing out multiline HTML and not a line per `print()`

- and I want my users to see the output immediately.

So if I am about to produce the results of the DB query, which might take some time to complete, I want users to get some titles ahead.

This improves the usability of my site.

Recall yourself: What do you like better: getting the output a bit slower, but steadily from the moment you've pressed the **Submit** button or having to watch the “falling stars” for awhile and then to receive the whole output at once, even a few milliseconds faster (if the client (browser) did not time out till then).

Conclusion: Do not blindly follow suggestions, but think what is best for you in every given case.

4.12 Profiling

Profiling process helps you to determine which subroutines or just snippets of code take the longest execution time and which subroutines are being called most often.

Probably you will want to optimize those, and to improve the code toward efficiency.

It is possible to profile code running under `mod_perl` with the **Devel::DProf** module, available on CPAN.

However, you must have apache version 1.3b3 or higher and the **PerlChildExitHandler** enabled (during the `httpd` build process).

When the server is started, `Devel::DProf` installs an **END** block to write the **tmon.out** file.

This block will be called at the server shutdown. Here is how to start and stop a server with the profiler enabled:

```
% setenv PERL5OPT -d:DProf
% httpd -X -d `pwd` &
... make some requests to the server here ...
% kill `cat logs/httpd.pid`
% unsetenv PERL5OPT
% dprofpp
```

The **Devel::DProf** package is a Perl code profiler.

It will collect information on the execution time of a Perl script and of the subs in that script (remember that **print()** and **map()** are just like any other subroutines you write, but they are come bundled with Perl!)

Another approach is to use **Apache::DProf**, which hooks **Devel::DProf** into `mod_perl`.

The **Apache::DProf** module will run a **Devel::DProf** profiler inside each child server and write the **tmon.out** file in the directory **\$ServerRoot/logs/dprof/\$\$** when the child is shutdown (where **\$\$** is a number of the child process).

All it takes is to add to **httpd.conf**:

```
PerlModule Apache::DProf
```

Remember that any PerlHandler that was pulled in before **Apache::DProf** in the **httpd.conf** or `<startup.pl>`, would not have its code debugging info inserted. To run **dprofpp**, `chdir` to **\$ServerRoot/logs/dprof/\$\$** and run:

% dprofpp

4.13 Sending plain HTML as a compressed output

4.13.1 Apache::GzipChain - compress HTML (or anything) in the OutputChain

Have you ever served a huge HTML file (e.g. a file bloated with JavaScript code) and wondered how could you send it compressed, thus dramatically cutting down the download times.

After all, java applets can be compressed into a jar and benefit from a faster download times. Why cannot we do the same with a plain ASCII (HTML,JS and etc), it is a known fact that ASCII text can be compressed by a factor of 10.

Apache::GzipChain comes to help you with this task.

If a client (browser) understands **gzip** encoding this module compresses the output and sends it downstream.

A client decompresses the data upon receive and renders the HTML as if it was a plain HTML fetch.

For example to compress all html files on the fly, do:

```
<Files *.html>
  SetHandler perl-script
  PerlHandler Apache::OutputChain
      Apache::GzipChain Apache::PassFile
</Files>
```

Remember that it will work only if the browser claims to accept compressed input, thru **Accept-Encoding** header.

Apache::GzipChain keeps a list of user-agents, thus it also looks at **User-Agent** header, for known to accept compressed output browsers.

For example if you want to return compressed files which should pass in addition through Embperl module, you would write:

```
<Location /test>
    SetHandler perl-script
    PerlHandler Apache::OutputChain
        Apache::GzipChain Apache::EmbperlChain
    Apache::PassFile
</Location>
```

Hint: Watch an **access_log** file to see how many bytes were actually send, compare with a regular configuration send.

(See **perldoc Apache::GzipChain**).

;o)

5 Choosing the Right Strategy

5.1 Do it like me?!

There are too many technologies out there to choose from, and it would take an enormous investment of time and money to try to validate each one before deciding which is the best choice for your situation.

Keeping this idea in mind, I will present some different combinations of `mod_perl` and other technologies or just standalone `mod_perl`.

I'll describe how these things work together, and offer my opinions on the pros and cons of each, the relative degree of difficulty in installing and maintaining them, some hints on approaches that should be used and things to avoid.

To be clear, I will not address all technologies and tools, but limit this discussion to those complementing `mod_perl`.

5.2 mod_perl Deployment Overview

There are several different ways to build, configure and deploy your mod_perl enabled server. Some of them are:

1. Having one binary and one config file (one big binary for mod_perl).
2. Having two binaries and two config files (one big binary for mod_perl and one small for static objects like images.)
3. Having one DSO-style binary, mod_perl loadable object and two config files (Dynamic linking lets you compile once and have a big and a small binary in memory BUT you have to deal with a freshly made solution that has weak

documentation and is still subject to change and is rather more complex.)

4. Any of the above plus a reverse proxy server in http accelerator mode.

If you are a newbie, I would recommend that you start with the first option and work on getting your feet wet with apache and mod_perl.

1. The first option will kill your production site if you serve a lot of static data with ~2-12 MB webserver processes. On the other hand, while testing you will have no other server interaction to mask or add to your errors.
2. The second option allows you to seriously tune the two servers for maximum performance. On the other hand you have to deal with proxying or fancy site design to keep the

two servers in synchronization. In this configuration, you also need to choose between running the two servers on multiple ports, multiple IPs, etc... This adds the burden of administrating more than one server.

3. The third option (DSO) -- as mentioned above -- means playing with the bleeding edge. In addition **mod_so** (the DSO module) adds size and complexity to your binaries. With DSO, modules can be added and removed without recompiling the server, and modules are even shared among multiple servers. Again, it is bleeding edge and still somewhat platform specific, but your mileage may vary.
4. The fourth option (proxy in http accelerator mode), once correctly configured and tuned, improves the performance of any of the above three options by caching and buffering page results.

The rest of this section discusses the pros and the cons of each of these presented configurations.

5.3 Standalone mod_perl Enabled Apache Server

The first approach is to implement a straightforward mod_perl server.

The advantages:

- Simplicity. You just follow the installation instructions, configure it, restart the server and you are done.
- No network changes. You do not have to worry about using additional ports as we will see later.

- Speed. You get a very fast server, you see an enormous speedup from the first moment you start to use it.

The disadvantages:

- mod_perl-enabled Apache processes are huge (4Mb to 10Mb and more). (Less if memory sharing is in place)

You should be very worried about having mod_perl processes serve static objects such as images and html files.

Each static request served by a mod_perl-enabled server means another large process running, competing for system resources such as memory and CPU cycles. The real overhead depends on static objects request rate.

Remember that if your `mod_perl` code produces HTML code which includes images, each one will turn into another static object request.

- Another drawback of this approach is that when serving output to a client with a slow connection. The huge `mod_perl`-enabled server process will be tied up until the response is completely written to the client.

While it might take a few milliseconds for your script to complete the request, there is a chance it will be still busy for some number of seconds or even minutes if the request is from a slow connection client.

Not that this is not disadvantage if all users are on a fast local Intranet. Still, remember that some of your Intranet users might work from home through the slow modem links.

If you are new to mod_perl, this is probably the best way to get yourself started.

And of course, if your site is serving only mod_perl scripts (close to zero static objects, like images), this might be the perfect choice for you!

5.4 One Plain and One mod_perl-enabled Apache Servers

A better approach is to run two servers: a very light, plain apache server to serve static objects and a heavier mod_perl-enabled apache server to serve requests for dynamic (generated) objects (aka CGI).

From here on, I will refer to these two servers as **httpd_docs** (vanilla apache) and **httpd_perl** (mod_perl enabled apache).

The advantages:

- The heavy `mod_perl` processes serve only dynamic requests, which allows the deployment of fewer of these large servers.
- **MaxClients**, `MaxRequestsPerChild` and related parameters can now be optimally tuned for both **httpd_docs** and `httpd_perl` servers, something we could not do before. This allows us to fine tune the memory usage and get a better server performance.

Now we can run many lightweight **httpd_docs** servers and just a few heavy **httpd_perl** servers.

An **important** note: When user browses static pages and the base URL in the **Location** window points to the static server, for example **`http://www.nowhere.com/index.html`** -- all relative URLs (e.g. ``) are being served by the light plain apache server.

But this is not the case with dynamically generated pages.

For example when the base URL in the **Location** window points to the dynamic server -- (e.g. **http://www.nowhere.com:8080/perl/index.pl**) all relative URLs in the dynamically generated HTML will be served by the heavy `mod_perl` processes.

You must use a fully qualified URLs and not the relative ones! **http://www.nowhere.com/icons/arrow.gif** is a full URL, while **/icons/arrow.gif** is a relative one.

Using One light non-Apache and One mod_perl enabled Apache Servers

If the only requirement from the light server is for it to serve static objects, then you can get away with non-apache servers having an even smaller memory footprint.

thttpd has been reported to be about 5 times faster than apache (especially under a heavy load), since it is very simple and uses almost no memory (260k) and does not spawn child processes.

The Advantages:

- All the advantages of the 2 servers scenario.
- More memory saving. Apache is about 4 times bigger than **thttpd**, if you spawn 30 children you use about 30M of memory, while **thttpd** uses only 260k - 100 times less! You could use the saved 30M to run more mod_perl servers.

Note that this is not true if your OS supports memory sharing and you configured apache to use it (it is a DSO approach). There is no memory sharing if apache modules are being statically compiled into httpd). If you do allow memory sharing -- 30 light apache servers ought to use about 3-4Mb only,

because most of it will be shared. If this is the case -- the save ups are much smaller with **thttpd**.

- Reported to be about 5 times faster than plain apache serving static objects.

The Disadvantages:

- Lacks some of apache's features, like access control, error redirection, customizable log file formats, and so on.

5.5 Adding a Proxy Server in http Accelerator Mode

At the beginning there were 2 servers: one - plain apache server, which was **very light**, and configured to serve static objects, the other -- mod_perl enabled, which was **very heavy** and aimed to serve mod_perl scripts.

We named them: **httpd_docs** and **httpd_perl** appropriately.

The two servers coexisted at the same IP (DNS) by listening to different ports: 80 -- for **httpd_docs** and 8080 -- for **httpd_perl**.

Now I am going to convince you that you **want** to use a proxy server (in the http accelerator mode).

The advantages are:

- Allow serving of static objects from the proxy's cache (objects that previously were entirely served by the httpd_docs server).
- You get less I/O activity reading static objects from the disk (proxy serves the most “popular” objects from the RAM memory - of course you benefit more if you allow the proxy server to consume more RAM).

Since you do not wait for the I/O to be completed you are able to serve the static objects much faster.

- The proxy server acts as a sort of output buffer for the dynamic content.

The mod_perl server sends the entire response to the proxy and is then free to deal with other requests.

The proxy server is responsible for sending the response to the browser.

So if the transfer is over a slow link, the mod_perl server is not waiting around for the data to move.

Let's take a user connected to your site with 28.8 kbps (bps == bits/sec) modem.

It means that the speed of the user's link is $28.8/8 = 3.6$ kbytes/sec.

I assume an average generated HTML page to be of 10kb (kb == kilobytes) and an average script that generates this output in 0.5 secs.

How much time will the server wait before the user gets the whole output response?

A simple calculation reveals pretty scary numbers - it will have to wait for another 6 secs (20kb/3.6kb), when it could serve another 12 (6/0.5) dynamic requests in this time.

But you know that nowadays scripts return pages which sometimes are being blown up with javascript code and similar, which makes them of 100kb size and download time to be of... (This calculation is left to you as an exercise :)

To make your download time numbers even worse, let me remind you that many users like to open many browser windows and do many things at once (download files and browse **heavy** sites). So the speed of 3.6kb/sec we were assuming before, may many times be 5-10 times slower.

- Proxy server helps us to hide the details of the server's implementation.

Users will never see ports in the URLs (more on that topic later).

And you can have a few boxes serving the requests, and only one serving as a front end, which spreads the jobs between the servers in a way you configured it too.

So you can actually put down one server down for upgrade, but end user will never notice that because the front end server will dispatch the jobs to other servers.

- For security reasons, using any httpd accelerator (or a proxy in httpd accelerator mode) is essential because you do not let your internal server get directly attacked by arbitrary packets from whomever.

The `httpd` accelerator and internal server communicate in expected HTTP requests. This allows for only your public “bastion” accelerating `www` server to get hosed in a successful attack, while leaving your internal data safe.

Of course there are drawbacks. Luckily, these are not functionality drawbacks, but more of administration hassle.

The disadvantages are:

- You add another daemon to worry about, and while proxies are generally stable, you have to make sure to prepare proper startup and shutdown scripts, which are being run at the boot and reboot appropriately.

Also, maybe a watchdog script running at the `crontab`.

- Proxy servers can be configured to be light or heavy, the admin must decide what gives the highest performance for his application.

A proxy server like squid is light in the concept of having only one process serving all requests. But it can appear pretty heavy when it loads objects into memory for faster service.

Have I succeeded in convincing you that you want the proxy server?

If you are on a local area network (LAN), then the big benefit of the proxy buffering the output and feeding a slow client is gone. You are probably better off sticking with a straight mod_perl server in this case.

As of this writing the two proxy implementations are known to be used in bundle with `mod_perl` - **Squid** proxy server and **mod_proxy** which is a part of the apache server.

Let's compare the two of them:

5.6 The Squid Server

The Advantages:

- Caching of static objects. So these are being served much faster assuming that your cache size is big enough to keep the most requested objects in the cache.
- Buffering of dynamic content, by taking the burden of returning the content generated by mod_perl servers to slow clients.
- Non-linear URL space / server setup. You can use Squid to play some tricks with the URL space and/or domain based virtual server support.

The Disadvantages:

- Proxying dynamic content is not going to help much if all the clients are on a fast local net.
- Also, a message on the squid mailing list implied that squid only buffers in 16k chunks so it would not allow a mod_perl to complete immediately if the output is larger.
- Speed. Squid is not very fast today when compared to plain file based web servers available.

Only if you are using a lot of dynamic features such as mod_perl or similar speed is a reason to use Squid, and then only if the application and server is designed with caching in mind.

- Memory usage. Squid uses quite a bit of memory.
- HTTP protocol level. Squid is pretty much a HTTP/1.0 server, which seriously limits the deployment of HTTP/1.1 features.
- HTTP headers / dates, freshness. The squid server might give out “old” pages, confusing downstream/client caches. Also chances are that you will be giving out stale pages.
- Stability. Compared to plain web servers Squid is not the most stable.

The presented pros and cons lead to an idea, that probably you might want squid more for its dynamic content buffering features, but only if your server serves mostly dynamic requests.

So in this situation it is better to have a plain apache server serving static objects, and squid proxying the mod_perl enabled server only. At least when performance is the goal.

5.7 An Apache's mod_proxy

I do not think the difference in speed between apache's **ProxyPass** and **squid** is relevant for most sites, since the real value of what they do is buffering for slow client connections.

However squid runs as a single process and probably consumes fewer system resources.

The trade-off is that mod_rewrite is easy to use if you want to spread parts of the site across different back end servers, and mod_proxy knows how to fix up redirects containing the back-end server's idea of the location.

With squid you can run a redirector process to proxy to more than one back end, but there is a problem in fixing redirects in a way that keeps the client's view of both server names and port

numbers in all cases.

The difficult case being where you have DNS aliases that map to the same IP address for an alias and you want the redirect to use port 80 (when the server is really on a different port) but you want it to keep the specific name the browser sent so it does not change in the client's **location** window.

The Advantages:

- No additional server is needed. We keep the one plain plus one mod_perl enabled apache servers. All you need is to enable the **mod_proxy** in the `httpd_docs` server and add a few lines to **httpd.conf** file.
- **ProxyPass** and `ProxyPassReverse` directives allow you to hide the internal redirects, so if **http://nowhere.com/modperl/** is actually

http://localhost:81/modperl/, it will be absolutely transparent for user.

ProxyPass redirects the request to the mod_perl server, and when it gets the respond, **ProxyPassReverse** rewrites the URL back to the original one, e.g:

```
ProxyPass      /modperl/ http://localhost:81/modperl/  
ProxyPassReverse /modperl/ http://localhost:81/modperl/
```

- It does mod_perl output buffering, like squid does.
- It even does caching. You have to produce correct **Content-Length**, **Last-Modified** and **Expires** http headers for it to work.

If some dynamic content is not to change constantly, you can dramatically increase performance by caching it with **ProxyPass**.

- **ProxyPass** happens before the authentication phase, so you do not have to worry about authenticating twice.
- Apache is able to accel https (secure) requests completely, while also doing http accel. (with squid you have to use an external redirection program for that).
- The latest (from apache 1.3.6) Apache proxy accel mode reported to be very stable.

The Disadvantages:

- Users reported that it might be a bit slow, but the latest version is fast enough.

;o)

6 Real World Scenarios Implementation

6.1 Standalone mod_perl Enabled Apache Server

6.1.1 Installation in 10 lines

The Installation is very very simple (example of installation on Linux OS):

```
% cd /usr/src
% lwp-download http://www.apache.org/dist/apache_x.x.x.tar.gz
% lwp-download http://perl.apache.org/dist/mod_perl-x.xx.tar.gz
% tar zvxf apache_x.xx.tar.gz
% tar zvxf mod_perl-x.xx.tar.gz
% cd mod_perl-x.xx
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
  DO_HTTPD=1 USE_APACI=1 PERL_MARK_WHERE=1 EVERYTHING=1
% make && make test && make install
% cd ../apache_x.x.x/src
% make install
```

That's all!

6.1.2 Installation in 10 paragraphs

This section goes over the details I was just talking about.

This page was left here in attempt to stay in sync with the section numbers in your handouts.

6.1.3 Configuration Process

After checking that your basic Apache works, add to the `httpd.conf` assuming that `/home/httpd/perl/` is a home of all our scripts:

```
Alias /perl/ /home/httpd/perl/  
  
PerlModule Apache::Registry  
<Location /perl>  
    SetHandler perl-script  
    PerlHandler Apache::Registry  
    Options ExecCGI  
    PerlSendHeader On  
    allow from all  
</Location>
```

Create a test script `/home/httpd/perl/test.pl`

```
#!/usr/bin/perl -w
use strict;
print "Content-type: text/html\r\n\r\n";
print "It worked!!!\n";
```

Make it executable and readable by server.

```
% chown nobody /home/httpd/perl/test.pl
% chmod u+rx /home/httpd/perl/test.pl
```

First test it:

```
% /home/httpd/perl/test.pl
```

You should see the following output:

```
Content-type: text/html
```

It worked!!!

Real testing by calling from a browser:

```
http://localhost/perl/test.pl
```

Make sure that you have a loop-back device configured, if not -- use the real server name for this test, for example:

```
http://www.nowhere.com/perl/test.pl
```

You should see:

It worked!!!

Now we can move the CGI scripts to **/home/httpd/perl/** and start testing them.

Some of your scripts will not work out of box and will demand some minor tweaking or major rewrite to make them work properly with mod_perl enabled server.

Chances are that if you are not practicing a sloppy programming techniques -- the scripts will work without any modifications at all.

The above setup is very basic, it will help you to get your feet wet with mod_perl enabled server.

6.2 One Plain and One mod_perl enabled Apache Servers

Since we are going to run two apache servers we will need two different sets of configuration, log and other files. We need a special directory layout.

From now on I will refer to these two servers as **httpd_docs** (vanilla Apache) and **httpd_perl** (Apache/mod_perl).

For this illustration, we will use **/usr/local** as our *root* directory. The Apache installation directories will be stored under this root (**/usr/local/bin**, **/usr/local/etc** and etc...)

First let's prepare the sources. We will assume that all the sources go into **/usr/src** dir.

It is better when you use two separate copies of apache sources. Since you probably will want to tune each apache version at separate and to do some modifications and recompilations as the time goes.

Make two subdirectories:

```
% mkdir /usr/src/httpd_docs  
% mkdir /usr/src/httpd_perl
```

Put the Apache sources into a **/usr/src/httpd_docs** directory:

```
% cd /usr/src/httpd_docs  
% tar xvzf /tmp/apache_x.x.x.tar.gz  
% ls -l  
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 apache_x.x.x/
```

Now we will prepare the **httpd_perl** server sources:

```
% cd /usr/src/httpd_perl
% tar xvzf /tmp/apache_x.x.x.tar.gz
% tar xvzf /tmp/modperl-x.xx.tar.gz
% ls -l
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 apache_x.x.x/
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 modperl-x.xx/
```

Time to decide on the desired directory structure layout (where the apache files go):

```
ROOT = /usr/local
```

The two servers can share the following directories (so we will not duplicate data):

```
/usr/local/bin/  
/usr/local/lib  
/usr/local/include/  
/usr/local/man/  
/usr/local/share/
```

Important: we assume that both servers are built from the same Apache source version.

Servers store their specific files either in **httpd_docs** or **httpd_perl** sub-directories:

```
/usr/local/etc/httpd_docs/  
    httpd_perl/  
  
/usr/local/sbin/httpd_docs/  
    httpd_perl/
```

```
/usr/local/var/httpd_docs/logs/  
proxy/  
run/  
httpd_perl/logs/  
proxy/  
run/
```

After completion of the compilation and the installation of the both servers, you will need to configure them.

To make things clear before we proceed into details, you should configure the **/usr/local/etc/httpd_docs/httpd.conf** as a plain apache and **Port** directive to be 80 for example.

And **/usr/local/etc/httpd_perl/httpd.conf** to configure for mod_perl server and of course whose **Port** should be different from the one **httpd_docs** server listens to (e.g. 8080).

6.2.1 Configuration and Compilation of the Sources.

Let's proceed with installation. I will use x.x.x instead of real version numbers so these slides will never become obsolete :).

6.2.1.1 Building the httpd_docs Server

Sources Configuration:

```
% cd /usr/src/httpd_docs/apache_x.x.x
% make clean
% env CC=gcc \
./configure --prefix=/usr/local \
--sbindir=/usr/local/sbin/httpd_docs \
--sysconfdir=/usr/local/etc/httpd_docs \
--localstatedir=/usr/local/var/httpd_docs \
--runtimedir=/usr/local/var/httpd_docs/run \
--logfiledir=/usr/local/var/httpd_docs/logs \
--proxycachedir=/usr/local/var/httpd_docs/proxy
```

If you need some other modules, like `mod_rewrite` and `mod_include` (SSI), add them here as well:

```
--enable-module=include \  
--enable-module=rewrite
```

Note: add `--layout` to see the resulting directories' layout without actually running the configuration process.

Sources Compilation:

```
% make  
% make install
```

Rename `httpd` to `http_docs`

```
% mv /usr/local/sbin/httpd_docs/httpd \  
/usr/local/sbin/httpd_docs/httpd_docs
```

Now update an **apachectl** utility to point to the renamed httpd via your favorite text editor or by using perl:

```
% perl -p -i -e \  
's|httpd_docs/httpd|httpd_docs/httpd_docs|' \  
/usr/local/sbin/httpd_docs/apachectl
```

6.2.1.2 Building the httpd_perl (mod_perl enabled) Server

First, install the required modules. **perl Makefile.PL** alerts you about the missing ones. Get them from <http://www.perl.com/CPAN> or use CPAN.pm.

```
% cd /usr/src/httpd_perl/apache_x.x.x
% make clean
% cd /usr/src/httpd_perl/mod_perl-x.xx
% make clean
```

```
% /usr/local/bin/perl Makefile.PL \  
APACHE_PREFIX=/usr/local/ \  
APACHE_SRC=../apache_x.x.x/src \  
DO_HTTPD=1 \  
USE_APACI=1 \  
PERL_MARK_WHERE=1 \  
PERL_STACKED_HANDLERS=1 \  
ALL_HOOKS=1 \  
APACI_ARGS=--sbindir=/usr/local/sbin/httpd_perl, \  
--sysconfdir=/usr/local/etc/httpd_perl, \  
--localstatedir=/usr/local/var/httpd_perl, \  
--runtimedir=/usr/local/var/httpd_perl/run, \  
--logfiledir=/usr/local/var/httpd_perl/logs, \  
--proxycachedir=/usr/local/var/httpd_perl/proxy
```

As with **httpd_docs** you might need other modules like **mod_rewrite**, so add them here:

```
--enable-module=rewrite
```

Note: **PERL_STACKED_HANDLERS=1** is needed for **Apache::DBI**

Now, build, test and install the **httpd_perl**.

```
% make && make test && make install
```

We did not run **make install** in the apache's source directory. When **USE_APACI** is enabled, **APACHE_PREFIX** will specify the **--prefix** option for apache's **configure** utility, specifying the installation path for apache.

When this option is used, **mod_perl's make install** will also **make install** on the apache side, installing the **httpd** binary, support tools, along with the configuration, log and document trees.

If **make test** fails, look into **t/logs** and see what is in there.

Now rename the **httpd** to **httpd_perl**:

```
% mv /usr/local/sbin/httpd_perl/httpd \  
/usr/local/sbin/httpd_perl/httpd_perl
```

Update the **apachectl** utility to point to renamed **httpd** name:

```
% perl -p -i -e \  
's|httpd_perl/httpd|httpd_perl/httpd_perl|' \  
/usr/local/sbin/httpd_perl/apachectl
```

6.2.2 Configuration of the servers

Now when we have completed the building process, the last stage before running the servers, is to configure them.

6.2.2.1 Basic httpd_docs Server's Configuration

Configure the `/usr/local/etc/httpd_docs/httpd.conf` (plain apache).

Make sure you configure the log files and other paths according to the directory layout we decided to use.

Start the server with:

```
/usr/local/sbin/httpd_docs/apachectl start
```

6.2.2.2 Basic httpd_perl Server's Configuration

Now edit the `/usr/local/etc/httpd_perl/httpd.conf` file. We configure for the plain Apache, as with `http_docs`, but:

```
Port 8080
```

since port 80 is already taken by `http_docs`.

mod_perl specific directives:

```
# mod_perl scripts will be called from  
Alias /perl/ /usr/local/myproject/perl/
```

```
PerlModule Apache::Registry
<Location /perl>
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options ExecCGI
    PerlSendHeader On
</Location>
```

If you omit **ExecCGI** option, the script will be printed to the user's browser as a plain text or will possibly trigger a '**Save-As**' window.

PerlSendHeader On sends an HTTP header to the browser. Turn it off for **nph** (non-parsed-headers) scripts.

Now start the server with:

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

6.3 Running 2 webservers and squid in httpd accelerator mode

First, install Squid. On Linux, it's just a matter of installing of rpm package and configuring `/etc/squid/squid.conf`.

Second, lets understand what do we have in hands and what do we want from squid:

- A single machine.
- `httpd_docs` and `httpd_perl` servers are listening on ports 81 and 8080 accordingly.

- Squid is listening on port 80, forwarding static object requests to **httpd_docs**, and dynamic requests to **httpd_perl**.
- Both servers return the data to the proxy server (unless it is already cached in the squid), so user never sees the other ports and never knows that there might be more than one server running.
- Proxy server makes all the magic behind it transparent to user. Do not confuse it with **mod_rewrite**, where a server redirects the request somewhere according to the rules and forgets about it.
- The described functionality is being known as **httpd accelerator mode** in proxy dialect.

Since we want to stay on the mod_perl course, squid configuration is quite off-topic. You will find a detailed explanation of the squid configuration in the handouts. The only thing I'll talk about is a redirector daemon (we aren't skipping perl code :). Here is an example of a simple one:

```
#!/usr/local/bin/perl
$|=1;
while (<>) {
    # redirect to mod_perl server
    print($_),
        next if s|(:81)?/perl/|:8080/perl/|o;

    # send it unchanged to plain apache
    print;
}
```

A few notes regarding redirector script:

You must disable buffering. `$|=1`; does the job. If you do not disable buffering, the **STDOUT** will be flushed only when the buffer becomes full and its default size is about 4096 characters.

So if you have an average URL of 70 chars, only after 59 (4096/70) requests the buffer will be flushed, and the requests will finally achieve the server in target. Your users will just wait till it will be filled up.

The daemon is quite effective, since it runs as a daemon. Exactly like `mod_perl --perl` is loaded all the time and the code was already compiled, so redirect is very fast (not slower if redirector was written in C or alike). Squid keeps an open pipe to each redirect daemon, thus there is even no overhead of the expensive system calls.

Now it is time to restart the server, at linux I do it with:

```
/etc/rc.d/init.d/squid restart
```

Now the setup is complete ...

Almost... When you try the new setup, you will be surprised and upset to discover the port 81 showing up in the URLs of the static objects (like htmls).

Hey, we did not want the user to see the port 81 and use it instead of 80, since than it will bypass the squid server and the hard work we went through was just a waste of time?

The solution is to run both squid and httpd_docs on the same port. This can be accomplished by binding each one to a specific interface.

Again you will find the details in your handouts :)

6.4 Running 1 webservers and squid in httpd accelerator mode

Why do we need the light httpd_docs server, when we have squid?

- An overhead of loading the objects into squid at first time.
- Unless a huge chunk of memory is devoted to squid, not all of them will be cached.

Result: mod_perl servers will have an overhead of serving the static objects.

That's why I decided to have even more administration overhead and to stick with squid, httpd_docs and httpd_perl scenario, where I can optimize and fine tune everything.

If you are feeling that the scenario from the previous section is too complicated for you, make it simpler. Have only one server with mod_perl built in and let the squid to do most of the job that plain light apache used to do.

Pick this setup only if you can make squid cache most of your static objects. If it cannot, your mod_perl server will do the work we do not want it to.

What's different:

- **httpd_docs** is not here anymore.

- No need for the redirector.

- No need to bind two servers on the same port, so the squid configuration is simpler.

Just like before, you will copy and paste the squid config file from your handouts, when you will actually install it.

6.5 Using mod_proxy

Now we will talk about apache's mod_proxy and understand how it works.

The server on port 80 answers http requests directly and proxies the mod_perl enabled server in the following way:

```
ProxyPass /modperl/ http://localhost:81/modperl/  
ProxyPassReverse /modperl/ http://localhost:81/modperl/
```

PPR is the saving grace here, that makes apache a win over Squid. It rewrites the redirect on its way back to the original URI.

You can control the buffering feature with

ProxyReceiveBufferSize directive:

ProxyReceiveBufferSize 1048576

The above setting will set a buffer size to be of 1Mb.

To get an immediate release of the mod_perl server from stale awaiting, **ProxyReceiveBufferSize** should be set to a value greater than the biggest generated respond produced by any mod_perl script.

As the name states, its buffering feature applies only to **downstream data** (coming from the origin server to the proxy) and not upstream (i.e. file uploads buffering)

Apache does caching as well. It's relevant to mod_perl only if you produce proper headers, so your scripts' output can be cached. See apache documentation for more details on configuration of this capability.

Ask Bjoern Hansen wrote a **mod_proxy_add_forward** module for apache, that sets the **X-Forwarded-For** field when doing a **ProxyPass**, similar to what squid can do.

Basically, that module adds an extra HTTP header to proxying requests. You can access that header in the `mod_perl-enabled` server, and set the IP of the remote server.

6.6 mod_perl server as DSO

To build the mod_perl as DSO add **USE_DSO=1** to the rest of configuration parameters (to build **libperl.so** instead of **libperl.a**), like:

```
perl Makefile.PL USE_DSO=1 ...
```

If you run **./configure** from apache source do not forget to add:
--enable-shared=perl

Then just add the **LoadModule** directive into your **httpd.conf**.

You will find a complete explanation in the **INSTALL.apaci** pod which can be found in the mod_perl distribution.

Some people reported that DSO compiled mod_perl would not run on specific OS/perl version. Also threads enabled perl reported sometimes to break the mod_perl/DSO. But it still can work for you.

```
;o)
```

7 Server Configuration

7.1 mod_perl Specific Configuration

The next step after building and installing your new mod_perl enabled apache server, is to configure the server.

Before you start with mod_perl specific configuration, first configure apache, and see that it works. When done, return here to continue...

7.1.1 *Alias Configurations*

First, you need to specify the locations on a file-system for the scripts to be found.

Add the following configuration directives:

```
# for plain cgi-bin:
scriptAlias /cgi-bin/ /usr/local/myproject/cgi/

# for Apache::Registry mode
Alias /perl/ /usr/local/myproject/cgi/

# Apache::PerlRun mode
Alias /cgi-perl/ /usr/local/myproject/cgi/
```

Alias provides a mapping of URL to file system object under **mod_perl**. **ScriptAlias** is being used for **mod_cgi**.

Alias defines the start of the URL path to the script you are referencing.

For example, using the above configuration, fetching **http://www.nowhere.com/perl/test.pl**, will cause the server to look for the file **test.pl** at **/usr/local/myproject/cgi**, and execute it as an **Apache::Registry** script if we define **Apache::Registry** to be the handler of **/perl** location (see below).

The URL **http://www.nowhere.com/perl/test.pl** will be mapped to **/usr/local/myproject/cgi/test.pl**.

This means that you can have all your CGIs located at the same place at the file-system, and call the script in any of three modes simply by changing the directory name component of the URL

(cgi-bin|perl|cgi-perl) - is not this neat?

(That is the configuration you see above - all three Aliases point to the same directory within your file system, but of course they can be different).

If your script does not seem to be working while running under mod_perl, you can easily call the script in straight mod_cgi mode without making any script changes (in most cases), but rather by changing the URL you invoke it by.

FYI: for modperl **ScriptAlias** is the same thing as:

```
Alias /foo/ /path/to/foo/  
SetHandler cgi-handler
```

where **SetHandler cgi-handler** invokes `mod_cgi`.

The latter will be overwritten if you enable **Apache::Registry**.

In other words, **ScriptAlias** does not work for `mod_perl`, it only appears to work when the additional configuration is in there.

If the **Apache::Registry** configuration came before the **ScriptAlias**, scripts would be run under `mod_cgi`.

While handy, **ScriptAlias** is a known kludge, always better to use **Alias** and **SetHandler**.

Of course you can choose any other **Alias** (you will use it later in **httpd.conf**), you can choose to use all three modes or only one of these.

It is undesirable to run scripts in plain mod_cgi from a mod_perl-enabled server - the price is too high, it is better to run these on plain apache server.

7.1.2 Location Configuration

Now we will work with the `httpd.conf` file. I add all the `mod_perl` stuff at the end of the file, after the native apache configurations.

First we add:

```
<Location /perl>  
    SetHandler perl-script  
    PerlHandler Apache::Registry  
    Options ExecCGI  
    allow from all  
    PerlSendHeader On  
</Location>
```

This configuration causes all scripts that are called with a `/perl` path prefix to be executed under the **Apache::Registry** module and as a CGI (so the **ExecCGI**, if you omit this option the script will be printed to the caller's browser as a plain text or possibly will trigger a 'Save-As' window).

You have nothing to do about **/cgi-bin** location (`mod_cgi`), since it has nothing to do with `mod_perl`.

Here is a similar location configuration for **Apache::PerlRun**:

```
<Location /cgi-perl>
    SetHandler perl-script
    PerlHandler Apache::PerlRun
    Options ExecCGI
    allow from all
    PerlSendHeader On
</Location>
```

7.1.3 *PerlFreshRestart*

To reload `PerlRequire`, `PerlModule`, other `use()`'d modules and flush the `Apache::Registry` cache on server restart, add:

```
PerlFreshRestart On
```

7.1.4 /perl-status location

A very useful feature. You can watch what happens to the perl guts of the server. Below you will find the instructions of configuration and usage of this feature

7.1.4.1 /perl-status configuration

Add this to `httpd.conf`:

```
<Location /perl-status>
    SetHandler perl-script
    PerlHandler Apache::Status
    order deny,allow
    #deny from all
    #allow from
</Location>
```

If you are going to use **Apache::Status**, it's important to put it as a first module in the start-up file, or in the `httpd.conf` (after **Apache::Registry**):

```
# startup.pl
use Apache::Registry ();
use Apache::Status ();
use Apache::DBI ();
```

If you don't put **Apache::Status** before **Apache::DBI** then you don't get **Apache::DBI**'s menu entry in status.

7.1.4.2 Usage

Assuming that your mod_perl server listens to port 81, fetch <http://www.nowhere.com:81/perl-status>

```
Embedded Perl version 5.00502 for Apache/1.3.2 (Unix) mod_perl/1.16  
process 187138, running since Thu Nov 19 09:50:33 1998
```

This is the linked menu that you should see:

- Signal Handlers**
- Enabled mod_perl Hooks**
- PerlRequire'd Files**
- Environment**
- Perl Section Configuration**
- Loaded Modules**
- Perl Configuration**

ISA Tree
Inherited Tree
Compiled Registry Scripts
Symbol Table Dump

Let's follow for example : **PerlRequire'd Files** -- we see:

PerlRequire	Location
<code>/usr/myproject/lib/apache-startup.pl</code>	<code>/usr/myproject/lib/apache-startup.pl</code>

From some menus you can continue deeper to peek at the perl internals of the server, to watch the values of the global variables in the packages, to the list of cached scripts and modules and much more. Just click around...

7.1.4.3 Compiled Registry Scripts section seems to be empty.

Sometimes when you fetch `/perl-status` you and follow the **Compiled Registry Scripts** link from the status menu -- you see no listing of scripts at all.

This is absolutely correct -- **Apache::Status** shows the registry scripts compiled in the httpd child which is serving your request for `/perl-status`.

If a child has not compiled yet the script you are asking for, `/perl-status` will just show you the main menu.

This usually happens when the child was just spawned.

7.1.5 *PerlSetVar*, *PerlSetEnv* and *PerlPassEnv*

```
PerlSetEnv key val  
PerlPassEnv key
```

PerlPassEnv passes, **PerlSetEnv** sets and passes the *ENV* variables to your scripts. you can access them in your scripts through **%ENV** (e.g. **\$ENV{"key"}**).

PerlSetVar is very similar to **PerlSetEnv**, but you extract it with another method. In `<Perl>` sections:

```
push @{$Location{"/"}}->{PerlSetVar} }, [ 'FOO' => BAR ];
```

and in the code you read it with:

```
my $r = Apache->request;  
print $r->dir_config( 'FOO' );
```

7.1.6 *perl-startup file*

Since many times you have to add many perl directives to the configuration file, it can be a good idea to put all of these into a one file, so the configuration file will be cleaner.

Add the following line to **httpd.conf**:

```
# startup.perl loads all functions that we want to use within
# mod_perl
PerlRequire /path/to/startup.pl
```

before the rest of the mod_perl configuration directives.

Also you can call **perl -c perl-startup** to test the file's syntax.
What does this take?

7.1.6.1 Sample perl-startup file

An example of perl-startup file:

```
use strict;

# extend @INC if needed
use lib qw(/dir/foo /dir/bar);

# make sure we are in a sane environment.
$ENV{GATEWAY_INTERFACE} =~ /^CGI-Perl/
    or die "GATEWAY_INTERFACE not Perl!";

# for things in the "/perl" URL
use Apache::Registry;

#load perl modules of your choice here
#this code is interpreted *once* when the server starts
use LWP::UserAgent ();
use DBI ();
```

```
# tell me more about warnings
use Carp ();
$SIG{__WARN__} = \&Carp::cluck;
```

```
# Load CGI.pm and call its compile() method to precompile
# (but not to import) its autoloading methods.
use CGI ();
CGI->compile(':all');
```

Note that starting with **\$CGI::VERSION 2.46**, the recommended method to precompile the code in **CGI.pm** is:

```
use CGI qw(-compile :all);
```

But the old method is still available for backward compatibility.

7.1.6.2 What modules should you add to the startup file and why.

Modules that are being loaded at the server startup will be shared among server children, so only one copy of each module will be loaded, thus saving a lot of RAM for you.

Usually I put most of the code I develop into modules and preload them from here.

You can even preload your CGI script with **Apache::RegistryLoader** and preopen the DB connections with **Apache::DBI**.

7.1.6.3 The confusion with `use()` clause at the server startup?

Many people wonder, why there is a need for duplication of `use()` clause both in startup file and in the script itself.

The question rises from misunderstanding of the `use()` operand. `use()` consists of two other operands, namely `require()` and `import()`. So when you write:

```
use Foo qw(bar);
```

perl actually does:

```
require Foo.pm;  
import qw(bar);
```

When you write:

```
use Foo qw( );
```

perl actually does:

```
require Foo.pm;  
import qw( );
```

which means that the caller does not want any symbols to be imported.

Why is this important? Since some modules has **@EXPORT** set to a list of tags to be exported by default and when you write:

```
use Foo;
```

and think nothing is being imported, the **import()** call is being executed and probably some symbols do being imported.

See the docs/source of the module in question to make sure you **use()** it correctly.

When you write your own modules, always remember that it's better to use **@EXPORT_OK** instead of **@EXPORT**, since the former doesn't export tags unless it was asked to.

Since the symbols that you might import into a startup's script namespace will be visible by none of the children, scripts that need a **Foo**'s module exported tags have to pull it in like if you did not preload **Foo** at the startup file.

For example, just because you have **use()**d **Apache::Constants** in the startup script, does not mean you can have the following handler:

```
package MyModule;

sub {
    my $r = shift;

    ## Cool stuff goes here

    return OK;
}
1;
```

You would either need to add:

```
use Apache::Constants qw( OK );
```

Or instead of **return OK**; say:

```
return Apache::Constants::OK;
```

See the manpage/perldoc on **Exporter** and **perlmod** for more on **import()**.

7.1.6.4 The confusion with defining globals in startup

PerlRequire allows you to execute code that preloads modules and does more things.

Imported or defined variables are visible in the scope of the startup file.

It is a wrong assumption that global variables that were defined in the startup file, will be accessible by child processes.

You do have to define/import variables in your scripts and they will be visible inside a child process who run this script.

They will be not shared between siblings.

Remember that every script is running in a specially (uniquely) named package - so it cannot access variables from other packages unless it inherits from them or **use()**'s them.

7.2 Running 'apachectl configtest' or 'httpd -t'

apachectl configtest tests the configuration file without starting the server.

You can safely modify the configuration file on your production server, if you run this test before you restart the server.

Of course it is not 100% error prone, but it will reveal any syntax errors you might do while editing the file.

'**apachectl configtest**' is the same as '**httpd -t**' and it actually executes the code in startup.pl, not just parses it.

Perl Sections

With `<Perl></Perl>` sections, it is possible to configure your server entirely in Perl.

`<Perl>` sections can contain `*any*` and as much Perl code as you wish.

These sections are compiled into a special package whose symbol table `mod_perl` can then walk and grind the names and values of Perl variables/structures through the apache core configuration gears.

Most of the configurations directives can be represented as scalars (**\$scalar**) or lists (**@list**).

An **@List** inside these sections is simply converted into a space delimited string for you inside. Here is an example:

```
#httpd.conf
<Perl>
@PerlModule = qw(Mail::Send Devel::Peek);

#run the server as whoever starts it
$user = getpwuid($> || $>;
$Group = getgrgid($> || $>;

$serverAdmin = $user;

</Perl>
```

Block sections such as `<Location>..</Location>` are represented in a `%Location` hash, e.g.:

```
$Location{"~/~dougm/"} = {  
  AuthUserFile => '/tmp/htpasswd',  
  AuthType => 'Basic',  
  AuthName => 'test',  
  DirectoryIndex => [qw(index.html index.htm)],  
  Limit => {  
    METHODS => 'GET POST',  
    require => 'user dougm',  
  },  
};
```

If a Directive can take two *or* three arguments you may push strings and the lowest number of arguments will be shifted off the **@List** or use array reference to handle any number greater than the minimum for that directive:

```
push @Redirect, "/foo", "http://www.foo.com/";;
```

```
push @Redirect, "/imdb", "http://www.imdb.com/";;
```

```
push @Redirect, [qw(temp "/here" "http://www.there.com");];
```

Other section counterparts include **%VirtualHost**, **%Directory** and **%Files**.

To pass all environment variables to the children with a single configuration directive, rather than listing each one via `PassEnv` or `PerlPassEnv`, a `<Perl>` section could read in a file and:

```
push @PerlPassEnv, [ $key => $val ];
```

or

```
Apache->httpd_conf( "PerlPassEnv $key $val" );
```

These are somewhat simple examples, but they should give you the basic idea. You can mix in any Perl code your heart desires. See `eg/httpd.conf.pl` and `eg/perl_sections.txt` in `mod_perl` distribution for some examples.

A tip for syntax checking outside of `httpd`:

```
<Perl>
# !perl

#... code here ...

__END__
</Perl>
```

Now you may run:

```
perl -cx httpd.conf
```

To enable >Perl> sections you should build mod_perl with **perl Makefile.PL PERL_SECTIONS=1**.

7.3 General pitfalls

7.3.1 My cgi/perl code is being returned as a plain text instead of being executed by the webserver?

Check your configuration files and make sure that the “+ExecCGI” is turned on in your configurations.

```
<Location /perl>  
    SetHandler perl-script  
    PerlHandler Apache::Registry  
    Options +ExecCGI  
    allow from all  
    PerlSendHeader On  
</Location>
```

7.3.2 My script works under cgi-bin, but when called via mod_perl I see A 'Save-As' prompt

Did you put **PerlSendHeader On** in the configuration part of the
<Location foo></Location>?
;o)

8 Installation Notes

8.1 Configuration and Installation

8.1.1 *perl*

Make sure you have perl installed -- the newer stable version you have the better (minimum perl.5.004!).

If you don't have it -- install it. Follow the instructions in the distribution's **INSTALL** file.

During the configuration stage (while running **./Configure**), make sure you answer **YES** to the question:

Do you wish to use dynamic loading? [y]

Answer **y** to be able to load dynamically Perl Modules extensions.

8.1.2 *apache*

It is a good idea to try to install the apache webserver without mod_perl first.

This way, if something goes wrong, you will know that it's not the apache server's problem.

But you can skip this stage if you already have a working (non-mod_perl) apache server, or if you are just the daring type.

In any case you should unpack the apache source distribution, preferably at the same level as the mod_perl distribution.

```
% ls -l /usr/src
drwxr-xr-x  8 stas  bar      2048 Oct  6 09:46 apache_x.x.x/
drwxr-xr-x 19 stas  bar     4096 Oct  2 14:33 mod_perl-x.xx/
```

8.1.3 *mod_perl*

Now we come to the main point of this document.

Here I will give only a short example of `mod_perl` installation. A dedicated section discusses this issue in more details.

As with any perl package, the installation of `mod_perl` is very easy and standard.

perldoc INSTALL will guide you through the configuration and the installation processes.

The fastest way to install would be:

```
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \  
DO_HTTPD=1 USE_APACI=1 PERL_MARK_WHERE=1 EVERYTHING=1  
% make && make test && make install
```

Note: replace x.x.x with the version numbers you actually use.

To change the installation target (either if you are not **root** or you need to install a second copy for testing purposes), assuming you use **/foo/server** as a base directory, you have to run this:

```
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \  
DO_HTTPD=1 PERL_MARK_WHERE=1 EVERYTHING=1 \  
APACHE_PREFIX=/foo/server PREFIX=/foo/server
```

Where **PREFIX** specifies where to install the perl modules, **APACHE_PREFIX** -- the same for the apache files.

The next step is to configure the `mod_perl` sections of the apache configuration file.

Fire up the server with **`/foo/server/sbin/apachectl start`**, Look for the error reports at the **`error_log`** file in case the server does not start up (No error message will be printed to the console!).

8.2 How can I tell whether mod_perl is running

There are a few ways. In older versions of apache (< 1.3.6 ?) you could check that by running **httpd -v**, it no longer works. Now you should use **httpd -l**.

Please notice that it is not enough to have it installed - you should of course configure it for mod_perl and restart the server.

8.2.1 Testing by checking the error_log file

When starting the server, just check the **error_log** file for the following message:

```
[Thu Dec 3 17:27:52 1998] [notice]
  Apache/1.3.1 (Unix) mod_perl/1.15 configured
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  -- resuming normal operations
```

8.2.2 Testing by viewing `/perl-status`

Assuming that you have configured the **Testing via telnet**

Knowing the port you have configured apache to listen on, you can use **telnet** to talk directly to it.

Assuming that your `mod_perl` enabled server listens to port 8080, telnet to your server at port 8080, and type **HEAD / HTTP/1.0** then press the `<ENTER>` key TWICE:

```
% telnet localhost 8080<ENTER>  
HEAD / HTTP/1.0<ENTER><ENTER>
```

You should see a response like this:

```
HTTP/1.1 200 OK
Date: Tue, 01 Dec 1998 12:27:52 GMT
Server: Apache/1.3.6 (Unix) mod_perl/1.19
Connection: close
Content-Type: text/html
```

Connection closed.

The line: **Server: Apache/1.3.6 (Unix) mod_perl/1.19 --confirms** that you **do** have mod_perl installed and its version is **1.19**. Of course in your case it would be the version you have installed.

However, just because you have got mod_perl linked in there, that does not mean that you have configured your server to handle Perl scripts with mod_perl. You will have to configure it first.

8.2.3 *Testing via a CGI script*

Another method is to invoke a CGI script which dumps the server's environment.

Copy and paste the script below (no need for the first perl calling (shebang) line!).

Let's say you named it **test.pl**, saved it at the root of the CGI scripts and CGI root is mapped directly to the **/perl** location of your server.

```
print "Content-type: text/html\n\n";  
print "Server's environment<P>\n";  
print "<TABLE>";
```

```
foreach ( keys %ENV ) {  
    print "<TR><TD>$_ </TD><TD>$ENV{$_}</TR></TD></TR></TD>" ;  
}  
print "</TABLE>" ;
```

Make it readable and executable by server:

```
% chmod a+rx test.pl
```

(you will want to tune permissions on the public host).

Now fetch the URL **`http://www.nowhere.com:8080/perl/test.pl`** (replace 8080 with the port your mod_perl enabled server is listening to).

You should see something like this (the generated output was trimmed):

```
SERVER_SOFTWARE      Apache/1.3.6 (Unix) mod_perl/1.19
GATEWAY_INTERFACE    CGI-Perl/1.1
REQUEST_METHOD        GET
HTTP_ACCEPT           image/gif, image/x-xbitmap, image/jpeg, */*
MOD_PERL              1.19
REQUEST_URI           /perl/test.pl
SCRIPT_NAME           /perl/test.pl
[...snipped]
```

Now if I run the same script in mod_cgi mode (configured with **/cgi-bin** Alias) (you will need to add the perl invocation line **#!/bin/perl** for the above script) and fetch **<http://www.nowhere.com/cgi-bin/test.pl>**.

```
SERVER_SOFTWARE      Apache/1.3.6 (Unix)
GATEWAY_INTERFACE    CGI/1.1
[...snipped]
```

You will see that two variables, **SERVER_SOFTWARE** and **GATEWAY_INTERFACE**, are different from the case above.

This gives you a hint of how to tell in what mode you are running in your cgi scripts. I start all my cgi scripts that are mod_perl aware with:

```
BEGIN {
    # Auto-detect if we are running under mod_perl or CGI.
    $USE_MOD_PERL = ((exists $ENV{'GATEWAY_INTERFACE'}
        and $ENV{'GATEWAY_INTERFACE'} =~ /CGI-Perl/)
        or exists $ENV{'MOD_PERL'} );
    # perl5.004 is a must under mod_perl
    require 5.004 if $USE_MOD_PERL;
}
```

You might wonder why in the world you would need to know in what mode you are running.

For example you will want to use **Apache::exit()** and not **CORE::exit()** in your modules, but if you think that your script might be used in both environments (mod_cgi vs. mod_perl), you

will have to override the **exit()** subroutine and to make the runtime decision of what method you will use.

Not that if you run scripts under **Apache::Registry** handler, it takes care of overriding the **exit()** call for you, so it's not an issue if this is your case.

8.2.4 Testing via *lwp-request*

Yet another one. Why do I show all these approaches? While here they are serving a very simple purpose, they can be helpful in other situations.

Assuming you have the **libwww-perl (LWP)** package installed (you will need it installed in order to pass `mod_perl's make test` anyway):

```
% lwp-request -e -d http://www.nowhere.com
```

Will show you all the headers. (The `-d` option disables printing the response content.)

```
% lwp-request -e -d http://www.nowhere.com | egrep '^Server:'
```

To see the server's version only.

Use **http://www.nowhere.com:port_number** if your server is listening to a non-default 80 port.

8.3 Is it possible to install and use apache/mod_perl without having a root access?

Yes, no problem with that. Follow the installation instructions and when you encounter **APACI_ARGS** use your home directory (or some other directory which you have write access to) as a prefix, (e.g. **/home/stas/www**), and everything will be installed there.

There is a chance that some perl libs will be not installed on your server by root and you will have to install these locally too.

See the

<http://www.singlesheaven.com/stas/TULARC/webmaster/myfaq.html#7>
for more information on local perl installations.

You will not be able to have the server listen to a port lower than 1024 if you are not starting it as **root**, so choose a port number above 1024. (I use 8080 in most cases).

Note that you will have to use a URL like **http://www.nowhere.com:8080** in that case, but that is not a problem since usually users do not directly access URLs to CGI scripts, but rather are directed to them from a link on a web page or as the '**ACTION**' of a HTML form, so they should not know at all that the port is different from the default port 80.

If you want your apache server to start automatically on system reboot, you will need to invoke the server startup script from somewhere within the init scripts on your host.

This is often somewhere under `/etc/rc.d`, but this path can vary depending upon the flavor of Unix you are using.

One more important thing to keep in mind is system resources. `mod_perl` is memory hungry -- if you run a lot of `mod_perl` processes on a public, multiuser (not dedicated) machine -- most likely the system administrator of the host will ask you to use less resources and even to shut down your `mod_perl` server and to find another home for it. You have a few solutions:

- Reduce resources usage.
- Ask your ISP if you can put a dedicated machine into their computer room and be root there.
- Look for another ISP with lots of resources or one that supports `mod_perl`. You can find a list of these ISP at <http://perl.apache.org> .

8.4 Is it possible to determine which options were given to modperl's Makefile.PL

It is possible to determine which options were given to modperl's **Makefile.PL** during the configuration stage, so to be used later in recreating the same build tree when rebuilding the server.

This is relevant only if you did not use the default config parameters and altered some of them during the configuration stage.

I was into this problem many times. I am going to build something by passing some non-default parameters to the config script and then later when I need to rebuild the tool either to upgrade it or to

make an identical copy at another machine, I would find that I do not remember what parameters I altered.

The best solution for this problem is to prepare the run file with all the parameters that are about to be used and then run it instead of typing it all by hand.

So later I will have the script handy to be reused.

mod_perl suggests using the **makepl_args.mod_perl** file which comes with mod_perl distribution. This is the file where you should specify all the parameters you are going to use.

But if you have found yourself with a compiled tool and no traces of the specified parameters left, usually you can still find them out, if the sources were not **make clean**'d.

You will find the apache specific parameters in **apache_x.x.x/config.status** and modperl's at in **mod_perl_x.xx/apaci/mod_perl.config**.

8.5 Server Installation problems

8.5.1 *make test fails*

There are two configuration parameters: **PREP_HTTPD** and **DO_HTTPD**, that you can use in:

```
perl Makefile.PL [options]
```

DO_HTTPD=1 means default to 'y' for the two apache's **configure** utility prompts: (a) 'which source tree to configure against' and (b) 'whether to build the httpd in that tree'.

PREP_HTTPD=1 just means default 'n' to the second prompt -- meaning, *do not build httpd (make) in the apache source tree*.

In other words if you use **PREP_HTTPD=1** the httpd will be not build. It will be build only if you use **DO_HTTPD=1** option and not use **PREP_HTTPD=1**.

If you did not build the `httpd`, `chdir` to the apache source, and execute:

```
make
```

Then return to the `mod_perl` source and run:

```
make test
```

```
make install
```

Note that you would have to do the same if you do not pass **APACHE_PREFIX=/path_to_installation_prefix** during the **perl Makefile.PL [options]** stage.

8.5.2 mod_perl.c is incompatible with this version of apache

You will see this message when you try to run a httpd, if you have had a stale old apache header layout in one of the **include** paths during the build process.

Do run **find** (or **locate**) utility in order to locate **ap_mmn.h** file. In my case I have had a **/usr/local/include/ap_mmn.h** which was installed by RedHat install process. If this is the case get rid of it, and rebuild it again.

For all RH fans, before you are going to build the apache by yourself, do:

rpm -e apache

to remove the pre-installed package first!

8.5.3 *Should I rebuild mod_perl if I have upgraded my perl?*

Yes, you should. You have to rebuild mod_perl enabled server since it has a hard coded **@INC** which points to the old perl and it is probably linked to the an old **libperl** library.

You can try to modify the **@INC** in the startup script (if you keep the old perl version around), but it is better to build a fresh one to save you a mess.

```
;o)
```

9 Controlling and Monitoring the Server

9.1 Restarting techniques

All of these techniques require that you know the server PID (Process ID).

The easiest way to find the PID is to look it up in the **httpd.pid**. (hint: look it up in **httpd.conf**)

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

Another way:

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

or maybe:

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

From the list of processes you get, pick the one that belongs to **root**.

You shouldn't ever need to send signals to any process except the parent.

There are three signals you should know about: **TERM**, **HUP**, and **USR1**.

9.2 Implications of sending TERM, HUP, and USR1 to the server

TERM Signal: stop now

Sending the **TERM** signal to the parent causes it to immediately attempt to kill off all of its children.

This process may take several seconds to complete, following which the parent itself exits.

Any requests in progress are terminated, and no further requests are served.

That's the moment that the accumulated **END** blocks will be executed!

Note that if you use **Apache::Registry** or **Apache::PerlRun**, then **END** blocks are being executed upon each request (at the end).

HUP Signal: restart now

Sending the **HUP** signal to the parent causes it to kill off its children like in **TERM** (Any requests in progress are terminated) but the parent doesn't exit.

It re-reads its configuration files, and re-opens any log files. Then it spawns a new set of children and continues serving hits.

The server will reread its configuration files, flush all the compiled and preloaded modules, and rerun any startup files. It's equivalent to stopping, then restarting a server.

Note: If your configuration file has errors in it when you issue a restart then your parent will not restart but exit with an error. See below for a method of avoiding this.

USR1 Signal: graceful restart

The **USR1** signal causes the parent process to advise the children to exit after their current request (or to exit immediately if they're not serving anything).

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 of the configuration, which begins serving new requests immediately.

The only difference between **USR1** and **HUP** is that **USR1** allows children to complete any in-progress request prior to killing them off.

By default, if a server is restarted (ala **kill -USR1 'cat logs/httpd.pid'** or with **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, enable with this directive:

PerlFreshRestart On (in httpd.conf)

It's worth mentioning that restart or termination can sometimes take quite a lot of time.

Check out the **PERL_DESTRUCT_LEVEL=-1** option during the `mod_perl Makefile.PL` stage, which speeds this up and leads to more robust operation in the face of problems, like running out of memory.

It is only usable if no significant cleanup has to be done by perl **END** blocks and **DESTROY** methods when the child terminates, of course.

What constitutes 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 is significant but closing an ordinary file isn't.

Some folks prefer to specify signals using numerical values, rather than symbolics. If you are looking for these, check out your **kill(3)** man page.

My page points to **/usr/include/sys/signal.h**, the relevant entries are:

```
#define SIGHUP  1
#define SIGTERM 15
#define SIGUSR1 30
```

9.3 Using `apachectl` to control the server

Apache's distribution provides a nice script to control the server.

It's called `apachectl` and it's installed into the same location with `httpd`.

In our scenario - it's `/usr/local/sbin/httpd_perl/apachectl`.

Start `httpd`:

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

Stop httpd:

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

Restart httpd if running by sending a **SIGHUP** or start if not running:

```
% /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
```

Do a configuration syntax test:

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

It's important to remember that this script is based on the PID file which is `/usr/local/var/httpd_perl/run/httpd.pid` in our case. If you delete the file by hand - `apachectl` will fail to run.

Also, notice that `apachectl` is suitable to use from within your Unix system's startup files so that your web server is automatically restarted upon system reboot.

Either copy the `apachectl` file to the appropriate location (`/etc/rc.d/rc3.d/S99apache` works on my RedHat Linux system) or create a symlink with that name pointing to the canonical location.

9.4 SUID start-up scripts

Covered in your handouts.

9.5 Monitoring the Server. A watchdog.

With mod_perl many things can happen to your server.

The worst one is the possibility that the server will die when you will be 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 called `apache.watchdog` and to put it into the crontab to be called every 30 minutes or even every minute - if it's so critical to make sure the server will be up all the time.

The crontab entry:

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

The script:

```
#!/bin/sh

EMAIL = webmaster@somewhere.far
PIDFILE = /usr/local/var/httpd_perl/run/httpd.pid
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 >& /dev/null
    else
        echo "$0 $ARG: httpd could not be started"
        mail $EMAIL -s "$0 $ARG: httpd could not be started" \
            </dev/null >& /dev/null
    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 server can be up as a process, it can be stuck and not working.

I call this script every minute from crontab.

Why so often? If your server starts to spin and trash your disk's space with multiply error messages, in a 5 minutes you might run out of free space, which might bring your system to its knees.

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 be not felt by the server at all.

So we end up with crontab entry:

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

And the watchdog itself:

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

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.stas.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/Stas" . $ua->agent);
# Uncomment the proxy if you don't use it!
# $proxy="http://www-proxy.com";
$ua->proxy('http', $proxy) if $proxy;
```

```
# If returns '1' it's we are alive
exit 1 if checkurl($test_script_url);

# We've got the problem - the server seems
# to be down. Try to restart it
my $status = system $restart_command;
# print "Status $status\n";

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 if success, 0 for fail
#####
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

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

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

    print MAIL <<__END_OF_MAIL__ ;
To: $to
From: $from
Subject: $subject

$messagebody
```

```
__END_OF_MAIL__  
  
close MAIL;  
}
```

9.6 Running server in a single mode

Often while developing new code, you will want to run the server in single process mode.

Running in single process mode inhibits the server from “daemonizing”, allowing you to run it more easily under debugger control.

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

When you execute the above the server will run in the fg (foreground) of the shell you have called it from.

Turn off **KeepAlive** in **httpd.conf** while working in this mode, if you return pages with references to images at the same server.

No control messages (Like “server started, server stopped and etc”.) that the parent server normally writes to the `error_log` will be written.

Since **-X** causes the server to handle all requests itself, without forking any children, there is no controlling parent to write status messages.

9.7 Starting a personal server for each developer

Sometimes you have a group of developers who need to concurrently develop their own `mod_perl` scripts.

This means that each one will want to have control over the server - to kill it, to run it in single server mode, to restart it again, etc., as well to have control over the location of the log files and other configuration settings like **MaxClients**, etc.

You can work around this problem by preparing a few `httpd.conf` file and forcing each developer to use:

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

I have approached it in other way. I have used the **-Dparameter** startup option of the server. I call my version of the server

```
% http_perl -Dsbekman
```

In **httpd.conf** I wrote:

```
# Personal development Server for sbekman
# sbekman use the server running on port 8000
<IfDefine sbekman>
Port 8000
PidFile /usr/local/var/httpd_perl/run/httpd.pid.sbekman
ErrorLog /usr/local/var/httpd_perl/logs/error_log.sbekman
Timeout 300
KeepAlive On
MinspareServers 2
MaxspareServers 2
StartServers 1
```

```
MaxClients 3
MaxRequestsPerChild 15
</IfDefine>

# Personal development server for userfoo
# userfoo use 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>
```

What we have achieved with this technique: Full control over start/stop, number of children, separate error log file, and port selection.

The above setup saves me from getting called every few minutes - “Stas, I’m going to restart the server” .

To make a picture complete, change the **apachectl**, and rename it to something like **apachectl-sbekman**.

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

Of course you think you can use only one control file and know who is calling by using uid, but since you have to be root to start the server - it is not so simple.

The last thing was to let developers an option to run in single process mode by:

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

In addition to making life easier, we decided to use relative links everywhere in the static docs (including the calls to CGIs).

You may ask how using the relative link you will get to the right server? Very simple - we have utilized the `mod_rewrite` to solve our problems:

In `access.conf` of the `httpd_docs` server we have the following code: (you have to configure your `httpd_docs` server with `--enable-module=rewrite`)

```
# sbekman' server
# port = 8000
RewriteCond %{REQUEST_URI} ^/(perl|cgi-perl)
RewriteCond %{REMOTE_ADDR} 123.34.45.56
RewriteRule ^(.*) http://nowhere.com:8000/$1 [R,L]

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

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

where IP numbers are the IPs of the developer client machines
(where they are running their web browser.)

So if I have a relative URL like `/perl/test.pl` in HTML code or `http://www.nowhere.com/perl/test.pl` it will be redirected by `httpd_docs` to `http://www.nowhere.com:8000/perl/test.pl` (`sbekman`'s port).

Of course you have another problem: The CGI generates some html, which should be called again. If it generates a URL with hard coded PORT the above scheme will not work. There 2 solutions:

First, generate relative URL so it will reuse the technique above, with redirect (which is transparent for user) but it will not work if you have something to **POST** (redirect looses all the data!).

Second, use a general configuration module which generates a correct full URL according to **REMOTE_USER**, so if `$ENV{REMOTE_USER} eq 'sbekman'`, I return `http://www.nowhere.com:8000/perl/` as `cgi_base_url`. Again this will work if the user is authenticated.

9.8 Wrapper to emulate the server environment

Many times you start off debugging your script by running it from your favorite shell.

Sometimes you encounter a very weird situation when script runs from the shell but dies when called as a CGI.

The real problem lies in the difference between the environment that is being used by your server and your shell.

An example can be a different perl path or having **PERL5LIB** env variable which includes paths that are not in the **@INC** of the perl compiled with mod_perl server and configured during the startup.

The best debugging approach is to write a wrapper that emulates the exact environment of the server, by first deleting the environment variables like **PERL5LIB** and 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 server startup and configuration files.

It will also allow you to remove completely the first line of the script - since mod_perl skips it and the wrapper knows how to call the script.

Below is the example of such a script. Note that we force the **-Tw** when we call the real script. (I have also added the ability to pass params, which will not happen when you call the cgi from the web)

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

# Usage: wrap.pl some_cgi.pl

BEGIN{
    use vars qw($basedir);
    $basedir = "/usr/local";

    # we want to make a complete emulation,
    # so we must remove the user's environment
    @INC = ();

    # local perl libs
    push @INC,
        qw($basedir/lib/perl5/5.00502/aix
           $basedir/lib/perl5/5.00502
           $basedir/lib/perl5/site_perl/5.005/aix
           $basedir/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 ($cgi =~ m|/|) {
    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 invoke warnings and Taint mode ON!!!
system qq{$basedir/bin/perl -I$PERL5LIB -Tw $cgi $params};
```

9.9 Log Rotation

Details in the handouts.

9.10 Preventing from modperl process from going wild

Sometimes calling an undefined subroutine in a module can cause a tight loop that consumes all memory.

Here is a way to catch such errors. Define an autoloader subroutine:

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

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

Sometimes an error happens and causes the server to write millions of lines into your **error_log** file and in a few minutes to put your server down on its knees.

For example I get an error **Callback called exit** show up in my **error_log** file many times. The **error_log** file grows to 300 Mbytes in size in a few minutes. You should run a cron job to make sure this does not happen and if it does to take care of it. e.g:

```
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 $$ kB on inx" root@localhost  
fi
```

;o)

10 mod_perl and Relational Databases

10.1 Why Relational (SQL) Databases

Nowadays millions of users surf the Internet. There are millions of Terabytes of data laying around. To manipulate that data new smart techniques and technologies were invented. One of the major inventions was a relational database, which allows to search and modify huge data storages in zero time.

It uses **SQL** (Structured Query Language) to manipulate contents of these databases. Of course once we started to use the web, we have found a need to write web interfaces to these databases and CGI was and is the mostly used technology for building such interfaces.

The main limitation for a CGI script driving a database versus application, is its statelessness - on every request the CGI script has to initiate a connection to the database, when the request is completed the connection is lost. **Apache::DBI** was written to remove this limitation.

When you use it, you have a persistent database connection over the process' life. As you understand this possible only with CGI running under mod_perl enabled server, since the child process does not quit when the request has been served.

So when a mod_perl script needs to `_talk_` to a database, he starts `_talking_` right away, without initiating a database connection first, **Apache::DBI** worries to provide a valid connection immediately.

Of course the are more nuances, which will be talked about in the following sections.

10.2 Apache::DBI - Initiate a persistent database connection

This module initiates a persistent database connection. It is possible only with `mod_perl`.

10.2.1 *Introduction*

When loading the DBI module (do not confuse this with the **Apache::DBI** module) it looks if the environment variable `GATEWAY_INTERFACE` starts with 'CGI-Perl' and if the module **Apache::DBI** has been loaded. In this case every connect request will be forwarded to the **Apache::DBI** module.

This looks if a database handle from a previous connect request is already stored and if this handle is still valid using the **ping()** method. If these two conditions are fulfilled it just returns the database handle. If there is no appropriate database handle or if the ping method fails, a new connection is established and the handle is stored for later re-use.

In other words when the script is run again from a child that has already (and is still) connected, the host/username/password is checked against the cache of open connections, and if one is available, uses that one.

There is no need to delete the disconnect statements from your code. They won't do anything because the **Apache::DBI** module overloads the disconnect method with a NOP (like an empty call).

You want to use this module if you are opening a **few** DB connections to the server. **Apache::DBI** will make them persistent per child, so if you have 10 children and each opens 2 different connections you will have in total 20 opened persistent connections.

Thus after initial connect you will save up the connection time for every connect request from your DBI module. Which is a huge benefit for the mod_perl apache server with high traffic of users deploying the relational DB.

As you understand you must **NOT** use this module if you are opening a special connection for each of your users, since each of them will stay persistent and in a short time the number of connections will be so big that your machine will scream and die.

If you want to use **Apache::DBI** in both situations, as of this moment the only available solution is to run 2 mod_perl servers, one using **Apache::DBI** and one not.

10.2.2 Configuration

After installing this module, the configuration is simple - add to the **httpd.conf** the following directive.

```
PerlModule Apache::DBI
```

Note that it is important, to load this module before any other ApacheDBI module !

You can skip preloading **DBI**, since **Apache::DBI** does that. But there is no harm if you leave it in.

10.2.3 Preopening DBI connections

If you want that when you call the script after server restart, the connection will be already preopened, you should use **connect_on_init()** method in the startup file to preload every connection you are going to use. For example:

```
Apache::DBI->connect_on_init
("DBI:mysql:mysql::myserver",
 "username",
 "passwd",
 {
   PrintError => 1, # warn() on errors
   RaiseError => 0, # don't die on error
   AutoCommit => 1, # commit executes immediately
 }
);
```

As noted before, it is wise to you this method only if you only want all of apache to be able to connect to the database server as one user (or few users).

10.2.4 Debugging Apache::DBI

If you are not sure this module is working as advertised, you should enable the Debug mode in the startup script by:

```
$Apache::DBI::DEBUG = 1;
```

Since now on you will see in the **error.log** file when **Apache::DBI** initializes a connection and when it just returns it from its cache. Use the following command to see it in the real time (your **error.log** file might be locate at a different path):

```
tail -f /usr/local/apache/logs/error_log
```

I use **alias** (in **tcsh**) so I will not have to remember the path:

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

Another approach is to add to **httpd.conf** (which does the same):

```
PerlModule Apache::DebugDBI
```

10.2.5 Problems and solutions

10.2.5.1 The morning bug

SQL server keeps the connection to the client open for a limited period of time. So many developers were bitten by so called **Morning bug** when every morning the first users to use the site were receiving: **No Data Returned** message, but then everything worked as usual.

The error caused by **Apache::DBI** returning a handle of the invalid connection (server closed it because of timeout), and the script was dying on that error.

The infamous and well documented in the man page, **ping()** method was introduced to solve this problem. But seems that people are still being beaten by this problem.

Another solution was found - to rise the timeout parameter at the SQL server startup.

Currently I startup **mysql** server with **safe_mysql** script, so I have updated it to use this option:

```
nohup $ledir/mysqld [snipped other options] -O wait_timeout=172800
```

Where 172800 secs equal to 48 hours. And it works.

Note that starting from ver. **0.82**, **Apache::DBI** implements ping inside the **eval** block, so if the handle has been timed out, it should reconnect without creating the morning bug.

10.2.5.2 Opening a connection with different parameters

Q: Currently I am running into a problem where I found out that **Apache::DBI** insists that the connection will be opened exactly the same way before it will decide to use a cached connection.

I.e. if I have one script that sets **LongReadLen** and one that does not, it will be two different connections. Instead of having a max of 40 open connections, I end up with 80.

A: indeed, **Apache::DBI** returns a cached database handle, if and only if all parameters including all options are identical.

But you are free to modify the handle right after you got it from the cache.

Initiate the connections always using the same parameters and set **LongReadLen** afterwards.

10.2.5.3 Cannot find the DBI handler

Q: I cannot find the handler name with which to manipulate my connection; hence I seem to be unable to do anything to my database.

A: You did not use **DBI::connect()** as with usual DBI usage to get your `$dbh` database handler. Using the **Apache::DBI** does not eliminate the need to write a proper **DBI** code. As the man page states - you should program as if you did not use **Apache::DBI** at all. **Apache::DBI** will override and return you a cached connection. And in case of **disconnect()** call it will be just ignored.

10.2.5.4 Apache:DBI does not work

Make sure you have it installed.

Make sure you configured `mod_perl` with **EVERYTHING=1**.

Use the example script `eg/startup.pl`, just remove the comment from the line:

```
#use Apache::DebugDBI;
```

and adapt the connect string. Do not change anything in your scripts, for using **Apache::DBI**.

```
;o)
```

11 mod_perl for ISPs

11.1 ISPs providing mod_perl services - a fantasy or reality.

You have fallen in love with mod_perl from the first sight, since the moment you have installed it at your home box.

But when you wanted to convert your CGI scripts, currently running on your favorite ISPs machine, to run under mod_perl - you have discovered, your ISPs either have never heard of such a beast, or refuse to install it for you.

You are an old sailor in the ISP business, you have seen it all, you know how many ISPs are out there and you know that the sales margins are too low to keep you happy.

You are looking for some new service almost no one provides, to attract more clients to become your users and hopefully to have a bigger slice than a neighbor ISP.

If you are a user asking for a `mod_perl` service or an ISP considering to provide this service, this section should make things clear for both of you.

an ISP has 3 choices to choose from:

1. ISP cannot afford having a user, running scripts under `mod_perl`, on the main server, since it will die very soon for one of the many reasons: either sloppy programming, or user testing just updated script which probably has some syntax errors and etc, no need to explain why if you are familiar with `mod_perl` peculiarities.

The only scripts that **CAN BE ALLOWED** to use, are the ones that were written by ISP and are not being modified by user (guest books, counters and etc - the same standard scripts ISPs providing since they were born). So you have to say **NO** for this choice.

2. But, hey why I cannot let my user to run his own server, so I clean my hands off and do not care how dirty and sloppy user's code is (assuming that user is running the server by his own username).

This option is fine as long as you are concerned about your new system requirements.

If you have even some very limited experience with mod_perl, you know that mod_perl enabled apache servers while freeing up your CPU and lets you run scripts much much faster, has a huge memory demands (5-20 times the plain

apache uses).

The size depends on the code length, sloppiness of the programmer, possible memory leaks the code might have and all that multiplied by the number of children each server spawns.

A very simple example : a server demanding 10Mb of memory which spawns 10 children, already rises your memory requirements by 100Mb (the real requirement are actually smaller if your OS allows code sharing between processes and a programmer exploits these features in her code).

Now multiply the received number by the number of users you intend to have and you will get the memory requirements.

Since ISPs never say no, you better use an opposite approach - think of a largest memory size you can afford then divide it by one user's requirements as I have shown in example, and you will know how much mod_perl users you can afford :)

But who am I to prognosticate how much memory your user may use.

His requirement from a single server can be very modest, but do you know how many of servers he will run (after all she has all the control over httpd.conf - and it has to be that way, since this is very essential for the user running mod_perl)?

All this rumberling about memory leads to a single question:
Can you restrict user from using more than X memory?

Or another variation of the question: Assuming you have as much memory as you want, can you charge user for the average memory usage?

If the answer for either of the above question is positive, you are all set and your clients will prize your name for letting them run mod_perl!

There are tools to restrict resources' usage (See for example `man pages` for **ulimit(3)**, **getrlimit(2)**, **setrlimit(2)** and **sysconf(3)**).

If you have picked this choice, you have to provide your client:

- Shutdown/startup scripts installed together with the rest of your daemon startup scripts (e.g **/etc/rc.d** directory) scripts, so when you reboot your machine user's server

will be correctly shutdowned and will be back online the moment your system comes back online.

Also make sure to start each server under username the server belongs to, if you are not looking for a big trouble.

- Proxy (in a forward or httpd accelerator mode) services for user's virtual host. Since user will have to run her server on unprivileged port (>1024), you will have to forward all requests from **user.given.virtual.hostname:80** (which is **user.given.virtual.hostname** without port - 80 is a default) to **your.machine.ip:port_assigned_to_user** and user to code his scripts to write self referencing URLs to be of **user.given.virtual.hostname** base of course.

Letting user to run a mod_perl server, immediately adds a requirement for user to be able to restart and configure their own server. But only root can bind port 80. That is why user has to use ports numbers >1024.

- Another problem you will have to solve is how to assign ports between users. Since user can pick any port above 1024 to run his server on, you will have to make some regulation here.

A simple example will stress the importance of this problem: I am a malicious user or I just a rival of some fellow who runs his own server on your ISP.

All I should do is to find out what port his server is listening to (e.g. with help of **netstat(8)**) and configure my own server to listen on the same port.

While I am unable to bind to this same port, imagine what will happen when you reboot your system and my startup script happen to be run before my rivals!

I get the port first, now all requests will be redirected to my server and let your imagination go wild about what nasty things might happen then.

Of course the ugly things will be revealed pretty soon, but the damage has been done.

3. A much better, but costly solution is **co-location**.

Let user to hook her (or ISP's) stand alone machine into your network, and forget about this user.

Of course either user or you will have to make all the system administration chores and it will cost your client more money.

All in all, who are the people who seek the mod_perl support?

The ones who run serious projects/businesses, who can afford a stand alone box, thus gaining their goal of self autonomy and keeping their ISP happy. So money is not an obstacle.

;o)

12 Getting Helped and Further Learning

12.1 Tutorial's Sources

This tutorial is based on the **mod_perl Guide** I wrote. The Guide includes many more details, I could not include in this tutorial due to a limited time I have had. Read the complete guide online at <http://perl.apache.org/guide/> .

12.2 Acknowledgements

I want to thank all the people who donated their time and efforts to make this amazing idea of mod_perl a reality.

This includes Doug MacEachern, the author of mod_perl and all the developers who contributed bug patches, modules and help.

And of course the numerous unseen users who helped to find bugs and advocate the mod_perl around the world.

12.3 Got into a trouble?

If after reading the available `mod_perl` documentation listed below, you feel that your question is not yet answered, please ask the `apache/mod_perl` mailing list to help you.

But first try to browse the mailing list archive.

Most of the time you will find the answer for your question by searching the mailing list archive, since there is a big chance someone else has already encountered the same problem and found a solution for it.

If you ignore this advice, do not be surprised if your question will be left unanswered - it bores people to answer the same question more than once.

It does not mean that you should avoid asking questions.

Just do not abuse the available help and **RTFM** before you call for **HELP**. (You have certainly heard the infamous fable of the shepherd boy and the wolves)

12.4 Get helped with mod_perl

mod_perl home

<http://perl.apache.org>

Getting mod_perl

Get the latest mod_perl sources and documentation from

<http://perl.apache.org> . Try the direct download link

<http://perl.apache.org/dist/> .

Apache Modules Book

<http://www.modperl.com> is the home site of The Apache Modules Book, a book about creating Web server modules using the Apache API, written by Lincoln Stein and Doug

MacEachern.

Now you can purchase the book at your local bookstore or from the online dealer. O'Reilly lists this book as:

**Writing Apache Modules with Perl and C
By Lincoln Stein & Doug MacEachern
1st Edition March 1999
1-56592-567-X, Order Number: 567X
746 pages, \$34.95**

- **mod_perl Guide**
by Stas Bekman at <http://perl.apache.org/guide/> .

- **mod_perl FAQ**

by Frank Cringle at <http://perl.apache.org/faq/> .

- **mod_perl performance tuning guide**

by Vivek Kherra at <http://perl.apache.org/tuning/> .

- **mod_perl plugin reference guide**

by Doug MacEachern at
http://perl.apache.org/src/mod_perl.html .

- **Quick guide for moving from CGI to mod_perl**

at http://perl.apache.org/dist/cgi_to_mod_perl.html .

- **mod_perl_traps, common traps and solutions for mod_perl users**

at http://perl.apache.org/dist/mod_perl_traps.html .

- **mod_perl Quick Reference Card**

<http://www.refcards.com> (Apache and other refcards are available from this link)

- **mod_perl mailing list**

The Apache/Perl mailing list (modperl@apache.org) is **available for mod_perl users and developers to share ideas, solve problems and discuss things related to mod_perl and the Apache::* modules.** To subscribe to this list, send mail to majordomo@apache.org with empty **Subject** and with **Body**:

subscribe modperl

A **searchable** mod_perl mailing list archive available at <http://forum.swarthmore.edu/epigone/modperl> . We owe it to Ken Williams.

Another archive: <http://www.geocrawler.com/lists/3/web/182/0/>

12.5 Get helped with Perl

- **Getting Perl**

Perl is most likely already installed on your machine, but you should at least check the version you using.

It is highly recommended that you have at least perl version 5.004 or higher. You can get the latest perl version from <http://www.perl.com/> .

Try the direct download link

<http://www.perl.com/pace/pub/perldocs/latest.html> . You can get a perl documentation from the same location.

- **The Perl FAQ**

<http://www.perl.com/CPAN/doc/FAQs/FAQ/PerlFAQ.html>

- **perl home**

<http://www.perl.com/>

- **The Perl Journal**

<http://www.tpj.com/>

- **Perl Module Mechanics**

http://world.std.com/~swmcd/steven/perl/module_mechanics.html
- This page describes the mechanics of creating, compiling, releasing and maintaining Perl modules.

12.6 Get helped with Perl/CGI

- **Perl/CGI FAQ**

at <http://www.perl.com/CPAN/doc/FAQs/cgi/perl-cgi-faq.html>

- **Answers to some bothering Perl and Perl/CGI questions**

<http://www.singlesheaven.com/stas/TULARC/webmaster/myfaq.html>

- **Idiot's Guide to CGI programming**

<http://www.perl.com/CPAN/doc/FAQs/cgi/idiots-guide.html>

- **WWW Security FAQ**

<http://www.w3.org/Security/Faq/www-security-faq.html>

- **CGI/Perl Taint Mode FAQ**

<http://www.gunther.web66.com/FAQS/taintmode.html> (by
Gunther Birznieks)

12.7 Get helped with Apache

- **Apache Project's Home**

<http://www.apache.org>

- **Getting apache**

Get the latest apache webserver and documentation from <http://www.apache.org> . Try the direct download link

<http://www.apache.org/dist/> .

- **Apache Quick Reference Card**

<http://www.refcards.com> (Apache and other refcards are available from this link)

- **The Apache FAQ**

<http://www.apache.org/docs/misc/FAQ.html>

- **Apache Server Documentation**

<http://www.apache.org/docs/>

- **Apache Handlers**

<http://www.apache.org/docs/handler.html>

12.8 Get helped with DBI

- **Perl DBI examples**

<http://eskimo.tamu.edu/~jbaker/dbi-examples.html> (by Jeffrey William Baker).

- **DBI at Hermetica**

<http://www.hermetica.com/technologia/DBI/>

- **DBI mailing list information**

<http://www.fugue.com/dbi/>

- **DBI mailing list archives**

<http://outside.organic.com/mail-archives/dbi-users/>

- **Persistent connections with mod_perl**

http://perl.apache.org/src/mod_perl.html#PERSISTENT_DATABASE_CONNECTIONS

12.9 Get helped with Squid - Internet Object Cache

- Home page - <http://squid.nlanr.net/>
- FAQ - <http://squid.nlanr.net/Squid/FAQ/FAQ.html>
- Users Guide - <http://squid.nlanr.net/Squid/Users-Guide/>
- Mailing lists - <http://squid.nlanr.net/Squid/mailling-lists.html>

12.10 Other Software that was mentioned

- **thttpd - tiny/turbo/throttling HTTP server**

<http://www.acme.com/software/thttpd/>

- **mod_proxy_add_forward**

Ask Bjoern Hansen has written a

mod_proxy_add_forward.c module for Apache, that sets the **X-Forwarded-For** field when doing a ProxyPass, similar to what Squid can do. His patch is at:

<http://modules.apache.org/search?id=124> or at
<ftp://ftp.netcetera.dk/pub/apache/>

;o)

