1 Protocol Handlers

1.1 Description

This chapter explains how to implement Protocol (Connection) Handlers in mod_perl.

1.2 Connection Cycle Phases

As we saw earlier, each child server (be it a thread or a process) is engaged in processing connections. Each connection may be served by different connection protocols, e.g., HTTP, POP3, SMTP, etc. Each connection may include more than one request, e.g., several HTTP requests can be served over a single connection, when several images are requested for the same webpage.

The following diagram depicts the connection life cycle and highlights which handlers are available to mod_perl 2.0:

connection cycle

When a connection is issued by a client, it's first run through `PerlPreConnectionHandler` and then passed to the `PerlProcessConnectionHandler`, which generates the response. When `PerlProcessConnectionHandler` is reading data from the client, it can be filtered by connection input filters. The generated response can be also filtered though connection output filters. Filters are usually used for modifying the data flowing though them, but can be used for other purposes as well (e.g., logging interesting information). For example the following diagram shows the connection cycle mapped to the time scale:

connection cycle timing

The arrows show the program control. In addition, the black-headed arrows also show the data flow. This diagram matches an interactive protocol, where a client send something to the server, the server filters the input, processes it and send it out through output filters. This cycle is repeated till the client or the server don't tell each other to go away or abort the connection. Before the cycle starts any registered pre_connection handlers are run.

Now let's discuss each of the `PerlPreConnectionHandler` and `PerlProcessConnectionHandler` handlers in detail.

1.2.1 PerlPreConnectionHandler

The *pre_connection* phase happens just after the server accepts the connection, but before it is handed off to a protocol module to be served. It gives modules an opportunity to modify the connection as soon as possible and insert filters if needed. The core server uses this phase to setup the connection record based on the type of connection that is being used. mod_perl itself uses this phase to register the connection input and output filters.

In mod_perl 1.0 during code development `Apache::Reload` was used to automatically reload modified since the last request Perl modules. It was invoked during `post_read_request`, the first HTTP request's phase. In mod_perl 2.0 *pre_connection* is the earliest phase, so if we want to make sure that all

modified Perl modules are reloaded for any protocols and its phases, it's the best to set the scope of the Perl interpreter to the lifetime of the connection via:

```
PerlInterpScope connection
```

and invoke the `Apache2::Reload` handler during the *pre_connection* phase. However this development-time advantage can become a disadvantage in production--for example if a connection, handled by HTTP protocol, is configured as `KeepAlive` and there are several requests coming on the same connection and only one handled by mod_perl and the others by the default images handler, the Perl interpreter won't be available to other threads while the images are being served.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `SRV`, because it's not known yet which resource the request will be mapped to.

Arguments

A *pre_connection* handler is passed a `connection record` as its argument:

```
sub handler {
    my $c = shift;
    # ...
    return Apache2::Const::OK;
}
```

[META: There is another argument passed (the actual client socket), but it is currently an undef]

Return

The handler should return `Apache2::Const::OK` if it completes successfully or `Apache2::Const::FORBIDDEN` if the request is forbidden.

Examples

Here is a useful *pre_connection* phase example: provide a facility to block remote clients by their IP, before too many resources were consumed. This is almost as good as a firewall blocking, as it's executed before Apache has started to do any work at all.

`MyApache2::BlockIP2` retrieves client's remote IP and looks it up in the black list (which should certainly live outside the code, e.g. dbm file, but a hardcoded list is good enough for our example).

```
#file:MyApache2/BlockIP2.pm
#-------------------------
package MyApache2::BlockIP2;

use strict;
use warnings;

use Apache2::Connection ();

use Apache2::Const -compile => qw(FORBIDDEN OK);
```

```
my %bad_ips = map {$_ => 1} qw(127.0.0.1 10.0.0.4);

sub handler {
    my Apache2::Connection $c = shift;

    my $ip = $c->remote_ip;
    if (exists $bad_ips{$ip}) {
        warn "IP $ip is blocked\n";
        return Apache2::Const::FORBIDDEN;
    }

    return Apache2::Const::OK;
}

1;
```

This all happens during the *pre_connection* phase:

```
PerlPreConnectionHandler MyApache2::BlockIP2
```

If a client connects from a blacklisted IP, Apache will simply abort the connection without sending any reply to the client, and move on to serving the next request.

1.2.2 PerlProcessConnectionHandler

The *process_connection* phase is used to process incoming connections. Only protocol modules should assign handlers for this phase, as it gives them an opportunity to replace the standard HTTP processing with processing for some other protocols (e.g., POP3, FTP, etc.).

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `SRV`. Therefore the only way to run protocol servers different than the core HTTP is inside dedicated virtual hosts.

Arguments

A *process_connection* handler is passed a `connection record` object as its only argument.

A socket object can be retrieved from the connection record object.

Return

The handler should return `Apache2::Const::OK` if it completes successfully.

Examples

Here is a simplified handler skeleton:

```
sub handler {
    my ($c) = @_;
    my $sock = $c->client_socket;
    $sock->opt_set(APR::Const::SO_NONBLOCK, 0);
    # ...
    return Apache2::Const::OK;
}
```

Most likely you'll need to set the socket to perform blocking IO. On some platforms (e.g. Linux) Apache gives us a socket which is set for blocking, on other platforms (.e.g. Solaris) it doesn't. Unless you know which platforms your application will be running on, always explicitly set it to the blocking IO mode as in the example above. Alternatively, you could query whether the socket is already set to a blocking IO mode with help of `the opt_get() method`.

Now let's look at the following two examples of connection handlers. The first using the connection socket to read and write the data and the second using bucket brigades to accomplish the same and allow for connection filters to do their work.

1.2.2.1 Socket-based Protocol Module

To demonstrate the workings of a protocol module, we'll take a look at the `MyApache2::EchoSocket` module, which simply echoes the data read back to the client. In this module we will use the implementation that works directly with the connection socket and therefore bypasses connection filters if any.

A protocol handler is configured using the `PerlProcessConnectionHandler` directive and we will use the `Listen` and `<VirtualHost>` directives to bind to the non-standard port **8010**:

```
Listen 8010
<VirtualHost _default_:8010>
    PerlModule                  MyApache2::EchoSocket
    PerlProcessConnectionHandler MyApache2::EchoSocket
</VirtualHost>
```

`MyApache2::EchoSocket` is then enabled when starting Apache:

```
panic% httpd
```

And we give it a whirl:

```
panic% telnet localhost 8010
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Hello
Hello

fOo BaR
fOo BaR

Connection closed by foreign host.
```

Here is the code:

```
#file:MyApache2/EchoSocket.pm
#---------------------------
package MyApache2::EchoSocket;

use strict;
use warnings FATAL => 'all';

use Apache2::Connection ();
use APR::Socket ();

use Apache2::Const -compile => 'OK';
use APR::Const     -compile => 'SO_NONBLOCK';

use constant BUFF_LEN => 1024;

sub handler {
    my $c = shift;
    my $sock = $c->client_socket;

    # set the socket to the blocking mode
    $sock->opt_set(APR::Const::SO_NONBLOCK => 0);

    while ($sock->recv(my $buff, BUFF_LEN)) {
        last if $buff =~ /^[\r\n]+$/;
        $sock->send($buff);
    }

    Apache2::Const::OK;
}
1;
```

The example handler starts 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 `Apache2::Connection` class. We have direct access to the client socket via `Apache2::Connection`'s *client_socket* method. This returns an object, blessed into the `APR::Socket` class. Before using the socket, we make sure that it's set to perform blocking IO, by using the `APR::Const::SO_NONBLOCK` constant, compiled earlier.

Inside the recv/send loop, the handler attempts to read `BUFF_LEN` bytes from the client socket into the `$buff` buffer. The handler breaks the loop if nothing was read (EOF) or if the buffer contains nothing but new line character(s), which is how we know to abort the connection in the interactive mode.

If the handler receives some data, it sends it unmodified back to the client with the `APR::Socket::send()` method. When the loop is finished the handler returns `Apache2::Const::OK`, telling Apache to terminate the connection. As mentioned earlier since this handler is working directly with the connection socket, no filters can be applied.

1.2.2.2 Bucket Brigades-based Protocol Module

Now let's look at the same module, but this time implemented by manipulating bucket brigades, and which runs its output through a connection output filter that turns all uppercase characters into their lowercase equivalents.

The following configuration defines a virtual host listening on port 8011 and which enables the `MyApache2::EchoBB` connection handler, which will run its output through `MyApache2::EchoBB::lowercase_filter` filter:

```
Listen 8011
<VirtualHost _default_:8011>
    PerlModule                   MyApache2::EchoBB
    PerlProcessConnectionHandler MyApache2::EchoBB
    PerlOutputFilterHandler      MyApache2::EchoBB::lowercase_filter
</VirtualHost>
```

As before we start the httpd server:

```
panic% httpd
```

And try the new connection handler in action:

```
panic% telnet localhost 8011
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Hello
hello

fOo BaR
foo bar

Connection closed by foreign host.
```

As you can see the response part this time was all in lower case, because of the output filter.

And here is the implementation of the connection and the filter handlers.

```
#file:MyApache2/EchoBB.pm
#-----------------------
package MyApache2::EchoBB;

use strict;
use warnings FATAL => 'all';

use Apache2::Connection ();
use APR::Socket ();
use APR::Bucket ();
use APR::Brigade ();
use APR::Error ();
use APR::Status ();

use APR::Const     -compile => qw(SUCCESS SO_NONBLOCK);
```

```
use Apache2::Const -compile => qw(OK MODE_GETLINE);

sub handler {
    my $c = shift;

    $c->client_socket->opt_set(APR::Const::SO_NONBLOCK => 0);

    my $bb_in  = APR::Brigade->new($c->pool, $c->bucket_alloc);
    my $bb_out = APR::Brigade->new($c->pool, $c->bucket_alloc);

    my $last = 0;
    while (1) {
        my $rc = $c->input_filters->get_brigade($bb_in,
                                                Apache2::Const::MODE_GETLINE);
        last if APR::Status::is_EOF($rc);
        die APR::Error::strerror($rc) unless $rc == APR::Const::SUCCESS;

        while (!$bb_in->is_empty) {
            my $b = $bb_in->first;

            $b->remove;

            if ($b->is_eos) {
                $bb_out->insert_tail($b);
                last;

            }

            if ($b->read(my $data)) {
                $last++ if $data =~ /^[\r\n]+$/;
                # could do some transformation on data here
                $b = APR::Bucket->new($bb_out->bucket_alloc, $data);
            }

            $bb_out->insert_tail($b);
        }

        my $fb = APR::Bucket::flush_create($c->bucket_alloc);
        $bb_out->insert_tail($fb);
        $c->output_filters->pass_brigade($bb_out);
        last if $last;
    }

    $bb_in->destroy;
    $bb_out->destroy;

    Apache2::Const::OK;
}

use base qw(Apache2::Filter);
use constant BUFF_LEN => 1024;

sub lowercase_filter : FilterConnectionHandler {
    my $filter = shift;

    while ($filter->read(my $buffer, BUFF_LEN)) {
        $filter->print(lc $buffer);
```

```
    }

    return Apache2::Const::OK;
}

1;
```

For the purpose of explaining how this connection handler works, we are going to simplify the handler. The whole handler can be represented by the following pseudo-code:

```
while ($bb_in = get_brigade()) {
    while ($b_in = $bb_in->get_bucket()) {
        $b_in->read(my $data);
        # do something with data
        $b_out = new_bucket($data);

        $bb_out->insert_tail($b_out);
    }
    $bb_out->insert_tail($flush_bucket);
    pass_brigade($bb_out);
}
```

The handler receives the incoming data via bucket bridges, one at a time in a loop. It then process each bridge, by retrieving the buckets contained in it, reading the data in, then creating new buckets using the received data, and attaching them to the outgoing brigade. When all the buckets from the incoming bucket brigade were transformed and attached to the outgoing bucket brigade, a flush bucket is created and added as the last bucket, so when the outgoing bucket brigade is passed out to the outgoing connection filters, it won't be buffered but sent to the client right away.

It's possible to make the flushing code simpler, by using a dedicated method `fflush()` that does just that -- flushing of the bucket brigade. It replaces 3 lines of code:

```
        my $fb = APR::Bucket::flush_create($c->bucket_alloc);
        $bb_out->insert_tail($fb);
        $c->output_filters->pass_brigade($bb_out);
```

with just one line:

```
        $c->output_filters->fflush($bb_out);
```

If you look at the complete handler, the loop is terminated when one of the following conditions occurs: an error happens, the end of stream status code (`EOF`) has been received (no more input at the connection) or when the received data contains nothing but new line characters which we used to to tell the server to terminate the connection.

Now that you've learned how to move buckets from one brigade to another, let's see how the presented handler can be reimplemented using a single bucket brigade. Here is the modified code:

```
sub handler {
    my $c = shift;

    $c->client_socket->opt_set(APR::Const::SO_NONBLOCK, 0);
```

```
    my $bb = APR::Brigade->new($c->pool, $c->bucket_alloc);

    while (1) {
        my $rc = $c->input_filters->get_brigade($bb,
                                                Apache2::Const::MODE_GETLINE);
        last if APR::Status::is_EOF($rc);
        die APR::Error::strerror($rc) unless $rc == APR::Const::SUCCESS;

        for (my $b = $bb->first; $b; $b = $bb->next($b)) {

            last if $b->is_eos;

            if ($b->read(my $data)) {
                last if $data =~ /^[\r\n]+$/;
                my $nb = APR::Bucket->new($bb->bucket_alloc, $data);
                # head->...->$nb->$b ->...->tail
                $b->insert_before($nb);
                $b->remove;
            }
        }

        $c->output_filters->fflush($bb);
    }

    $bb->destroy;

    Apache2::Const::OK;
}
```

This code is shorter and simpler. Since it sends out the same bucket brigade it got from the incoming filters, it only needs to replace buckets that get modified, which is probably the only tricky part here. The code:

```
                # head->...->$nb->$b ->...->tail
                $b->insert_before($nb);
                $b->remove;
```

inserts a new bucket in front of the currently processed bucket, so that when the latter removed the former takes place of the latter.

Notice that this handler could be much simpler, since we don’t modify the data. We could simply pass the whole brigade unmodified without even looking at the buckets. But from this example you can see how to write a connection handler where you actually want to read and/or modify the data. To accomplish that modification simply add a code that transforms the data which has been read from the bucket before it’s inserted to the outgoing brigade.

We will skip the filter discussion here, since we are going to talk in depth about filters in the dedicated to filters tutorial. But all you need to know at this stage is that the data sent from the connection handler is filtered by the outgoing filter and which transforms it to be all lowercase.

And here is the simplified version of this handler, which doesn’t attempt to do any transformation, but simply passes the data though:

```
sub handler {
    my $c = shift;

    $c->client_socket->opt_set(APR::Const::SO_NONBLOCK => 0);

    my $bb = APR::Brigade->new($c->pool, $c->bucket_alloc);

    while (1) {
        my $rc = $c->input_filters->get_brigade($bb,
                                                Apache2::Const::MODE_GETLINE);
        last if APR::Status::is_EOF($rc);
        die APR::Error::strerror($rc) unless $rc == APR::Const::SUCCESS;

        $c->output_filters->fflush($bb);
    }

    $bb->destroy;

    Apache2::Const::OK;
}
```

Since the simplified handler no longer has the condition:

```
$last++ if $data =~ /^[\r\n]+$/;
```

which was used to know when to break from the external `while(1)` loop, it will not work in the interactive mode, because when telnet is used we always end the line with `/[\r\n]/`, which will always send data back to the protocol handler and the condition:

```
last if $bb->is_empty;
```

will never be true. However, this latter version works fine when the client is a script and when it stops sending data, our shorter handler breaks out of the loop.

So let's do one more tweak and make the last version work in the interactive telnet mode without manipulating each bucket separately. This time we will use `flatten()` to slurp all the data from all the buckets, which saves us the explicit loop over the buckets in the brigade. The handler now becomes:

```
sub handler {
    my $c = shift;

    $c->client_socket->opt_set(APR::Const::SO_NONBLOCK => 0);

    my $bb = APR::Brigade->new($c->pool, $c->bucket_alloc);

    while (1) {
        my $rc = $c->input_filters->get_brigade($bb,
                                                Apache2::Const::MODE_GETLINE);
        last if APR::Status::is_EOF($rc);
        die APR::Error::strerror($rc) unless $rc == APR::Const::SUCCESS;

        next unless $bb->flatten(my $data);
        $bb->cleanup;
        last if $data =~ /^[\r\n]+$/;
```

```
            # could transform data here
            my $b = APR::Bucket->new($bb->bucket_alloc, $data);
            $bb->insert_tail($b);

            $c->output_filters->fflush($bb);
        }

        $bb->destroy;

        Apache2::Const::OK;
    }
```

Notice, that once we slurped the data in the buckets, we had to strip the brigade of its buckets, since we re-used the same brigade to send the data out. We used `cleanup()` to get rid of the buckets.

1.3 Examples

Following are some practical examples.

META: If you have written an interesting, but not too complicated module, which others can learn from, please submit a pod to the mailing list so we can include it here.

1.3.1 Command Server

The `MyApache2::CommandServer` example is based on the example in the "TCP Servers with IO::Socket" section of the *perlipc* manpage. 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 `MyApache2::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.

Here is the whole module:

```
  package MyApache2::CommandServer;

  use strict;
  use warnings FATAL => 'all';

  use Apache2::Connection ();
  use Apache2::RequestRec ();
  use Apache2::RequestUtil ();
  use Apache2::HookRun ();
  use Apache2::Access ();
  use APR::Socket ();

  use Apache2::Const -compile => qw(OK DONE DECLINED);
```

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

sub handler {
    my $c = shift;
    my $socket = $c->client_socket;

    if ((my $rc = login($c)) != Apache2::Const::OK) {
        $socket->send("Access Denied\n");
        return $rc;
    }

    $socket->send("Welcome to " . __PACKAGE__ .
                  "\nAvailable commands: @cmds\n");

    while (1) {
        my $cmd;
        next unless $cmd = getline($socket);

        if (my $sub = $commands{$cmd}) {
            last unless $sub->($socket) == Apache2::Const::OK;
        }
        else {
            $socket->send("Commands: @cmds\n");
        }
    }


    return Apache2::Const::OK;
}

sub login {
    my $c = shift;

    my $r = Apache2::RequestRec->new($c);
    $r->location_merge(__PACKAGE__);

    for my $method (qw(run_access_checker run_check_user_id
                       run_auth_checker)) {
        my $rc = $r->$method();

        if ($rc != Apache2::Const::OK and $rc != Apache2::Const::DECLINED) {
            return $rc;
        }

        last unless $r->some_auth_required;

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

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

    return Apache2::Const::OK;
```

```
}

sub getline {
    my $socket = shift;

    my $line;
    $socket->recv($line, 1024);
    return unless $line;
    $line =~ s/[\r\n]*$//;

    return $line;
}

sub prompt {
    my ($socket, $msg) = @_;

    $socket->send("$msg: ");
    getline($socket);
}

sub motd {
    my $socket = shift;

    open my $fh, '/etc/motd' or return;
    local $/;
    $socket->send(scalar <$fh>);
    close $fh;

    return Apache2::Const::OK;
}

sub date {
    my $socket = shift;

    $socket->send(scalar(localtime) . "\n");

    return Apache2::Const::OK;
}

sub who {
    my $socket = shift;

    # make -T happy
    local $ENV{PATH} = "/bin:/usr/bin";

    $socket->send(scalar `who`);

    return Apache2::Const::OK;
}

sub quit { Apache2::Const::DONE }

1;
__END__
```

Next, let's explain what this module does in details.

As with all `PerlProcessConnectionHandlers`, we are passed an `Apache2::Connection` object as the first argument. Again, we will be directly accessing the client socket via the *client_socket* method. 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 `Apache2::RequestRec new()` method, which returns a *request_rec* object, just like that, which is passed at request time to HTTP protocol `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 `location_merge()` method is passed the `location` for this request, it will look up the `<Location>` section that matches the given name and merge it with the default server configuration. For example, should we only wish to allow access to this server from certain locations:

```
<Location MyApache2::CommandServer>
    Order Deny,Allow
    Deny from all
    Allow from 10.*
</Location>
```

The `location_merge()` method only looks up and merges the configuration, we still need to apply it. This is done in *for* loop, iterating over three methods: `run_access_checker()`, `run_check_user_id()` and `run_auth_checker()`. 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 `Apache2::Const::OK`, `Apache2::Const::DECLINED` or `Apache2::Const::FORBIDDEN`. If *run_access_check* returns something other than `Apache2::Const::OK` or `Apache2::Const::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 presence of a `require` directive, such as:

```
<Location MyApache2::CommandServer>
    Order Deny,Allow
    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 *run_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 `run_auth_checker()`. 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()` function 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 `Apache2::Const::OK`, we break out of the loop.

Let's use this configuration:

```
Listen 8085
<VirtualHost _default_:8085>
    PerlProcessConnectionHandler MyApache2::CommandServer

    <Location MyApache2::CommandServer>
        Order Deny,Allow
        Allow from 127.0.0.1
        Require user dougm
        Satisfy any
        AuthUserFile /tmp/basic-auth
    </Location>
</VirtualHost>
```

Since we are using `mod_auth` directives here, you need to make sure that it's available and loaded for this example to work as explained.

The auth file can be created with the help of `htpasswd` utility coming bundled with the Apache server. For example to create a file */tmp/basic-auth* and add a password entry for user *dougm* with password *foobar* we do:

```
% htpasswd -bc /tmp/basic-auth dougm foobar
```

Now we are ready to try the command server:

```
% telnet localhost 8085
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Login: dougm
Password: foobar
Welcome to MyApache2::CommandServer
Available commands: motd date who quit
motd
Have a lot of fun...
date
Mon Mar 12 19:20:10 PST 2001
who
dougm    tty1     Mar 12 00:49
dougm    pts/0    Mar 12 11:23
dougm    pts/1    Mar 12 14:08
dougm    pts/2    Mar 12 17:09
quit
Connection closed by foreign host.
```

1.4 CPAN Modules

Some of the CPAN modules that implement mod_perl 2.0 protocols:

- **`Apache::SMTP` - An SMTP server**

  http://search.cpan.org/dist/Apache-SMTP/

1.5 Maintainers

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

- Stas Bekman [http://stason.org/]

1.6 Authors

-

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

Table of Contents:

1 Protocol Handlers 1
1.1 Description 2
1.2 Connection Cycle Phases 2
1.2.1 PerlPreConnectionHandler 2
1.2.2 PerlProcessConnectionHandler 4
1.2.2.1 Socket-based Protocol Module 5
1.2.2.2 Bucket Brigades-based Protocol Module 7
1.3 Examples 12
1.3.1 Command Server 12
1.4 CPAN Modules 17
1.5 Maintainers 17
1.6 Authors 17