

1 Choosing the Right Strategy

1.1 Description

This document discusses various mod_perl setup strategies used to get the best performance and scalability of the services.

1.2 Do it like I do it!?

There is no such thing as the **right** strategy in the web server business, although there are many wrong ones. Never believe a person who says: *"Do it this way, this is the best!"*. As the old saying goes: *"Trust but verify"*. 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.

With this in mind, I will present some ways of using standalone mod_perl, and some combinations of mod_perl and other technologies. I'll describe how these things work together, offer my opinions on the pros and cons of each, the relative degree of difficulty in installing and maintaining them, and 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.

Please let me stress it again: **do not** blindly copy someone's setup and hope for a good result. Choose what is best for your situation -- it might take **some** effort to find out what that is.

In this chapter we will discuss

- **Deployment of mod_perl in Overview, with the pros and cons.**
- **Alternative architectures for running one and two servers.**
- **Proxy servers (Squid, and Apache's mod_proxy).**

1.3 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 configuration file (one big binary for mod_perl).
2. Having two binaries and two configuration files (one big binary for mod_perl and one small binary for static objects like images.)
3. Having one DSO-style binary and two configuration files, with mod_perl available as a loadable object.
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`. Later, you can decide whether to move to the second one which allows better tuning at the expense of more complicated administration, or to the third option -- the more state-of-the-art-yet-suspiciously-new DSO system, or to the fourth option which gives you even more power.

1. The first option will kill your production site if you serve a lot of static data from large (4 to 15MB) webserver processes. On the other hand, while testing you will have no other server interaction to mask or add to your errors.
2. This option allows you to tune the two servers individually, for maximum performance.

However, you need to choose between running the two servers on multiple ports, multiple IPs, etc., and you have the burden of administering more than one server. You have to deal with proxying or fancy site design to keep the two servers in synchronization.

3. With DSO, modules can be added and removed without recompiling the server, and their code is even shared among multiple servers.

You can compile just once and yet have more than one binary, by using different configuration files to load different sets of modules. The different Apache servers loaded in this way can run simultaneously to give a setup such as described in the second option above.

On the down side, you are playing at the bleeding edge.

You are dealing with a new solution that has weak documentation and is still subject to change. It is still somewhat platform specific. Your mileage may vary.

The DSO module (`mod_so`) adds size and complexity to your binaries.

Refer to the section "Pros and Cons of Building `mod_perl` as DSO for more information.

Build details: Build `mod_perl` as DSO inside Apache source tree via APACI

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.

1.4 Alternative architectures for running one and two servers

The next part of this chapter discusses the pros and the cons of each of these presented configurations. Real World Scenarios Implementation describes the implementation techniques of these schemes.

We will look at the following installations:

- **Standalone mod_perl Enabled Apache Server**
- **One Plain Apache and One mod_perl-enabled Apache Servers**
- **One light non-Apache and One mod_perl enabled Apache Servers**
- **Adding a Proxy Server in http Accelerator Mode**

1.4.1 Standalone mod_perl Enabled Apache Server

The first approach is to implement a straightforward mod_perl server. Just take your plain Apache server and add mod_perl, like you add any other Apache module. You continue to run it at the port it was using before. You probably want to try this before you proceed to more sophisticated and complex techniques.

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 and you see an enormous speedup from the first moment you start to use it.

The disadvantages:

- The process size of a mod_perl-enabled Apache server is huge (maybe 4MB at startup and growing to 10MB and more, depending on how you use it) compared to typical plain Apache. Of course if memory sharing is in place, RAM requirements will be smaller.

You probably have a few tens of child processes. The additional memory requirements add up in direct relation to the number of child processes. Your memory demands are growing by an order of magnitude, but this is the price you pay for the additional performance boost of mod_perl. With memory prices so cheap nowadays, the additional cost is low -- especially when you consider the dramatic performance boost mod_perl gives to your services with every 100MB of RAM you add.

While you will be happy to have these monster processes serving your scripts with monster speed, you should be very worried about having them 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 the static object request rate. Remember that if your mod_perl code produces HTML code which includes images, each one will turn into another static object request. Having another plain webserver to serve the static objects solves this unpleasant obstacle. Having a proxy server as a front end, caching the static objects and freeing the mod_perl processes from this burden is another solution. We will discuss both below.

- Another drawback of this approach is that when serving output to a client with a slow connection, the huge mod_perl-enabled server process (with all of its system resources) 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. As in the previous drawback, a proxy solution

can solve this problem. More on proxies later.

Proxying dynamic content is not going to help much if all the clients are on a fast local net (for example, if you are administering an Intranet.) On the contrary, it can decrease performance. Still, remember that some of your Intranet users might work from home through 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!

For implementation notes, see the "One Plain and One mod_perl enabled Apache Servers" section in the implementations chapter.

1.4.2 One Plain Apache and One mod_perl-enabled Apache Servers

As I have mentioned before, when running scripts under mod_perl you will notice that the httpd processes consume a huge amount of virtual memory -- from 5MB to 15MB and even more. That is the price you pay for the enormous speed improvements under mod_perl. (Again -- shared memory keeps the real memory that is being used much smaller :)

Using these large processes to serve static objects like images and html documents is overkill. 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 better server performance.

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

An **important** note: When a user browses static pages and the base URL in the **Location** window points to the static server, for example `http://www.example.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.example.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 fully qualified URLs and not relative ones!

`http://www.example.com/icons/arrow.gif` is a full URL, while `/icons/arrow.gif` is a

relative one. Using `<BASE HREF="http://www.example.com/">` in the generated HTML is another way to handle this problem. Also, the `httpd_perl` server could rewrite the requests back to `httpd_docs` (much slower) and you still need the attention of the heavy servers. This is not an issue if you hide the internal port implementations, so the client sees only one server running on port 80. (See Publishing Port Numbers other than 80)

The disadvantages:

- An administration overhead.
 - The need for two configuration files.
 - The need for two sets of controlling scripts (startup/shutdown) and watchdogs.
 - If you are processing log files, now you probably will have to merge the two separate log files into one before processing them.
- Just as in the one server approach, we still have the problem of a `mod_perl` process spending its precious time serving slow clients, when the processing portion of the request was completed a long time ago. Deploying a proxy solves this, and will be covered in the next section.

As with the single server approach, this is not a major disadvantage if you are on a fast network (i.e. Intranet). It is likely that you do not want a buffering server in this case.

Before you go on with this solution you really want to look at the Adding a Proxy Server in http Accelerator Mode section.

For implementation notes see the "One Plain and One `mod_perl` enabled Apache Servers" section in implementations chapter.

1.4.3 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 30M you've saved to run a few more `mod_perl` servers.

The memory savings are significantly smaller if your OS supports memory sharing with Dynamically Shared Objects (DSO) and you have configured Apache to use it. If you do allow memory sharing, 30 light Apache servers ought to use only about 3 to 4MB, because most of it will be shared. There is no memory sharing if Apache modules are statically compiled into the httpd executable.

- 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.

Another interesting choice is a kHTTPd webserver for Linux. kHTTPd is different from other web servers in that it runs from within the Linux-kernel as a module (device-driver). kHTTPd handles only static (file based) web-pages, and passes all requests for non-static information to a regular userspace-webserver such as Apache. For more information see <http://www.fenrus.demon.nl/>.

Also check out the Boa webserver: <http://www.boa.org/>

1.5 Adding a Proxy Server in http Accelerator Mode

At the beginning there were two 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 and handlers. As you remember we named them `httpd_docs` and `httpd_perl` respectively.

In the dual-server setup presented earlier the two servers coexist at the same IP address by listening to different ports: `httpd_docs` listens to port 80 (e.g. <http://www.example.com/images/test.gif>) and `httpd_perl` listens to port 8080 (e.g. <http://www.example.com:8080/perl/test.pl>). Note that we did not write <http://www.example.com:80> for the first example, since port 80 is the default port for the http service. Later on, we will be changing the configuration of the `httpd_docs` server to make it listen to port 81.

This section will attempt to convince you that you really **want** to deploy a proxy server in the http accelerator mode. This is a special mode that in addition to providing the normal caching mechanism, accelerates your CGI and mod_perl scripts.

The advantages of using the proxy server in conjunction with mod_perl are:

- Certainly the benefits of the usual use of the proxy server which allows serving of static objects from the proxy's cache. 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.
- And the extra functionality provided by the http-acceleration mode, which makes the proxy server act 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 than just words. So we are going to show a simple example from the real world.

First let's explain the abbreviation used in the networking world. If someone claims to have a 56 kbps connection -- it means that the connection is of 56 kilo-bits per second (~56000 bits/sec). It's not 56 kilo-bytes per second, but 7 kilo-bytes per second, because 1 byte equals to 8 bits. So don't let the merchants fool you--your modem gives you 7 kilo-bytes per second connection at most and not 56 kilo-bytes per second as one might think.

Another convention used in computer literature is that if you see 10Kb it usually means 10 kilo-bits and 10KB is 10 kilo-bytes. So if you see upper case B it generally refers to bytes, and lower case b to bits (and K of course means kilo and equals to 1024 or to 1000 depending on the field it's used in). Remember that the latter convention is not followed everywhere, so use this knowledge with care. This document is following this convention.

So here is the real world example. Let's look at the typical scenario with a user connected to your site with 56Kbps modem. It means that the speed of the user's link is $56/8 = 7\text{KB/s}$ per sec. Let's assume an average generated HTML page to be of 42KB and an average mod_perl script that generates this response in 0.5 second. How many responses this script could produce during the time it took for the output to be delivered to user? A simple calculation reveals pretty scary numbers:

$$42\text{KB} / (0.5\text{s} * 7\text{KB/s}) = 12$$

12 other dynamic requests could be served at the same time, if we could put mod_perl to do only what it's best at: generating responses.

This very simple example shows us that we need only one twelfth the number of children running, which means that we will need only one twelfth of the memory (not quite true because some parts of the code are shared).

But you know that nowadays scripts often return pages which are blown up with JavaScript code and similar, which can make them 100kb size and the download time will be of the order of... (This calculation is left to you as an exercise :)

Moreover many users like to open many browser windows and do many things at once (download files and browse graphically *heavy* sites). So in the speed of 7KB/sec we have assumed before, may in reality be 5-10 times slower.

- This technique allows us 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 *Load Balancing* and it's a pretty big issue which will take a book on its own to cover and therefore will not be discussed here. There is a plenty of information available at the Internet though. For more information see 'High-Availability Linux Project')

- 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.

This is true if your server runs on your localhost (127.0.0.1) which makes it impossible to connect to you back end machine from the outside. But you don't need to connect from the outside anymore. You will see why when you proceed to this techniques' implementation notes.

The disadvantages are:

- Of course there are drawbacks. Luckily, these are not functionality drawbacks, but they are more administration hassle. 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. This is something that you do once and never come back to this issue again. Also, you might want to set up the crontab to run a watchdog script that will make sure that the proxying server is running and restart it if it detects a problem, reporting the problem to the administrator on the way.
- Proxy servers can be configured to be light or heavy. The administrator must decide what gives the highest performance for his application. A proxy server like *squid* is light in the sense of having only one process serving all requests. But it can consume a lot of memory when it loads objects into memory for faster service.
- If you use the default logging mechanism for all requests on the front- and back-end servers the requests that will be forwarded to the back-end server will be logged twice, which makes it tricky to merge the two logfiles, should you want to.

One solution is to tell the heavy Apache not to bother logging requests that seem to come from the light Apache's machine. You might do this by installing a custom `PerlLogHandler` or just piping to *access_log* via `grep -v` (match all but this pattern) for the requests coming from the light Apache server. In this scenario, the *access_log* written by the light Apache is the file you should work with. But you need to look for any direct accesses to the heavy server in case the proxy server is sometimes bypassed, which can be eliminated if the server is listening only to the localhost (127.0.0.1).

If you still decide to log proxied requests at the back-end server they will be useless since instead of real remote IP of the user, you will get always the same IP of the front-end server. Later in this Chapter on page XXX (`mod_proxy_add_forward`) we present a solution for this problem.

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

Of course if you are on a very fast local area network (LAN) (which means that all your users are connected from this LAN and not from the outside), 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.

1.6 Implementations of Proxy Servers

As of this writing, two proxy implementations are known to be widely used with mod_perl, the **squid** proxy server and **mod_proxy** which is a part of the Apache server. Let's compare them.

1.6.1 *The Squid Server*

The Advantages:

- Caching of static objects. These are served much faster, assuming that your cache size is big enough to keep the most frequently 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, thus freeing mod_perl servers from waiting for the slow clients to download the data. Freed servers immediately switch to serve other requests, thus your number of required servers goes down dramatically.
- 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, by default squid only buffers in 16KB chunks so it would not allow mod_perl to complete immediately if the output is larger. (READ_AHEAD_GAP which is 16KB by default, can be enlarged in defines.h if your OS allows that).
- Speed. Squid is not very fast today when compared with the plain file based web servers available. Only if you are using a lot of dynamic features such as mod_perl or similar is there a reason to use Squid, and then only if the application and the server are designed with caching in mind.
- Memory usage. Squid uses quite a bit of memory. In fact, it caches a good part of its content in memory, to be able to serve it directly from RAM, a technique which is a lot quicker than serving from disk. However, as you already have your mod_perl server caching its code in RAM, you might not want another RAM-hogging beast taking up your precious memory (see the Squid FAQ for reference: <http://www.squid-cache.org/Doc/FAQ/FAQ-8.html>).
- HTTP protocol level. Squid is pretty much a HTTP/1.0 server, which seriously limits the deployment of HTTP/1.1 features, such as Keep-Alive requests.
- HTTP headers, dates and freshness. The squid server might give out stale pages, confusing downstream/client caches. (You update some documents on the site, but squid will still serve the old ones.)
- Stability. Compared to plain web servers, Squid is not the most stable.

The pros and cons presented above lead to the idea that you might want to use squid for its dynamic content buffering features, but only if your server serves mostly dynamic requests. So in this situation, when performance is the goal, it is better to have a plain Apache server serving static objects, and squid proxying the mod_perl enabled server only.

For implementation details, see the sections Running One Webserver and Squid in httpd Accelerator Mode and Running Two Webservers and Squid in httpd Accelerator Mode in the implementations chapter.

1.6.2 Apache's mod_proxy

I do not think the difference in speed between Apache's **mod_proxy** 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, while 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 is where you have DNS aliases that map to the same IP address and you want the redirect to port 80 and the server is on a different port and you want to keep the specific name the browser has already sent, so that 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 mod_proxy in the httpd_docs server and add a few lines to httpd.conf file.
- The ProxyPass and ProxyPassReverse directives allow you to hide the internal redirects, so if `http://example.com/modperl/` is actually `http://localhost:81/modperl/`, it will be absolutely transparent to the user. ProxyPass redirects the request to the mod_perl server, and when it gets the response, 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. See the Using mod_proxy notes for more details.
- It even does caching. You have to produce correct Content-Length, Last-Modified and Expires http headers for it to work. If some of your dynamic content does not change frequently, you can dramatically increase performance by caching it with mod_proxy.
- ProxyPass happens before the authentication phase, so you do not have to worry about authenticating twice.

- Apache is able to accelerate secure HTTP requests completely, while also doing accelerated HTTP. With Squid you have to use an external redirection program for that.
- The latest (Apache 1.3.6 and later) Apache proxy accelerated module is reported to be very stable.

For implementation see the "Using mod_proxy" section in the implementation chapter.

1.6.3 Closing Lingered Connections with Lingerd

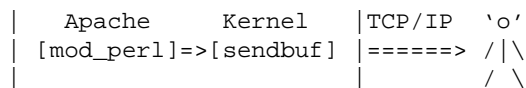
Because of some technical complications in TCP/IP, at the end of each client connection, it is not enough for Apache to close the socket and forget about it; instead, it needs to spend about one second *lingering* on the client. (More details can be found at http://httpd.apache.org/docs/misc/fin_wait_2.html)

Lingerd is a daemon (service) designed to take over the job of properly closing network connections from an http server like Apache and immediately freeing it to handle a new connection.

lingerd can only do an effective job if HTTP Keep-Alives are turned off; since Keep-Alives are useful for images, the recommended setup is to have lingerd serving mod_perl enabled Apache and plain Apache for images and other static objects.

With a lingerd setup, you don't have the proxy, so the buffering chain we have presented before for the proxy setup is much shorter here:

FIGURE:



Hence in this setup it becomes more important to have a big enough kernel send buffer.

With lingerd, a big enough kernel send buffer, and keep-alives off, the job of spoonfeeding the data to a slow client is done by the OS kernel in the background. As a result, lingerd makes it possible to serve the same load using considerably fewer Apache processes. This translates into a reduced load on the server. It can be used as an alternative to the proxy setups we have seen so far.

For more information about lingerd see: <http://www.iagora.com/about/software/lingerd/>

1.7 When One Machine is not Enough for RDBMS Database and mod_perl

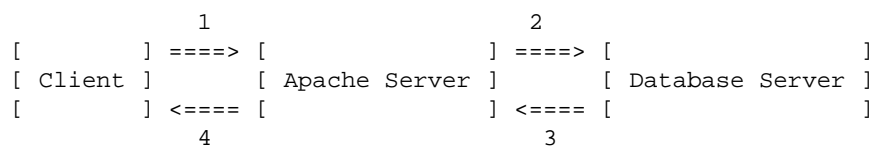
Imagine a scenario where you start your business as a small service providing web-site. After a while your business becomes very popular and at some point you understand that it has outgrown the capacity of your machine. Therefore you decide to upgrade your current machine with lots of memory, the cutting edge super expensive CPU and an ultra-fast hard disk. As a result the load goes back to normal but not for a long, as the demand for your services keeps on growing and just a little time after you've upgraded your machine, once again it cannot cope the load. Should you buy an even stronger and very expensive machine

or start looking for another solution? Let's explore the possible solution for this problem.

A typical web service consists of two main software components, the database server and the web server.

A typical user-server interaction consists of accepting the query parameters entered into an HTML form and submitted to the web server by a user, converting these parameters into a database query, sending it to the database server, accepting the results of the executed query, formatting them into a nice HTML page, and sending it to a user's Internet browser or another application that created the request (e.g. WAP).

This figure depicts the above description:



This schema is known as a 3-tier architecture in the computing world.

A 3-tier architecture means splitting up several processes of your computing solution between different machines.

- **Tier 1**

The client, who will see the data on its screen and can give instructions to modify or process the data. In our case, an Internet browser.

- **Tier 2**

The application server, which does the actual processing of the data and sends it back to the client. In our case, a mod_perl enabled Apache server.

- **Tier 3**

The database server, which stores and retrieves all the data for the application server.

We are interested only in the second and the third tiers; we don't specify user machine requirements, since mod_perl is all about server side programming. The only thing the client should be able to do is to render the generated HTML from the response, which any simple browser will do. Of course I'm not talking about the case where you return some heavy Java applets, but that movie is screened in another theater.

1.7.1 Servers' Requirements

Let's first understand what kind of software the web and database servers are, what they need to run fast and what implications they have on the rest of the system software.

The three important machine components are the hard disk, the amount of RAM and the CPU type.

Typically the mod_perl server is mostly RAM hungry, while the SQL database server mostly needs a very fast hard-disk. Of course if your mod_perl process reads a lot from the disk (which is a quite infrequent phenomenon) you will need a fast disk too. And if your database server has to do a lot of sorting of big tables and do lots of big table joins, you will need a lot of RAM too.

If we would specify average "virtual" requirements for each machine, that's what we'd get:

An *"ideal"* mod_perl machine:

- * HD: low-end (no real IO, mostly logging)
- * RAM: the more the better
- * CPU: medium to high (according to needs)

An *"ideal"* database server machine:

- * HD: high-end
- * RAM: large amounts (big joins, sorting of many records)
small amounts (otherwise)
- * CPU: medium to high (according to needs)

1.7.2 The Problem

With the database and the httpd on the same machine, you have conflicting interests.

During peak loads, Apache will spawn more processes and use RAM that the database server might have been using, or that the kernel was using on its behalf in the form of cache. You will starve your database of resources at the time when it needs those resources the most.

Disk I/O contention is the biggest time issue. Adding another disk wouldn't cut I/O times because the database is the only one who does I/O - since mod_perl processes have all their code loaded in memory. (I'm talking about code that does pure perl and SQL processing) so it's clear that the DB is I/O and CPU bounded (RAM only if there are big joins to make) and mod_perl CPU and mostly RAM bounded.

The problem exists, but it doesn't mean that you cannot run the application and the web servers on the same machine. There is a very high degree of parallelism in modern PC architecture. The I/O hardware is helpful here. The machine can do many things while a SCSI subsystem is processing a command, or the network hardware is writing a buffer over the wire.

If a process is not runnable (that is, it is blocked waiting for I/O or similar), it is not using significant CPU time. The only CPU time that will be required to maintain a blocked process is the time it takes for the operating system's scheduler to look at the process, decide that it is still not runnable, and move on to the next process in the list. This is hardly any time at all. If there are two processes and one of them is blocked on I/O and the other is CPU bound, the blocked process is getting 0% CPU time, the runnable process is getting 99.9% CPU time, and the kernel scheduler is using the remainder.

1.7.3 *The Solution*

Adding another machine, which allows a set-up where both the database and the web servers run on their own dedicated machines.

1.7.3.1 Pros

- **Hardware Requirements**

That allows you to scale two requirements independently.

If your httpd processes are heavily weighted with respect to RAM consumption, you can easily add another machine to accommodate more httpd processes, without changing your database machine.

If your database is CPU intensive, but your httpd doesn't need much CPU time, you can get low end machines for the httpd and a high end machine with a very fast CPU for the database server.

- **Scalability**

Since your web server is not depending on the database server location any more, you can add more web servers hitting the same database server, using the existing infrastructure.

- **Database Security**

Once you have multiple web server boxes the backend database becomes a single point of failure so it's a good idea to shield it from direct Internet access, something you couldn't do when you had both servers on the same machine.

1.7.3.2 Cons

- **Network latency**

A database request from a webserver to a database server running on the same machine uses UNIX sockets, compared to the TCP/IP sockets used when the client submits the query from another machine. UNIX sockets are very fast since all the communications happens within the same box, eliminating network delays. TCP/IP sockets communication totally depends on the quality and the speed of the network the two machines are connected with.

Basically, you can have almost the same client-server speed if you install a very fast and dedicated network between the two machines. It might impose a cost of additional NICs but it's probably insignificant compared to the speed up you gain.

But even the normal network that you have would probably fit as well, because the networks delays are probably much smaller than the time it takes to execute the query. In contrast to the previous paragraph, you really want to test the added overhead, since the network can be quite slow especially at the peak hours.

How do you know what overhead is a significant one? All you have to measure is the average time spent in the web server and the database server. If any of the two numbers is at least 20 times bigger than the added overhead of the network you are all set.

To give you some numbers -- if your query takes about 20 milliseconds to process and only 1 millisecond to deliver the results, it's good. If the delivery takes about half of the time the processing takes you should start considering switching to a faster and/or dedicated network.

The consequences of a slow network can be quite bad. If the network is slow mod_perl processes remain open waiting for data from the database server and eat even more RAM as new child processes pop up to handle new requests. So the overall machine performance can be worse than it was originally when you had just a single machine for both servers.

1.7.4 Three Machines Model

Since we are talking about using a dedicated machine for each server, you might consider adding a third machine to do the proxy work; this will make your setup even more flexible since it will enable you to proxy-pass all request to not just one mod_perl running box, but to many of them. It will enable you to do load balancing if and when you need it.

Generally the proxy machine can be very light when it serves just a little traffic and mainly proxy-passes to the mod_perl processes. Of course you can use this machine to serve the static content and then the hardware requirement will depend on the number of objects you will have to serve and the rate at which they are requested.

1.8 Running More than One mod_perl Server on the Same Machine.

Let's assume that you have two different sets of code which have little or nothing in common--different Perl modules, no code sharing. Typical numbers can be four megabytes of unshared and four megabytes of shared memory for each code set, plus three megabytes of shared basic mod_perl stuff. Which makes each process 17MB in size when the two code sets are loaded. (3MB (server core shared) + 4MB (shared 1st code set) + 4MB (unshared 1st code set) + 4MB (shared 2nd code set) + 4MB (unshared 2nd code set). Under this scenario:

```
Shared_RAM_per_Child : 11MB
Max_Process_Size     : 17MB
Total_RAM            : 251MB
```

We assume that four megabytes is the size of each code sets unshared memory. This is a pretty typical size of unshared memory, especially when connecting to databases, as the database connections cannot be shared. Databases like Oracle can take even more RAM per connection on top of this.

Let's assume that we have 251 megabytes of RAM dedicated to the webserver.

According to the equation developed in the section: "Choosing MaxClients":

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

$$\text{MaxClients} = (251 - 11) / (17 - 11) = 40$$

We see that we can run 40 processes, using the given memory and the two code sets in the same server.

Now consider this practical decision. Since we have recognized that the code sets are very distinct in nature and there is no significant memory sharing in place, the wise thing to do is to split the two code sets between two mod_perl servers (a single mod_perl server actually is a set of the parent process and a number of the child processes). So instead of running everything on one server, now we move the second code set onto another mod_perl server. At this point we are talking about a single machine.

Let's look at the figures again. After the split we will have 20 servers of eleven megabytes (4MB unshared + 7mb shared) and another 20 more of the same kind.

How much memory do we need now? From the above equation we derive:

$$\text{Total_RAM} = \text{MaxClients} * (\text{Max_Process_Size} - \text{Shared_RAM_per_Child}) + \text{Shared_RAM_per_Child}$$

And using the numbers (the total of 40 servers):

$$\text{Total_RAM} = 2 * (20 * (11 - 7) + 7) = 174$$

A total of 174MB of memory required. But, hey, we have 251MB of memory. We've got 77MB of memory freed up. If we recalculate again the MaxClients we will see that we can run almost 60 servers:

$$\text{MaxClients} = (251 - 7 * 2) / (11 - 7) = 59$$

So we can run about 19 more servers using the same memory size. Almost 30 servers for each code set instead of 20 originally. We have enlarged the servers pool by half without changing the machine's hardware.

Moreover this new setup allows us to fine tune the two code sets, since in reality the smaller in size code base might have a higher hit rate, so we can benefit even more.

Let's assume that based on the usage statistics we know that the first code set is called in 70% of requests and the other 30% are used by the second set. Now we assume that the first code set requires only 5MB of RAM (3MB shared plus 2MB unshared) over the basic mod_perl server size, and the second set needs 11MBytes (7MB shared and 4MB unshared).

Lets compare this new requirement with our original 50:50 setup (here we have assigned the same number of clients for each code set).

So now the first mod_perl server running the first code set will have all its processes using 8MB (3MB (server shared) + 3MB (code shared) + 2MB (code unshared)), and the second 14MB (3+7+4). Given that we have a 70:30 hits relation and that we have 251MB of available memory, we have to solve these two equations:

$$X/Y = 7/3$$

$$X*(8-6) + 6 + Y*(14-10) + 10 = 251$$

where X is the total number of the processes the first code set can use and Y the second. The first equation reflect the 70:30 hits relation, and the second uses the equation for the total memory requirements for the given number of servers and the shared and unshared memory sizes.

When we solve these equations, we find that X equals 63 and Y equals 27. So we have a total of 90 servers -- two and a half times the number of servers running compared to the original setup using the same memory size.

The hits rate optimized solution and the fact that the code sets can be different in their memory requirements, allowed us to run 30 more servers in total and gave us 33 more servers (63 versus 30) for the most wanted code base, relative to the simple 50:50 split as in the first example.

Of course if you identify more than two distinct sets of code based on your hit rate statistics, more complicated solutions may be required. You could make even more splits and run three or more mod_perl servers.

Remember that having too many running processes doesn't necessarily mean better performance because all of them will contend for CPU time slices. The more processes that are running the less CPU time each gets and the slower overall performance will be. Therefore after hitting a certain load you might want to start spreading servers over different machines.

In addition to the obvious memory saving you gain the power to troubleshoot problems that occur more easily when you have different components running on different servers. It's quite possible that a small change in the server configuration to fix or improve something for one code set, might completely break the second code set. For example if you upgrade the first code set and it requires an update of some modules that both code bases rely on. But there is a chance that the second code set won't work with a new version of a module it was relying on.

1.9 SSL functionality and a mod_perl Server

If you need SSL functionality, you can get it by adding the mod_ssl or equivalent Apache_ssl to the light front-end server (httpd_docs) or the heavy back-end mod_perl server (httpd_perl). (The configuration and installation instructions are located here.)

The question is: Is it a good idea to add mod_ssl into the back-end mod_perl enabled server? Given that your internal network is secured, or if both the front and back end servers are running on the same machine and you can ensure a safe communication between the processes, there is no need for an encrypted traffic between them.

If this is the situation you don't have to put `mod_ssl` into the already too heavy `mod_perl` server. You will have the external traffic encrypted by the front-end server, which will proxy-pass the unencrypted request and response data internally.

Another important point is if you put the `mod_ssl` on the back-end, you have to tunnel back your images to it (i.e. have the back-end serve the images) defeating the whole purpose of having the front-end lightweight server.

You cannot serve a secure page which includes non-secured information. If you fetch an html page over SSL and have an `` tag that fetches the image from the non-secure server, the image is shown broken. This is true for any other non-secured objects as well. Of course if the generated response doesn't include any embedded objects, like images -- this is not a problem.

Choosing the front-end machine to have an SSL functionality also simplifies configuration of `mod_perl` by eliminating VirtualHost duplication for SSL. `mod_perl` configuration files can be plenty difficult without the `mod_ssl` overhead.

Also assuming that you have front-end machines under-worked anyway, especially if you run a high-volume web service deploying a cluster of machines to serve requests, you save some CPU as it's known that SSL connections are about 100 times more CPU intensive than non-SSL connections.

Of course caching session keys so you don't have to set up a new symmetric key for every single connection, improves the situation. If you use the shared memory session caching mechanism that `mod_ssl` supports, then the overhead is actually rather small except for the initial connection.

But then on the other hand, why even bother to run a full scale `mod_ssl` in front? You might as well just choose a small tunnel/port forwarding application like Stunnel or one of the many other mentioned at <http://www.openssl.org/related/apps.html>.

Of course if you do heavy SSL processing ideally you should really be offloading it to a dedicated cryptography server. But this advice can be misleading based on the current status of the crypto hardware. If you use hardware you get extra speed now, but you're locked into a proprietary solution; in 6 months/one year software will have caught up with whatever hardware you're using and because software is easier to adapt you'll have more freedom to change what software you're using and more control of things. So the choice is in your hand.

1.10 Maintainers

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

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

1.11 Authors

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

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

Table of Contents:

1	Choosing the Right Strategy	1
1.1	Description	2
1.2	Do it like I do it!?	2
1.3	mod_perl Deployment Overview	2
1.4	Alternative architectures for running one and two servers	3
1.4.1	Standalone mod_perl Enabled Apache Server	4
1.4.2	One Plain Apache and One mod_perl-enabled Apache Servers	5
1.4.3	One light non-Apache and One mod_perl enabled Apache Servers	6
1.5	Adding a Proxy Server in http Accelerator Mode	7
1.6	Implementations of Proxy Servers	10
1.6.1	The Squid Server	10
1.6.2	Apache's mod_proxy	11
1.6.3	Closing Lingering Connections with Lingerd	12
1.7	When One Machine is not Enough for RDBMS Database and mod_perl	12
1.7.1	Servers' Requirements	13
1.7.2	The Problem	14
1.7.3	The Solution	15
1.7.3.1	Pros	15
1.7.3.2	Cons	15
1.7.4	Three Machines Model	16
1.8	Running More than One mod_perl Server on the Same Machine.	16
1.9	SSL functionality and a mod_perl Server	18
1.10	Maintainers	19
1.11	Authors	20