

The ApacheCon 2000  
March 9, 2000  
Orlando, Florida

***Tutorial:***  
***Improving Script Performance***  
***Under mod\_perl***

***By Stas Bekman***  
***Internet and Intranet programmer***  
***<http://stason.org/>***  
***<stas@stason.org>***

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 Getting Started Fast

# 1.1 mod\_perl in Four Slides

- Installation
- Configuration
- The “mod\_perl rules” Apache::Registry Scripts
- The “mod\_perl rules” Apache Perl Module

# 1.2 What is mod\_perl?

Solves numerous mod\_cgi shortcomings:

- Embedded Perl Interpreter -- no loading overhead
- Code compiled only once per process life -- no compilation overhead
- No forking per request -- process reuse
- Response processing is now reduced to running your code.
- Response times improve by a factor of 10 to 100

- A bigger size, but just a few processes can handle a much bigger load
- `mod_cgi` compatibility preserved (Apache::`Registry` and Apache::`PerlRun` modules)
- Persistent database connections

## Extended mod\_cgi's functionality:

- A complete Perl API added to the Apache core
- Handling of all phases of request processing in Perl.
- Writing complete Apache modules in Perl
- Complete server configuration in Perl.
- Numerous 3rd party modules are available

## Logistics:

- Developed by Doug MacEachern
- Licensed under the “Artistic License” as Perl itself.
- Home page <http://perl.apache.org>
- Mailing list: send email to [modperl-subscribe@apache.org](mailto:modperl-subscribe@apache.org) with the string “*subscribe modperl*” in the body.
- December 1999 -- 412000 mod\_perl hosts (according to <http://perl.apache.org/netcraft/>)

# 1.3 Installation

```
% 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 xzvf apache_x.x.x.tar.gz  
% tar xzvf 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 EVERYTHING=1  
% make && make test && make install  
% cd ../apache_x.x.x && make install
```

That's all!

# 1.4 Configuration

Add the following to the configuration file *httpd.conf*:

```
# for Apache::Registry mode
Alias /perl/ /home/httpd/perl/

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

# 1.5 The "mod\_perl rules" Apache::Registry Scripts

You can write plain perl/CGI scripts just as under mod\_cgi:

```
mod_perl_rules1.pl
-----
print "Content-type: text/plain\r\n\r\n";
print "mod_perl rules!\n";
```

Of course you can write them in the Apache Perl API:

```
mod_perl_rules2.pl
-----
my $r = shift;
$r->send_http_header('text/plain');
$r->print("mod_perl rules!\n");
```

Save both files under the */home/httpd/perl* directory, make them executable and readable by server, and issue these requests using your favorite browser:

[http://localhost/perl/mod\\_perl\\_rules1.pl](http://localhost/perl/mod_perl_rules1.pl)

[http://localhost/perl/mod\\_perl\\_rules2.pl](http://localhost/perl/mod_perl_rules2.pl)

In both cases you will see on the following response:

**mod\_perl rules!**

# 1.6 The "mod\_perl rules" Apache Perl Module

To create an Apache Perl module, all you have to do is to wrap the code into a handler subroutine:

```
ModPerl/Rules.pm
-----
use Apache::Constants;

sub handler{
    my $r = shift;
    $r->send_http_header('text/plain');
    print "mod_perl rules!\n";
    return OK;
}
```

Create a directory called *ModPerl* under one of the directories in @INC, and put *Rules.pm* into it. Then add the following snippet to *httpd.conf*:

```
PerlModule ModPerl::Rules  
<Location /mod_perl_rules>  
    SetHandler perl-script  
    PerlHandler ModPerl::Rules  
</Location>
```

Now you can issue a request to:

[http://localhost/perl/mod\\_perl\\_rules](http://localhost/perl/mod_perl_rules)

and just as with our *mod\_perl\_rules.pl* scripts you will see:

**mod\_perl rules!**

as the response.

# 1.7 Is That All I Need To Know About `mod_perl`?

- Definitely not! These slides are intended to show you that you can install and start using a `mod_perl` server within 30 minutes of downloading the sources.
- There is much more to `mod_perl` than this.
- Fortunately, there are many resources and lots of help freely available to you. See the last chapter of this tutorial for the help references.

;o)

# 2 Performance. Benchmarks.

# 2.1 What we will learn in this chapter

- Performance: An Overall picture
- Analysis of SW and HW Requirements
- Sharing Memory
- How Shared My Memory Is
- Preload Perl modules at server startup
- Preload Registry Scripts

- **Global vs Fully Qualified Variables**
- **Avoid Importing Functions**
- **PerlSetupEnv Off**
- **Adding a Proxy Server in http Accelerator Mode**
- **KeepAlive**
- **Upload/Download of Big Files**
- **Forking or Executing subprocesses from mod\_perl**
- **Memory leakage**

- Checking script modification times
- Cached `stat()` calls
- Be carefull with symbolic links
- Limiting the size of the processes
- Limiting the resources used by `httpd` children
- Limiting the request rate speed (robots blocking)
- Benchmarks. Impressing your Boss and Colleagues.
- Tuning the Apache's configuration variables for the best performance

- Persistent DB Connections
- Using `$|=1` under `mod_perl` and better `print()` techniques.
- Object Methods Calls Versus Function Calls
- Sending plain HTML as a compressed output

## 2.2 Performance: An Overall picture

- The definition of *Performance*
- 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.
- We will see more examples later
- 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.

## 2.3 Analysis of SW and HW Requirements

You need to analyze all of the problem's dimensions.

There are several things that need to be considered:

- How long does it take to process each request
- How many requests can you process simultaneously
- How many simultaneous requests are you planning to get

## **Solutions:**

- Request processing time is a function of optimized code among other factors.
- A concurrency level is a function of RAM you have in your server.
- Is it better to switch to another, possibly just as inefficient language will actually cost more than throwing another Ultra 2 into the rack.
- Ask yourself whether switching to another language will even help.
- In some applications, a huge chunk of memory is needed e.g. to link in Oracle runtime libraries. So you would pay this price even if you switch from Perl to C.

- How many simultaneous requests?
- Are you really expecting 8 million hits per day?
- What is the expected peak load?
- What kind of response time do you need to guarantee?
- Remember that these numbers might change drastically when you apply code changes and your site becomes more popular.
- Remember that when the you get a very high hits rate, the requirements wouldn't grow lineary by exponential!

## 2.4 Sharing Memory

- A very important point is the sharing of memory.
- Saving more memory by sharing it between child processes.
- OS support is required
- This is only possible when you preload code at server startup
- During a child process' life, its memory pages becomes unshared
- 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 explain in a moment) will hit almost at random.
- Using `MaxRequestsPerChild` to balance the memory that stays shared against the time
- 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.

- Your goal is not to have `MaxRequestsPerChild` to be 10000.
- Having a child serving 300 requests on precompiled code is already a huge speedup
- Copy-n-Write happens when child's memory pages are getting dirty
- 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 lose the benefits of the memory sharing.

## 2.5 How Shared My Memory Is

- How much shared memory do you have?
- You can see it by either using the memory utils that comes with your system like `top(1)` and `ps(1)`.
- or you can deploy `GTop` module:

```
print "shared memory of the current process: ",  
      GTop->new->proc_mem($$)->share,"\n";  
  
print "Total shared memory: ",  
      GTop->new->mem->share,"\n";
```

## 2.6 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.

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

- But even a better approach is to create a separate startup file 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 with `mod_perl` you will want to precompile the methods at initialization time.

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

- 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.

## 2.6.1 Preload Perl modules - Real Numbers

- I have conducted a few tests to benchmark the memory usage when some modules are preloaded.
- The first set of tests checks the memory use with Perl Modules preload (only `CGI.pm`).
- The second set checks the compile method of `CGI.pm`.
- The third test checks the benefit of Perl Modules preloading but a few of them and also the effect of precompiling the Registry modules with `Apache::RegistryLoader`.

## Test One

```
use strict;  
use CGI ();  
my $q = new CGI;  
print $q->header;  
print $q->start_html, $q->p( "Hello" );
```

- **Server restarted**

- Before the CGI.pm preload: (No other modules preloaded)

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	87004	0.0	0.0	1060	1524	-	A	16:51:14	0:00	httpd
httpd	240864	0.0	0.0	1304	1784	-	A	16:51:13	0:00	httpd

- After running a script which uses CGI's methods (no imports):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	188068	0.0	0.0	1052	1524	-	A	17:04:16	0:00	httpd
httpd	86952	0.0	1.0	2520	3052	-	A	17:04:16	0:00	httpd

- Observation: child httpd has grown up by 1268K

- **Server restarted**

- After the CGI.pm preload:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	240796	0.0	0.0	1456	1552	-	A	16:55:30	0:00	httpd
httpd	86944	0.0	0.0	1688	1800	-	A	16:55:30	0:00	httpd

- after running a script which uses CGI's methods (no imports):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	86872	0.0	0.0	1448	1552	-	A	17:02:56	0:00	httpd
httpd	187996	0.0	1.0	2808	2968	-	A	17:02:56	0:00	httpd

- Observation: child httpd has grown up by 1168K, 100K less then without preload - good!

- **Server restarted**

- After CGI.pm preloaded and compiled with  
CGI->compile(' :all' );

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	86980	0.0	0.0	2836	1524	-	A	17:05:27	0:00	httpd
httpd	188104	0.0	0.0	3064	1768	-	A	17:05:27	0:00	httpd

- After running a script which uses CGI's methods (no imports):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	86980	0.0	0.0	2828	1524	-	A	17:05:27	0:00	httpd
httpd	188104	0.0	1.0	4188	2940	-	A	17:05:27	0:00	httpd

- Observation: child httpd has grown up by 1172K No change!
- So what does CGI->compile(' :all' ) help?

- it's because we never use all of the methods CGI provides -- so in real use it's faster.
- So you might want to compile only the tags you are about to use -- then you will benefit for sure.

## Test Two

```
use strict;  
use CGI qw(:all);  
print header, start_html, p("Hello");
```

- **Server restarted**

- After `CGI.pm` was preloaded and NOT compiled with `CGI->compile(':all')`:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	17268	0.0	0.0	1456	1552	-	A	18:02:49	0:00	httpd
httpd	86904	0.0	0.0	1688	1800	-	A	18:02:49	0:00	httpd

- After running a script which imports symbols (all of them):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	17268	0.0	0.0	1448	1552	-	A	18:02:49	0:00	httpd
httpd	86904	0.0	1.0	2952	3112	-	A	18:02:49	0:00	httpd

- Observation: child httpd has grown up by 1264K

- **Server restarted**

- After CGI.pm was preloaded and compiled with  
CGI->compile(':all'):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	86812	0.0	0.0	2836	1524	-	A	17:59:52	0:00	httpd
httpd	99104	0.0	0.0	3064	1768	-	A	17:59:52	0:00	httpd

- After running a script which imports symbols (all of them):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	86812	0.0	0.0	2832	1436	-	A	17:59:52	0:00	httpd
httpd	99104	0.0	1.0	4884	3636	-	A	17:59:52	0:00	httpd

- Observation: child httpd has grown by 1868K. Why?
- Isn't CGI::compile(':all') supposed to make children to share the compiled code with parent?

- It does work as advertised, but if you pay attention in the code we have called only three `CGI.pm`'s methods
- Just saying `use CGI qw(:all)` doesn't mean we compile the all available methods - we just import their names.
- So actually this test is misleading.
- Execute `compile()` only on the methods you are actually using and then you will see the difference.

## Test Three

```
use strict;  
use CGI;  
use Data::Dumper;  
use Storable;  
[and many lines of code, lots of globals - so the code is huge!]
```

- **Server restarted**

- Nothing preloaded at startup:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	90962	0.0	0.0	1060	1524	-	A	17:16:45	0:00	httpd
httpd	86870	0.0	0.0	1304	1784	-	A	17:16:45	0:00	httpd

- Script using CGI (methods), Storable, Data::Dumper called:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	90962	0.0	0.0	1064	1436	-	A	17:16:45	0:00	httpd
httpd	86870	0.0	1.0	4024	4548	-	A	17:16:45	0:00	httpd

- Observation: child httpd has grown by 2764K

- **Server restarted**

- Preloaded CGI (compiled), Storable, Data::Dumper at startup:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	26792	0.0	0.0	3120	1528	-	A	17:19:21	0:00	httpd
httpd	91052	0.0	0.0	3340	1764	-	A	17:19:21	0:00	httpd

- Script using CGI (methods), Storable, Data::Dumper called

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	26792	0.0	0.0	3124	1440	-	A	17:19:21	0:00	httpd
httpd	91052	0.0	1.0	6568	5040	-	A	17:19:21	0:00	httpd

- Observation: child httpd has grown by 3276K. Ouch: 512K more!!!

- The reason is that when you preload at the startup all of the methods, they all are being precompiled, there are many of them and they take a big chunk of memory.

- If you don't use the `compile()` method, only the functions that are being used will be compiled.
- Yes, it will slightly slow down the first response of each process, but the actual memory usage will be lower.
- BTW, if you write in the script:

```
use CGI qw(all);
```

- Only the symbols of all functions are being imported.
- While they are taking some space, it's smaller than the space that a compiled code of these functions might occupy.

- **Server restarted**

- All the above modules + the above script PreCompiled with Apache::RegistryLoader at startup:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	43224	0.0	0.0	3256	1528	-	A	17:23:12	0:00	httpd
httpd	26844	0.0	0.0	3488	1776	-	A	17:23:12	0:00	httpd

- Script using CGI (methods), Storable, Data::Dumper called:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	43224	0.0	0.0	3252	1440	-	A	17:23:12	0:00	httpd
httpd	26844	0.0	1.0	6748	5092	-	A	17:23:12	0:00	httpd

- Observation: child httpd has grown even more 3316K ! Does not seem to be good!

- **Summary:**
- 1. Perl Modules Preloading gave good results everywhere.
- 2. CGI.pm's `compile()` method seems to use even more memory. It's because we never use all of the methods CGI provides. Do `compile()` only the tags that you are going to use and you will save the overhead of the first call for each has not yet been called method, and the memory - since compiled code will be shared across all the children.
- 3. Apache: `:RegistryLoader` might make scripts load faster on the first request after the child has just started but the memory usage is worse!! See the numbers by yourself.
- HW/SW used : The server is apache 1.3.2, mod\_perl 1.16 running on AIX 4.1.5 RS6000 1G RAM.

## 2.7 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.
- The code should be added to the startup file

```
use Apache::RegistryLoader ( );  
my $url = Apache::RegistryLoader->new;  
$url->handler($url);
```

- This code recursively loads all the script it finds

```
use File::Find 'finddepth';
use Apache::RegistryLoader ();
{
    my $perl_dir = "perl/";
    my $r1 = Apache::RegistryLoader->new;
    finddepth(sub {
        return unless /\.pl$/;
        my $url = "$File::Find::dir/$_";
        print "pre-loading $url\n";

        my $status = $r1->handler($url);
        unless($status == 200) {
            warn "pre-load of '$url' failed, status=$status\n";
        }
    }, $perl_dir);
}
```

- You might need to provide a second argument to `handler()`, to help it translate URIs into a filepaths.

## 2.8 Global vs Fully Qualified 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";
```

## 2.9 PerlSetupEnv Off

- PerlSetupEnv Off is another optimization you might consider.
- *mod\_perl* fiddles with the environment to make it appear as if the script were being called under the CGI protocol.
- For example, the `$ENV{QUERY_STRING}` environment variable is initialized with the contents of *Apache::args()*,
- and `$ENV{SERVER_NAME}` is filled in from the value returned by *Apache::server\_hostname()*.
- But `%ENV` population is expensive.

- Those who have moved to the Perl Apache API no longer need this extra %ENV population, can gain by turning it **Off**.
- By default it is On.
- Note that you can still set ENV variables. e.g. when you use the following configuration:

```
<Location /perl>  
PerlSetupEnv Off  
PerlSetEnv TEST hi  
SetHandler perl-script  
PerlHandler Apache::RegistryNG->handler  
Options +ExecCGI  
</Location>
```

- A script having a `print Data::Dumper(\%ENV)` line, prints:

```
$VAR1 = {
    'GATEWAY_INTERFACE' => 'CGI-Perl/1.1',
    'MOD_PERL' => 'mod_perl/1.21_01-dev',
    'PATH' => '/usr/lib/perl5/5.00503:... snipped ...',
    'TEST' => 'hi'
};
```

## 2.10 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 (*very heavy*) and configured to serve mod\_perl scripts.
- We named them `httpd_docs` and `httpd_perl` respectively.

- The two servers coexist at the same IP address by listening to different ports:
- `httpd_docs` listens to port 80
- and `httpd_perl` listens to port 8080
- Now I am going to convince you that you **want** to use a proxy server (in the http accelerator mode).

## The advantages:

- **Proxy cache**
  - Allow serving of static objects from the proxy's cache (objects that previously were entirely served by the `httpd_docs` server).
- **Less IO**
  - You get less I/O activity reading static objects from the disk (proxy serves the most “popular” objects from RAM - 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 static objects much faster.

- **Output Buffering**

- 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.
- Using numbers is always more convincing :)

- 28.8 kbps connection => **3.6 kbytes/sec**.
- An average generated HTML page to be of **10kb**
- An average script that generates this output in **0.5 secs**.
- How long will the server wait before the user gets the whole output response?
- A simple calculation reveals pretty scary numbers:  
$$(0.5 \text{ sec} * 10 \text{ kb}) / 3.6 \text{ kb/sec} \sim 6 \text{ sec}$$
- It will have to wait for another 6 secs (20kb/3.6kb)
- When it could serve 11 more dynamic requests in this time.

6 sec / 0.5 sec - 1 = 11

- But the generated pages are generally much bigger than 10Kb
- and users tend to open more than one browser at the same time
- Result: The waiting time can grow 10 times and more

- **Hiding Implementation Details**

- We are going to hide the details of the server's implementation.
- Users will never see ports in the URLs (more on that topic later).

- 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 that you can control.
- You can actually shut down a server, without the user even noticing, because the front end server will dispatch the jobs to other servers.
- (This is called a Load Ballancing and it's a pretty big issue, which will not be discussed here)

- **Security protection**

- 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.

## The disadvantages

- **Administration overhead**
  - You have 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 run at boot and reboot as appropriate.
  - Also, you might want to set up the crontab to run a watchdog script.

- **Memory Usage**

- 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 a 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.

## 2.11 KeepAlive

- If your `mod_perl` server's `httpd.conf` includes the following directives:

```
KeepAlive On  
MaxKeepAliveRequests 100  
KeepAliveTimeout 15
```

- you've gotten a real performance penalty,
- since after completing each request processing, the process will wait for `KeepAliveTimeout` seconds before closing the connection and thus not serving other requests at this time.

- You will need many more processes on a server with high traffic.
- If you use some server status reporting tools, you will see the process in *K* status when it's in `KeepAlive` status.
- Most chances are that you don't want this feature to be enabled. So set it Off with:

### `KeepAlive Off`

- the other two directives don't matter anymore.

## 2.12 Upload/Download of Big Files

- If some particular script's main functionality is uploading or downloading of big files, you probably want it to be executed on plain apache server under `mod_cgi`.
- Taken of course that the script requires none of the functionalities the `mod_perl` server provides. Like custom authentication handlers.
- You don't want to tie up your precious `mod_perl` backend server children doing something as long and dumb as transferring a file.

- Also, the user won't really see any important performance benefits from `mod_perl` anyway, since the upload may take up to several minutes, and the overhead saved by `mod_perl` is typically under one second.

## 2.13 Forking or Executing 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*:

```
use POSIX qw(setsid);  
@params=FreezeThaw::thaw(shift @ARGV);  
# check that @params is ok  
close STDIN;  
close STDOUT;  
close STDERR;  
# you might need to reopen the STDERR  
# 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`.

## Forking example:

```
if (fork) {  
    #do nothing  
} else {  
    system("echo Hi");  
    CORE::exit(0);  
}
```

- Parent immediately continues with the code that comes up after the fork
- Child executes `system("echo Hi")` and then terminates itself.
- Notice that I use `CORE::exit (exit() == Apache::exit` under Registry and friends)

## Forking gory details:

- Normally, every process has its parent.
- Many processes are children of the `init` process, whose `PID` equals to 1.
- When you fork a process you must `wait()` or `waitpid()` for it to finish.
- If you don't wait for it becomes a zombie.
- Zombie, is a process that doesn't have a father.
- When the child quits, it reports the termination to his parent.

- If no one `wait()`s to collect the exit status of the child, it gets “confused” and becomes a ghost process, that can be seen, but not killed.
- It will be killed only when you stop the `httpd` process that spawned it!
- (generally `top()`/`ps()` utilities display these processes with `<defunc>` tag, and you will see an increment of the zombies counter reported when doing `top()`.)
- These zombie processes can take up system resources and are generally undesirable.

## The proper fork is:

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

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    waitpid($kid,0);
    print "Parent has finished\n";
} else {
    # do something
    CORE::exit(0);
}
```

- But in most cases the only reason you would want to fork is when you need to spawn a process that would take a lot of time to complete.

- So if the server child that spawns this process has to wait for it to finish, you gained nothing.
- You cannot neither wait for its completion, nor continue because you will get yet another zombie process.
- The simplest solution is to ignore your dead children (this doesn't work everywhere, however).

```
$SIG{CHLD} = IGNORE;
```

- All the processes will be collected by the `init` process and prevent from them to become zombies.
- Note, that you cannot localize this setting with `local()`. If you do, it wouldn't take the desired effect.

- The child must close all the pipes to the connection socket that were opened by the parent process (STDIN and STDOUT)
- You may need to close and reopen a STDERR filehandler
- (It's opened to append to the error\_log file as inherited by parent, so chances are that you want it to leave untouched).

**So now the code would look like:**

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

$SIG{CHLD} = IGNORE;

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    print "Parent has finished\n";
} else {
    close STDIN;
    close STDOUT;
    close STDERR;
    # do something
    CORE::exit(0);
}
```

- Notice that `waitpid()` call has gone

## A double fork approach:

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

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    waitpid($kid,0);
} else {
    defined (my $grandkid = fork) or die "Kid cannot fork: $!\n";
    if ($grandkid) {
        CORE::exit(0);
    } else {
        # code here
        close STDIN;
        close STDOUT;
        close STDERR;
        # do something long lasting
        CORE::exit(0);
    }
}
```

- Grandkid becomes a "*child of init*" (parent process ID is 1).
- Note that the last two solutions do allow you to know the exit status of the process, but in our case we don't want to.

## use a different *SIGCHLD* handler:

```
use POSIX 'WNOHANG';
$SIG{CHLD} = sub { while( waitpid(-1,WNOHANG)>0 ) {} };
```

- Which is useful when you `fork()` more than once process.
- The arguments tell `waitpid()` to reap the next child that's available,
- and prevent the call from blocking if there happens to be no child ready from reaping.
- The handler will loop until `waitpid()` returns a negative number or zero, indicating that no more reappable children remain.

## 2.14 Memory leakage

- Scripts under `mod_perl` can very easily leak memory!
- Global variables stay around indefinitely
- Lexical variables 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 $!;  
local $/ = undef; # will read the whole file in  
$content = <IN>;  
close IN;
```

- If your file is 5Mb, the child who served that script will grow exactly by that size.
- 20 children == 20\*5M = 100M of RAM!
- Solutions: Processing line at a time. Using temp files.

**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.
- We have some `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.
- $(5+5) * 20 = 200\text{Mb}$

- Solution: whenever you think the variable can grow bigger than few Kb, pass it by reference!
- For example a test flat file database can be small while testing
- Once put in production it grows and suddenly the code slows down.

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

- 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;  
    # nothing returned - the variable $content outside has been  
    # already modified  
}
```

### **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.
- Let the DB engine to do as much processing as possible.
- Don't copy the records into your program's namespace for processing
- Get only the records you need
- Get the sorted records
- We will use DBI for this:

```
$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>
```

- The httpd process will grow by the size of the variables that have been allocated for the records that matched the query.
- Remember to multiply it by the number of the children
- 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>";
```

- 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.
- It should be much faster, since you can remove any **SORTBY** and alike attributes.

## 2.15 Checking script modification times

- Under `Apache::Registry` the requested CGI script is always being `stat()`'ed to check whether it was modified.
- It adds a very little overhead, but if you are into squeezing all the juices from the server, you might want to save this call.
- If you do -- take a look at `Apache::RegistryBB` (`BareBones`) module.

## 2.16 Cached stat() calls

- When you do a `stat()` or its variations (`-M` - modification time, `-A` last access time, `-C` inode-change time, and other),
- the information is being cached, so if you need to make an additional check for the same file,
- save the overhead of this check and use a `_` variable instead.
- For example when testing for existence and read permissions you might use:

```
my $filename = "./test";
# two stat() calls
print "OK\n" if -e $filename and -r $filename;
my $mod_time = (-M $filename) * 24 * 60 * 60;
print "$filename was modified $mod_time seconds ago\n";
```

- or the more efficient (two `stat()` syscalls saved)!:

```
my $filename = "./test";
# two stat() calls
print "OK\n" if -e $filename and -r _;
my $mod_time = (-M _) * 24 * 60 * 60;
print "$filename was modified $mod_time seconds ago\n";
```

## 2.17 Be carefull with symbolic links

- As you know Apache::Registry caches the scripts based on their URI.

```
% ln -s /home/httpd/perl/news/news.pl /home/httpd/perl/news.pl
```

- Now the script can be reached as `/news/news.pl` and `/news.pl` URIs.
- It doesn't really matter until you advertise the two URIs, and users reach the same script from both of them.

- The moment this happens, you will get the same script cached twice!
- Use `/perl-status` location handler to see all the compiled scripts and their packages.
- In our example when requesting:  
<http://localhost/perl-status?rgysubs> you would see:  
  
`Apache::ROOT::perl::news::news_2ep1`  
`Apache::ROOT::perl::news_2ep1`
- After the both URIs have been requested from the same child process that happened to serve your request.
- To make the debug easier run the server in a single mode.

## 2.18 Limiting the size of the processes

- Apache::SizeLimit allows you to kill off Apache httpd processes if they grow too large.

```
# in your startup.pl:  
use Apache::SizeLimit;  
$Apache::SizeLimit::MAX_PROCESS_SIZE = 10000; # in KB
```

```
# in your httpd.conf:  
PerlFixupHandler Apache::SizeLimit
```

- By using this module, you should be able to discontinue using the Apache configuration directive `MaxRequestsPerChild`,

- although for some users, using both in combination does the job.

## 2.19 Limiting the resources used by httpd children

- Apache::Resource uses the BSD::Resource module, which uses the C function `setrlimit()` to set limits on system resources such as memory and cpu usage.
- To configure use:

```
PerlModule Apache::Resource
# set child memory limit in megabytes
# (default is 64 Meg)
PerlSetEnv PERL_RLIMIT_DATA 32:48

# set child CPU limit in seconds
```

```
# (default is 360 seconds)
PerlSetEnv PERL_RLIMIT_CPU 120
```

### **PerlChildInitHandler Apache::Resource**

- If you configure `Apache::Status`, it will let you review the resources set this way.
- The following limit values are in megabytes: `DATA`, `RSS`, `STACK`, `FSIZE`, `CORE`, `MEMLOCK`;
- all others are treated as their natural unit.
- Prepend `PERL_RLIMIT_` for each one you want to use.
- Refer to `setrlimit` man page on your OS for other possible resources.

- If the value of the variable is of the form S:H, S is treated as the soft limit, and H is the hard limit.
- If it is just a single number, it is used for both soft and hard limits.
- To debug add:

```
<Perl>  
    $Apache::Resource::Debug = 1;  
    require Apache::Resource;  
</Perl>  
PerlChildInitHandler Apache::Resource
```
- and look in the `error_log` to see what it's doing.

## 2.20 Limiting the request rate speed (robots blocking)

- A limitation of using pattern matching to identify robots is that it only catches the robots that you know about, and only those that identify themselves by name.
- A few devious robots masquerade as users by using user agent strings that identify themselves as conventional browsers.
- To catch such robots, you'll have to be more sophisticated.
- Apache::SpeedLimit comes for you to help, see:

- [http://www.modperl.com/chapters/ch6.html#Blocking\\_Greedy\\_Clients](http://www.modperl.com/chapters/ch6.html#Blocking_Greedy_Clients)

## 2.21 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)

## 2.21.1 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."
```

## 2.21.2 *PerlHandler's Benchmarking*

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

## 2.22 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.
- 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.

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

## Get armed

- 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 ApacheBench (`ab`) utility that comes with apache distribution, a `crashme` script which uses `LWP::Parallel::UserAgent` or `httperf`.

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

## 2.22.1 *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 :).

- We will simulate 10 users concurrently requesting a very light script at `www.example.com:81/test/test.pl`.

- Each “user” makes 10 requests.

```
% ./ab -n 100 -c 10 www.example.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:

Completed 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.example.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.example.com:81/perl/access/access.cgi
Complete requests:    1000
Failed requests:      0
Requests per second: 133.01
```

- We 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 each 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.example.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!)

- 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
- 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 (back in 1998):

**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 mysql bounded)

- [http://www.example.com:81/perl/access/access.cgi?do\\_sub=query\\_form](http://www.example.com:81/perl/access/access.cgi?do_sub=query_form)

- with configuration:

```
MinSpaServers      8
MaxSpaServers     16
StartServers      10
MaxClients        50
MaxRequestsPerChild 5000
```

- gives us:

NR	NC	RPS	comment
10	10	3.33	# not a reliable figure
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 *Requests per second* is almost the same
- given that all the requests have been served,
- as 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.example.com:81/perl/access/access.cgi>), it's the same script that just returns a HTML form, without making any SQL queries.

```

MinpareServers      8
MaxpareServers     16
StartServers       10
MaxClients         50
MaxRequestsPerChild 5000

```

```

NR    NC    RPS    comment
-----

```

```

10    10    26.95  # not a reliable figure
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 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.

```
Min spare servers      8
Max spare servers     10
Start servers         10
Max clients           10
Max requests per child 5000
```

```
NR    NC    RPS    comment
-----
  10   10   23.87  # not a reliable figure
 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.

- 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!

## 2.22.2 Tuning with *httperf*

- *httperf* is an utility written by David Mosberger.
- Just like ApacheBench--it measures the performance of the webserver.
- A sample command line is shown below:

```
% httperf --server hostname --port 80 --uri /test.html \  
--rate 150 --num-conn 27000 --num-call 1 --timeout 5
```

- And here is a result that one might get:

```
Total: connections 27000 requests 26701 replies 26701 test-duration 179.996 s  
  
Connection rate: 150.0 conn/s (6.7 ms/conn, <=47 concurrent connections)  
Connection time [ms]: min 1.1 avg 5.0 max 315.0 median 2.5 stddev 13.0  
Connection time [ms]: connect 0.3  
  
Request rate: 148.3 req/s (6.7 ms/req)
```

Request size [B]: 72.0

Reply rate [replies/s]: min 139.8 avg 148.3 max 150.3 stddev 2.7 (36 samples)

Reply time [ms]: response 4.6 transfer 0.0

Reply size [B]: header 222.0 content 1024.0 footer 0.0 (total 1246.0)

Reply status: 1xx=0 2xx=26701 3xx=0 4xx=0 5xx=0

CPU time [s]: user 55.31 system 124.41 (user 30.7% system 69.1% total 99.8%)

Net I/O: 190.9 KB/s (1.6\*10<sup>6</sup> bps)

Errors: total 299 client-timo 299 socket-timo 0 connrefused 0 connreset 0

Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0

## 2.22.3 *Tuning with crashme script*

- This is another crashme suite originally written by Michael Schilli
- The handouts include the code.
- 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 :).

- **Sample output:**

```
URL(s) : http://www.example.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
```

## 2.22.4 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 -- 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 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.

## Real Memory Occupation:

Your children can share the memory between them.

- 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:

**Total\_RAM** = 500Mb

**Max\_Process\_Size** = 10Mb

**Shared\_RAM\_per\_Child** = 4Mb

500 - 10

**MaxClients** = ----- = 81

10 - 4

- 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 the sharing level, let's say:

**Total\_RAM** = 500Mb  
**Max\_Process\_size** = 10Mb  
**Shared\_RAM\_per\_Child** = 8Mb

500 - 10

**MaxClients** = ----- = 245

10 - 8

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

## 2.22.5 *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 solves some memory leakage problems caused by sloppy programming practices,

- whereas a child process consumes 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 has consumed, when it exits after `MaxRequestsPerChild` requests.

- if you set this number too low, you will lose a fraction of the speed bonus you receive with `mod_perl`.
- Consider using `Apache::PerlRun` if this is the case.
- Another approach is to use `Apache::SizeLimit`.

## 2.22.6 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

## 2.22.7 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.
- 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.

## 2.23 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 DBI, DBD::\* and ApacheDBI\* modules!

```
db.pl
-----
use DBI;
use strict;

my $dbh = DBI->connect( 'DBI:mysql:database', 'user', 'password',
    { autocommit => 0 }
    ) || die $DBI::errstr;

...rest of the program
```

## 2.23.1 Preopening Connections at the Child Process' Fork Time

- 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.
- It will establish a connection when a child is started in that child process.

## 2.23.2 Caching `prepare()` statements

- You can also benefit from persistent connections by replacing `prepare()` with `prepare_cached()`.
- That way you will always be sure that you have a good statement handle and you will get some caching benefit.
- The downside is that you are going to pay for DBI to parse your SQL and do a cache lookup every time you call `prepare_cached()`.
- Be warned that some databases doesn't support caches of prepared plans. (e.g PostgreSQL and Sybase).

## 2.23.3 Handling Timeouts

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

## 2.24 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 multi-line 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).

- An even better solution is to keep the buffering enabled, and use a Perl API `rflush()` call to flush the buffers when wanted.
- This way you can aggregate in the buffer the top of the page you are going to send to user,
- and flush it a moment before you are going to do some lengthy operation, like DB query.
- So you kill the two birds in one shoot:
- You show some of the data to the user immediately,
- so user will feel that something is actually happening,

- and you almost have no performance hit caused by disabled buffering.

```
use CGI ();
my $r = shift;
my $q = new CGI;
print $q->header( 'text/html' );
print $q->start_html;
print $q->p( "Searching...Please wait" );
$r->rflush;
# imitate a lengthy operation
for (1..5) {
    sleep 1;
}
print $q->p( "Done!" );
```

## 2.25 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.

# 2.26 Object Methods Calls Versus Function Calls

- Which subroutine calling form is more efficient: OOP methods or functions?

## The Overhead with Light Subroutines

- Let's do a benchmarking.
- Using an empty subroutine

```
package Foo;  
use strict;  
use Benchmark;  
sub bar { };  
  
timethese(50_000, {  
    method => sub { Foo->bar() },  
    function => sub { Foo::bar('Foo') };  
});
```

- The benchmarking result:

```
Benchmark: timing 50000 iterations of function, method...  
function:  0 wallclock secs ( 0.80 usr + 0.05 sys = 0.85 CPU)  
method:   1 wallclock secs ( 1.51 usr + 0.08 sys = 1.59 CPU)
```

- *methods* is almost twice slower, than *functions*.
- 0.85 CPU compared to 1.59 CPU real execution time.
- Reasons:
  - the time taken to resolve the pointer from the object
  - to find the module it belongs to
  - and then the actual method.

## The Overhead with Heavy Subroutines

- But that doesn't mean that you shouldn't use methods.
- Subroutines generally do something.
- The more they do the smaller the overhead will be

```
package Foo;  
use strict;  
use Benchmark;  
  
sub bar {  
    my $class = shift;  
  
    my ($x, $y) = (100, 100);  
    $y = log ($x ** 10) for (0..20);  
};
```

```
timethese(50_000, {
    method => sub { Foo->bar() },
    function => sub { Foo::bar('Foo') };
});
```

- We get a very close benchmarks!

```
function: 33 wallclock secs (15.81 usr + 1.12 sys = 16.93 CPU)
method: 32 wallclock secs (18.02 usr + 1.34 sys = 19.36 CPU)
```

## Summary

- In case your functions do very little, like the functions that generate HTML tags in CGI.pm,
- the overhead might become a significant one.
- If your goal is speed you might consider to use the *function* form.
- But if you write a big and complicated application, it's much better to use the *method* form, as it will make your code easier to develop, maintain and debug.

## Are All Methods Slower than Functions?

- `CGI.pm` allows you to execute its subroutines as functions and methods.

```
use CGI;  
my $q = new CGI;  
$q->param( 'x' , 5 );  
my $x = $q->param( 'x' );
```

versus

```
use CGI qw( :standard );  
param( 'x' , 5 );  
my $x = param( 'x' );
```

- As usual, let's benchmark some very light calls and compare.
- Ideally we would expect the *methods* to be slower than *functions* based on the previous benchmarks:

```
use Benchmark;

use CGI qw(:standard);
$CGI::NO_DEBUG = 1;
my $q = new CGI;
my $x;
timethese
(20000, {
    method => sub {$q->param('x',5)}; $x = $q->param('x'); },
    function => sub { param('x',5)}; $x = param('x'); },
});
```

- The benchmark is written in such a way that all the initializations are done at the beginning, so that we get as accurate performance figures as possible.

```
function: 51 wallclock secs (28.16 usr + 2.58 sys = 30.74 CPU)
method: 39 wallclock secs (21.88 usr + 1.74 sys = 23.62 CPU)
```

- As we can see *methods* are faster than *functions*, which seems to be wrong.
- The explanation lays in the way `CGI.pm` is implemented.
- `CGI.pm` uses some *fancy* tricks to make the same routine act both as a *method* and a plain *function*.
- The overhead of checking whether the arguments list looks like a *method* invocation or not, will mask the slight difference in time for the way the function was called.
- If you are intrigued and want to investigate further by yourself the subroutine you want to explore is called *self\_or\_default*.

- The first line of this function short-circuits if you are using the object methods, but the whole function is called if you are using the functional forms.
- Therefore, the functional form should be slightly slower than the object form.

- Aside of namespace pollution, when importing symbols from any module any script, its size grows by the size of the allocated space for those symbols.
- The more you import (e.g. `qw(:standard)` vs `qw(:all)`) the more memory will be used. Let's say the overhead is of size X.
- Now take the number of scripts you deploy the function method interface, let's call it Y.
- Finally let's say that you have Z number of processes.
- You will need  $X*Y*Z$  size of additional memory, taking  $X=10k$ ,  $Y=10$ ,  $Z=30$ , we get  $10k*10*30 = 3Mb!!$

- Let's benchmark the CGI.pm using GTop.pm.
- First with no exporting at all.  

```
use GTop ();  
use CGI ();  
print GTop->new->proc_mem( $$ )->size;  
  
1,949,696
```

- Now exporting a few dozens symbols:  

```
use GTop ();  
use CGI qw( :standard );  
print GTop->new->proc_mem( $$ )->size;
```

1,966,080

- And finally exporting all the symbols (about 130)

```
use GTop ();  
use CGI qw(:all);  
print GTop->new->proc_mem($$)->size;
```

1,970,176

- Results:

```
import symbols size(bytes) delta(bytes) relative to (  
-----  
( ) 1949696 0  
qw(:standard) 1966080 16384  
qw(:all) 1970176 20480
```

- So in my example above  $X=20k \Rightarrow 20K*10*30 = 6Mb$ .
- You will need 6Mb more when importing all the CGI.pm's symbols versus not importing at all.
- But generally you use more scripts, more processes and probably import more symbols from the additional modules that use deploy.
- If you are heading to performance improving direction, you will have to face the fact, that having to type `My::Module::my_method` might save you a good chunk of memory if the above call must not be called with a reference to an object, but even then it can be passed by value.

- I strongly endorse `Apache::Request (libapreq)` -- Generic Apache Request Library.
- Its guts are all written in C, giving it a significant memory and performance benefit.
- It has all the functionality `CGI.pm` has, but HTML generation functions.

## 2.27 Sending plain HTML as a compressed output

- 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.
- The 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`

- Notice that the rightmost PerlHandler must be a content producer. Use `Apache::PassFile` or another similar module.

;o)

# 3 Getting Help and Further Learning

# 3.1 What we will learn in this chapter

- Getting help
- Get help with mod\_perl
- Get help with Perl
- Get help with Perl/CGI
- Get help with Apache
- Get help with DBI

- **Get help with Squid**

## **3.2 Getting help**

1. Searchable Mailing list archive
2. The Books and Online Documentation
3. Mailing List as a last resort.

## 3.3 Get help with mod\_perl

- **mod\_perl home**

<http://perl.apache.org>

- **mod\_perl Garden project**

<http://modperl.sourceforge.org>

- **mod\_perl Books**

- **'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**

- **'Enabling web services with mod\_perl' Book**

<http://www.modperlbook.com> is the home site of the new mod\_perl book, that Eric Cholet and Stas Bekman are co-authoring together. We expect the book to be published in fall 2000.

Ideas, suggestions and comments are welcome. You may send them to [info@modperlbook.com](mailto:info@modperlbook.com) .

- **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](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](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](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 Resources Page**

[http://www.perlreference.com/mod\\_perl/](http://www.perlreference.com/mod_perl/)

- **mod\_perl mailing list**

The Apache/Perl mailing list ([modperl@apache.org](mailto: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 [modperl-subscribe@apache.org](mailto:modperl-subscribe@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.

## More archives available:

- <http://www.geocrawler.com/lists/3/web/182/0/>
- <http://www.bitmechanic.com/mail-archives/modperl/>
- <http://www.mail-archive.com/modperl%40apache.org/>
- <http://www.davin.ottawa.on.ca/archive/modperl/>
- <http://www.progressive-comp.com/Lists/?l=apache-modperl&r=1&w=2#apache-modperl>
- <http://www.egroups.com/group/modperl/>

## 3.4 Get help with Perl

- **The Perl FAQ**

<http://www.perl.com/CPAN/doc/FAQs/FAQ/PerlFAQ.html>

- **The 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](http://world.std.com/~swmcd/steven/perl/module_mechanics.html)

- This page describes the mechanics of creating, compiling, releasing and maintaining Perl modules.

## 3.5 Get help 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.stason.org/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)

## 3.6 Get help with Apache

- **Apache Project's Home**

<http://www.apache.org>

- **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>

- **mod\_rewrite Guide**

<http://www.engelschall.com/pw/apache/rewriteguide/>

## 3.7 Get help with DBI

- **Perl DBI examples**

<http://www.saturn5.com/~jwb/dbi-examples.html> (by Jeffrey William Baker).

- **DBI Homepage**

<http://www.symbolstone.org/technology/perl/DBI/>

- **DBI mailing list information**

<http://www.fugue.com/dbi/>

- **DBI mailing list archives**

<http://outside.organic.com/mail-archives/dbi-users/>

<http://www.xray.mpe.mpg.de/mailling-lists/dbi/>

- **Persistent connections with mod\_perl**

[http://perl.apache.org/src/mod\\_perl.html#PERSISTENT\\_DATABASE\\_CONNECTIONS](http://perl.apache.org/src/mod_perl.html#PERSISTENT_DATABASE_CONNECTIONS)

## 3.8 Get help 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>
- ;o)

