mod_tcl
TCL inside the Apache web server

Olly Stephens, *Systems Architect, ARM Ltd*

***Abstract* — mod_tcl is an apache httpd module that allows TCL code to be executed inside the web server during any of its phases whilst handling requests. This tight integration makes possible many things over and above simple page generation.**

Version 1.0 of mod_tcl was released in 2002. Since then, very little work has been done on the codebase publically. However, some significant changes and improvements have recently been made and it is this new – yet to be published – version that is discussed in this paper.

This paper will start by discussing how apache serves requests and how mod_tcl integrates into its pipeline. It will follow this with examples demonstrating some of the more interesting things you can do with this infrastructure; authentication handling, URL rewriting, dynamic logging. Finally it will discuss the current state of the mod_tcl codebase and future directions.

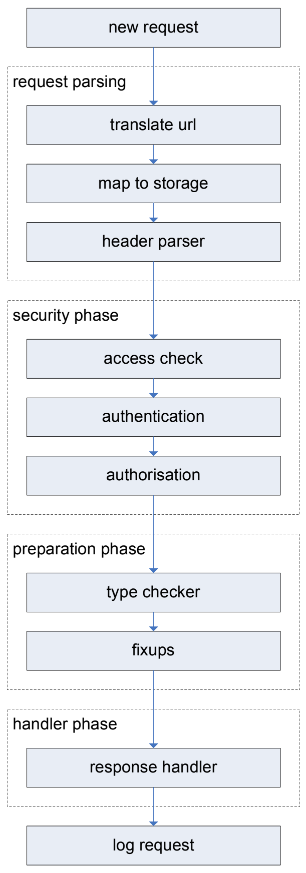
I. Introduction

The apache httpd web server is one of the wider known and heavily used web servers available. It is an extremely robust and production proven system for serving all manner of web pages. One of its key benefits is its module system which allows arbitrary code to hook into a number of key points during the processing of a request. Apache as standard comes with a large array of available modules, from complex URL rewriting infrastructures through custom authentication handlers such as LDAP and, of course, custom renderers such as CGI. In addition to these core modules, there is a large body of contributed work available. Many of these contributed modules include language bindings, allowing code written in a scripting language (typically) to be embedded into the request processing flow. All of the popular scripting languages of today (TCL, Perl, Python, PHP, Ruby, Lua) have at least one integration available. In the case of TCL, there are three. Websh[1] and Rivet[2] are implementations of web frameworks; allowing a mix of code and content to simplify the generation of web pages; both can be optionally built as apache modules, improving the performance under the apache server by leveraging persistent processes.

mod_tcl differs from the other two in that it provides a low-level interface to all of the different hook points that apache makes available during its request processing. This means that you can use mod_tcl to do things other than page generation as we shall see later. mod_tcl does not attempt to provide a higher level framework for page generation as the others do, but can easily be used as the basis for one.

II. Anatomy of an HTTP Request

When apache receives a new request, it goes through a set of stages as indicated in the figure below.



The initial phase – request parsing – is responsible for understanding the URI and mapping it onto an underlying file or service. The second phase – security – applies any access, authentication and authorization checks. There then follows a fixup phase where things like the MIME type of the response are determined. Finally, the handler phase actually generates the response.

For each of these phases, apache provides a number of hook points. Code can be written that will be called during this phase and is capable of handling that phase, rejecting the request or ignoring the request – in the latter case, apache will drop through to the next handler for the phase with each one having a default internal implementation.

A typical web framework implementation will implement a hook for the response handler.

III. MOD_TCL IMPLEMENTATION

mod_tcl implements hooks for all 11 of the stages shown in the previous diagram. Each hook is benign unless additional configuration is provided which wires specific TCL code into a specific hook.

TCL hooks are enabled using directives in the apache configuration file. These take the form:

```
TclPhaseHandler funcname [file]
```

where *phase* determines the place the hook will fire and *funcname* is the function that will be called when the hook fires. If *file* is specified, it indicates a TCL file that will be sourced into the interpreter before the function is called. The file will be sourced into its own namespace (derived from the name of the file) in order to isolate its contents. *funcname* – if not fully qualified – is assumed to be defined in the file, and so is assumed to be in the same derived namespace. If *file* is not specified, then it is assumed that *funcname* will be fully qualified and will refer to a library function. The exception to this assumption is the response handler which is dealt with slightly differently.

Typical examples of TCL directives are:

```
TclTranslateHandler trans tcllib/mapper.tcl
TclLogHandler ::myapp::log_request
```

In the first example above, the file `tcllib/mapper.tcl` will be sourced if it hasn't yet in the current interpreter or if the file has changed since it was last sourced. It will be sourced into its own namespace. Then the `trans` function within that namespace will be invoked to handle the translation.

In the second example above, no file is sourced but the `::myapp::log_request` function will be called to handle the log phase.

The TCL functions that implement the handlers need to understand what apache needs from them. The interface provided by mod_tcl is a low-level one where each TCL API function corresponds to a core apache function and not much more. The functions are called without arguments. A special command `apache::request` gives read/write access to the apache request object. The function is expected to return the appropriate apache return code (e.g. OK, DECLINED, etc.) which are all available as constants in the `::apache` namespace.

For the special case response handler, it is usual for the URI to refer to a specific TCL file. So *file* defaults to be a file based on the URI and it is an error (404 – file not found) if it does not exist. Response handling also requires that the apache configuration associate the file with the TCL handler. This is standard for a response module.

Example directives for response handling would therefore be:

```
AddHandler tcl-script .tcl
TclResponseHandler response
```

In this example, all files that end with .tcl will be handled by the TCL code. For each one, it will load it into the interpreter – in its own namespace as usual – then invoke the response function from that namespace to generate the response.

As with other handlers, if the file has changed since it was last sourced into the interpreter, this step will be redone before the function is called. This is particularly convenient for response handlers as it avoids the necessity of restarting the server just because a content page has changed.

mod_tcl maintains a pool of TCL interpreters from which it allocates them per request. An interpreter that is allocated to a request is used for all phases of that request. Interpreter allocation is deferred until absolutely necessary to avoid the overhead of allocation for requests that do not involve any TCL processing.

Apache can handle concurrent requests by either running in a traditional child-fork model, a threaded single-process model, or a hybrid of the two. The new version of mod_tcl supports all three modes; the old version – like many other language modules – only supported prefork. Interpreters are never simultaneously accessed by separate threads due to their binding to individual requests – which are also only ever single threaded, so there are no contentions. Only interpreter allocation needs to be protected by a mutex, and this is achieved using a standard APR "resource pool" object to manage them. This resource pool object lets you control – through apache configuration directives – the maximum and minimum number of spare interpreters that will be maintained in the pool. This allows the number to swell when the server is busy, and to automatically reclaim that resource when the load lessens.

In addition to the request phase handlers, the apache config can provide TCL code that is invoked in any new interpreter before it is used and code that is invoked before any interpreter is disposed of.

IV. Add-On Modules

In addition to the core functionality provided to TCL by the module, mod_tcl also provides some runtime hooks which allow extension modules to add their own functionality. The core distribution contains the following additional modules. Each one can be optionally included at runtime through a directive in the configuration file.

mod_tcl_io

This module provides an IO abstraction to the TCL code so that `stdin` represents the request content, `stdout` can be used for the response and `stderr` maps to the apache error log.

In addition, this module implements a `filter` function which allows you to use TCL to implement either an input or output filter.

mod_tcl_apreq

This module provides an interface in TCL to the apreq module which is an input filter that parses form requests and makes them available through a standard API. Consequently, it can be used by response handlers to easily process forms, including file uploads. It also provides high-level functions for dealing with cookies which are exposed to TCL as well.

The apreq module/interface is also used by Rivet through a very similar interface.

mod_tcl_ssl

This module provides an interface in TCL to SSL parameters if the request is made to a secure web server.

mod_tcl_apr

This module provides an interface to some of the useful functions implemented in the APR layer. These include functions such as password checking. The interface is far from complete – APR and APRutil have a huge amount of useful and interesting functionality – and should really be factored out into a standalone TCL package instead.

V. Examples

The code snippets for these examples are located at the end of the paper.

A. Simple response handler

This example shows the implementation of a typical simple response handler. Note that the actual response implementation is fairly cluttered and not at all appropriate for large web pages. It is trivial, however, to use other TCL packages to raise the abstraction to a level that is more productive.

B. Authentication handler

This example shows the implementation of a simple authentication handler. Apache authentication has three stages to it; access check (anonymous so usually just remote host checking), authentication (no longer anonymous; is this person who they say they are?) and authorization (can this person – who is authentic – do what they are asking to do). The example uses the latter two to ensure that only one person can access the secret area, and only if he logs in correctly. The example uses basic authentication as it is trivial to setup. A more modern example using forms and cookies is equally viable using mod_tcl – we have implemented one – but would be too much code to be used in this context.

C. Uploading via PUT

This example shows the implementation of a custom file upload process server-side. Browsers don't typically support the PUT protocol, but in custom clients it's simple to implement and server-side it's much more lightweight than a POST request as the file doesn't have to be parsed out of the message.

In this example, a translate handler is employed to ensure that any URL within the uploads area will be directed to the putfile code. The code then provides a custom response handler which processes the upload and returns a plain text status message.

This example also shows the use of standard streams, made possible by the mod_log_io add-on.

D. Downloading (with range support)

This final example shows the implementation of a download manager. Responding to requests with the content of a file is trivial to implement at the response handler layer. But if you want to support the full HTTP standard, you need to also allow for partial range-based downloads. This increases the complexity of the required code considerably. The built-in apache response handler, however, has all this support within it. It is also by far the most efficient way of streaming a large file as it's capable of using numerous tricks to increase the throughput. Consequently, it makes sense – if possible – to redirect static file downloads back into the apache core. This example does just that; it uses the translation phase to map the request to an actual file that will be returned. It then demonstrates another mod_tcl feature – the ability to register additional handlers on a per-request basis. It uses this method to register a type handler – necessary to avoid apache guessing the type from the spool file which in this case may have an odd name – and a log handler to record the download speed at the end of the transfer.

VI. Implementation challenges

A number of the key new features that have been added since mod_tcl 1.0 have already been mentioned in passing. The dynamic handlers demonstrated above is one such example. Another is the support for interpreter pools. This has proved quite challenging to get to a stage where it functions in an obvious and safe way. The thread-safe nature of the core TCL interpreter has made it possible for this module to work in a threaded example – something that isn't true of some other scripting languages' apache modules.

However, there are three elements in particular that have presented specific challenges during the implementation and they are detailed below.

A. Request object handling

Each apache request generates an internal request object. This object is passed to the hooks, and is used to lookup or set information based on the request.

During the handling of a request, sub-requests can be spawned which are children of the initial request but have their own object. Consequently it is sometimes necessary to walk the parent/child relationships of objects in order to determine the correct response.

Some language interfaces – e.g. mod_perl – expose the request object in a variable that is passed as a parameter to the hook code. mod_tcl does not do this because of the added complexity involved in ensuring stale request handles do not get cached in interpreters. Instead, mod_tcl uses a special access function apache::request to give read/write access to the elements of a request through an extensive list of sub-commands. This function additionally has special modifier sub-commands prev and next which allow tests for the presence of previous/next (parent/child) requests and for querying those requests instead. This is best explained using an example:

```
apache::request uri
```

This command returns the URI of the current request

```
apache::request prev
```

This command returns true if a previous request exists

```
apache::request prev uri
```

This command returns the URI of the previous request.

Storing the actual request object for reference when required by TCL commands originally proved troublesome too. However, the ability in 8.5 to attach arbitrary data to a namespace solved this particular conundrum.

B. APR table support

The APR library used extensively by the apache httpd code has a table object which is not dissimilar from a TCL dictionary. However, there is one key difference – APR tables can contain duplicate keys. Duplicate keys are seldom used, but do occur in key places, such as the headers_in and headers_out sections of a request.

In order to support these tables, and to exploit the commonality of these tables to reduce the amount of core code, a TCL interface to the tables was written that is hooked into the necessary places whenever the object being referenced is an APR table. The interface is designed to mimic the dict interface to a certain degree, with the usual `set`, `get` and `keys` sub-commands. However, it also has an `add` sub-command which differs from set in that it adds an additional identical key rather than replacing an existing one. It also has a `dict` command which returns the complete object as a dict for further processing – this ignores duplicate keys so is not applicable for all use cases but covers most.

C. Thread management in the IO add-on

The mod_tcl_io module manipulates TCL streams, associating them with particular requests, flushing them when a request completes, etc. Early implementations tried to define just three main streams, to implement stdin, stdout and stderr. But their use between multiple threads caused problems. The current solution instead maintains three streams per thread, making use of TCL's thread-specific data storage functions.

VII. Future Work

The new version of mod_tcl discussed in this paper is currently in production use within ARM. It has proven itself to be robust, stable and fast. However, the production servers run in prefork mode so more testing is required in multi-threaded more before this claim can be extended to any apache environment. In particular, whilst the code should work no problem under Windows, it has not even been compiled let alone tested.

I am in the final stages of arranging to contribute the code back to the apache project so that people can test it themselves. I hope, some point soon, to have tested it sufficiently to declare it a worthy candidate for a mod_tcl 2.0 core release.

The obvious next step for the code would be to implement additional add-on modules mod_tcl_rivet and/or mod_tcl_websh. Both of these technologies can run under different environments and this would allow them to run inside an apache server that also uses mod_tcl. At the same time, the potential to refactor some code out to a common library shared by all of the implementations may be possible.

Finally, it would be very interesting to implement mod_tcl_dav – an add-on that would expose DAV requests to TCL code. This would enable an implementation that mapped a TCL VFS into the WebDAV protocol for example.

References

[1] http://tcl.apache.org/websh/
[2] http://tcl.apache.org/rivet/

EXAMPLE CODE

A. Simple response handler

conf/httpd.conf

```
<FilesMatch "\.tcl$">
  SetHandler tcl-script
  TclResponseHandler response
</FilesMatch>
```

htdocs/test.tcl

```
proc response {} {
  apache::request content_type text/html
  apache::send_http_header
  apache::rputs "<html><head>"
  apache::rputs "<title>TEST</title>"
  apache::rputs "</head><body>"
  apache::rputs "<h1>TEST</h1>"
  apache::rputs "<p><b>TCL</b> is [info patchlevel]</p>"
  apache::rputs "<p><b>APACHE</b> is [apache::get_server_version]</p>"
  apache::rputs "</body></html>"
  return $::apache::OK
}
```

B. Authentication handler

conf/httpd.conf

```
<Location /secret>
  AuthType Basic
  AuthName "My Secret Stuff"
  Require valid-user
  TclAuthenHandler authen tcllib/auth/handlers.tcl
  TclAuthzHandler  authz  tcllib/auth/handlers.tcl
</Location>
```

tcllib/auth/handlers.tcl

```
proc authen {} {
  foreach {rc pw} [apache::get_basic_auth_pw] break
  if {$rc != $::apache::OK} {
    return $rc
  }
  if {$pw eq [apache::request user]} {
    return $::apache::OK
  }
  apache::note_basic_auth_failure
  return $::apache::HTTP_FORBIDDEN
}

proc authz {} {
  if {[apache::request user] eq "olly"} {
    return $::apache::OK
  } else {
    return $::apache::HTTP_FORBIDDEN
  }
}
```

EXAMPLE CODE

C. *uploading via PUT*

`conf/httpd.conf`

```
<Location /uploads>
  TclTransHandler translate tcllib/putfile.tcl
  TclResponseHandler response
</Location>
```

`tcllib/putfile.tcl`

```
proc translate {} {
  apache::request filename tcllib/putfile.tcl
  return $::apache::OK
}

proc response {} {
  apache::request content_type text/plain
  if {[catch {
    if {[apache::request method] ne "PUT"} {
      error "not a PUT request"
    }
    set nm [file join $::TMPDIR [file tail [apache::request uri]]]
    set fd [open $nm w]
    set sz [fcopy stdin $fd]
    close $fd
    puts "uploaded $nm ($sz bytes)"
  } err] != 0} {
    puts $err
    return $::apache::HTTP_INTERNAL_SERVER_ERROR
  }
  return $::apache::OK
}
```

D. *download wrapping*

`conf/httpd.conf`

```
<Location /downloads>
  TclTransHandler translate tcllib/getfile.tcl
</Location>
```

`tcllib/getfile.tcl`

```
proc translate {} {
  foreach {key fnm typ} [spool_file [apache::request uri]] break
  apache::request filename $fnm
  apache::handler type [namespace code "set_type $typ"]
  apache::handler log  [namespace code "log_dnld $key [clock seconds]"]
  return $::apache::OK
}

proc set_type {typ} {
  apache::request content_type $typ
}

proc log_dnld {key start} {
  log_download $key [apache::request bytes_sent] [expr [clock seconds] - $start]
}
```