ApacheCon Europe
October 25, 2000
London, UK

Overview of mod_perl version 2.0

by **Doug MacEachern**
<dougm@covalent.net>
Covalent Technologies

This document is originally written in **POD**, converted to **HTML**, **PostScript** and **PDF** by `Pod::HtmlPsPdf` Perl module.

(you will find a Table of Contents at the end of the Tutorial)

1 Introduction

1.1 Apache 2.0 Summary

Note: This section will give you a brief overview of the changes in Apache 2.0, just enough to understand where mod_perl will fit in. For more details on Apache 2.0 consult the papers by Ryan Bloom.

1.1.1 MPMs - Multi-Processing Model Modules

In Apache 1.3.x concurrent requests were handled by multiple processes, and the logic to manage these processes lived in one place, *http_main.c*, 7200 some odd lines of code. If Apache 1.3.x is compiled on a Win32 system large parts of this source file are redefined to handle requests using threads. Now suppose you want to change the way Apache 1.3.x processes requests, say, into a DCE RPC listener. This is possible only by slicing and dicing *http_main.c* into more pieces or by redefining the *standalone_main* function, with a `-DSTANDALONE_MAIN=your_function` compile time flag. Neither of which is a clean, modular mechanism.

Apache-2.0 solves this problem by intoducing *Multi Processing Model modules*, better known as *MPMs*. The task of managing incoming requests is left to the MPMs, shrinking *http_main.c* to less than 500 lines of code. Several MPMs are included with Apache 2.0 in the *src/modules/mpm* directory:

prefork

The *prefork* module emulates 1.3.x's preforking model, where each request is handled by a different process.

pthread/dexter

These two MPMs implement a hybrid multi-process multi-threaded approach based on the *pthreads* standard, but each offers different fine-tuning configuration.

os2/winnt/beos

These MPMs also implement the hybrid multi-process/multi-threaded model, with each based on native OS thread implementations.

perchild

The *perchild* MPM is based on the *dexter* MPM, but is extended with a mechanism which allows mapping of requests to virtual hosts to a process running under the user id and group configured for that host. This provides a robust replacement for the *suexec* mechanism.

1.1.2 APR - Apache Portable Runtime

Apache 1.3.x has been ported to a very large number of platforms including various flavors of unix, win32, os/2, the list goes on. However, in 1.3.x there was no clear-cut, pre-designed portability layer for third-party modules to take advantage of. APR provides this API layer in a very clean way. For mod_perl, APR will assist a great deal with portability. Combined with the portablity of Perl, mod_perl-2.0 needs only to implement a portable build system, the rest comes ‘‘for free’’. A Perl interface will be provided for certain areas of APR, such as the shared memory abstraction, but the majority of APR will be used by mod_perl ‘‘under the covers’’.

1.1.3 New Hook Scheme

In Apache 1.3, modules were registered using the *module* structure, normally static to *mod_foo.c*. This structure contains pointers to the command table, config create/merge functions, response handler table and function pointers for all of the other hooks, such as *child_init* and *check_user_id*. In 2.0, this structure has been pruned down to the first three items mention and a new function pointer added called *register_hooks*. It is the job of *register_hooks* to register functions for all other hooks (such as *child_init* and *check_user_id*). Not only is hook registration now dynamic, it is also possible for modules to register more than one function per hook, unlike 1.3. The new hook mechanism also makes it possible to sort registered functions, unlike 1.3 with function pointers hardwired into the module structure, and each module structure into a linked list. Order in 1.3 depended on this list, which was possible to order using compile-time and configuration-time configuration, but that was left to the user. Whereas in 2.0, the add_hook functions accept an order preference parameter, those commonly used are:

FIRST

MIDDLE

LAST

For mod_perl, dynamic registration provides a cleaner way to bypass the *Perl*Handler* configuration. By simply adding this configuration:

```
PerlModule Apache::Foo
```

Apache/Foo.pm can register hooks itself at server startup:

```
Apache::Hook->add(PerlAuthenHandler => \&authenticate, Apache::Hook::MIDDLE);
Apache::Hook->add(PerlLogHandler => \&logger, Apache::Hook::LAST);
```

However, this means that Perl subroutines registered via this mechanism will be called for *every* request. It will be left to that subroutine to decide if it was to handle or decline the given phase. As there is overhead in entering the Perl runtime, it will most likely be to your advantage to continue using *Perl*Handler* configuration to reduce this overhead. If it is the case that your *Perl*Handler* should be invoked for every request, the hook registration mechanism will save some configuration keystrokes.

1.1.4 Configuration Tree

When configuration files are read by Apache 1.3, it hands off the parsed text to module configuration directive handlers and discards that text afterwards. With Apache 2.0, the configuration files are first parsed into a tree structure, which is then walked to pass data down to the modules. This tree is then left in memory with an API for accessing it at request time. The tree can be quite useful for other modules. For example, in 1.3, mod_info has it’s own configuration parser and parses the configuration files each time you access it. With 2.0 there is already a parse tree in memory, which mod_info can then walk to output it's information.

If a mod_perl 1.xx module wants access to configuration information, there are two approaches. A module can ‘‘subclass’’ directive handlers, saving a copy of the data for itself, then returning **DECLINE_CMD** so the other modules are also handed the info. Or, the `$Apache::Server::SaveConfig` variable can be set to save <Perl> configuration in the `%Apache::ReadConfig::` namespace. Both methods are rather kludgy, version 2.0 will provide a Perl interface to the Apache configuration tree.

1.1.5 I/O Filtering

Filtering of Perl modules output has been possible for years since tied filehandle support was added to Perl. There are several modules, such as *Apache::Filter* and *Apache::OutputChain* which have been written to provide mechanisms for filtering the `STDOUT` ‘‘stream’’. There are several of these modules because no one approach has quite been able to offer the ease of use one would expect, which is due simply to limitations of the Perl tied filehandle design. Another problem is that these filters can only filter the output of other Perl modules. C modules in Apache 1.3 send data directly to the client and there is no clean way to capture this stream. Apache 2.0 has solved this problem by introducing a filtering API. With the baseline i/o stream tied to this filter mechansim, any module can filter the output of any other module, with any number of filters in between.

1.1.6 Protocol Modules

Apache 1.3 is hardwired to speak only one protocol, HTTP. Apache 2.0 has moved to more of a ‘‘server framework’’ architecture making it possible to plugin handlers for protocols other than HTTP. The protocol module design also abstracts the transport layer so protocols such as SSL can be hooked into the server without requiring modifications to the Apache source code. This allows Apache to be extended much further than in the past, making it possible to add support for protocols such as FTP, SMTP, RPC flavors and the like. The main advantage being that protocol plugins can take advantage of Apache's portability, process/thread management, configuration mechanism and plugin API.

1.2 mod_perl and Threaded MPMs

1.2.1 Perl 5.6

Thread safe Perl interpreters, also known as ‘‘ithreads’’ (Intepreter Threads) provide the mechanism need for mod_perl to adapt to the Apache 2.0 thread architecture. This mechanism is a compile time option which encapsulates the Perl runtime inside of a single *PerlInterpreter* structure. With each interpreter instance containing its own symbol tables, stacks and other Perl runtime mechanisms, it is possible for any number of threads in the same process to concurrently callback into Perl. This of course requires each thread to have it's own *PerlInterpreter* object, or at least that each instance is only access by one thread at any given time.

mod_perl-1.xx has only a single *PerlInterpreter*, which is contructed by the parent process, then inherited across the forks to child processes. mod_perl-2.0 has a configurable number of *PerlInterpreters* and two classes of interpreters, *parent* and *clone*. A *parent* is like that in 1.xx, the main interpreter created at startup time which compiles any pre-loaded Perl code. A *clone* is created from the parent using the Perl API *perl_clone()* function. At request time, *parent* interpreters are only used for making more *clones*, as they are the interpreters which actually handle requests. Care is taken by Perl to copy only mutable data, which means that no runtime locking is required and read-only data such as the syntax tree is shared from the *parent*.

1.2.2 New mod_perl Directives for Threaded MPMs

Rather than create a *PerlInterperter* per-thread by default, mod_perl creates a pool of interpreters. The pool mechanism helps cut down memory usage a great deal. As already mentioned, the syntax tree is shared between all cloned interpreters. If your server is serving more than mod_perl requests, having a smaller number of PerlInterpreters than the number of threads will clearly cut down on memory usage. Finally and perhaps the biggest win is memory reuse. That is, as calls are made into Perl subroutines, memory allocations are made for variables when they are used for the first time. Subsequent use of variables may allocate more memory, e.g. if the string needs to hold a larger than it did before, or an array more elements than in the past. As an optimization, Perl hangs onto these allocations, even though their values ‘‘go out of scope’’. With the 1.xx model, random children would be hit with these allocations. With 2.0, mod_perl has much better control over which PerlInterpreters are used for incoming requests. The intepreters are stored in two linked lists, one for available interpreters one for busy. When needed to handle a request, one is taken from the head of the available list and put back into the head of the list when done. This means if you have, say, 10 interpreters configured to be cloned at startup time, but no more than 5 are ever used concurrently, those 5 continue to reuse Perls allocations, while the other 5 remain much smaller, but ready to go if the need arises.

Various attributes of the pools are configurable with the following configuration directives:

PerlInterpStart

The number of intepreters to clone at startup time.

PerlInterpMax

If all running interpreters are in use, mod_perl will clone new interpreters to handle the request, up until this number of interpreters is reached. When Max is reached, mod_perl will block until one becomes available.

PerlInterpMinSpare

The minimum number of available interpreters this parameter will clone interpreters up to Max, before a request comes in.

PerlInterpMaxSpare

mod_perl will throttle down the number of interpreters to this number as those in use become available.

PerlInterpMaxRequests

The maximum number of requests an interpreter should serve, the interpreter is destroyed when the number is reached and replaced with a fresh clone.

1.2.3 Issues with Threading

The Perl ‘‘ithreads’’ implementation ensures that Perl code is thread safe, at least with respect to the Apache threads in which it is running. However, it does not ensure that extensions which call into third-party C/C++ libraries are thread safe. In the case of non-threadsafe extensions, if it is not possible to fix those routines, care will need to be taken to serialize calls into such functions (either at the xs or Perl level).

1.3 Thread Item Pool API

As we discussed, mod_perl implements a pool mechanism to manage *PerlInterpreters* between threads. This mechanism has been abstracted into an API known as ‘‘tipool’’, *Thread Item Pool*. This pool can be used to manage any data structure, in which you wish to have a smaller number than the number of configured threads. A good example of such a data structure is a database connection handle. The *Apache::DBI* module implements persisent connections for 1.xx, but may result in each child maintaining its own connection, when it is most often the case that number of connections is never needed concurrently. The TIPool API provides a mechanism to solve this problem, consisting of the following methods:

new

Create a new thread item pool. This constructor is passed an *Apache::Pool* object, a hash reference to pool configuration parameters, a hash reference to pool callbacks and an optional userdata variable which is passed to callbacks:

```
my $tip = Apache::TIPool->new($p,
                              {Start => 3, Max => 6},
                              {grow => \&new_connection,
                               shrink => \&close_connection},
                              \%my_config);
```

The configuration parameters, *Start*, *Max*, *MinSpare*, *MaxSpare* and *MaxRequests* configure the pool for your items, just as the *PerlInterp** directives do for *PerlInterpreters*.

The *grow* callback is called to create new items to be added to the pool, *shrink* is called when an item is removed from the pool.

pop

This method will return an item from the pool, from the head of the available list. If the current number of items are all busy, and that number is less than the configured maximum, a new item will be created by calling the configured *grow* callback. Otherwise, the *pop* method will block until an item is available.

```
my $item = $tip->pop;
```

putback

This method gives an item (returned from *pop*) back to the pool, which is pushed into the head of the available list:

```
$tip->putback($item);
```

Future improvements will be made to the TIPool API, such as the ability to sort the *available* and *busy* lists and specify if items should be popped and putback to/from the head or tail of the list.

1.3.1 Apache::DBIPool

Now we will take a look at how to make *DBI* take advantage of *TIPool* API with the *Apache::DBIPool* module. The module configuration in httpd.conf will look something like so:

```
PerlModule Apache::DBIPool
```

```
<DBIPool dbi:mysql:db_name>
  DBIPoolStart 10
  DBIPoolMax   20
  DBIPoolMaxSpare 10
  DBIPoolMinSpare 5
  DBIUserName dougm
  DBIPassWord XxXx
</DBIPool>
```

The module is loaded using the *PerlModule* directive just as with other modules. TIPools are then configured using *DBIPool* configuration sections. The argument given to the container is the *dsn* and within are the pool directives *Start*, *Max*, *MaxSpare* and *MinSpare*. The *UserName* and *PassWord* directives will be passed to the *DBI connect* method. There can be any number of *DBIPool* containers, provided each *dsn* is different, and/or each container is inside a different *VirtualHost* container.

Now let’s examine the source code, keeping in mind this module contains the basics and the official release (tbd) will likely contain more details, such as how it hooks into *DBI.pm* to provide transparency the way *Apache::DBI* currently does.

After pulling in the modules needed *Apache::TIPool*, *Apache::ModuleConfig* and *DBI*, we setup a callback table. The *new_connection* function will be called with the TIP needs to add a new item and *close_connection* when an item is being removed from the pool. The *Apache::Hook add* method registers a *PerlPostConfigHandler* which will be called after Apache has read the configuration files.

This handler (our *init* function) is passed 3 *Apache::Pool* objects and one *Apache::Server* object. Each *Apache::Pool* has a different lifetime, the first will be alive until configuration is read again, such as during restarts. The second will be alive until logs are re-opened and the third is a temporary pool which is cleared before Apache starts serving requests. Since the DBI connection pool is associated with configuration in httpd.conf, we will use that pool.

The *Apache::ModuleConfig get* method is called with the *Apache::Server* object to give us the configuration associated with the given server. Next is a while loop which iterates over the configuration parsed by the *DBIPool* directive handler. The keys of this hash are the configured *dsn*, of which there is one per *DBIPool* configuration section. The values will be a hash reference to the pool configuration, *Start*, *Max*, *MinSpare*, *MaxSpare* and *MaxRequests*.

A *new Apache::TIPool* is then contructed, passing it the `$pconf` *Apache::Pool*, configuration `$params`, the *$callbacks* table and `$conn` hash ref. The *TIPool* is then saved into the `$cfg` object, indexed by the *dsn*.

At the time *Apache::TIPool::new* is called, the *new_connection* callback will be called the number of time to which *Start* is configured. This callback localizes *Apache::DBIPool::connect* to a code reference which makes the real database connection.

At request time *Apache::DBIPool::connect* will fetch a database handle from the *TIPool*. It does so by digging into the configuration object associated with the current virtual host to obtain a reference to the *TIPool* object. It then calls the *pop* method, which will immediatly return a database handle if one is available. If all opened connection are in used and the current number of connections is less than the configured *Max*, the call to *pop* will result in a call to *new_connection*. If *Max* has already been reached, then *pop* will block until a handle is *putback* into the pool.

Finally, the handle is blessed into the *Apache::DBIPool::db* class which will override the dbd class *disconnect* method. The overridden *disconnect* method obtains a reference to the *TIPool* object and passes it to the *putback* method, making it available for use by other threads. Should the Perl code using this handle neglect to call the *disconnect* method, the overridden *connect* method has already registered a cleanup function to make sure it is *putback*.

1.3.2 Apache::DBIPool Source

```
package Apache::DBIPool;
```

```
use strict;
use Apache::TIPool ();
use Apache::ModuleConfig ();
use DBI ();
```

```
my $callbacks = {
   grow => \&new_connection,     #add new connection to the pool
   shrink => \&close_connection, #handle removed connection from pool
};

Apache::Hook->add(PerlPostConfigHandler => \&init); #called at startup

sub init {
    my($pconf, $plog, $ptemp, $s) = @_;

    my $cfg = Apache::ModuleConfig->get($s, __PACKAGE__);

    #create a TIPool for each dsn
    while (my($conn, $params) = each %{ $cfg->{DBIPool} }) {
        my $tip = Apache::TIPool->new($pconf, $params, $callbacks, $conn);
        $cfg->{TIPool}->{ $conn->{dsn} } = $tip;
    }
}

sub new_connection {
    my($tip, $conn) = @_;

    #make actual connection to the database
    local *Apache::DBIPool::connect = sub {
        my($class, $drh) = (shift, shift);
        $drh->connect($dbname, @_);
    };

    return DBI->connect(@{$conn}{qw(dsn username password attr)});
}

sub close_connection {
    my($tip, $conn, $dbh) = @_;
    my $driver = (split $conn->{dsn}, ':')[1];
    my $method = join '::', 'DBD', $driver, 'db', 'disconnect';
    $dbh->$method(); #call the real disconnect method
}

my $EndToken = '</DBIPool>';

#parse <DBIPool dbi:mysql:...>...
```

```
sub DBIPool ($$$;*) {
    my($cfg, $parms, $dsn, $cfg_fh) = @_;
    $dsn =~ s/>$//;

    $cfg->{DBIPool}->{$dsn}->{dsn} = $dsn;

    while((my $line = <$cfg_fh>) !~ m:^$EndToken:o) {
        my($name, $value) = split $line, /\s+/, 2;
        $name =~ s/^DBIPool(\w+)/lc $1/ei;
        $cfg->{DBIPool}->{$dsn}->{$name} = $value;
    }
}

sub config {
    my $r = Apache->request;
    return Apache::ModuleConfig->get($r, __PACKAGE__);
}

#called from DBI::connect
sub connect {
    my($class, $drh) = (shift, shift);

    $drh->{DSN} = join ':', 'dbi', $drh->{Name}, $_[0];
    my $cfg = config();

    my $tip = $cfg->{TIPool}->{ $drh->{DSN} };

    unless ($tip) {
        #XXX: do a real connect or fallback to Apache::DBI
    }

    my $item = $tip->pop; #select a connection from the pool

    $r->register_cleanup(sub { #incase disconnect() is not called
        $tip->putback($item);
    });

    return bless 'Apache::DBIPool::db', $item->data; #the dbh
}

package Apache::DBIPool::db;
```

```
our @ISA = qw(DBI::db);
```

```
#override disconnect, puts database handle back in the pool
sub disconnect {
    my $dbh = shift;
    my $tip = config()->{TIPool}->{ $dbh->{DSN} };
    $tip->putback($dbh);
    1;
}
```

```
1;
__END__
```

1.4 PerlOptions Directive

A new configuration directive to mod_perl-2.0, *PerlOptions*, provides fine-grained configuration for what were compile-time only options in mod_perl-1.xx. In addition, this directive provides control over what class of *PerlInterpreter* is used for a *VirtualHost* or location configured with *Location*, *Directory*, etc.

These are all best explained with examples, first here's how to disable mod_perl for a certain host:

```
<VirtualHost ...>
   PerlOptions -Enable
</VirtualHost>
```

Suppose a one of the hosts does not want to allow users to configure *PerlAuthenHandler*, *PerlAuthzHandler* or *PerlAccessHandler* or <Perl> sections:

```
<VirtualHost ...>
   PerlOptions -Authen -Authz -Access -Sections
</VirtualHost>
```

Or maybe everything but the response handler:

```
<VirtualHost ...>
   PerlOptions None +Response
</VirtualHost>
```

A common problem with mod_perl-1.xx was the shared namespace between all code within the process. Consider two developers using the same server and each which to run a different version of a module with the same name. This example will create two *parent* Perls, one for each *VirtualHost*, each with its own namespace and pointing to a different paths in `@INC`:

```
<VirtualHost ...>
   ServerName dev1
   PerlOptions +Parent
   PerlSwitches -Mblib=/home/dev1/lib/perl
</VirtualHost>
```

```
<VirtualHost ...>
   ServerName dev2
   PerlOptions +Parent
   PerlSwitches -Mblib=/home/dev2/lib/perl
</VirtualHost>
```

Or even for a given location, for something like ‘‘dirty’’ cgi scripts:

```
<Location /cgi-bin>
   PerlOptions +Parent
   PerlInterpMaxRequests 1
   PerlInterpStart 1
   PerlInterpMax 1
   PerlHandler Apache::Registry
</Location>
```

Will use a fresh interpreter with its own namespace to handle each request.

Should you wish to fine tune Interpreter pools for a given host:

```
<VirtualHost ...>
   PerlOptions +Clone
   PerlInterpStart 2
   PerlInterpMax 2
</VirtualHost>
```

This might be worthwhile in the case where certain hosts have their own sets of large-ish modules, used only in each host. By tuning each host to have it’s own pool, that host will continue to reuse the Perl allocations in their specific modules.

1.5 Integration with 2.0 Filtering

The mod_perl-2.0 interface to the Apache filter API is much simpler than the C API, hiding most of the details underneath. Perl filters are configured using the *PerlFilterHandler* directive, for example:

```
PerlFilterHandler Apache::ReverseFilter
```

This simply registers the filter, which can then be turned on using the core *AddFilter* directive:

```
<Location /foo>
   AddFilter Apache::ReverseFilter
</Location>
```

The *Apache::ReverseFilter* handler will now be called for anything accessed in the */foo* url space. The *AddFilter* directive takes any number of filters, for example, this configuration will first send the output to *mod_include*, which will in turn pass its output down to *Apache::ReverseFilter*:

```
AddFilter INCLUDE Apache::ReverseFilter
```

For our example, *Apache::ReverseFilter* simply reverses all of the output characters and then sends them downstream. The first argument to a filter handler is an *Apache::Filter* object, which at the moment provides two methods *read* and *write*. The *read* method pulls down a chunk of the output stream into the given buffer, returning the length read into the buffer. An optional size argument may be given to specify the maximum size to read into the buffer. If omitted, an arbitrary size will fill the buffer, depending on the upstream filter. The *write* method passes data down to the next filter. In our case `scalar reverse` takes advantage of Perl's builtins to reverse the upstream buffer:

```
package Apache::ReverseFilter;

use strict;

sub handler {
    my $filter = shift;

    while ($filter->read(my $buffer, 1024)) {
        $filter->write(scalar reverse $buffer);
    }

    return Apache::OK;
}

1;
```

1.6 Protocol Modules with mod_perl-2.0

1.6.1 *Apache::Echo*

Apache 2.0 ships with an example protocol module, *mod_echo*, which simply reads data from the client and echos it right back. Here we'll take a look at a Perl version of that module, called *Apache::Echo*. A protocol handler is configured using the *PerlProcessConnectionHandler* directive and we'll use an *IfDefine* section so it's only enabled via the command line and binds to a different Port **8084**:

```
<IfDefine Apache::Echo>
    Port 8084
    PerlProcessConnectionHandler Apache::Echo
</IfDefine>
```

Apache::Echo is then enabled by starting Apache like so:

```
% httpd -DApache::Echo
```

And we give it a whirl:

```
% telnet localhost 8084
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
hello apachecon
hello apachecon
^]
```

The code is just a few lines of code, with the standard *package* declaration and of course, `use strict;`. As with all *Perl*Handler*s, the subroutine name defaults to *handler*. However, in the case of a protocol handler, the first argument is not a *request_rec*, but a *conn_rec* blessed into the *Apache::Connection* class. Right away we enter the echo loop, stopping if the *eof* method returns true, indicating that the client has disconnected. Next the *read* method is called with a maximum of 1024 bytes placed in `$buff` and returns the actual length read into `$rlen`. If no bytes were read we break out of the while loop. Otherwise, attempt to echo the data back using the *write* method. The *flush* method is called so the buffer is flushed to the client right away, otherwise the client would not see any data until the buffer was full (with around 8k or so worth). Once the client has disconnected, the module returns **OK**, telling Apache we have handled the connection:

```
package Apache::Echo;
```

```
use strict;
```

```
sub handler {
    my Apache::Connection $c = shift;
```

```
    while (!$c->eof) {
        my $rlen = $c->read(my $buff, 1024);
```

```
        last unless $rlen > 0 and $c->write($buff);
```

```
            $c->flush;
        }

        return Apache::OK;
    }

    1;
    __END__
```

1.6.2 *Apache::CommandServer*

Our first protocol handler example took advange of Apache's server framework, but did not tap into any other modules. The next example is based on the example in the ‘‘TCP Servers with IO::Socket’’ section of *perlipc*. Of course, we don't need *IO::Socket* since Apache takes care of those details for us. The rest of that example can still be used to illustrate implementing a simple text protocol. In this case, one where a command is sent by the client to be executed on the server side, with results sent back to the client.

The *Apache::CommandServer* handler will support four commands: *motd*, *date*, *who* and *quit*. These are probably not commands which can be exploited, but should we add such commands, we'll want to limit access based on ip address/hostname, authentication and authorization. Protocol handlers need to take care of these tasks themselves, since we bypass the HTTP protocol handler.

As with all *PerlProcessConnectionHandlers*, we are passed an *Apache::Connection* object as the first argument. After every call to the *write* method we want the client to see the data right away, so first *autoflush* is turned on to take care of that for us. Next, the *login* subroutine is called to check if access by this client should be allowed. This routine makes up for what we lost with the core HTTP protocol handler bypassed. First we call the *fake_request* method, which returns a *request_rec* object, just like that which is passed into request time *Perl*Handlers* and returned by the subrequest API methods, *lookup_uri* and *lookup_file*. However, this ‘‘fake request’’ does not run handlers for any of the phases, it simply returns an object which we can use to do that ourselves. The `__PACKAGE__` argument is given as our ‘‘location’’ for this request, mainly used for looking up configuration. For example, should we only wish to allow access to this server from certain locations:

```
    <Location Apache::CommandServer>
        deny from all
        allow from 10.*
    </Location>
```

The *fake_request* method only looks up the configuration, we still need to apply it. This is done in *for* loop, iterating over three methods: *check_access*, *check_user_id* and *check_authz*. These methods will call directly into the Apache functions that invoke module handlers for these phases and will return an integer status code, such as **OK**, **DECLINED** or **FORBIDDEN**. If *check_access* returns something other than **OK** or **DECLINED**, that status will be propagated up to the handler routine and then back up to Apache. Otherwise the access check passed and the loop will break unless *some_auth_required* returns true. This would be false given the previous configuration example, but would be true in the presense of a *require* directive, such as:

```
      <Location Apache::CommandServer>
          deny from all
          allow from 10.*
          require user dougm
      </Location>
```

Given this configuration, *some_auth_required* will return true. The *user* method is then called, which will return false if we have not yet authenticated. A *prompt* utility is called to read the username and password, which are then injected into the *headers_in* table using the *set_basic_credentials* method. The *Authenticate* field in this table is set to a base64 encoded value of the username:password pair, exactly the same format a browser would send for *Basic authentication*. Next time through the loop *check_user_id* is called, which will in turn invoke any authentication handlers, such as *mod_auth*. When *mod_auth* calls the *ap_get_basic_auth_pw()* API function (as all Basic auth modules do), it will get back the username and password we injected. If we fail authentication a **401** status code is returned which we propagate up. Otherwise, authorization handlers are run via *check_authz*. Authorization handlers normally need the *user* field of the *request_rec* for its checks and that field was filled in when *mod_auth* called *ap_get_basic_auth_pw()*.

Provided login is a success, a welcome message is printed and main request loop entered. Inside the loop the *getline* method returns just one line of data, with newline characters stripped. If the string sent by the client is in our command table, the command is then invoked, otherwise a usage message is sent. If the command does not return a true value, we break out of the loop. Let’s give it a try with this configuration:

```
   <IfDefine Apache::CommandServer>
       Port 8085
       PerlProcessConnectionHandler Apache::CommandServer

       <Location Apache::CommandServer>
           allow from 127.0.0.1
           require user dougm
           satisfy any
           AuthUserFile /tmp/basic-auth
       </Location>
   </IfDefine>
```

```
% telnet localhost 8085
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Login: dougm
Password: foo
Welcome to Apache::CommandServer
Available commands: motd date who quit
motd
Have a lot of fun...
date
Wed Sep 13 23:47:26 2000
who
dougm    tty1     Sep  7 11:40
dougm    ttyp0    Sep 12 11:38 (:0.0)
dougm    ttyp1    Sep 12 15:50 (:0.0)
quit
Connection closed by foreign host.
```

1.6.3 Apache::CommandServer Source

```
package Apache::CommandServer;
```

```
use strict;
```

```
my @cmds = qw(motd date who quit);
my %commands = map { $_, \&{$_} } @cmds;
```

```
sub handler {
    my Apache::Connection $c = shift;
```

```
    $c->autoflush(1);
```

```
    if ((my $rc = login($c)) != Apache::OK) {
        $c->write("Access Denied\n");
        return $rc;
    }
```

```
    $c->write("Welcome to ", __PACKAGE__,
              "\nAvailable commands: @cmds\n");
```

```
    while (!$c->eof) {
        my $cmd;
        next unless $cmd = $c->getline;
```

```
            if (my $sub = $commands{$cmd}) {
                last unless $sub->($c);
            }
            else {
                $c->write("Commands: @cmds\n");
            }
        }

        return Apache::OK;
    }

    sub login {
        my $c = shift;

        my $r = $c->fake_request(__PACKAGE__);

        for my $method (qw(check_access check_user_id check_authz)) {
            my $rc = $r->$method();

            if ($rc != Apache::OK and $rc != Apache::DECLINED) {
                return $rc;
            }

            last unless $r->some_auth_required;

            unless ($r->user) {
                my $username = prompt($c, "Login");
                my $password = prompt($c, "Password");

                $r->set_basic_credentials($username, $password);
            }
        }

        return Apache::OK;
    }

    sub prompt {
        my($c, $msg) = @_;
        $c->write("$msg: ");
        $c->getline;
    }
```

```
sub motd {
    my $c = shift;
    open my $fh, '/etc/motd' or return;
    local $/;
    $c->write(<$fh>);
    close $fh;
}

sub date {
    my $c = shift;
    $c->write(scalar localtime, "\n");
}

sub who {
    my $c = shift;
    $c->write(`who`);
}

sub quit {0}

1;
__END__
```

1.7 mod_perl-2.0 Optimizations

As mentioned in the introduction, the rewrite of mod_perl gives us the chances to build a smarter, stronger and faster implementation based on lessons learned over the 4.5 years since mod_perl was introduced. There are optimizations which can be made in the mod_perl source code, some which can be made in the Perl space by optimizing its syntax tree and some a combination of both. In this section we'll take a brief look at some of the optimizations that are being considered.

The details of these optimizations will from the most part be hidden from mod_perl users, the exeception being that some will only be turned on with configuration directives. The explanation of these optimization ideas are best left for the live talk, a few which will be overviewed then include:

"Compiled" Perl*Handlers

Method calls faster than subroutine calls!

'print' enhancements

Inlined Apache::*.xs calls

Use of Apache Pools for memory allocations

Copy-on-write strings

1.8 References

http://perl.apache.org/

The mod_perl homepage will announce mod_perl-2.0 developments as they become available.

Table of Contents: