Creating High Performance Web Applications using Tcl, Display Templates, XML, and Database Content

Alex Shah and Tony Darugar
Binary Evolution, Inc.

Creating High Performance Web Applications using Tcl, Display Templates, XML, and Database Content

Alex Shah
Technical Director
Binary Evolution, Inc.
ashah@binevolve.com

Tony Darugar
President
Binary Evolution, Inc.
tdarugar@binevolve.com

Abstract

We describe an online system that provides a framework for the rapid creation of high performance, database driven web sites based on content from XML files. The software that "glues" the content to the presentation is written in Tcl. The proposed architecture uses a pool of persistent Tcl engines to substantially improve performance and robustness as compared to traditional server-side programming techniques.

Introduction

Today's online applications demand more from web technology than C-based CGI programming can provide. A web site is a living document: the content, the presentation, and the software that drives that presentation need to change often. To meet the day to day requirements of a dynamic web site, a developer must use tools and technology that maximize flexibility and minimize development time.

We will describe an online system that provides a framework for the creation of high performance, database driven dynamic web sites. Simple HTML templates that can be manipulated in WYSIWYG editors will be used for display of the content. Content will be stored in XML documents or within a database, allowing publishers to update the site easily. The software that glues the database and XML content to the display templates will be written with one of the most stable, well supported scripting technologies available: Tcl. The Tcl language was chosen because it is non-proprietary, platform and operating system independent and is familiar to most web developers. Superior web performance will be achieved through the use of Tcl engines that maintain persistent database connections and communicate directly with the web server via the ISAPI or NSAPI protocols.

Performance Issues

Traditional server-side programming has relied heavily on the Common Gateway Interface (CGI) standard, first implemented in the NCSA server. The CGI protocol is fairly easy to understand and allows developers to write server-side applications in the language of their choice. Since applications run in separate processes, CGI based applications can crash without bringing down the entire web server. Nearly every web server, regardless of platform, has implemented the CGI standard.

CGI has some significant performance drawbacks. The operating system overhead associated with starting a new process for each request often utilizes more resources than the script itself. Under load, incoming requests are scattered by the scheduler: some requests are processed immediately; others may wait for hundreds of seconds. When content is stored in a database, the additional overhead from opening and closing the connection often results in huge lags, even for the simplest transaction.

In response to the performance problems associated with CGI, several vendors have developed high performance, proprietary APIs for their servers. The most notable are Netscape's NSAPI, Microsoft's ISAPI and Apache's server API. Applications compiled using one of these proprietary server APIs are faster than CGI programs. By running an application within the server process, the overhead of starting up and initializing a new process for each request is removed. Requests are handled in the order received and are not subject to rearrangement by the operating system's process scheduler. Database connections can be opened once, and reused.

Unfortunately, server APIs are missing many of CGI's benefits. Server APIs are difficult to learn and force the developer to use C or C++ to develop their application. Since the application is loaded into the server process, a bug in the application can bring down the entire server. Once written, an application cannot be easily ported to a new server or platform without rewriting the code.

Several solutions try to combine the ease of use of CGI and the performance of server APIs. One approach is the creation of an entirely new programming language and API for the sole purpose of server programming. Cold Fusion is an example of this approach. While easier to use than server APIs, Cold Fusion has several obvious disadvantages, such as the need to learn a new, proprietary language. Cold Fusion users do not have access to a large, established development community and must rely on a single vendor for all advances and extensions to the language.

Unlike Cold Fusion, NeoWebScript and others use established open languages such as Tcl that are already familiar to many developers and have a large user and development community [Lehen]. NeoWebScript takes the approach of adding a Tcl interpreter to the Apache server process and allows the web developer to embed Tcl code within their HTML pages. Since a new Tcl process is not created for each request, NeoWebScript's performance is substantially better than CGI. Despite NeoWebScript's ease of use and performance, a few drawbacks do exist. The approach of embedding the Tcl interpreter directly into the http process only works for non-multithreaded servers such as Apache. Existing CGI-Tcl scripts must be rewritten to conform to NeoWebScript's programming paradigm.

Another approach is to improve and extend the CGI protocol by removing the overhead of starting a new process for each new request. FastCGI implements this approach by introducing an accept loop, whereby the script accepts an incoming request, processes it, and goes back the accepting state [FastCGI]. Many of the benefits of CGI are retained by this approach, including being language independent and protecting the http server process by executing the scripts in a separate process. However, FastCGI's improved performance comes at the expense of greater effort on the part of the developer . Since the scripts are required to be persistent, they must be memory and resource leak free. The cleanup automatically done by the operating system when using CGI (by restarting the process for each request) must now be handled by the developer. FastCGI encourages the creation of large, monolithic programs, as opposed to small, modular scripts in order to have less processes to manage [FastCGI2].

Ideally, an improved CGI would increase performance as well as ease server-side programming, rather than complicate it. A successful CGI adaptation would take the benefits of the existing CGI protocol, combine them with the performance advantages of server APIs, be easy to use, and be based on open languages.

High Performance Architecture

The architecture we will use for our sample application uses Binary Evolution's *VelociGen*™ [VelociGen]. The *VelociGen for Tcl*™ (VET) combines the performance associated with server APIs with the benefits of CGI (see Figure 1).

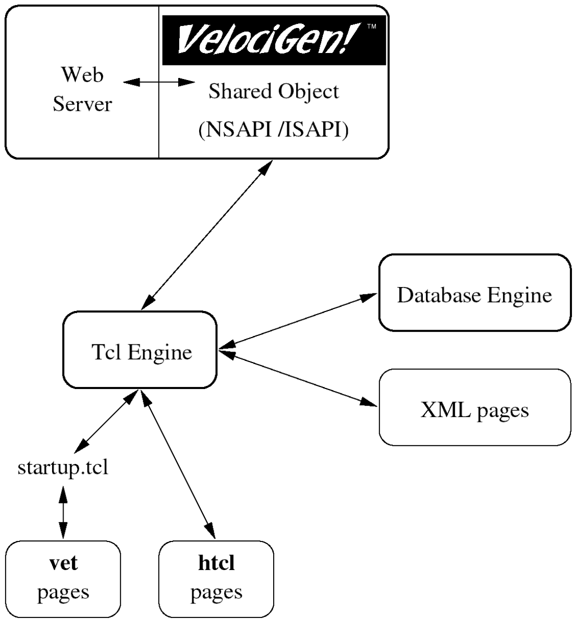

Figure 1. *VelociGen*™ Architecture

In VET's default configuration, requests for files that end with **.vet** or **.htcl** are processed by the *VelociGen*™ interface. VET can be configured to run scripts with different file extensions, or from a specific directory, for example **/cgi-bin**.

When a Tcl request is received by the http server, the *VelociGen*™ shared object searches for an available Tcl engine to handle the request. If no Tcl engines are available and the user defined maximum number of Tcl processes has not been exceeded, a new Tcl engine is spawned. The request is queued when all of the Tcl engines are busy processing prior requests.

Upon startup, the Tcl engine creates a master interpreter which remains in memory until the Tcl engine is exited. The master interpreter evaluates a file called **persistent.tcl** which contains user generated Tcl code that will be shared by all requests. Code for creating persistent socket or database connections should be added to the **persistent.tcl** file. Since code evaluated by the master interpreter is pre-compiled and cached, maximum Tcl performance can be obtained by moving procedures from your Tcl script into **persistent.tcl**.

After executing **persistent.tcl**, the Tcl engine waits for a request to arrive from the parent process. The *VelociGen*TM shared object will pass the filename for the script, CGI variables, and POST data to the Tcl engine. After this information is received, the Tcl engine will evaluate a file called **slave.tcl**. The purpose of **slave.tcl** is to create a new slave interpreter and alias any necessary procedures from the master interpreter. By modifying **slave.tcl** the developer can choose whether the Tcl script should be run using a safe or unrestricted interpreter.

If the script to be run is mostly HTML with some embedded Tcl code (**.vet**), the **startup.tcl** file will be evaluated next. The **startup.tcl** file contains Tcl code to simplify CGI programming. For example, it stores the query string in a Tcl associative array called QUERY and cookies into the COOKIE associative array. It also provides support for multipart form data, http file uploads and miscellaneous procedures for common web programming tasks. Depending on user preference, the **startup.tcl** file can be easily replaced with another CGI Tcl library such as the cgi.tcl package written by Don Libes [Libes]. After executing **startup.tcl**, the **.vet** file is processed as follows: HTML is passed onto the web server without further processing by the Tcl engine. Tcl code defined within user specified tags, for example: <tcl>...</tcl> is evaluated by the slave interpreter; standard output is passed onto the web server.

Scripts ending with **.htcl** run in VET's CGI compatible mode. In this mode, **startup.tcl** is not evaluated. CGI variables are copied to Tcl's *env* array. POST data is placed on *stdin*; data written to *stdout* will be communicated to the web server and eventually reach the client browser.

In the event that a Tcl script crashes the interpreter, only the current request is affected. Other Tcl engines, and the server http process continue to run without incident. The *VelociGen*TM shared object can detect when a Tcl engine has crashed, display an appropriate error message on the client browser, and restart the engine when the next Tcl request is received. *VelociGen*TM can also be configured to forcefully terminate a script after a user defined number of seconds.

After script termination, the slave interpreter is destroyed, freeing variables, file handles, and other resources used in the processing of the **.vet** or **.htcl** script. The master interpreter is not cleared out: Tcl code, variables, and socket or database connections defined in **persistent.tcl** remain cached and pre-compiled in the Tcl engine. The Tcl engine now waits for another Tcl request to arrive. The connection between the http server and Tcl engine is never closed.

Comparison of *VelociGen*TM to Other Server-Side Programming Solutions

Like CGI and FastCGI, applications written with VET run in separate, isolated processes. Scripts submitted to the Tcl engine for processing can crash, block on IO, or go into infinite loops, without affecting the web server. Only a small amount of well tested, thread-safe code for managing and communicating with the Tcl engines needs to be loaded into the web server.

Unlike CGI, the Tcl process or engine remains persistent and does not exit after each request, thus eliminating substantial operating system overhead. By keeping the Tcl engine in memory, persistent connections can be made to the database, further increasing performance. VET requests are less susceptible to rearrangement by the operating system process scheduler. Since requests are handled in the order that they are received, response time stays consistent, even under heavy load.

By separating the web server and Tcl into separate processes, the potential exists for running the Tcl engines on remote machines. The architecture provides scalability: request handling and server performance can be easily increased by distributing Tcl processing over several machines.

Unlike FastCGI, VET is capable of handling existing CGI-Tcl scripts without modification. Developers can continue to code small scripts without performance loss. Freeing of system resources is handled by VET rather than the developer. Several requests to the same Tcl script are processed simultaneously, rather than serialized in an accept loop.

Rather than taking Cold Fusion's approach of creating a entirely new language for server-side programming, VET shares the approach taken by NeoWebScript. The VET interface is based on the well established and proven Tcl language. VET pages can also parse and process Tcl embedded within HTML pages. The embedded Tcl code is placed within user defined tags, for example: <tcl>,

</tcl>, <%, %>, [[,]]. Unlike NeoWebScript, VET is backward compatible with CGI scripts written in Tcl. By separating the Tcl interpreter from the http server process, thread-safe *VelociGen*™ technology supports multithreaded servers such as Netscape Enterprise, Microsoft IIS, and O'Reilly Web Site Pro, as well as Apache.

The VET interface is not dependent on a particular platform or server architecture. Any web server that provides a high performance interface compatible with Netscape's NSAPI, Microsoft's ISAPI, or Apache's server API will work with VET. Since Tcl has been designed to run on nearly any platform, the VET interface allows developers to write an application once and then deploy on nearly any operating system, platform, or web server.

Why use Tcl?

Scripting languages such as Tcl represent a very different style of programming than system programming languages such as C or Java™. Scripting languages are designed to achieve a higher level of programming and more rapid application development than system programming languages. Scripting languages also provide a simplified environment for performing online tasks. Content can be pulled from databases or XML and manipulated using regular expressions or other built-in string functions without having to build data structures and algorithms from scratch.

The Tcl language was chosen as a base for our system because it has only a few fundamental constructs and relatively little syntax, which makes it easy to learn. The Tcl syntax is meant to be simple and flexible. Developers can easily add commands and functionality to Tcl using C or C++ code when needed. As John Ousterhout, the creator of Tcl puts it, Tcl is designed for "gluing" applications together.

Support and extensions for Tcl are widely available, making it a very attractive language for online development. In addition to the text manipulation functions that come with Tcl, a powerful regular expression library can be installed into the interpreter. Support for nearly any database is available for Tcl. Steve Ball has recently made an XML extension available as well [Ball]. Extensions also exist for creating and manipulating graphs and GIF images.

Tcl 8.0 also comes with a byte compiler. VET's ability to pre-compile and cache Tcl code is ideal in a web environment where scripts are run over and over. By using pre-compiled Tcl, the overhead of interpreting Tcl source code is removed. Since most Tcl routines map to compiled C, pre-compiled Tcl can execute at speeds that come very close to the performance of server APIs.

With respect to online publishing, Tcl scripts are especially powerful. A developer can quickly and easily update a script and see the changes on the next server request, without having to recompile the entire application. Tcl allows editors, graphics artists and developers to update the content, the presentation of that content, and the logic driving that presentation on a daily basis, while still maintaining high performance and flexibility.

Online Recipe Application based on XML and Tcl

A vast amount of documentation and information is available on the Internet, with more being added at an incredible pace every day. Most of this information is stored in HTML, the Web's ubiquitous formatting language. This has been a very successful model both because of the ease of authoring HTML documents and because of the relative ease of displaying them.

While HTML is effective as a formatting and display language, it does not allow the author of the document to store structural or fielded data within the document. In fact, the only information maintained within the document is how to display it. More and more, the need for structured, fielded documents is becoming obvious.

To demonstrate this, imagine all of the different cooking recipes available on the Internet. Each has its own look, order for placement of sections and wording for the headings. If you wanted to collect these disparate recipes into a single collection with a uniform look, you would have to expend a large amount of energy figuring out how each recipe is laid out and converting it to your own look. Or, imagine if you wanted to place the ingredient list of each recipe into a central database, so that you could search for recipes that used a particular combination of ingredients. Since each recipe lists its ingredients in a different way, a large amount of effort would be needed. These data management issues also apply to collections of movie reviews and stock portfolio information.

XML can be used to remedy this situation. An XML tagged document resembles HTML, although the meaning of those tags are defined by the author of the document [XMLRec]. Unlike HTML, XML tags add meta information to the document rather than determine how the document will be displayed. By separating the display characteristics of the document from the content, XML allows one to change the presentation without having to modify content spread across hundreds or thousands of documents. Also, since the tags are structured and can be nested, meta-data can be stored within the document itself. For example, a recipe document could have the tag <ingredient>carrot</ingredient> to indicate that carrots are used in that recipe, making it very obvious what each ingredient is. The following listing shows a partial recipe for carrot cake in XML:

```
<recipe course="dessert" calories="325">
  <name>Carrot Cake</name>
  <description>
    A favorite classic. Stays moist and freezes well.
  </description>
  <note status="credit">Betty Crocker’s Cookbook</note>
  <ingredient-list yields="16">
    <ingredient quantity="1 1/2 cups">sugar</ingredient>
   ...
    <ingredient quantity="3 cups">shredded carrots</ingredient>
    <ingredient quantity="1 cup">coarsely chopped nuts</ingredient>
  </ingredient-list>
  <preparation preptime="about 1 1/2 hours">
    <step>Heat oven to 350 degrees</step>
    <step>Grease and flour rectangular pan, 13 x 9 x 2 inches</step>
    ...
   <step>Frost with cream cheese frosting if desired</step>
  </preparation>
</recipe>
```

Listing 1. Carrot Cake Recipe in XML

Using XML important information about the structure of a document can be placed within the document itself.

The following example demonstrates how *VelociGen for Tcl*™ can be used in conjunction with an XML parser to display recipes stored in XML format.

Since XML tags are guaranteed to be balanced, valid XML documents can be parsed and placed into a tree data structure. For our example, we use Steve Ball’s XML parser for Tcl [Ball] which transforms the XML document into a tree structure using Tcl lists. Each node in the tree is made up of four elements:

node: type tag attribute content

The *type* element is set to one of two values: ‘parse:elem’ or ‘parse:text’. When *type* is ‘parse:elem’, *tag* is set to the name of the XML tag, *attribute* is set to a list of attributes placed within the XML tag, and *content* is set to one or more *nodes* found between the start and end tags. When *type* is ‘parse:text’, *tag* is set to the text found between the start and end tags, *attribute* and *content* are set to empty Tcl lists: ‘{}’.

A simple approach to giving XML presentation is to simply map the XML tags to appropriate HTML tags. Listing 2 shows this approach. The *tag_map* defines these simple mappings - one for the start of each tag, and one for the end. As the tree is recursively traversed, using the *process* function, each tag is handled as follows: the HTML code defined by the *tag_map* is displayed, followed by the content of the tag, followed by the end HTML code defined by the *tag_map*.

```
# ----------------------------------------
# The tag map: Each item consists of the tag name, the HTML to output at the beginning
# of processing the tag, and the HTML to output at the end of processing the tag.
set tag_map {
{RECIPE ""      "" }
{DESCRIPTION    {<blockquote>}  {</blockquote>} }
{NAME   {<center><h2>}  {</h2></center>} }
{NOTE   {<!--}  {-->} }
{INGREDIENT-LIST        {<p><h3>Ingredients</h3><ul>}   {</ul>} }
{INGREDIENT     {<li>} "" }
{PREPARATION    {<p><h3>Preparation</h3><ol>}   {</ol>} }
{STEP   {<li>}  "" }
}
```

Listing 2. XML to HTML conversion tag_map

The XML tags are processed using a simple recursive traversal of the parse tree:

```
# Output start HTML, XML tag contents, and
# end HTML as defined in tag_map for the
# XML tag “$tag”

process_tag start $tag
process $content
process_tag end $tag
```

Using this approach XML documents can be converted to HTML quickly and easily. However, Listing 2 does not allow for the handling of tag attributes - they are simply ignored. To deal with the more complicated aspects of XML we need a more powerful mapping.

The more powerful approach involves defining Tcl procedures to handle the XML tags. Rather than using text substitution, each XML tag becomes a Tcl procedure, taking the *attributes* and *content* as arguments. This approach provides the full power of Tcl for handling the XML tags.

Listing 3 shows this approach. *tag_map* is now a namespace instead of a Tcl list. Each procedure in this namespace corresponds to an XML tag and defines what is to be done with that tag.

```
namespace eval tag_map {

proc DESCRIPTION {attr content} {
    puts "<blockquote>"
    process $content
    puts "</blockquote>"
}

proc NOTE {attr content} {
    set value     [lindex $attr 1]
    switch $value {
        credit {
           puts "<br>From: <i><b>" ; process $content ; puts "</b></i><br>"
        }
        default { }
    }
}

proc INGREDIENT-LIST {attr content} {
    puts "<table border=0 bgcolor=yellow width=100%><tr><td>"
    puts "<h3>Ingredients:</h3><ul>"
    process $content
    puts "</ul></td></tr></table>"
}

proc INGREDIENT {attr content} {
    puts "<li><b>"
    process $content
    puts "</b><i>[lindex $attr 1]</i><br>"
}

...
# End tag_map namespace
}
```

Listing 3. XML to HTML conversion namespace.

A typical procedure inspects and processes the tag attributes, outputs some HTML code, calls *process* to recursively handle the contents of the tag, and outputs the ending HTML code. The call to *process* is needed in order to reach the current tag’s child *nodes*; omitting it allows you to suppress the display of the tag’s contents, including its subtrees. For example, by removing the *process* call from the *INGREDIENT-LIST* procedure, the listing of ingredients can be suppressed.

Using this system the documents can be written and maintained in XML, providing all of the benefits discussed above, and displayed in HTML, allowing the system to be deployed today, using existing web technologies. Further benefits of authoring and storing the documents can be seen in the following section, where the XML document is very easily converted to SQL and stored in a database.

Online Recipe Application based on Databases and Tcl

XML provides structure to a document, but does not provide a mechanism to search and query the content. For such purposes, it is often useful to store content within a database. Our next example will demonstrate how recipe information can be stored in a database and then retrieved and displayed in HTML using Tcl.

Listing 4 shows the schema for storing the recipe example in a SQL database. The recipe is stored in three tables: *recipe*, which stores the meta level information about the recipe; *ingredients*, which contains the list of ingredients; and *prep*, which contains the preparation steps.

Listing 3, which defines the mapping from XML to HTML, can be modified to map from XML to SQL statements, allowing us to store our document in a SQL database. Listing 5 shows the necessary modifications. Notes are ignored simply by not processing their contents.

```
create table recipe (
        id              int,
        course          char(1),
        calories        int,
        name            char(30),
        description     char(254)
);

create table ingredients (
        rec_id          int,
        quantity        char(20),
        name            char(20)
);

create table prep (
        rec_id          int,
        step_num        int,
        description     char(254)
);
```

Listing 4. Database Schema

```
namespace eval tag_map {

proc RECIPE {attr content} {
    puts
       "insert into recipe (id) values (1);"
    process $content
}

proc NAME {attr content} {
    puts "update recipe set name='[process $content]';"
}

proc DESCRIPTION {attr content} {
    puts "update recipe set description='[process $content]';"
}

proc NOTE {attr content} {
    # Ignored
    return
}

proc INGREDIENT-LIST {attr content} {
    process $content
}

proc INGREDIENT {attr content} {
    puts "insert into ingredients values (1, '[lindex $attr 1]', '[process $content]');"
}
...
# End tag_map namespace
}
```

Listing 5. XML to SQL tagmap namespace

To create SQL output instead of HTML, only the mappings need to be redefined. This demonstrates the flexibility and power of XML. By adding structure to a document without including presentation information, XML allows easy conversion of the document to other formats, including HTML and SQL.

Listing 6 shows a code segment for displaying the recipe, drawing data from the database. The necessary data is retrieved from the database using SQL statements and made into HTML via an HTML template. The HTML template can be manipulated using an HTML editor, allowing site designers to modify the look of the page without having to understand the Tcl code. The following listing uses the MySQL database with Tcl-GDBI [Darugar] as the Tcl interface.

```
...
<tcl>
set select "select * from recipe where id=1"
set nrows [sql query $conn $select]
set row     [sql fetchrow $conn]

# get the fields out of the returned row.
set idx 0
foreach field {id course calories name description} {
        set $field [lindex $row $idx]
        incr idx
}
</tcl>

<center><h2><tcl>puts $name</tcl></h2></center>
<blockquote>
<tcl>puts $description</tcl>
</blockquote>

<!-- Ingredients ---------------------------------------- -->
<table border=0 bgcolor=yellow width=100%><tr><td>
<h3>Ingredients:</h3>
<ul>
<tcl>
set select "select * from ingredients where rec_id=1"
set nrows [sql query $conn $select]

while {[set row [sql fetchrow $conn]] != ""} {
        set quantity [lindex $row 1]
        set name     [lindex $row 2]
</tcl>

<li><b><tcl>puts $name</tcl></b><i><tcl>puts $quantity</tcl></i><br>

<tcl>
# end of while loop for ingredients
}
</tcl>

</ul></td></tr></table>
...
```

Listing 6. Displaying content from a database.

In real life situations drawing content from a database or using XML/SGML to serve popular sites has often been avoided because of the performance implications: each request requires accessing a database and formatting the content into HTML, or parsing and translating XML into HTML. Traditional server programming systems such as CGI make this overhead prohibitive, forcing the developer to use static HTML instead.

Performance Results

VET shrinks the performance gap between static HTML and dynamically generated HTML. Figure 2 shows response times for displaying recipes at various user loads using CGI-Tcl scripts, VET driven Tcl scripts, and static HTML. The Tcl scripts used for the performance comparison retrieve and display recipe data from an Oracle database (see Listing 6). For testing purposes, slight modifications were made to Listing 6 in order to use the exact same script under CGI and VET. The script was converted into CGI-Tcl by placing **puts** before each line of HTML and removing the <tcl>, </tcl> tags. We also replaced our MySQL interface [Darugar] with the more widely used Oratcl library [Poindexter]. Additional performance was attained by maintaining persistent database connections in VET. Tests were done on a Sun IPX running Solaris 2.5.1, Netscape Fasttrack Server 2.01, Oracle Workgroup Server 7.3.2.2.0 and *VelociGen for Tcl*^{TM} v1.0c.

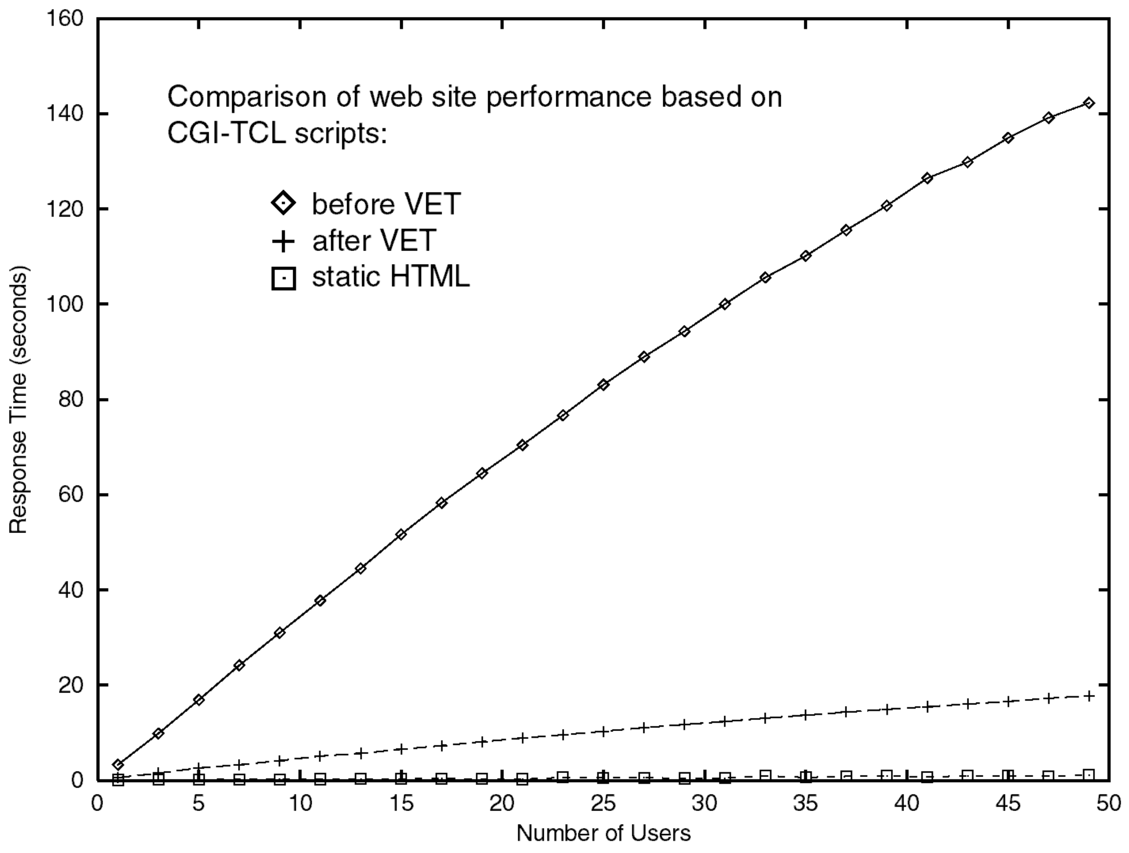

Figure 2. *VelociGen for Tcl*™ vs. CGI Performance

As seen in Figure 2, VET's performance is much better than an equivalent system written with CGI. As the load increases to 50 simultaneous requests, the average response time for the CGI based system jumps to over 140 seconds, which is unacceptable for most web applications. The VET based system handles the load well, taking about 10-20 seconds under stress. Requesting the recipe as a static HTML page does outperform the VET solution slightly, a small price to pay for the ability to search, query your XML data and manipulate the presentation.

Conclusion

Tcl is very well suited for online applications. The language provides a simple syntax and built-in functions for handling many online tasks. It is multiplatform, well tested and supported, and is non-proprietary. It also has the additional benefit of being supported by an entire community of developers rather than a single vendor. This allows users to take advantage of the latest technologies as they appear, well before they can be implemented into proprietary systems. In our simple recipe example, we have demonstrated Tcl's ability to easily load and display content from both databases and XML.

When combined with Binary Evolution's *VelociGen*™, the use of Tcl for the creation of high performance web sites becomes possible. The proposed system cleanly separates content from presentation, allowing easy manipulation of both the site's look and substance. Rather than waiting 2 to 3 years for widespread XML support in the browser, this solution allows developers to use XML technology today. By tightly integrating database technology, Tcl and HTML templates, our proposed system provides a flexible framework for meeting the day to day needs of today's dynamic web sites.

References

[Ball]	Steve Ball, Steve.Ball@tcltk.anu.edu.au, "TclXML", URL: http://tcltk.anu.edu.au/XML/
[Darugar]	Parand T. Darugar, tdarugar@binevolve.com, "Tcl-GDBI : Tcl MySQL interface", URL: http://www.binevolve.com/~tdarugar/tcl-sql/
[FastCGI]	Open Market Inc., "FastCGI", URL: http://www.fastcgi.com/
[FastCGI2]	Open Market Inc., "FastCGI: A High-Performance Web Server Interface", April 1996, URL: http://www.fastcgi.com/kit/doc/fastcgi-whitepaper/fastcgi.htm
[Lehen]	Karl Lehenbauer, karl@neosoft.com, "NeoWebScript", URL: http://www.neosoft.com/neowebscript/
[Libes]	Don Libes, libes@nist.gov, "cgi.tcl", URL: http://expect.nist.gov/cgi.tcl/
[Libes96]	Don Libes, libes@nist.gov, "Writing CGI scripts in Tcl", Fourth Annual USENIX Tcl/Tk Workshop, 1996, URL: http://www.usenix.org/publications/library/proceedings/tcl96/libes.html, URL: http://www.mel.nist.gov/msidlibrary/summary/9622.html
[Poindexter]	Tom Poindexter, "Oratcl" and "Sybtcl", URL: http://www.nyx.net/~tpoindex/tcl.html#Oratcl URL: http://www.nyx.net/~tpoindex/tcl.html#Sybtcl
[VelociGen]	Binary Evolution, Inc., info@binevolve.com, "VelociGen: Fast, Efficient, and Simple Server Programming with Perl and Tcl", URL: http://www.binevolve.com
[XMLRec]	Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0" W3C Recommendation 10-February-1998, URL: http://www.xml.com/axml/target.html