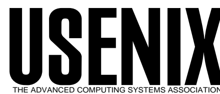
Proceedings of the 7th USENIX Tcl/Tk Conference

Austin, Texas, USA, February 14–18, 2000

THE TCLHTTPD WEB SERVER

Brent Welch



© 2000 by The USENIX Association. All Rights Reserved. For more information about the USENIX Association: Phone: 1 510 528 8649; FAX: 1 510 548 5738; Email: office@usenix.org; WWW: http://www.usenix.org. Rights to individual papers remain with the author or the author's employer. Permission is granted for noncommercial reproduction of the work for educational or research purposes.This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

The TclHttpd Web Server

Brent Welch <welch@scriptics.com>
Scriptics Corporation

Abstract

This paper describes TclHttpd, a web server built entirely in Tcl. The web server can be used as a stand-alone server or it can be embedded into applications to web-enable them. TclHttpd provides a Tcl+HTML template facility that is useful for maintaining site-wide look and feel, and an application-direct URL that invokes a Tcl procedure in an application. This paper describes the architecture of the application and relates our experience using the system to host www.scriptics.com.

Introduction

TclHttpd started out as about 175 lines of Tcl that could serve up HTML pages and images. The Tcl `socket` and I/O commands make this easy. Of course, there are lots of features in web servers like Apache or Netscape that were not present in the first prototype. Steve Uhler took my prototype, refined the HTTP handling, and aimed to keep the basic server under 250 lines. I went the other direction, setting up a modular architecture, adding in features found in other web servers, and adding some interesting ways to connect TclHttpd to Tcl applications.

Today TclHttpd is used both as a general-purpose Web server, and as a framework for building server applications. It implements `www.scriptics.com`, including the Tcl Resource Center and Scriptics' electronic commerce facilities. It is also built into several commercial applications such as license servers and mail spam filters.

Integrating with TclHttpd

TclHttpd is interesting because, as a Tcl script, it is easy to add to your application. Suddenly your application has an interface that is accessible to Web browsers in your company's intranet or the global Internet. The Web server provides several ways you can connect it to your application:

- *Static pages*. As a "normal" web server, you can serve static documents that describe your application.
- *Domain handlers*. You can arrange for all URL requests in a section of your web site to be handled by your application. This is a very general interface where you interpret what the URL means and what sort of pages to return to each request. For example, `http://www.scriptics.com/resource` is implemented this way. The URL past `/resource` selects an index in a simple database, and the server returns a page describing the pages under that index.
- *Application-Direct URLs*. This is a domain handler that maps URLs onto Tcl procedures. The form query data that is part of the HTTP GET or POST request is automatically mapped onto the parameters of the application-direct procedure. The procedure simply computes the page as its return value. This is an elegant and efficient alternative to the CGI interface. For example, in TclHttpd the URLs under `/status` report various statistics about the web server's operation.
- *Document handlers*. You can define a Tcl procedure that handles all files of a particular type. For example, the server has a handler for CGI scripts, HTML files, image maps, and HTML+Tcl template files.
- *HTML+Tcl Templates*. These are web pages that mix Tcl and HTML markup. The server replaces the Tcl using the `subst` command and returns the result. The server can cache the result in a regular HTML file to avoid the overhead of template processing on future requests. Templates are a great way to maintain common look and feel to a family of web pages, as well as to implement more advanced dynamic HTML features like self-checking forms.

TclHttpd Architecture

Figure 1 shows the basic components of the server. At the core is the `Httpd` module, which implements the server side of the `HTTP` protocol. This module manages network requests, dispatches them to the `Url` module, and provides routines used to return the results to requests. The `Url` module divides the web site into *domains*, which are subtrees of the URL hierarchy provided by the server. The idea is that different domains may have completely different implementations. For example, the Document domain maps its URLs into files and directories on your hard disk, while the Application-Direct domain maps URLs into Tcl procedure calls within your application. The CGI domain maps URLs onto other programs that compute web pages.

Domain Handlers

You can implement new kinds of domains that provide your own interpretation of a URL. This is the most flexible interface available to extend the web server. You provide a callback that is invoked to handle every request in a domain, or subtree, of the URL hierarchy. The callback interprets the URL, using routines from the `Httpd` module.

Example 1 defines a simple domain that always returns the same page to every request. The domain is registered with the `Url_PrefixInstall` command. The arguments to `Url_PrefixInstall` are the URL prefix and a callback that is called to handle all URLs that match that prefix. In the example, all URLs that have the prefix `/simple` are dispatched to the `SimpleDomain` procedure.

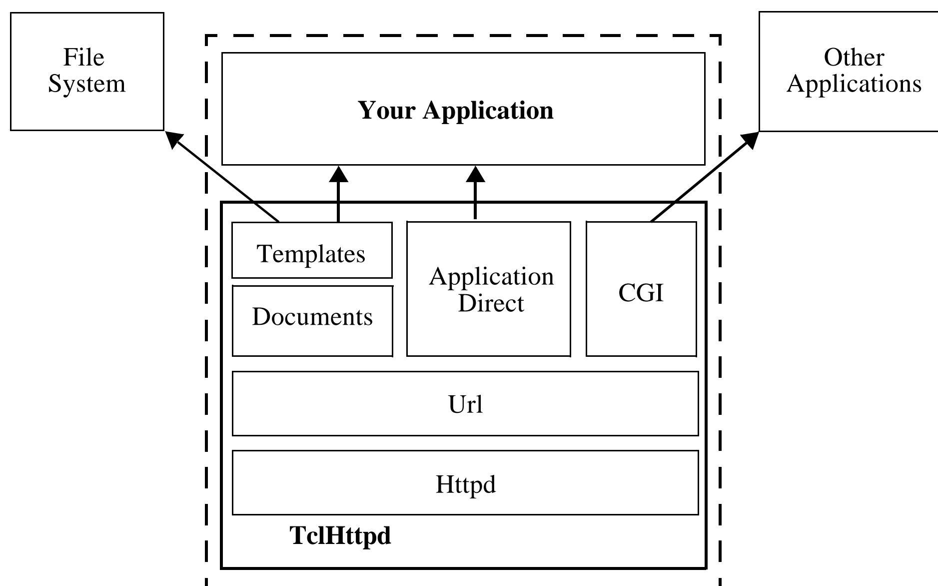


Figure 1 The dotted box represents one application that embeds TclHttpd. Document templates and Application Direct URLs provide direct connections from an HTTP request to your application.

Example 1 A simple URL domain.

```
Url_PrefixInstall /simple [list SimpleDomain /simple]

proc SimpleDomain {prefix sock suffix} {
    upvar #0 Httpd$sock data

    # Generate page header

    set html "<title>A simple page</title>\n"
    append html "<h1>$prefix$suffix</h1>\n"
    append html "<h1>Date and Time</h1>\n"
    append html [clock format [clock seconds]]
    # Display query data

    if {[info exist data(query)]} {
        append html "<h1>Query Data</h1>\n"
        append html "<table>\n"
        foreach {name value} [Url_DecodeQuery $data(query)] {
            append html "<tr><td>$name</td>\n"
            append html "<td>$value</td></tr>\n"
        }
        append html "</table>\n"
    }
    Httpd_ReturnData $sock text/html $html
}
```

The `SimpleDomain` handler illustrates several properties of domain handlers. The `sock` and `suffix` arguments to `SimpleDomain` are appended by `Url_Dispatch` when it invokes the domain handler. The `suffix` parameter is the part of the URL after the prefix. The `prefix` is passed in as part of the callback definition so the domain handler can recreate the complete URL. For example, if the server receives a request for the url `/simple/page`, then the prefix is `/simple`, the suffix is `/page`.

The `sock` parameter is a handle on the socket connection to the remote client. This variable is also used to name a state variable that the `Httpd` module maintains about the connection. The name of the state array is `Httpd$sock`, and `SimpleDomain` uses `upvar` to get a more convenient name for this array (i.e., `data`):

```
upvar #0 Httpd$sock data
```

The only module in the server that uses the socket handle directly is the Httpd module. The rest of the code treats `$sock` as an opaque handle, and uses the `upvar` trick to map that handle into a locally accessible array.

An important element of the state array is the query data, `data(query)`. This is the information that comes from HTML forms. The query data arrives in an encoded format, and the `Url_DecodeQuery` procedure is used to decode the data into a list of names and values.

Finally, once the page has been computed, the `Httpd_ReturnData` procedure is used to return the page to the client. This takes care of the HTTP protocol as well as returning the data. There are three related procedures, `Httpd_ReturnFile`, `Httpd_Error`, and `Httpd_Redirect`.

Application Direct URLs

The Application Direct domain implementation provides the simplest way to extend the web server. It hides the details associated with query data, decoding URL paths, and returning results. All you do is define Tcl procedures that correspond to URLs. Their arguments are automatically matched up to the query data. The Tcl procedures compute a string that is the result data, which is usually HTML. That’s all there is to it.

The `Direct_Url` procedure defines a URL prefix and a corresponding Tcl command prefix. Any URL that begins with the URL prefix will be handled by a corresponding Tcl procedure that starts with the Tcl command prefix. This is shown in Example 2:

Example 2 Application Direct URLs

```
Direct_Url /demo Demo

proc Demo {} {
    return "<html><head><title>Demo page</title></head>\n\
        <body><h1>Demo page</h1>\n\
        <a href=/demo/time>What time is it?</a>\n\
        <form action=/demo/echo>\n\
        Data: <input type=text name=data>\n\
        <br>\n\
        <input type=submit name=echo value='Echo Data'>\n\
        </form>\n\
        </body></html>"
}
proc Demo/time {{format "%H:%M:%S"}} {
    return [clock format [clock seconds] -format $format]
}
proc Demo/echo {args} {
    # Compute a page that echos the query data

    set html "<head><title>Echo</title></head>\n"
    append html "<body><table>\n"
    foreach {name value} $args {
        append html "<tr><td>$name</td><td>$value</td></tr>\n"
    }
    append html "</tr></table>\n"
    return $html
}
```

Example 2 defines `/demo` as an Application Direct URL domain that is implemented by procedures that begin with `Demo`. There are just three URLs defined:

```
/demo
/demo/time
/demo/echo
```

The `/demo` page displays a hypertext link to the `/demo/time` page, and a simple form that will be handled by the `/demo/echo` page. This page is static, and so there is just one `return` command in the procedure body.

The `/demo/time` procedure just returns the result of `clock format`. It doesn't even bother adding `<html>`, `<head>`, or `<body>` tags, which you can get away with in today's browsers. A simple result like this is also useful if you are using programs to fetch information via HTTP requests. The `/demo/time` procedure is defined with an optional `format` argument. If a `format` value is present in the query data then it overrides the default value given in the procedure definition.

Using Query Data

The /demo/echo procedure creates a table that shows its query data. Its `args` parameter gets filled in with a name-value list of all query data. You can have named parameters, named parameters with default values, and the `args` parameter in your application-direct URL procedures. The server automatically matches up incoming form values with the procedure declaration. For example, suppose you have an application direct procedure declared like this:

```
proc Demo/param { a b {c cdef} args} body
```

You could create an HTML form that had elements named `a`, `b`, and `c`, and specified /demo/param for the `ACTION` parameter of the `FORM` tag. Or, you could type the following into your browser to embed the query data right into the URL:

```
/demo/param?a=5&b=7&c=red&d=%7ewelch&e=two+words
```

In this case, when your procedure is called, `a` is `5`, `b` is `7`, `c` is `red`, and the `args` parameter becomes a list of:

```
d ~welch e {two words}
```

Returning Other Content Types

The default content type for application direct URLs is `text/html`. You can specify other content types by using a global variable with the same name as your procedure. (Yes, this is a crude way to craft an interface.) Example 3 shows part of the `faces.tcl` file that implements an interface to a database of picons, or personal icons, that is organized by user and domain names. The idea is that the database contains images corresponding to your email correspondents. The `Faces_ByEmail` procedure, which is not shown, looks up an appropriate image file. The application direct procedure is `Faces/byemail`, and it sets the global variable `Faces/byemail` to the correct value based on the filename extension. This value is used for the `Content-Type` header in the result part of the `HTTP` protocol.

Document Types

The Document domain (`doc.tcl`) maps URLs onto files and directories. It provides more ways to extend the server by registering different document type handlers. This occurs in a two step process. First the type of a file is determined by its suffix. The `mime.types` file contains a map from suffixes to `MIME` types such as `text/html` or `image/gif`. This map is controlled by the `Mtype` module in `mtype.tcl`. Second, the server checks for a Tcl procedure with the appropriate name:

`Doc_`*`mimetype`*

The matching procedure, if any, is called to handle the URL request. The procedure should use routines in the `Httpd` module to return data for the request. If there is no matching `Doc_mimetype` procedure, then the default document handler uses `Httpd_ReturnFile` and specifies the Content Type based on the file extension:

```
Httpd_ReturnFile $sock [Mtype $path] $path
```

Example 3 Alternate types for Application Direct URLs.

```
Direct_Url /faces Faces
proc Faces/byemail {email} {
    global Faces/byemail
    set filename [Faces_ByEmail $email]
    set Faces/byemail [Mtype $filename]
    set in [open $filename]
    fconfigure $in -translation binary
    set X [read $in]
    close $in
    return $X
}
```

Example 4 A sample document type handler.

```
# Add this line to mime.types
application/myjunk      .junk

# Define the document handler procedure
#   path is the name of the file on disk
#   suffix is part of the URL after the domain prefix
#   sock is the handle on the client connection

proc Doc_application/myjunk {path suffix sock} {
    upvar #0 Httpd$sock data
    # data(url) is more useful than the suffix parameter.

    # Use the contents of file $path to compute a page
    set contents [somefunc $path]

    # Determine your content type
    set type text/html

    # Return the page
    Httpd_ReturnData $sock $type $data
}
```

You can make up new types to support your application. Example 4 shows the pieces need to create a handler for a fictitious document type application/myjunk that is invoked to handle files with the .junk suffix. You need to edit the mime.types file and add a document handler procedure to the server:

As another example, the HTML+Tcl templates use the .tml suffix that is mapped to the application/x-tcl-template type. The TclHttpd distribution also includes support for files with a .snmp extension that implement a template-based web interface to the Scotty SNMP Tcl extension.

HTML + Tcl Templates

The template system uses HTML pages that embed Tcl commands and Tcl variable references. The server replaces these using the subst command and returns the results. The server comes with a general template system, but using subst is so easy you could create your own template system. The general template framework has these components:

- Each .html file has a corresponding .tml template file. This feature is enabled with the Doc_CheckTemplates command in the server's configuration file. Normally, the server returns the .html file unless the corresponding .tml file has been modified more recently. In this case the server processes the template, caches the result in the .html file, and returns the result.
- A dynamic template (e.g., a form handler) must be processed each time it is requested. If you put the Doc_Dynamic command into your page it turns off the caching of the result in the .html page. The server responds to a request for a .html page by processing the .tml page. Or, you can just reference the .tml file directly, in which case the server always processes the template.
- The server creates a page global Tcl variable that has context about the page being processed.
- The server initializes the env global Tcl variable with similar information, but in the standard way for CGI scripts.
- The server supports per-directory ".tml" files that contain Tcl source code. These files are designed to contain procedure definitions and variable settings that are shared among pages. The name of the file is simply ".tml", with nothing before the period. The server will source the ".tml" files in all directories leading down to the directory containing the template file. The server compares the modify time of these files against the template file and will process the template if these ".tml" files are newer than the cached .html file. So, by modifying the ".tml" file in the root of your URL hierarchy you invalidate all the cached .html files.
- The server supports a script library for the procedures called from templates. The Doc_TemplateLibrary procedure registers this directory. The server adds the directory to its auto_path, which assumes you have a tclIndex or pkgIndex.tcl file in the directory so the procedures are loaded when needed.

Where to put your Tcl Code

There are three places you can put the code of your application: directly in your template pages, in the per-directory ".tml" files, or in the library directory. The advantage of putting procedure definitions in the library is that they are defined one time but executed many times. This works well with the Tcl byte-code compiler. The disadvantage is that if you modify procedures in these files you have to explicitly source them into the server for these changes to take effect. A built-in URL makes this possible. The /debug/source URL accepts a source parameter that indicates what file to load. For safety reasons, it only loads files from the script library directory.

The advantage of putting code into the per-directory ".tml" files is that changes are picked up immediately with no effort on your part. The server automatically checks if these files are modified, and sources them each time it processes your templates. However, that code is only run one time, so the byte-code compiler just adds overhead. In general, I try to limit the code in the actual pages to simple procedure calls. Complex code directly in pages cannot be shared, and is more awkward to edit.

Form Handlers

TclHttpd provides alternatives to CGI that are more efficient because they are built right into the server. This eliminates the overhead that comes from running an external program to compute the page. Another advantage is that the Web server can maintain state between client requests in Tcl variables. If you use CGI, you must use some sort of database or file storage to maintain information between requests.

Application Direct Handlers

The server comes with several built-in forms handlers that you can use with little effort. The `/mail/forminfo` URL will package up the query data and mail it to you. You use form fields to set various mail headers, and the rest of the data is packaged up into a Tcl-readable mail message. Example 5 shows a form that uses this handler.

The mail message sent by `/mail/forminfo` is shown in Example 6.

It is easy to write a script that strips the headers, defines a `data` procedure, and uses `eval` to process the message body. Whenever you send data via email, if you format it with Tcl list structure you can process it quite easily.

Template Form Handlers

The drawback of using application-direct URL form handlers is that you have to modify their Tcl implementation to change the resulting page. Another approach is to use templates for the result page that embed a command that handles the form data. The `Mail_FormInfo` procedure, for example, mails form data. It takes no arguments. Instead, it looks in the query data for `sendto` and `subject` values, and if they are present it sends the rest of the data in an email. It returns an HTML comment that flags that mail was sent.

A *self-posting form* is a form that posts the form data to back to the page containing the form. The page embeds a Tcl command to check its own form data. Once the data is correct the page triggers a redirect to the next page in the flow. This is a powerful trick, which I learned from Monty Swiryn of Cuesta Technologies, that you can use to create complex page flows using templates.

Example 5 Mail form results with /mail/forminfo.

```
<form action=/mail/forminfo method=post>
   <input type=hidden name=sendto value=mailreader@my.com>
   <input type=hidden name=subject value="Name and Address">
   <table>
      <tr><td>Name</td><td><input name=name></td></tr>
      <tr><td>Address</td><td><input name=addr1></td></tr>
      <tr><td> </td><td><input name=addr2></td></tr>
      <tr><td>City</td><td><input name=city></td></tr>
      <tr><td>State</td><td><input name=state></td></tr>
      <tr><td>Zip/Postal</td><td><input name=zip></td></tr>
      <tr><td>Country</td><td><input name=country></td></tr>
   </table>
</form>
```

Example 6 Mail message sent by /mail/forminfo

```
To: mailreader@my.com
Subject: Name and Address

data {
   name   {Joe Visitor}
   addr1  {Acme Company}
   addr2  {100 Main Street}
   city   {Mountain View}
   state  California
   zip    12345
   country   USA
}
```

Of course, you need to save the form data at each step. You can put the data in Tcl variables, use the data to control your application, or store it into a database. TclHttpd comes with a `Session` module that is one way to manage this information.

Example 7 shows the `Form_Simple` procedure that generates a simple self-checking form. Its arguments are a unique id for the form, a description of the form fields, and the URL of the next page in the flow. The field description is a list with three elements for each field: a required flag, a form element name, and a label to display with the form element. You can see this structure in the template shown in Example 8 on page 9. The procedure does two things at once. It computes the HTML form, and it also checks if the required fields are present. It uses some procedures from the `form` module, which is described on page 9, to generate form elements that retain values from the previous page. If all the required fields are present, it discards the HTML, saves the data, and triggers a redirect by calling `Doc_Redirect`.

Example 7 A self-checking form procedure.

```
proc Form_Simple {id fields nextpage} {
    global page
    if {![form::empty formid]} {
        # Incoming form values, check them
        set check 1
    } else {
        # First time through the page
        set check 0
    }
    set html "<!-- Self-posting. Next page is $nextpage -->\n"
    append html "<form action=\"$page(url)\" method=post>\n"
    append html "<input type=hidden name=formid value=$id>\n"
    append html "<table border=1>\n"
    foreach {required key label} $fields {
        append html "<tr><td>"
        if {$check && $required && [form::empty $key]} {
            lappend missing $label
            append html "<font color=red>*</font>"
        }
        append html "</td><td>$label</td>\n"
        append html "<td><input [form::value $key]></td>\n"
        append html "</tr>\n"
    }
    append html "</table>\n"
    if {$check} {
        if {![info exist missing]} {

            # No missing fields, so advance to the next page.
            # In practice, you must save the existing fields
            # at this point before redirecting to the next page.

            Doc_Redirect $nextpage
        } else {
            set msg "<font color=red>Please fill in "
            append msg [join $missing ", "]
            append msg "</font>"
            set html <p>$msg\n$html
        }
    }
    append html "<input type=submit>\n</form>\n"
    return $html
}
```

Example 8 A page with a self-checking form.

```
<html><head>
    <title>Name and Address Form</title>
</head>
<body bgcolor=white text=black>
    <h1>Name and Address</h1>
    Please enter your name and address.
    [myform::simple nameaddr {
        1 name    "Name"
        1 addr1   "Address"
        0 addr2"  "Address"
        1 city    "City"
        0 state   "State"
        1 zip     "Zip Code"
        0 country "Country"
    } nameok.html]
</body></html>
```

Example 8 shows a page template that calls `Form_Simple` with the required field description.

The `form` package

TclHttpd comes with a `form` package that is designed to support self-posting forms. The `Form_Simple` procedure uses `form::empty` to test if particular form values are present in the query data. The `form::value` procedure is useful for constructing form elements on self-posting form pages. It returns:

```
name="name" value="value"
```

The *value* is the value of form element *name* based on incoming query data, or just the empty string if the query value for *name* is undefined. This way the form can post to itself and retain values from the previous version of the page. It is used like this:

```
<input type=text [form::value name]>
```

The `form::checkvalue` and `form::radiovalue` procedures are similar to `form::value` but designed for checkbuttons and radio buttons. The `form::select` procedure formats a selection list and highlights the selected values. The `form::data` procedure simply returns the value of a given form element.

Experiences with the Server

I have used TclHttpd on two main servers, sunscript.sun.com and www.scriptics.com, and many internal web sites. During a recent week, www.scriptics.com got over 18,000 home page hits, over 200,000 HTTP requests, over 4 gigabytes of data transferred, and over 26,000 page views in the Tcl Resource Center. If you visit http://www.scriptics.com/status you can get a live view of the statistics. This page shows per-minute hit rates over the last hour, per hour hit rates over the last day, and daily hit rates since the server was started. These "hits" are URL requests, which are larger than the number of page views because of images on a page. This traffic is high compared to an average companies site, but low compared to a large portal site.

The current per-minute rates are 100 to 200 hits/minute. Previously it was as high as 500 hits/minute due to the large number of images on our pages. We recently split the image traffic to another web server. On two occasions a bug in the server trapped a remote client by accidentally redirecting it to the same page that the client was requesting. This caused the client to fetch the same page again and again. When this happened, I observed sustained per-minute hit rates of over 700 hits/minute. Both problems were fixed by loading an explicit redirect that aimed the client at the page they really wanted; it did not require a server restart.

I performed some basic comparisons of servers on a test network of four machines on 100Mbit ethernet. The machines were a dual-processor Sparc-20 running at 75 MHz, an Ultra-5 Sparc running at 270 Mhz, a Pentium II running Linux at 400 MHz, and a Pentium III running Windows NT at 450 MHz. TclHttpd was run on all platforms, while Apache, Netscape, AOLserver, and IIS were run on a subset of the platforms. No performance tuning was done on any of the servers.

The test simply performs a number of HTTP

requests to various URLs: a URL implemented by a simple Tcl procedure, a small image, a medium sized image, and a large image.

Three charts are shown in the appendix. The Dell-450 chart shows the performance of TclHttpd and IIS on the fastest machine. TclHttpd adds overhead that is relatively high for small transfers (about 3.5 msec vs 12.5 msec for 200 bytes) and less so for big transfers (about 23 msec vs 37 for 120K.) The Dynamic Pages chart shows the performance of dynamic pages. This shows the obvious benefit of building page generation right into the server. The fastest is AOLserver at about 8 msec. The mod_tcl plugin for Apache was close behind at 11 msec. TclHttpd was about 23 msec, and CGI from Apache was about 72 msec. The 32 Kbyte chart compares the time for all different servers to deliver a 32Kbyte image. TclHttpd runs from 2 to 3 times slower than the fastest server on the platform for this sized transfer.

Perhaps the most notable experience from using TclHttpd on www.scriptics.com is that it is extremely robust. The server is a Sparc-20 running Solaris 2.5.1, and in a two year period I experienced one OS crash, and two or three occasions where I was forced to reboot the machine for various administrative reasons. Tcl, of course, never crashed, so TclHttpd ran for months at a time.

The other exciting thing about TclHttpd is the ability to modify the application without restarting the server. In the early days at sunscript.sun.com I fixed various bugs in the core TclHttpd code. It is more common that the bug fixes are in various the form handlers we have at www.scriptics.com. When a page generates a Tcl error, an error page is displayed in the browser. This contains a form with two options: view the `errorInfo` from the error, or mail that information to `webmaster@scriptics.com`. Of course, TclHttpd continues to function. We can usually diagnose the problem quickly just by looking at `errorInfo`. For difficult bugs we start another copy of the server and connect to it remotely with TclPro Debugger. Once the bug is fixed we simply load new code into the server to fix the problem. One danger of continuously modifying the server is that you can have a server that is running fine but cannot be restarted because of bugs in the startup code. After significant server changes I either restart the server or test the startup sequence by starting the application on a different port.

There have been many applications of TclHttpd as an embedded server. We use it for the Scriptics License Server that implements our shared licences for TclPro.

Current work on TclHttpd includes exploiting the threading capabilities of Tcl 8.2. A threaded server can eliminate the need to use CGI for long-running template code. In addition, Matt Newman has used the built-in stacked channel support in Tcl 8.2 to create a clean SSL extension to the server.

Related Work

There are a number of other interesting Tcl-based Web servers. Karl Lehenbaur presented a paper on the NeoWebScript Apache plugin in an earlier Tcl/Tk conference. David Welton wrote the mod_dtcl plugin for Apache. Probably the most mature Tcl-based web server is the AOLserver. This has used a multi-threaded version of Tcl for some time: first 7.4, then 7.6, and now 8.2. All of these support HTML+Tcl templates, although the syntax used to embed Tcl on the page varies somewhat from server to server.

What makes TclHttpd novel is the ability to embed the server into another application. The event-based model simplifies the integration of the server into the application. In contrast, an Apache or IIS plugin is forced to deal with the multi-process or multi-thread architecture of the hosting web server. Once you have embedded TclHttpd, you have a variety of ways to integrate it with your application, including Application-Direct URLs, custom domain handlers, document handlers, and dynamic page templates.

Web Links

The TclHttpd home page:
`http://www.scriptics.com/products/tclhttpd/`

The AOLserver home page:
`http://www.aolserver.com/`

The mod_dtcl home page:
`http://comanche.com.dtu.dk/dave/`

The NeoWebScript home page:
`http://www.NeoSoft.com/neowebscript/`

Appendix A: Templates for Site Structure

This appendix shows a simple template system used to maintain a common look at feel across the pages of a site. Example 9 shows a simple one-level site definition that is kept in the root `.tml` file. This structure lists the title and URL of each page in the site:

Each page includes two commands, `SitePage` and `SiteFooter` that generate HTML for the navigational part of the page. Between these commands is regular HTML for the page content. Example 10 shows a sample template file:

The `SitePage` procedure takes the page title as an argument. It generates HTML to implement a standard navigational structure. Example 11 has a simple implementation of `SitePage`:

The `foreach` loop that computes the simple menu of links turns out to be useful in many places. Example 12 splits out the loop and uses it in the `SitePage` and `SiteFooter` procedures. This version of the templates creates a left column for the navigation and a right column for the page content:

Of course, a real site will have more elaborate graphics and probably a two-level, three-level, or more complex tree structure that describes its structure.You can also define a family of templates so that each page doesn't have to fit the same mold. Once you start using templates, it is fairly easy to change both the template implementation and to move pages around among different sections of your web site.

There are many other applications for "macros" that make repetitive HTML coding chores easy. Take, for example, the link to `/ordering.html` in Example 10. The proper label for this is already defined in `$site(pages)`, so we could introduce a `SiteLink` procedure that uses this:

If your pages embed calls to `SiteLink`, then you can change the URL associated with the page name by changing the value of `site(pages)`. If this is stored in the top-level "`.tml`" file, the templates will automatically track the changes.

Example 9 A one-level site structure.

```
set site(pages) {
    Home                /index.html
    "Ordering Computers"/ordering.html
    "New Machine Setup" /setup.html
    "Adding a New User" /newuser.html
    "Network Addresses" /network.html
}
```

Example 10 A HTML + Tcl template file.

```
[SitePage "New Machine Setup"]
This page describes the steps to take when setting up a new
computer in our environment. See
<a href=/ordering.html>Ordering Computers</a>
for instructions on ordering machines.
<ol>
<li>Unpack and setup the machine.
<li>Use the Network control panel to set the IP address
and hostname.
<!-- Several steps omitted -->
<li>Reboot for the last time.
</ol>
[SiteFooter]
```

Example 11 SitePage template procedure. Simple horizontal menu along the top of the page.

```
proc SitePage {title} {
    global site
    set html "<html><head><title>$title</title></head>\n"
    append html "<body bgcolor=white text=black>\n"
    append html "<h1>$title</h1>\n"
    set sep ""
    foreach {label url} $site(pages) {
```

```
        append html $sep
        if {[string compare $label $title] == 0} {
            append html "$label"
        } else {
            append html "<a href='$url'>$label</a>"
        }
        set sep " | "
    }
    return $html
}
```

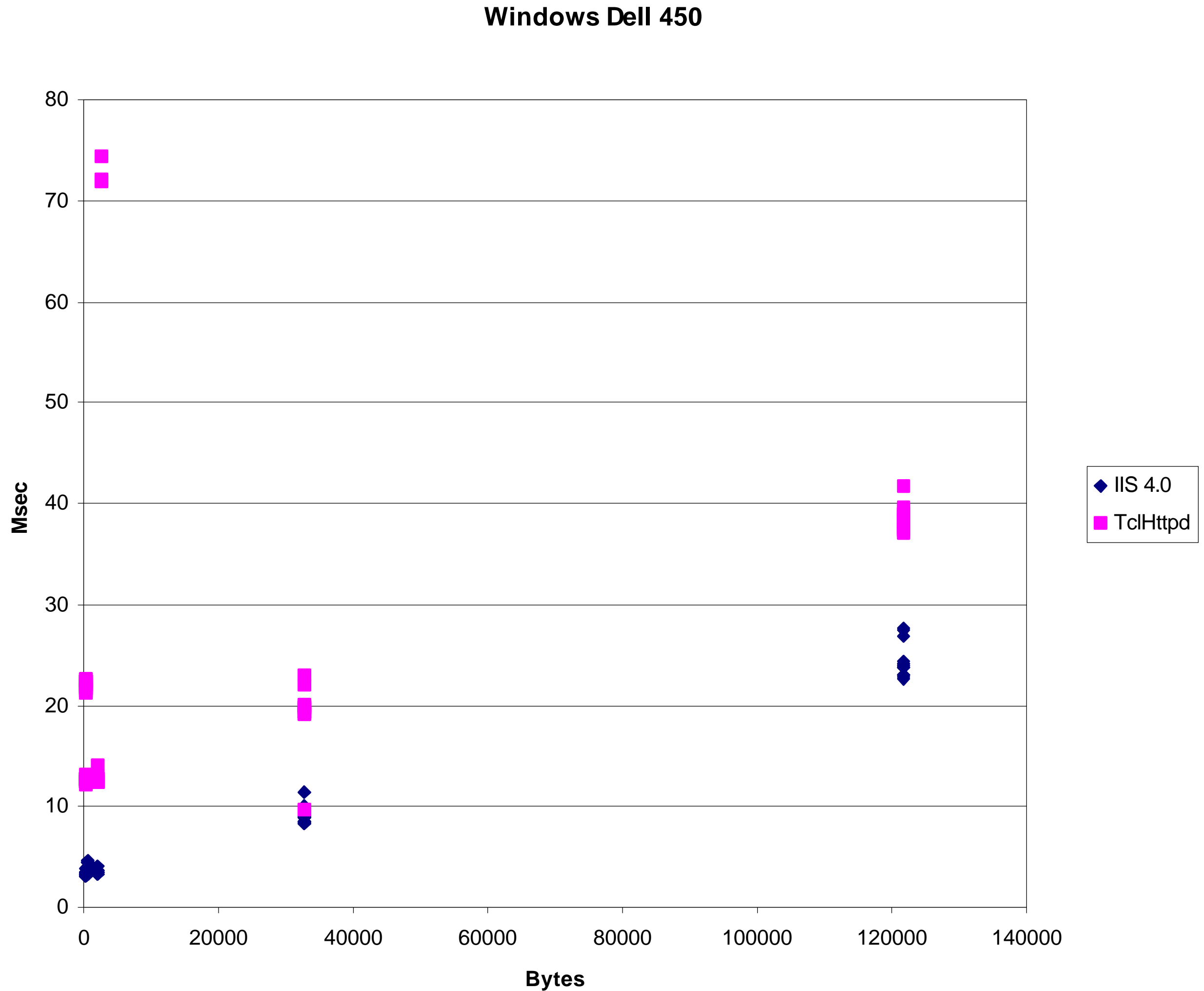
Example 12 SiteMenu and SiteFooter template procedures. Two-column format with menu in the left column.

```
proc SitePage {title} {
    global site
    set html "<html><head><title>$title</title></head>\n\
        <body bgcolor=$site(bg) text=$site(fg)>\n\
        <!-- Two Column Layout -->\n\
        <table cellpadding=0>\n\
        <tr><td>\n\
        <!-- Left Column -->\n\
        <img src='$site(mainlogo)'>\n\
        <font size=+1>\n\
        [SiteMenu <br> $site(pages)]\n\
        </font></td><td>\n\
        <!-- Right Column -->\n\
        <h1>$title</h1>\n\
        <p>\n"
    return $html
}
proc SiteFooter {} {
    global site
    set html "<p><hr>\n\
        <font size=-1>[SiteMenu | $site(pages)]</font>\n\
        </td></tr></table>\n"
    return $html
}
proc SiteMenu {sep list} {
    global page
    set s "" ; set html ""
    foreach {label url} $list {
        if {[string compare $page(url) $url] == 0} {
            append html $s$label
        } else {
            append html "$s<a href='$url'>$label</a>"
        }
        set s $sep
    }
    return $html
}
```

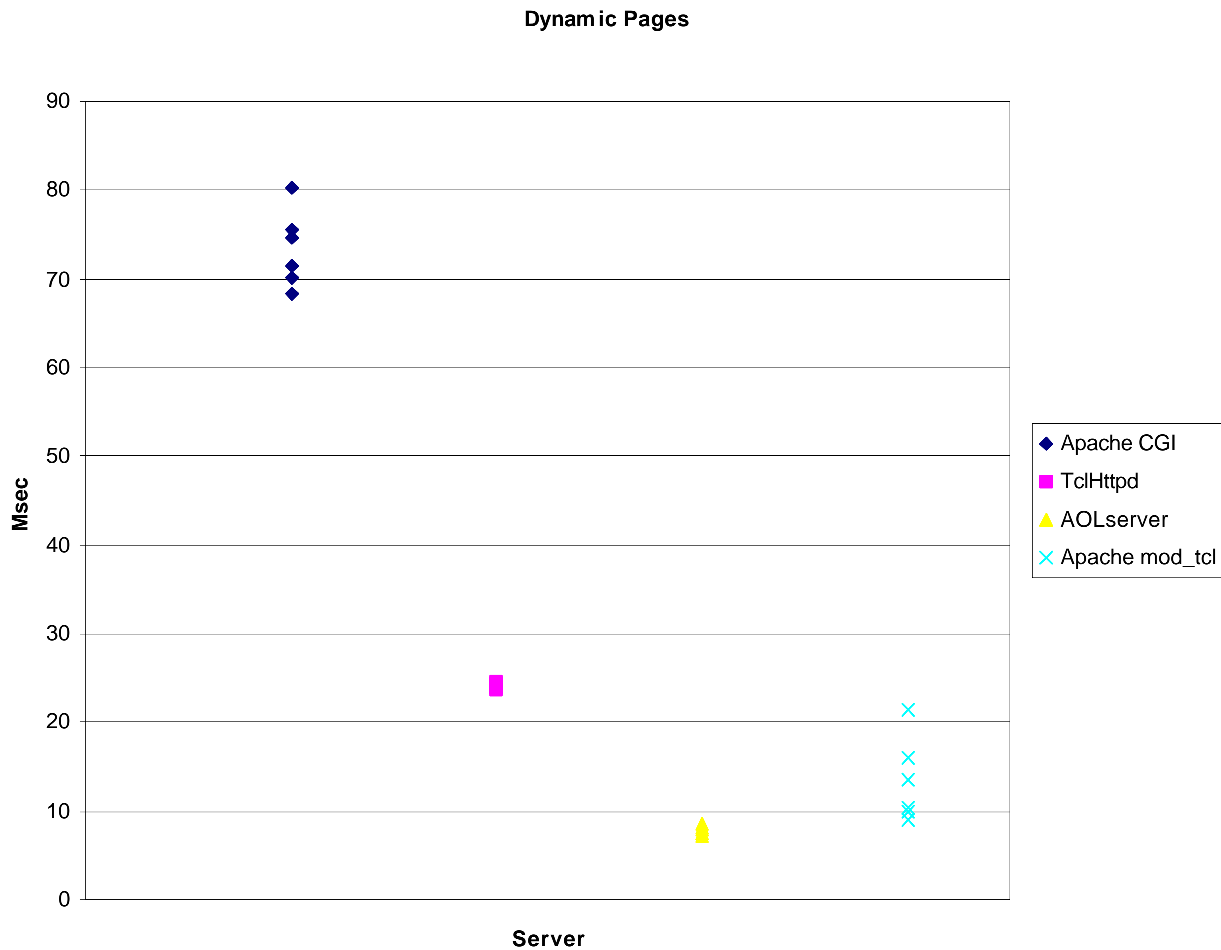
Example 13 The SiteLink procedure.

```
proc SiteLink {label} {
    global site
    array set map $site(pages)
    if {[info exist map($label)]} {
        return "<a href='$map($label)'>$label</a>"
    } else {
        return $label
    }
}
```

Appendix B: Performance Charts



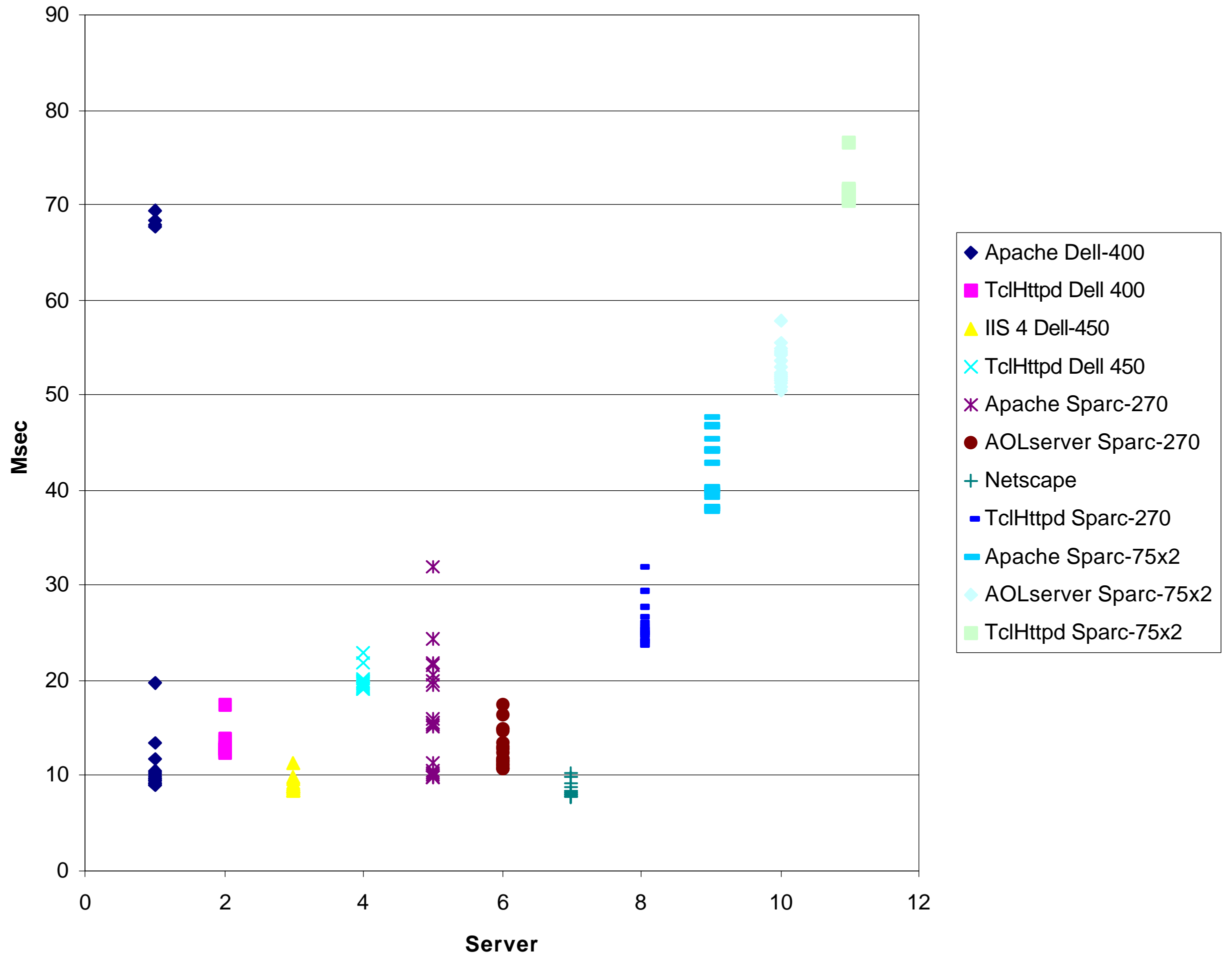
This chart shows two web servers, IIS 4 and TclHttpd, running on a 450 MHz Pentium-III under Windows NT. There is a plot for each run so you can see the variation across runs. Each run fetched the same URL repeatedly from the server using a single-threaded client. The runs performed either 100, 200, or 1000 repetitions, although these are not distinguished in the graph. These runs include some dynamic pages as well as static. The outlying points for TclHttpd are CGI scripts. There were no CGI tests done on IIS.



This chart shows the cost of creating dynamic pages on different platforms. The tests were repeated several times, and points are plotted for each run. All tests were run on the Sparc-270. CGI is slowest, of course, because Apache must fork a process. AOLserver is fastest, with the mod_tcl plugin for Apache close behind. TclHttpd is about three times faster than Apache CGI, and AOLserver is about 10 times faster than Apache CGI.

The dynamic page was very trivial, equivalent to:

puts “hello, world”



This figure compares all web servers when fetching a 32 Kbyte image file. Note that both the hardware and the web server are changing. For each hardware platform, TclHttpd and one or more other servers were compared. Overall TclHttpd runs from 1.5 to 3 times slower for moderate sized transfers.