1 Real World Scenarios

1.1 Description

This chapter provides step by step installation guide for the various setups discussed in Choosing the Right Strategy.

1.2 Assumptions

I will assume for this section that you are familiar with plain (not mod_perl enabled) Apache, its compilation and configuration. In all configuration and code examples I will use *localhost* or *www.example.com* as a hostname. For the testing on a local machine, *localhost* would be just fine. If you are using the real name of your machine make sure to replace *www.example.com* with the name of your machine.

1.3 Standalone mod_perl Enabled Apache Server

1.3.1 Installation in 10 lines

The Installation is very simple. This example shows installation on the Linux operating system.

```
% cd /usr/src
% lwp-download http://www.apache.org/dist/apache_x.x.x.tar.gz
% lwp-download http://perl.apache.org/dist/mod_perl-x.xx.tar.gz
% tar 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!

Notes: Replace x.xx and x.x.x with the real version numbers of mod_perl and Apache respectively. The `z` flag tells Gnu `tar` to uncompress the archive as well as extract the files. You might need superuser permissions to do the make install steps.

1.3.2 Installation in 10 paragraphs

If you have the `lwp-download` utility installed, you can use it to download the sources of both packages:

```
% 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
```

`lwp-download` is a part of the LWP module (from the `libwww` package), and you will need to have it installed in order for mod_perl's `make test` step to pass.

Extract both sources. Usually I open all the sources in */usr/src/*, but your mileage may vary. So move the sources and `chdir` to the directory that you want to put the sources in. If you have a non-GNU `tar` utility it will be unable to decompress so you will have to unpack in two steps: first uncompress the packages with:

```
gzip -d apache_x.x.x.tar.gz
gzip -d mod_perl-x.xx.tar.gz
```

then un-tar them with:

```
tar xvf apache_x.x.x.tar
tar xvf mod_perl-x.xx.tar
```

You can probably use gunzip instead of gzip -d if you prefer.

```
% cd /usr/src
% tar xzvf apache_x.x.x.tar.gz
% tar xzvf mod_perl-x.xx.tar.gz
```

`chdir` to the mod_perl source directory:

```
% cd mod_perl-x.xx
```

Now build the Makefile. For your first installation and most basic work the parameters in the example below are the only ones you will need. `APACHE_SRC` tells the Makefile.PL where to find the Apache *src* directory. If you have followed my suggestion and have extracted both sources under the directory */usr/src*, then issue the command:

```
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
```

There are many additional optional parameters. You can find some of them later in this section and in the Server Configuration section.

While running `perl Makefile.PL ...` the process will check for prerequisites and tell you if something is missing. If you are missing some of the perl packages or other software, you will have to install them before you proceed.

Next `make` the project. The command `make` builds the mod_perl extension and also calls `make` in the Apache source directory to build `httpd`. Then we run the *test* suite, and finally *install* the mod_perl modules in their proper places.

```
% make && make test && make install
```

Note that if `make` fails, neither `make test` nor `make install` will be executed. If `make test` fails, `make install` will be not executed.

Now change to the Apache source directory and run `make install`. This will install Apache's headers, default configuration files, build the Apache directory tree and put `httpd` in it.

```
% cd ../apache_x.x.x
% make install
```

When you execute the above command, the Apache installation process will tell you how to start a freshly built webserver (you need to know the path of `apachectl`, more about that later) and where to find the configuration files. Write down both, since you will need this information very soon. On my machine the two important paths are:

```
/usr/local/apache/bin/apachectl
/usr/local/apache/conf/httpd.conf
```

Now the build and installation processes are complete.

1.3.3 Configuration

First, a simple configuration. Configure Apache as you usually would (set `Port`, `User`, `Group`, `ErrorLog`, other file paths etc).

Start the server and make sure it works, then shut it down. The `apachectl` utility can be used to start and stop the server:

```
% /usr/local/apache/bin/apachectl start
% /usr/local/apache/bin/apachectl stop
```

Now we will configure Apache to run perl CGI scripts under the `Apache::Registry` handler.

You can put configuration directives in a separate file and tell *httpd.conf* to include it, but for now we will simply add them to the main configuration file. We will add the mod_perl configuration directives to the end of *httpd.conf*. In fact you can place them anywhere in the file, but they are easier to find at the end.

For the moment we will assume that you will put all the scripts which you want to be executed by the mod_perl enabled server under the directory */home/httpd/perl*. We will alias this directory to the URI */perl*

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

```
Alias /perl/ /home/httpd/perl/

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

Now create a four-line test script in */home/httpd/perl/*:

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

Note that the server is probably running as a user with a restricted set of privileges, perhaps as user `nobody` or `www`. Look for the `User` directive in *httpd.conf* to find the userid of the server.

Make sure that you have read and execute permissions for *test.pl*.

```
% chmod u+rx /home/httpd/perl/test.pl
```

Test that the script works from the command line, by executing it:

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

You should see:

```
Content-type: text/html

It worked!!!
```

Assuming that the server's userid is `nobody`, make the script owned by this user. We already made it executable and readable by user.

```
% chown nobody /home/httpd/perl/test.pl
```

Now it is time to test that mod_perl enabled Apache can execute the script.

Start the server ('`apachectl start`'). Check in *logs/error_log* to see that the server has indeed started--verify the correct date and time of the log entry.

To get Apache to execute the script we simply fetch its URI. Assuming that your *httpd.conf* has been configured with the directive `Port 80`, start your favorite browser and fetch the following URI:

```
http://www.example.com/perl/test.pl
```

If you have the loop-back device (127.0.0.1) configured, you can use the URI:

```
http://localhost/perl/test.pl
```

In either case, you should see:

```
It worked!!!
```

If your server is listening on a port other than 80, for example 8000, then fetch the URI:

```
http://www.example.com:8000/perl/test.pl
```

or whatever is appropriate.

If something went wrong, go through the installation process again, and make sure you didn't make a mistake. If that doesn't help, read the `INSTALL` pod document (`perlpod INSTALL`) in the mod_perl distribution directory.

Now that your mod_perl server is working, copy some of your Perl CGI scripts into the directory */home/httpd/perl/* or below it.

If your programming techniques are good, chances are that your scripts will work with no modifications at all. With the mod_perl enabled server you will see them working very much faster.

If your programming techniques are sloppy, some of your scripts will not work and they may exhibit strange behaviour. Depending on the degree of sloppiness they may need anything from minor tweaking to a major rewrite to make them work properly. (See Sometimes My Script Works, Sometimes It Does Not)

The above setup is very basic, but as with Perl, you can start to benefit from mod_perl from the very first moment you try it. As you become more familiar with mod_perl you will want to start writing Apache handlers and make more use of its power.

1.4 One Plain and One mod_perl enabled Apache Servers

Since we are going to run two Apache servers we will need two complete (and different) sets of configuration, log and other files. In this scenario we'll use a dedicated root directory for each server, which is a personal choice. You can choose to have both servers living under the same roof, but it might lead to a mess, since it requires a slightly more complicated configuration. This decision might be nice since you will be able to share some directories like *include* (which contains Apache headers), but in fact this can become a problem later, when you decide to upgrade one server but not the other. You will have to solve this problem then, so why not to avoid it in first place.

From now on we will refer to these two servers as **httpd_docs** (plain Apache) and **httpd_perl** (Apache/mod_perl). We will use */usr/local* as our *root* directory.

First let's prepare the sources. We will assume that all the sources go into the */usr/src* directory. Since you will probably want to tune each copy of Apache separately, it is better to use two separate copies of the Apache source for this configuration. For example you might want only the httpd_docs server to be built with the mod_rewrite module.

Having two independent source trees will prove helpful unless you use dynamically shared objects (`DSO`) which is covered later in this chapter.

Make two subdirectories:

```
  % mkdir /usr/src/httpd_docs
  % mkdir /usr/src/httpd_perl
```

Next put a set of the Apache sources into the */usr/src/httpd_docs* directory (replace the directory */tmp* with the path to the downloaded file and `x.x.x` with the version of Apache that you have downloaded):

```
% cd /usr/src/httpd_docs
% gzip -dc /tmp/apache_x.x.x.tar.gz | tar xvf -
```

or if you have GNU tar:

```
% tar xvzf /tmp/apache_x.x.x.tar.gz
```

Just to check we have extracted in the right way:

```
% ls -l
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 apache_x.x.x/
```

Now prepare the httpd_perl server sources:

```
% cd /usr/src/httpd_perl
% gzip -dc /tmp/apache_x.x.x.tar.gz | tar xvf -
% gzip -dc /tmp/modperl-x.xx.tar.gz | tar xvf -

% ls -l
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 apache_x.x.x/
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 modperl-x.xx/
```

We are going to use a default Apache directory layout, and place each server directories under its dedicated directory. The two directories are as you have already guessed:

```
/usr/local/httpd_perl/
/usr/local/httpd_docs/
```

The next step is to configure and compile the sources: Below are the procedures to compile both servers, using the directory layout I have just suggested.

1.4.1 Configuration and Compilation of the Sources.

As usual we use *x.x.x* instead of real version numbers so this document will never become obsolete. But the most important thing -- it's not misleading. It's quite possible that since the moment this document was written a new version has come out and you will be not aware of this fact if you will not check for it.

1.4.1.1 Building the httpd_docs Server

- **Sources Configuration:**

  ```
  % cd /usr/src/httpd_docs/apache_x.x.x
  % make clean
  %  ./configure --prefix=/usr/local/httpd_docs \
     --enable-module=rewrite --enable-module=proxy
  ```

  We need the *mod_rewrite* and *mod_proxy* modules as we will see later, so we tell *./configure* to build them in.

You might want to add `--layout` to see the resulting directories' layout without actually running the configuration process.

- **Source Compilation and Installation**

  ```
  % make
  % make install
  ```

  Rename `httpd` to `http_docs`:

  ```
  % mv /usr/local/httpd_docs/bin/httpd \
       /usr/local/httpd_docs/bin/httpd_docs
  ```

  Now modify the **apachectl** utility to point to the renamed httpd via your favorite text editor or by using perl:

  ```
  % perl -pi -e 's|bin/httpd|bin/httpd_docs|' \
    /usr/local/httpd_docs/bin/apachectl
  ```

Another approach would be to use the --*target* option while configuring the source, which makes the last two commands unnecessary.

```
% ./configure --prefix=/usr/local/httpd_docs \
   --target=httpd_docs \
   --enable-module=rewrite --enable-module=proxy
% make && make install
```

Since we told *./configure* that we want the executable to be called `httpd_docs` (via `--target=httpd_docs`) -- it performs all the naming adjustment for us.

The only thing that you might find unusual, is that *apachectl* will be now called *httpd_docsctl* and the configuration file *httpd.conf* now will be called *httpd_docs.conf*.

We will leave the decision making about the preferred configuration and installation way to the reader. In the rest of the guide we will continue using the regular names resulted from using the standard configuration and the manual executable name adjustment as described at the beginning of this section .

1.4.1.2 Building the httpd_perl Server

Now we proceed with the sources configuration and installation of the *httpd_perl* server. First make sure the sources are clean:

```
% cd /usr/src/httpd_perl/apache_x.x.x
% make clean
% cd /usr/src/httpd_perl/mod_perl-x.xx
% make clean
```

It is important to **make clean** since some of the versions are not binary compatible (e.g apache 1.3.3 vs 1.3.4) so any "third-party" C modules need to be re-compiled against the latest header files.

```
  % cd /usr/src/httpd_perl/mod_perl-x.xx

  % /usr/bin/perl Makefile.PL \
        APACHE_SRC=../apache_x.x.x/src \
    DO_HTTPD=1 USE_APACI=1 EVERYTHING=1 \
        APACHE_PREFIX=/usr/local/httpd_perl \
    APACI_ARGS='--prefix=/usr/local/httpd_perl'
```

If you need to pass any other configuration options to Apache's `configure`, add them after the *--prefix* option. e.g:

```
    APACI_ARGS='--prefix=/usr/local/httpd_perl \
                --enable-module=status'
```

Notice that **all** `APACI_ARGS` (above) must be passed as one long line if you work with `t?csh`!!! However with `(ba)?sh` it works correctly the way it is shown above, breaking the long lines with '\'. As of tcsh version 6.08.0, when it passes the `APACI_ARGS` arguments to `configure` it does not alter the newlines, but it strips the backslashes, thus breaking the configuration process.

Notice that just like in httpd_docs configuration you can use `--target=httpd_perl` instead of specifying each directory separately. Note that this option has to be the very last argument in `APACI_ARGS`, otherwise 'make test' tries to run `'httpd_perl'`, which fails.

[META: It's very important to use the same compiler you build the perl with. See the section 'What Compiler Should Be Used to Build mod_perl' for more information.

[META: --- Hmm, what's the option that overrides the compiler when building Apache from mod_perl. Check also whether mod_perl supplies the right compiler (the one used for building itself) -- if it does there is no need for the above note.]

Now, build, test and install the `httpd_perl`.

```
  % make && make test && make install
```

Upon installation Apache puts a stripped version of `httpd` at */usr/local/httpd_perl/bin/httpd*. The original version which includes debugging symbols (if you need to run a debugger on this executable) is located at */usr/src/httpd_perl/apache_x.x.x/src/httpd*.

You may have noticed that we did not run `make install` in the Apache source directory. When `USE_APACI` is enabled, `APACHE_PREFIX` will specify the `--prefix` option for Apache's `configure` utility, which gives the installation path for Apache. When this option is used, mod_perl's `make install` will also `make install` for Apache, installing the httpd binary, the support tools, and the configuration, log and document trees. If this option is not used you will have to explicitly run make install in the Apache source directory.

If `make test` fails, look into */usr/src/httpd_perl/mod_perl-x.xx/t/logs* and read the error_log file. Also see make test fails.

While doing `perl Makefile.PL ...` mod_perl might complain by warning you about a missing library `libgdbm`. This is a crucial warning. See Missing or Misconfigured libgdbm.so for more info.

Now rename `httpd` to `httpd_perl`:

```
  % mv /usr/local/httpd_perl/bin/httpd \
    /usr/local/httpd_perl/bin/httpd_perl
```

Update the *apachectl* utility to drive the renamed httpd:

```
  % perl -p -i -e ’s|bin/httpd|bin/httpd_perl|’ \
    /usr/local/httpd_perl/bin/apachectl
```

1.4.2 Configuration of the servers

Now when we have completed the building process, the last stage before running the servers is to configure them.

1.4.2.1 Basic httpd_docs Server Configuration

Configuring of the `httpd_docs` server is a very easy task. Starting from version 1.3.4 of Apache, there is only one file to edit. Open */usr/local/httpd_docs/conf/httpd.conf* in your favorite text editor and configure it as you usually would, except make sure that you configure the log file directory (*/usr/local/httpd_docs/logs* and so on) and the other paths according to the layout you have decided to use.

Start the server with:

```
  /usr/local/httpd_docs/bin/apachectl start
```

1.4.2.2 Basic httpd_perl Server Configuration

Edit the */usr/local/httpd_perl/conf/httpd.conf*. As with the `httpd_docs` server configuration, make sure that `ErrorLog` and other file location directives are set to point to the right places, according to the chosen directory layout.

The first thing to do is to set a `Port` directive - it should be different from that used by the plain Apache server (`Port 80`) since we cannot bind two servers to the same port number on the same machine. Here we will use `8080`. Some developers use port `81`, but you can bind to ports below 1024 only if the server has root permissions. If you are running on a multiuser machine, there is a chance that someone already uses that port, or will start using it in the future, which could cause problems. If you are the only user on your machine, basically you can pick any unused port number. Many organizations use firewalls which may block some of the ports, so port number choice can be a controversial topic. From my experience the most popular port numbers are: `80`, `81`, `8000` and `8080`. Personally, I prefer the port `8080`. Of course with the two server scenario you can hide the nonstandard port number from firewalls and users, by using either mod_proxy’s `ProxyPass` directive or a proxy server like Squid.

For more details see Publishing Port Numbers other than 80, Running One Webserver and Squid in httpd Accelerator Mode, Running Two Webservers and Squid in httpd Accelerator Mode and Using mod_proxy.

Now we proceed to the mod_perl specific directives. It will be a good idea to add them all at the end of `httpd.conf`, since you are going to fiddle with them a lot in the early stages.

First, you need to specify the location where all mod_perl scripts will be located.

Add the following configuration directive:

```
    # mod_perl scripts will be called from
  Alias /perl/ /usr/local/httpd_perl/perl/
```

From now on, all requests for URIs starting with */perl* will be executed under mod_perl and will be mapped to the files in */usr/local/httpd_perl/perl/*.

Now we configure the */perl* location.

```
  PerlModule Apache::Registry

  <Location /perl>
    #AllowOverride None
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options ExecCGI
    allow from all
    PerlSendHeader On
  </Location>
```

This configuration causes any script that is called with a path prefixed with */perl* to be executed under the `Apache::Registry` module and as a CGI (hence the `ExecCGI`--if you omit this option the script will be printed to the user's browser as plain text or will possibly trigger a '**Save-As**' window). The `Apache::Registry` module lets you run your (carefully written) Perl CGI scripts virtually unchanged under mod_perl. The `PerlModule` directive is the equivalent of Perl's require(). We load the `Apache::Registry` module before we use it by giving the `PerlHandler Apache::Registry` directive.

`PerlSendHeader On` tells the server to send an HTTP header to the browser on every script invocation. You will want to turn this off for nph (non-parsed-headers) scripts.

This is only a very basic configuration. The Server Configuration section covers the rest of the details.

Now start the server with:

```
  /usr/local/httpd_perl/bin/apachectl start
```

1.5 Running Two webservers and Squid in httpd Accelerator Mode

While I have detailed the mod_perl server installation, you are on your own with installing the Squid server (See Getting Helped for more details). I run Linux, so I downloaded the RPM package, installed it, configured the */etc/squid/squid.conf*, fired off the server and all was set.

Basically once you have Squid installed, you just need to modify the default squid.conf as I will explain below, then you are ready to run it.

The configuration that I'm going to present works with Squid server version 2.3.STABLE2. It's possible that some directives will change in future versions.

First, let's take a look at what we have already running and what we want from squid.

Previously we have had the httpd_docs and httpd_perl servers listening on ports 80 and 8080. Now we want squid to listen on port 80, to forward requests for static objects (plain HTML pages, images and so on) to the port which the httpd_docs server listens to, and dynamic requests to httpd_perl's port. And of course collecting the generated responses, which will be delivered to the client by Squid. As mentioned before this mode is known as *httpd-accelerator* mode in proxy dialect.

Therefore we have to reconfigure the httpd_docs server to listen to port 81 instead, since port 80 will be taken by Squid. Remember that in our scenario both copies of Apache will reside on the same machine as Squid.

A proxy server makes all the magic behind it transparent to users. Both Apache servers return the data to Squid (unless it was already cached by Squid). The client never sees the other ports and never knows that there might be more than one server running. Do not confuse this scenario with mod_rewrite, where a server redirects the request somewhere according to the rewrite rules and forgets all about it. (i.e. works as a one way dispatcher, which dispatches the jobs but is not responsible for.)

Squid can be used as a straightforward proxy server. ISPs and other companies generally use it to cut down the incoming traffic by caching the most popular requests. However we want to run it in httpd accelerator mode. Two directives (httpd_accel_host and httpd_accel_port) enable this mode. We will see more details shortly.

If you are currently using Squid in the regular proxy mode, you can extend its functionality by running both modes concurrently. To accomplish this, you can extend the existing Squid configuration with **httpd accelerator mode**'s related directives or you can just create one from scratch.

Let's go through the changes we should make to the default configuration file. Since the file with default settings (*/etc/squid/squid.conf*) is huge (about 60KB) and we will not alter 95% of its default settings, my suggestion is to write a new one including only the modified directives.

We want to enable the redirect feature, to be able to serve requests by more than one server (in our case we have two: the httpd_docs and httpd_perl servers). So we specify httpd_accel_host as virtual. This assumes that your server has multiple interfaces - Squid will bind to all of them.

```
  httpd_accel_host virtual
```

Then we define the default port the requests will be sent to, unless redirected. We assume that most requests will be for static documents (also it's easier to define redirect rules for the mod_perl server because of the URI that starts with *perl* or similar). We have our httpd_docs listening on port 81:

```
  httpd_accel_port 81
```

And as described before, squid listens to port 80.

```
  http_port 80
```

We do not use `icp` (`icp` is used for cache sharing between neighboring machines, which is more relevant in the proxy mode).

```
  icp_port 0
```

`hierarchy_stoplist` defines a list of words which, if found in a URL, causes the object to be handled directly by the cache. Since we told Squid in the previous directive that we aren't going to share the cache between neighboring machines this directive is irrelevant. In case that you do use this feature, make sure to set this directive to something like:

```
  hierarchy_stoplist /cgi-bin /perl
```

where the */cgi-bin* and */perl* are aliases for the locations which handle the dynamic requests.

Now we tell Squid not to cache dynamically generated pages.

```
  acl QUERY urlpath_regex /cgi-bin /perl
  no_cache deny QUERY
```

Please note that the last two directives are controversial ones. If you want your scripts to be more compliant with the HTTP standards, according to the HTTP specification the headers of your scripts should carry the *Caching Directives*: `Last-Modified` and `Expires`.

What are they for? If you set the headers correctly, there is no need to tell the Squid accelerator **NOT** to try to cache anything. Squid will not bother your mod_perl servers a second time if a request is (a) cacheable and (b) still in the cache. Many mod_perl applications will produce identical results on identical requests if not much time has elapsed between the requests. So your Squid might have a hit ratio of 50%, which means that the mod_perl servers will have only half as much work to do as they did before you installed Squid (or mod_proxy).

Even if you insert a user-ID and date in your page, caching can save resources when you set the expiration time to 1 second. A user might double click where a single click would do, thus sending two requests in parallel. Squid could serve the second request.

But this is only possible if you set the headers correctly. Refer to the chapter Correct Headers - A quick guide for mod_perl users to learn more about generating the proper caching headers under mod_perl. In case where only the scripts under */perl/caching-unfriendly* are not *caching friendly* fix the above setting to be:

```
  acl QUERY urlpath_regex /cgi-bin /perl/caching-unfriendly
  no_cache deny QUERY
```

But if you are lazy, or just have too many things to deal with, you can leave the above directives the way we described. Just keep in mind that one day you will want to reread this section and the headers generation tutorial to squeeze even more power from your servers without investing money in more memory and better hardware.

While testing you might want to enable the debugging options and watch the log files in the directory */var/log/squid/*. But make sure to turn debugging off in your production server. Below we show it commented out, which makes it disabled, since it's disabled by default. Debug option 28 enables the debugging of the access control routes, for other debug codes see the documentation embedded in the default configuration file that comes with squid.

```
# debug_options 28
```

We need to provide a way for Squid to dispatch requests to the correct servers. Static object requests should be redirected to httpd_docs unless they are already cached, while requests for dynamic documents should go to the httpd_perl server. The configuration below tells Squid to fire off 10 redirect daemons at the specified path of the redirect daemon and (as suggested by Squid's documentation) disables rewriting of any `Host:` headers in redirected requests. The redirection daemon script is listed below.

```
redirect_program /usr/lib/squid/redirect.pl
redirect_children 10
redirect_rewrites_host_header off
```

The maximum allowed request size is in kilobytes, which is mainly useful during `PUT` and `POST` requests. A user who attempts to send a request with a body larger than this limit receives an "Invalid Request" error message. If you set this parameter to a zero, there will be no limit imposed. If you are using `POST` to upload files, then set this to the largest file's size plus a few extra KB.

```
request_body_max_size 1000 KB
```

Then we have access permissions, which we will not explain. You might want to read the documentation, so as to avoid any security problems.

```
acl all src 0.0.0.0/0.0.0.0
acl manager proto cache_object
acl localhost src 127.0.0.1/255.255.255.255
acl myserver src 127.0.0.1/255.255.255.255
acl SSL_ports port 443 563
acl Safe_ports port 80 81 8080 81 443 563
acl CONNECT method CONNECT

http_access allow manager localhost
http_access allow manager myserver
http_access deny manager
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
# http_access allow all
```

Since Squid should be run as a non-root user, you need these if you are invoking the Squid as root. The user *squid* is created when the Squid server is installed.

```
cache_effective_user squid
cache_effective_group squid
```

Now configure a memory size to be used for caching. The Squid documentation warns that the actual size of Squid can grow to be three times larger than the value you set.

```
cache_mem 20 MB
```

We want to keep pools of allocated (but unused) memory available for future use if we have the memory available of course. Otherwise turn it off.

```
memory_pools on
```

Now tighten the runtime permissions of the cache manager CGI script (`cachemgr.cgi`, which comes bundled with squid) on your production server.

```
cachemgr_passwd disable shutdown
```

If you are not using this script to manage the Squid server from remote, you should disable it:

```
cachemgr_passwd disable all
```

Now the redirection daemon script (you should put it at the location you have specified in the `redirect_program` parameter in the config file above, and make it executable by the webserver of course):

```
#!/usr/local/bin/perl -p
BEGIN{ $|=1 }
s|www.example.com(?::81)?/perl/|www.example.com:8080/perl/|o ;
```

Here is what the regular expression from above does; it matches all the URIs that include either the string *www.example.com/perl/* or the string *www.example.com:81/perl/* and replaces either of these strings with *www.example.com:8080/perl*. No matter whether the regular expression worked or not, the `$_` variable is automagically printed.

We can write the above code as the following code as well:

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

$|=1;

while (<>) {
    # redirect to mod_perl server (httpd_perl)
  print($_), next
    if s|www.example.com(:81)?/perl/|www.example.com:8080/perl/|o;

    # send it unchanged to plain apache server (http_docs)
  print;
}
```

The above redirector can be more complex of course, but you know Perl, right?

A few notes regarding the redirector script:

You must disable buffering. `$|=1;` does the job. If you do not disable buffering, `STDOUT` will be flushed only when its buffer becomes full--and its default size is about 4096 characters. So if you have an average URL of 70 chars, only after about 59 (4096/70) requests will the buffer be flushed, and the requests will finally reach the server. Your users will not wait that long, unless you have hundreds requests per second and then the buffer will be flushed very frequently because it'll get full very fast.

If you think that this is a very ineffective way to redirect, you should consider the following explanation. The redirector runs as a daemon, it fires up N redirect daemons, so there is no problem with Perl interpreter loading. Exactly as with mod_perl, the perl interpreter is loaded all the time in memory and the code has already been compiled, so the redirect is very fast (not much slower than if the redirector was written in C). Squid keeps an open pipe to each redirect daemon, thus there is no overhead of the system calls.

Now it is time to restart the server, at linux I do it with:

```
  /etc/rc.d/init.d/squid restart
```

Now the Squid server setup is complete.

Almost... When you try the new setup, you will be surprised and upset to discover port 81 showing up in the URLs of the static objects (like htmls). Hey, we did not want the user to see the port 81 and use it instead of 80, since then it will bypass the squid server and the hard work we went through was just a waste of time!

The solution is to make both squid and httpd_docs listen to the same port. This can be accomplished by binding each one to a specific interface (so they are listening to different **sockets**). Modify *httpd_docs/conf/httpd.conf*:

```
  Port 80
  BindAddress 127.0.0.1
  Listen 127.0.0.1:80
```

Now the httpd_docs server is listening only to requests coming from the local server. You cannot access it directly from the outside. Squid becomes a gateway that all the packets go through on the way to the httpd_docs server.

Modify *squid.conf*:

```
  http_port 80
  tcp_outgoing_address 127.0.0.1
  httpd_accel_host 127.0.0.1
  httpd_accel_port 80
```

Now restart the Squid and httpd_docs servers (it doesn't matter which one you start first), and voila--the port number has gone.

You must also have in the file */etc/hosts* the following entry (chances are that it's already there):

```
127.0.0.1 localhost.localdomain localhost
```

Now if your scripts are generating HTML including fully qualified self references, using 8080 or the other port, you should fix them to generate links to point to port 80 (which means not using the port at all in the URI). If you do not do this, users will bypass Squid and will make direct requests to the mod_perl server's port. As we will see later just like with httpd_docs, the httpd_perl server can be configured to listen only to requests coming from the localhost (with Squid forwarding these requests from the outside) and therefore users will not be able to bypass Squid.

To save you some keystrokes, here is the whole modified `squid.conf`:

```
http_port 80
tcp_outgoing_address 127.0.0.1
httpd_accel_host 127.0.0.1
httpd_accel_port 80

icp_port 0

acl QUERY urlpath_regex /cgi-bin /perl
no_cache deny QUERY

# debug_options 28

redirect_program /usr/lib/squid/redirect.pl
redirect_children 10
redirect_rewrites_host_header off

request_body_max_size 1000 KB

acl all src 0.0.0.0/0.0.0.0
acl manager proto cache_object
acl localhost src 127.0.0.1/255.255.255.255
acl myserver src 127.0.0.1/255.255.255.255
acl SSL_ports port 443 563
acl Safe_ports port 80 81 8080 81 443 563
acl CONNECT method CONNECT

http_access allow manager localhost
http_access allow manager myserver
http_access deny manager
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
# http_access allow all

cache_effective_user squid
cache_effective_group squid

cache_mem 20 MB

memory_pools on

cachemgr_passwd disable shutdown
```

Note that all directives should start at the beginning of the line, so if you cut and paste from the text make sure you remove the leading whitespace from each line.

1.6 Running One Webserver and Squid in httpd Accelerator Mode

When I was first told about Squid, I thought: "Hey, now I can drop the `httpd_docs` server and have just Squid and the `httpd_perl` servers". Since all my static objects will be cached by squid, I do not need the light `httpd_docs` server.

But I was a wrong. Why? Because I still have the overhead of loading the objects into Squid the first time. If a site has many of them, unless a huge chunk of memory is devoted to Squid they won't all be cached and the heavy mod_perl server will still have the task of serving static objects.

How do we measure the overhead? The difference between the two servers is in memory consumption, everything else (e.g. I/O) should be equal. So you have to estimate the time needed to fetch each static object for the first time at a peak period and thus the number of additional servers you need for serving the static objects. This will allow you to calculate the additional memory requirements. I imagine that this amount could be significant in some installations.

So on for production servers I have decided to stick with the Squid, httpd_docs and httpd_perl scenario, where I can optimize and fine tune everything. But if in your case there is almost no static objects to serve, the httpd_docs server is definitely redundant. And all you need are the mod_perl server and Squid to buffer the output from it.

If you want to proceed with this setup, install mod_perl enabled Apache and Squid. Then use a configuration similar to the previous section, but now httpd_docs is not there anymore. Also we do not need the redirector anymore and we specify `httpd_accel_host` as a name of the server and not `virtual`. Because we do not redirect there is no need to bind two servers on the same port so there are neither `Bind` nor `Listen` directives in *httpd.conf*.

The modified configuration for this simplified setup (see the explanations in the previous section):

```
httpd_accel_host put.your.hostname.here
httpd_accel_port 8080
http_port 80
icp_port 0

acl QUERY urlpath_regex /cgi-bin /perl
no_cache deny QUERY

# debug_options 28

# redirect_program /usr/lib/squid/redirect.pl
# redirect_children 10
# redirect_rewrites_host_header off

request_body_max_size 1000 KB
```

```
  acl all src 0.0.0.0/0.0.0.0
  acl manager proto cache_object
  acl localhost src 127.0.0.1/255.255.255.255
  acl myserver src 127.0.0.1/255.255.255.255
  acl SSL_ports port 443 563
  acl Safe_ports port 80 81 8080 81 443 563
  acl CONNECT method CONNECT

  http_access allow manager localhost
  http_access allow manager myserver
  http_access deny manager
  http_access deny !Safe_ports
  http_access deny CONNECT !SSL_ports
  # http_access allow all

  cache_effective_user squid
  cache_effective_group squid

  cache_mem 20 MB

  memory_pools on

  cachemgr_passwd disable shutdown
```

1.7 mod_proxy

mod_proxy implements a proxy/cache for Apache. It implements proxying capability for FTP, CONNECT (for SSL), HTTP/0.9, and HTTP/1.0. The module can be configured to connect to other proxy modules for these and other protocols.

1.7.1 Concepts and Configuration Directives

In the following explanation, we will use *www.example.com* as the main server users access when they want to get some kind of service and *backend.example.com* as a machine that does the heavy work. The main and the back-end are different servers, they may or may not coexist on the same machine.

The mod_proxy module is built into the server that answers requests to the *www.example.com* hostname. For the matter of this discussion it doesn’t matter what functionality is built into the *backend.example.com* server, obviously it’ll be mod_perl for most of us.

1.7.1.1 ProxyPass

You can use the `ProxyPass` configuration directive for mapping remote hosts into the space of the local server; the local server does not act as a proxy in the conventional sense, but appears to be a mirror of the remote server.

Let’s explore what this rule does:

```
  ProxyPass   /modperl/ http://backend.example.com/modperl/
```

When a user initiates a request to http://www.example.com/modperl/foo.pl, the request will be redirected to http://backend.example.com/modperl/foo.pl, and starting from this moment user will see http://backend.example.com/ in her location window, instead of http://www.example.com/.

You have probably noticed many examples of this from real life Internet sites you've visited. Free-email service providers and other similar heavy online services display the login or the main page from their main server, and then when you log-in you see something like *x11.example.com*, then *w59.example.com*, etc. These are the back-end servers that do the actual work.

Obviously this is not an ideal solution, but usually users don't really care about what they see in the location window. So you can get away with this approach. As I'll show in a minute there is a better solution which removes this caveat and provides even more useful functionalities.

1.7.1.2 ProxyPassReverse

This directive lets Apache adjust the URL in the `Location` header on HTTP redirect responses. This is essential for example, when Apache is used as a reverse proxy to avoid by-passing the reverse proxy because of HTTP redirects on the back-end servers which stay behind the reverse proxy. Generally used in conjunction with the `ProxyPass` directive to build a complete front-end proxy server.

```
  ProxyPass          /modperl/  http://backend.example.com/modperl/
  ProxyPassReverse   /modperl/  http://backend.example.com/modperl/
```

When a user initiates a request to http://www.example.com/modperl/foo.pl, the request will be redirected to http://backend.example.com/modperl/foo.pl but on the way back `ProxyPassReverse` will correct the location URL to become http://www.example.com/modperl/foo.pl . This happens completely transparently. The end user will never know that something has happened to his request behind the scenes.

Note that this `ProxyPassReverse` directive can also be used in conjunction with the proxy pass-through feature:

```
  RewriteRule ... [P]
```

from mod_rewrite because its doesn't depend on a corresponding `ProxyPass` directive.

1.7.1.3 Security Issues

Whenever you use mod_proxy you need to make sure that your server will not become a proxy for free riders. Allowing clients to issue proxy requests is controlled by the `ProxyRequests` directive. Its default setting is `off`, which means proxy requests are handled only if generated internally (by `ProxyPass` or `RewriteRule...[P]` directives.) Do not use the `ProxyRequests` directive on your reverse proxy servers.

1.7.2 Buffering Feature

In addition to correcting the URI on its way back from the back-end server, mod_proxy also provides buffering services which mod_perl and similar heavy modules benefit from. The buffering feature allows mod_perl to pass the generated data to mod_proxy and move on to serve new requests, instead of waiting for a possibly slow client to receive all the data.

This figure depicts this feature:

```
              [socket]                   wire     ‘o’
[mod_perl] => [      ] => [mod_proxy] => ____ =>  /|\
              [buffer]                            / \
```

From looking at this figure it’s easy to see that the bottleneck is the socket buffer; it has to be able to absorb all the data that mod_perl has generated in order to free the mod_perl process immediately; mod_proxy will take the data as fast as mod_perl can deliver it, freeing the mod_perl server to service new requests as soon as possible while mod_proxy feeds the client at whatever rate the client requires.

`ProxyReceiveBufferSize` is the name of the parameter that specifies the size of the socket buffer. Configuring:

```
ProxyReceiveBufferSize 16384
```

will create a buffer of 16KB in size. If mod_perl generates output which is less than 16KB, the process will be immediately untied and allowed to serve new requests, if the output is bigger than 16KB, the following process will take place:

1. The first 16KB will enter the system buffer.

2. mod_proxy picks the first 8KB and sends it down the wire.

3. mod_perl writes the next 8KB into the place of the 8KB of data that was just sent off by mod_proxy.

Stages 2 and 3 are repeated until mod_perl has no more data to send. When this happens, mod_perl can serve a new request while stage 2 is repeated until all the data was picked from the system buffer and sent down the wire.

Of course you want to set the buffer size as large as possible, since you want the heavy mod_perl processes to be utilized in the most efficient way, so you don’t want them to waste their time waiting for a client to receive the data, especially if a client has a slow downstream connection.

As the `ProxyReceiveBufferSize` name states, its buffering feature applies only to *downstream data* (coming from the origin server to the proxy) and not upstream data. There is no buffering of data uploaded from the client browser to the proxy, thus you cannot use this technique to prevent the heavy mod_perl server from being tied up during a large POST such as a file upload. Falling back to mod_cgi seems to be the best solution for these specific scripts whose major function is receiving large amounts of upstream data.

[META: check this: --]

Of course just like mod_perl, mod_proxy writes the data it proxy-passes into its outgoing socket buffer, therefore the mod_proxy process gets released as soon as the last chunk of data is deposited into this buffer, even if the client didn't complete the download. Its the OS's problem to complete the transfer and release the TCP socket used for this transfer.

Therefore if you don't use mod_proxy and mod_perl sends its data directly to the client, and you have a big socket buffer, the mod_perl process will be released as soon as the last chunk of data enters the buffer. Just like with mod_proxy, the OS will deal with completing the data transfer.

[based on this comment] yes, too (but receive and transmit buffer may be of different size, depending on the OS)

The problem I don't know is, does the call to close the socket wait, until all data is actually send successfully or not. If it doesn't wait, you may not be noticed of any failure, but because the proxying Apache can write as fast to the socket transmission buffer as it can read, it should be possible that the proxying Apache copies all the data from the receive to the transmission buffer and after that releasing the receive buffer, so the mod_perl Apache is free to do other things, while the proxying Apache still wait until the client returns the success of data transmission. (The last, is the part I am not sure on)

[/META]

Unfortunately you cannot set the socket buffer size as large as you want because there is a limit of the available physical memory and OSs have their own upper limits on the possible buffer size.

This doesn't mean that you cannot change the OS imposed limits, but to do that you have to know the techniques for doing that. In the next section we will present a few OSs and the ways to increase their socket buffer sizes.

To increase the physical memory limits you just have to add more memory.

1.7.3 Setting the Buffering Limits on Various OSs

As we just saw there are a few kinds of parameters we might want to adjust for our needs.

1.7.3.1 IOBUFSIZE Source Code Definition

The first parameter is used by *proxy_util.c:ap_proxy_send_fb()* to loop over content being proxy passed in 8KB chunks (as of this writing), passing that on to the client. In other words it specifies the size of the data that is sent down the wire.

This parameter is defined by the `IOBUFSIZE`:

```
  #define IOBUFSIZE 8192
```

You have no control over this setting in the server configuration file, therefore you might want to change it in the source files, before you compile the server.

1.7.3.2 ProxyReceiveBufferSize Configuration Directive

You can control the socket buffer size with the `ProxyReceiveBufferSize` directive:

```
ProxyReceiveBufferSize 16384
```

The above setting will set a buffer size of 16KB. If it is not set explicitly, or if it is set to 0, then the default buffer size is used. The number should be an integral multiple of 512.

Note that if you set the value of `ProxyReceiveBufferSize` larger than the OS limit, the default value will be used.

Both the default and the maximum possible value of `ProxyReceiveBufferSize` depend on the Operating System.

- **Linux**

  For 2.2 kernels the maximum limit is in */proc/sys/net/core/rmem_max* and the default value is in */proc/sys/net/core/rmem_default*. If you want to increase `RCVBUF` size above 65535, the default maximum value, you have to raise first the absolute limit in */proc/sys/net/core/rmem_max*. To do that at the run time, execute this command to raise it to 128KB:

  ```
  % echo 131072 > /proc/sys/net/core/rmem_max
  ```

  You probably want to put this command into */etc/rc.d/rc.local* so the change will take effect at system reboot.

  On Linux OS with kernel 2.2.5 the maximum and default values are either 32KB or 64KB. You can also change the default and maximum values during kernel compilation; for that you should alter the `SK_RMEM_DEFAULT` and `SK_RMEM_MAX` definitions respectively. (Since kernel source files tend to change, use grep(1) utility to find the files.)

- **FreeBSD**

  Under FreeBSD it's possible to configure the kernel to have bigger socket buffers:

  ```
  % sysctl -w kern.ipc.maxsockbuf=2621440
  ```

- **Solaris**

  Under Solaris this upper limit is specified by *tcp_max_buf* parameter and is 256KB.

- **Other OSs**

  [ReaderMeta]: If you use an OS that is not listed here and know how to increase the socket buffer size please let me know.

When you tell the kernel to use bigger sockets you can set bigger values for *ProxyReceiveBufferSize*. e.g. 1048576 (1MB).

1.7.3.3 Hacking the Code

Some folks have patched the Apache's 1.3.x source code to make the application buffer configurable as well. After the patch there are two configuration directives available:

- ProxyReceiveBufferSize -- sets the socket buffer size
- ProxyInternalBufferSize -- sets the application buffer size

To patch the source, rename ap_breate() to ap_bcreate_size() and add a size parameter, which defaults to *IOBUFSIZE* if 0 is passed. Then add

```
  #define ap_bcreate(p,flags) ap_bcreate(p,flags,0)
```

and add a new ap_bcreate() which calls ap_bcreate_size() for binary compatibility.

Actually the `ProxyReceiveBufferSize` should be called `ProxySocketBufferSize`. This would also remove some of the confusion about what it actually does.

1.7.4 Caching Feature

META: complete the conf details

Apache does caching as well. It's relevant to mod_perl only if you produce proper headers, so your scripts' output can be cached. See the Apache documentation for more details on the configuration of this capability.

1.7.5 Build Process

To build mod_proxy into Apache just add --*enable-module=proxy* during the Apache **./configure** stage. Since you probably will need the mod_rewrite capability enable it as well with --*enable-module=rewrite*.

1.8 Front-end Back-end Proxying with Virtual Hosts

This section explains a configuration setup for proxying your back-end mod_perl servers when you need to use Virtual Hosts.

The term *Virtual Host* refers to the practice of maintaining more than one server on one machine, as differentiated by their apparent hostname. For example, it is often desirable for companies sharing a web server to have their own domains, with web servers accessible as *www.company1.com* and *www.company2.com*, without requiring the user to know any extra path information.

The approach is to use a unique port number for each virtual host at the back-end server, so you can redirect from the front-end server to localhost:1234, and name-based virtual servers on the front end, though any technique on the front-end will do.

If you run the front-end and the back-end servers on the same machine you can prevent any direct outside connections to the back-end server if you bind tightly to address `127.0.0.1` (*localhost*) as you will see in the following configuration example.

The front-end (light) server configuration:

```
<VirtualHost 10.10.10.10>
  ServerName www.example.com
  ServerAlias example.com
  RewriteEngine On
  RewriteOptions 'inherit'
  RewriteRule \.(gif|jpg|png|txt|html)$ - [last]
  RewriteRule ^/(.*)$ http://localhost:4077/$1 [proxy]
</VirtualHost>

<VirtualHost 10.10.10.10>
  ServerName foo.example.com
  RewriteEngine On
  RewriteOptions 'inherit'
  RewriteRule \.(gif|jpg|png|txt|html)$ - [last]
  RewriteRule ^/(.*)$ http://localhost:4078/$1 [proxy]
</VirtualHost>
```

The above front-end configuration handles two virtual hosts: *www.example.com* and *foo.example.com*. The two setups are almost identical.

The front-end server will handle files with the extensions *.gif*, *.jpg*, *.png*, *.txt* and *.html* internally, the rest will be proxied to be handled by the back-end server.

The only difference between the two virtual hosts settings is that the former rewrites requests to port `4077` at the back-end machine and the latter to port `4078`.

If your server is configured to run traditional CGI scripts (under mod_cgi) as well as mod_perl CGI programs, then it would be beneficial to configure the front-end server to run the traditional CGI scripts directly. This can be done by altering the `gif|jpg|png|txt` *Rewrite* rule to add `|cgi` at the end if all your mod_cgi scripts have the *.cgi* extension, or adding a new rule to handle all `/cgi-bin/*` locations locally.

The back-end (heavy) server configuration:

```
Port 80

PerlPostReadRequestHandler My::ProxyRemoteAddr

Listen 4077
<VirtualHost localhost:4077>
  ServerName www.example.com
  DocumentRoot /home/httpd/docs/www.example.com
  DirectoryIndex index.shtml index.html
```

```
</VirtualHost>

Listen 4078
<VirtualHost localhost:4078>
  ServerName foo.example.com
  DocumentRoot /home/httpd/docs/foo.example.com
  DirectoryIndex index.shtml index.html
</VirtualHost>
```

The back-end server knows to tell which virtual host the request is made to, by checking the port number the request was proxied to and using the appropriate virtual host section to handle it.

We set "Port 80" so that any redirects don't get sent directly to the back-end port.

To get the *real* remote IP addresses from proxy, the My::ProxyRemoteAddr handler is used based on the `mod_proxy_add_forward` Apache module. Prior to mod_perl 1.22 this setting must have been set per-virtual host, since it wasn't inherited by the virtual hosts.

The following configuration is yet another useful example showing the other way around. It specifies what to be proxied and then the rest is served by the front end:

```
RewriteEngine     on
RewriteLogLevel   0
RewriteRule       ^/(perl.*)$  http://127.0.0.1:8052/$1   [P,L]
NoCache           *
ProxyPassReverse  /  http://www.example.com/
```

So we don't have to specify the rule for static objects to be served by the front-end as we did in the previous example to handle files with the extensions *.gif*, *.jpg*, *.png* and *.txt* internally.

1.9 Getting the Remote Server IP in the Back-end server in the Proxy Setup

Ask Bjoern Hansen has written the `mod_proxy_add_forward` module for Apache. It sets the `X-Forwarded-For` field when doing a `ProxyPass`, similar to what Squid can do. Its location is specified in the download section.

Basically, this module adds an extra HTTP header to proxying requests. You can access that header in the mod_perl-enabled server, and set the IP address of the remote server. You won't need to compile anything into the back-end server.

1.9.1 Build

Download the module and use its location as a value of the *--activate-module* argument for the *./configure* utility within the Apache source code, so the module can be found.

```
./configure \
"--with-layout=Apache" \
"--activate-module=src/modules/extra/mod_proxy_add_forward.c" \
"--enable-module=proxy_add_forward" \
... other options ...
```

--enable-module=proxy_add_forward enables this module as you have guessed already.

1.9.2 Usage

If you are using `Apache::Registry` or `Apache::PerlRun` modules just put the following code into *startup.pl*:

```
use Apache::Constants ();
sub My::ProxyRemoteAddr ($) {
  my $r = shift;

  # we'll only look at the X-Forwarded-For header if the requests
  # comes from our proxy at localhost
  return Apache::Constants::OK
      unless ($r->connection->remote_ip eq "127.0.0.1")
          and $r->header_in('X-Forwarded-For');

  # Select last value in the chain -- original client's ip
  if (my ($ip) = $r->headers_in->{'X-Forwarded-For'} =~ /([^,\s]+)$/) {
    $r->connection->remote_ip($ip);
  }

  return Apache::Constants::OK;
}
```

And in the mod_perl's *httpd.conf*:

```
PerlPostReadRequestHandler My::ProxyRemoteAddr
```

and the right thing will happen transparently for your scripts. Otherwise if you write your own mod_perl content handler, you can retrieve it directly in your code using a similar code.

1.9.3 Security

Different sites have different needs. If you use the header to set the IP address, Apache believes it. This is reflected in the logging for example. You really don't want anyone but your own system to set the header, which is why the *recommended code* above checks where the request really came from before changing `remote_ip`.

Generally you shouldn't trust the `X-Forwarded-For` header. You only want to rely on `X-Forwarded-For` headers from proxies you control yourself. If you know how to spoof a cookie you've probably got the general idea on making HTTP headers and can spoof the `X-Forwarded-For` header as well. The only address you can count on as being a reliable value is the one from `r->connection->remote_ip`.

From that point on, the remote IP address is correct. You should be able to access $ENV{REMOTE_ADDR} environment variable as usual.

1.9.4 Caveats

It was reported that Ben Laurie's Apache-SSL does not seem to put the IP addresses in the X-Forwarded-For header--it does not set up such a header at all. However, the $ENV{REMOTE_ADDR} environment variable it sets up contains the IP address of the original client machine.

Prior to mod_perl 1.22 there was a need to repeat the PerlPostReadRequestHandler My::ProxyRemoteAddr directive for each virtual host, since it wasn't inherited by the virtual hosts.

1.9.5 mod_proxy_add_forward Module's Order Precedence

Some users report that they cannot get this module to work as advertised. They verify that the module is built in, but the front-end server is not generating the X-Forwarded-For header when requests are being proxied to the back-end server. As a result, the back-end server has no idea what the remote IP is.

As it turns out, *mod_proxy_add_forward* needs to be configured in Apache before *mod_proxy* in order to operate properly, since Apache gives highest precedence to the last defined module.

Moving the two build options required to enable *mod_proxy_add_forward* while configuring Apache appears to have no effect on the default configuration order of modules, since in each case, the resulting builds show *mod_proxy_add_forward* last in the list (or first via */server-info*).

One solution is to explicitly define the configuration order in the *http.conf* file, so that *mod_proxy_add_forward* appears before *mod_proxy*, and therefore gets executed after *mod_proxy*. (Modules are being executed in *reverse* order, i.e. module that was *Added* first will be executed last.)

Obviously, this list would need to be tailored to match the build environment, but to ease this task just insert an AddModule directive before each entry reported by httpd -l (and removing *httpd_core.c*, of course):

```
ClearModuleList
AddModule mod_env.c
[more modules snipped]
AddModule mod_proxy_add_forward.c
AddModule mod_proxy.c
AddModule mod_rewrite.c
AddModule mod_setenvif.c
```

Note that the above snippet is added to *httpd.conf* of the front-end server.

Another solution is to reorder the module list during configuration by using one or more --permute-module arguments to the *./configure* utility. (Try ./configure --help to see if your version of Apache supports this option.) --permute-module=foo:bar will swap the position of *mod_foo* and *mod_bar* in the list, --permute-module=BEGIN:foo will move *mod_foo* to the beginning of the list, and --permute-module=foo:END will move *mod_foo* to the end. For example suppose your module list from `httpd -l` looks like:

```
http_core.c
[more modules snipped]
mod_proxy.c
mod_setenvif.c
mod_proxy_add_forward.c
```

You might add the following arguments to *./configure* to move *mod_proxy_add_forward* to the position in the list just before *mod_proxy*:

```
./configure \
"--with-layout=Apache" \
"--activate-module=src/modules/extra/mod_proxy_add_forward.c" \
"--enable-module=proxy_add_forward" \
... other options ...
"--permute-module=proxy:proxy_add_forward" \
"--permute-module=setenvif:END"
```

With this change, the `X-Forwarded-For` header is now being sent to the back-end server, and the remote IP appears in the back-end server's *access_log* file.

1.10 HTTP Authentication With Two Servers Plus a Proxy

Assuming that you have a setup of one "front-end" server, which proxies the "back-end" (mod_perl) server, if you need to perform authentication in the "back-end" server it should handle all authentication itself. If Apache proxies correctly, it will pass through all authentication information, making the "front-end" Apache somewhat "dumb", as it does nothing but pass through the information.

In the configuration file your `Auth` configuration directives need to be inside the `<Directory ...>` ... `</Directory>` sections because if you use the section `<Location ...>` ... `</Location>` the proxy server will take the authentication information for itself and not pass it on.

The same applies to mod_ssl and similar Apache SSL modules. If it gets plugged into a front-end server, it will properly encode/decode all the SSL requests. So if your machine is secured from inside, your back-end server can do secure transactions.

1.11 mod_rewrite Examples

Example code for using mod_rewrite with mod_perl application servers. Several examples were taken from the mailing list.

1.11.1 Rewriting Requests Based on File Extension

In the mod_proxy + mod_perl servers scenario, `ProxyPass` was used to redirect all requests to the mod_perl server, by matching the beginning of the relative URI (e.g. */perl*). What should you do if you want everything, but files with extensions like *.gif*, *.cgi* and similar, to be proxypassed to the mod_perl server. These files are to be served by the light Apache server which carries the mod_proxy module.

The following example rewrites everything to the mod_perl server. It locally handles all requests for files with extensions *gif*, *jpg*, *png*, *css*, *txt*, *cgi* and relative URIs starting with */cgi-bin* (e.g. if you want some scripts to be executed under mod_cgi).

```
RewriteEngine On
# handle GIF and JPG images and traditional CGI's directly
RewriteRule \.(gif|jpg|png|css|txt|cgi)$ - [last]
RewriteRule ^/cgi-bin - [last]
# pass off everything but images to the heavy-weight server via proxy
RewriteRule ^/(.*)$ http://localhost:4077/$1 [proxy]
```

That is, first, handle locally what you want to handle locally, then hand off everything else to the back-end guy.

This is the configuration of the logging facilities.

```
RewriteLogLevel 1
RewriteLog "| /usr/local/apache_proxy/bin/rotatelogs \
/usr/local/apache-common/logs/r_log 86400"
```

It says: log all the rewrites thru the pipe to the `rotatelogs` utility which will rotate the logs every 2 hours (86400 secs).

1.11.2 Internet Exporer 5 favicon.ico 404

Redirect all those IE5 requests for *favicon.ico* to a central image:

```
RewriteRule .*favicon.ico /wherever/favicon.ico [PT,NS]
```

1.11.3 Hiding Extensions for Dynamic Pages

A quick way to make dynamic pages look static:

```
RewriteRule ^/wherever/([a-zA-Z]+).html /perl-bin/$1.cgi [PT]
```

1.11.4 Serving Static Content Locally and Rewriting Everything Else

Instead of keeping all your Perl scripts in */perl* and your static content everywhere else, you could keep your static content in special directories and keep your Perl scripts everywhere else. You can still use the light/heavy apache separation approach described before, with a few minor modifications.

In the *light* Apache's *httpd.conf* file, turn rewriting on:

```
RewriteEngine On
```

Now list all directories that contain only static objects. For example if the only relative to `DocumentRoot` directories are */images* and *style* you can set the following rule:

```
  RewriteRule ^/(images|style) - [L]
```

The `[L]` (*Last*) means that the rewrite engine should stop if it has a match. This is necessary because the very last rewrite rule proxies everything to the *heavy* server:

```
  RewriteRule ^/(.*) http://www.example.com:8080/$1 [P]
```

This line is the difference between a server for which static content is the default and one for which dynamic (perlish) content is the default.

You should also add the *reverse rewrite rule* as before:

```
  ProxyPassReverse / http://www.example.com/
```

so that the user doesn’t see the port number `:8080` in the browser’s location window.

It is possible to use `localhost` in the `RewriteRule` above if the heavy and light servers are on the same machine. So if we sum up the above setup we get:

```
  RewriteEngine On
  RewriteRule ^/(images|style) - [L]
  RewriteRule ^/(.*) http://www.example.com:8080/$1 [P]
  ProxyPassReverse / http://www.example.com/
```

1.11.5 Upgrading mod_perl Heavy Application Instances

When using a light/heavy separation method one of the challenges of running a production environment is being able to upgrade to newer versions of mod_perl or your own application. The following method can be used without having to do a server restart.

Add the following rewrite rule to your httpd.conf file:

```
  RewriteEngine On
  RewriteMap maps txt:/etc/httpd.maps
  RewriteRule ^(.*) http://${maps:appserver}$1 [proxy]
```

Create the file /etc/httpd.maps and add the following entry:

```
  appserver foo.com:9999
```

Mod_rewrite rereads (or checks the mtime of) the file on every request so the change takes effect immediately. To seamlessly upgrade your application server to a new version, install a new version on a different port. After checking for a quality installation, edit /etc/httpd.maps to point to the new server. After the file is written the next request the server processes will be redirected to the new installation.

1.11.6 Blocking IP Addresses

The following rewrite code blocks IP addresses:

```
  RewriteCond /web/site/var/blocked/REMOTE_ADDR-%{REMOTE_ADDR} -f
  RewriteRule .* http://YOUR-HOST-BLOCKED-FOR-EXCESSIVE-CONSUMPTION [redirect,last]
```

To block IP address 10.1.2.3, simply touch

```
  /web/site/var/blocked/REMOTE_ADDR-10.1.2.3
```

This has an advantage over Apache parsing a long file of addresses in that the OS is better at a file lookup.

1.12 Caching in mod_proxy

This is not really mod_perl related, so I'll just stress one point. If you want the caching to work the following HTTP headers should be supplied: `Last-Modified`, `Content-Length` and `Expires`.

1.13 Maintainers

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

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

1.14 Authors

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

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

Table of Contents: