;login:
THE MAGAZINE OF USENIX & SAGE
February 2002 • Volume 27 • Number 1

inside:

PROGRAMMING
THE TCLSH SPOT
by Clif Flynt

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

the tclsh spot

by Clif Flynt

Clif Flynt is president of Noumena Corp., which offers training and consulting services for Tcl/Tk and Internet applications. He is the author of *Tcl/Tk for Real Programmers* and the *TclTutor* instruction package. He has been programming computers since 1970 and a Tcl advocate since 1994.

clif@cflynt.com

The previous Tclsh Spot article described a simple Tcl script to reformat the output from tcpdump into a more human-readable format. Once you can read the traffic between two systems, many things become easier. In this article I'll discuss one thing that can be done with the processed tcpdump output.

When I wrote the tcpdump reformatting program, I needed to view the traffic between a couple of systems that weren't quite talking. A month or two later I needed to perform regression testing on a set of CGI scripts I was reworking. Being able to read the interactions helped me see what was going on, but didn't quite fulfill my desire for a fully automated regression test.

I wanted to browse the pages using a normal HTML browser, fill in forms, etc., and use the reformatted tcpdump transcript of the session to automatically generate a Tcl script that would duplicate my actions. The next goal was to confirm that the results were what I expected and report the elapsed time, so I could be certain that my modifications actually improved performance. There is probably a package out there that would do what I wanted, but I figured that in the time it would take me to track it down, install, learn and customize it, I could write what I needed from scratch.

Between Tcl's string manipulation tools and HTTP support, this application is pretty simple. It consists of a main module with some utility procedures and a GUI to report progress and results; the tests are simply Tcl scripts loaded with the Tcl source command.

The Tcl http package was described in a few Tclsh Spot articles about a year ago. Briefly, the http package includes functions that enable a Tcl script to interact with a Web server. These functions will send GET, POST, or HEAD requests, as a single execution stream, or with multiple operations in process simultaneously. You can configure the calls to work directly connected to the Net, or through a firewall.

Tcl implements http support in the http:: namespace. Like a Java or C++ class, the Tcl namespace command hides implementation details from the application developer.

This application uses the following two http commands:

http::geturl *url* downloads data from a URL and returns a token to use to access this data.

http::data *token* returns the data associated with a token.

The automatic test building script converts HTTP GET commands that resemble this:

```
GET /cgi_bin/search.tcl?search=books&name=flynt HTTP/1.0
```

to a geturl command like this:

```
set token10 [http::geturl \
    http://$Test(httpServer)/cgi_bin/search.tcl?search=books&name=flynt ]
```

These lines instruct the server to invoke the cgi_bin/search.tcl script, passing search=books&name=flynt as the query string. The CGI script is responsible for unpacking the string into keyword and value pairs.

The HTTP protocol supports two data retrieval operations, GET and POST. The GET command is the simplest and most common command. It can simply request a static HTML document, or can be used to pass additional data to be used by a CGI-type

script by adding a set of keyword/value pairs to the end of the script URL. The data being passed is separated from the main URL with a question mark, and individual keyword value pairs are represented as keyword=value and are separated by ampersands.

There is a limit to the number of characters you can transmit on a single line, and some CGI submissions can exceed this length (for instance, if you are filing a software bug report).

The POST command solves this problem by treating the keyword/value pairs as the body of an HTML message. A dump of the search query done as a POST command would resemble:

```
POST /cgi_bin/search.tcl HTTP/1.0
Content-type: application/x-www-form-urlencoded
Content-length: 23

search=books&name=flynt
```

The first line is the POST command and the next two lines are the HTTP header.

HTTP messages are formatted like email messages. There is a header in which each line consists of a keyword followed by a colon, followed by data, terminated with a new line. The header is terminated with a blank line, and the body of the message follows that.

By default, the http::geturl command will generate a GET command. If your script uses the -query option, a POST is generated. This script would generate the POST command.

```
set token10 [http::geturl \
      http://$Test(httpServer)/cgi_bin/search.tcl \
      -query "search=books&name=flynt"
```

Since the HTTP protocol is stateless, most Web servers use a cookie to link a user to some state information that is being maintained on the server (e.g., items in a shopping cart). The cookie value (and other information) is passed in the HTTP header block.

The http package generates a simple header to declare that this HTTP message was generated by the Tcl http package, etc. If you want to pass other parameters in the HTTP header, you can do this with the -headers option.

The -headers option accepts a list of keyword and value pairs that it will reformat as a MIME-style HTTP header.

For example, this code would add the line

```
Cookie: chocolatechip
```

to the header:

```
http::geturl http://$Test(httpServer)/cgi_bin/search.tcl\
      -query search=books&name=flynt \
      -headers {Cookie chocolatechip}
```

By default, the http::geturl procedure will block until the URL has been retrieved. For many Web robots this is a good technique, but for an interactive test application, you can't freeze the test platform GUI while the Web server is busy thinking.

The -command option will register a Tcl script to be evaluated when the URL has been retrieved. When the callback script is evaluated, a token to identify the data will be appended to the callback script. You can use this token as an argument to a procedure to retrieve the HTML page to process.

This code would request an HTML page and process the page within the checkPage procedure when it becomes available:

```
proc checkPage {identifier token} {
    global correctPages
    set data [http::data $token]
    set dataLength [string length $data]
    set correctLength [string length $correctPages($identifier)]
    if {([string first $correctPages($identifier) $data] == 0) &&
        ($dataLength == $correctLength)} {
        # Report OK
    } else {
        # Report mismatch
    }

}

http::geturl -command "checkPage searchPage" \
    http://$Test(httpServer)/cgi_bin/search.tcl?search=books&name=flynt
```

We can also use the -command option to start several simultaneous searches running on a server with code like:

```
set tclAuthors {ousterhout welch flynt harris mclennan smith nelson}
foreach author $tclAuthors {
    http::geturl -command "checkPage $author" \
        http://$Test(httpServer)/cgi_bin/search.tcl?search=books&name=$author
}
```

As each search completes, Tcl will invoke the checkPage procedure with the author's name as an identifier, and a token to use to access the page retrieved from the Web server.

For this application, I didn't want multiple geturl commands active at once. I wanted just one HTTP interaction active at a time, but I still needed to allow the GUI to update (and a cancel button to interrupt the test). This means I needed to pause the script execution after issuing each http::geturl request and wait until the page was retrieved.

The vwait command causes the interpreter to enter the event loop and process events until the registered variable is assigned a new value.

Syntax: vwait *varName*

varName The variable name to watch. The script following the vwait command will be evaluated after the variable's value is modified.

A script like this will initiate an http::geturl interaction, return control to the Tcl event loop, and wait for the HTML page to be retrieved before going on to the next command.

```
proc checkPage {pageFile token} {
    global doneFlag
```

```
    set newPage [http::data $token]
    # Compare newPage to pageFile

    set doneFlag 1
}

http::geturl $site -command {checkPage fileName}
set doneFlag 0
vwait doneFlag
```

The next problem is validating the page that was just retrieved.

One simple solution to this problem is to create a good set of pages, and compare the new page to a known good page. That's the reason for the pageFile argument to the checkPage procedure. It's the name of a file to compare to the new page.

The Tcl gets command will read a single line of data from a channel. For this application we want to read an entire file, so it's simpler to use the read command which will read the file in a single action.

Syntax: read *channelID ?numBytes?*
The *numBytes* parameter is optional. If it's provided, the Tcl interpreter will read that many bytes (or up to the EOF). If there is no *numBytes*, the read command will read data until it reaches the EOF.

The code to compare pages looks like this:

```
set newPage [http::data $token]

set if [open $pageFile]
set oldPage [read $if]

if {([string first $oldPage $newPage] != 0) ||
    ([string length $oldPage] != [string length $oldPage])} {
    # fail
} else {
# success
}
```

My first thought for comparing the two pages was to use the string match command. However, the string match command will match a glob-style pattern to a string, rather than comparing two exact strings. The string match command will work for most simple tests, but will break when the real data includes metacharacters like *, ? or square braces.

The string first and string last commands compare characters with no wildcards, so these commands can be used to compare strings that may have magic characters in them. If the strings are identical, the first character where they match will be the first character of the string: position 0, since Tcl uses 0-based strings and arrays.

The string length command returns the number of characters in a string. This is used to compare the lengths of the two pages, to be certain that there is no trailing data to worry about.

The last item on my want list was for my test harness to report how long each HTTP interaction took. The Tcl clock command will report or format a time in seconds. The clock command can also report time in the smallest unit that the platform will support

(usually milliseconds), but won't reformat that value directly into a human-readable string.

Syntax: clock *subcommand args*

subcommand The clock command supports several subcommands including:

seconds Returns current time and date in seconds since a system defined epoch.

clicks Returns current time and date as a system dependant integer, usually milliseconds since the last clock rollover.

format Converts a time in seconds to a human-readable format. There are many formatting commands to fine-tune the output.

For this application, seconds were adequate, and the elapsed time can be calculated with a simple expr command like:

```
set startTime [clock seconds]
# Do stuff
set elapsedSeconds [expr [clock seconds] - $startTime]
```

Since the checkPage procedure can compare pages and calculate elapsed time, we might as well make the return value a human-readable string for a final report.

The format command is ideal for generating reports with columns of data.

Syntax: format *formatString value1 ?value2?...*

The formatString resembles a C language printf string, with %d to substitute an integer, or %s to substitute a string at a location, etc.

Like the C language printf command, the format command format string can have a number between the percent symbol and the format identifier to define how many characters wide the field should be, and whether to make the string flush to the left or right margin.

So, a final version of checkPage with page checking, time calculation, and formatted output resembles:

```
proc checkPage {pageFile startTime identifier token} {
    global resultString
    set elapsedSeconds [expr [clock seconds] - $startTime]

    set newPage [http::data $token]

    set if [open $pageFile]
    set oldPage [read $if]

    if {([string first $oldPage $newPage] != 0) ||
        ([string length $oldPage] != [string length $oldPage])} {
        set result "error"
    } else {
        set result "ok"
    }
    set formatString {%-10s %-30s %8s seconds}
    set resultString [format $formatString $result $identifier $elapsedTime]
}
```

When this is invoked with a command like:

```
http::geturl $site -query $query -headers $headerList\
      -command "checkPage page1 [clock seconds] $name"
vwait resultString
```

it will assign resultString a string like this:

```
fail      /booksearch.tcl?author=flynt        23 seconds
```

which can be displayed in a text widget.

A simple GUI for this harness would be a label to show which CGI script is being tested, an exit button, and a simple text widget to display the results.

```
label .l -textvar statusLabelVar
grid .l -row 0 -column 0

button .b  -text exit -command exit
grid .b -row 0 -column 1

text .t -height 23 -width 80 -font {courier 12}
grid .t -row 1 -column 0 -columnspan 2
```

By default, a text widget is created using a proportional font. This makes a nice, easy-to-read display, but is difficult to use for columnar output, since different characters have different widths. The courier font is a fixed-width font that's supported on all platforms.

The test scripts are generated by stepping through the readable tcpdump text looking for GET and POST commands sent from the browser to the Web server. When one of these commands is found, the script will extract the URL, header information, and message body and generate a Tcl script to duplicate the browser action and display the report lines in the text widget.

Each HTTP interaction in the test script resembles this:

```
http::geturl testsite.com:/booksearch.tcl?author=flynt \\
      -command "checkPage  page50 [clock seconds] 50 booksearch.tcl?author=flynt \\"
-headers {Cookie {cookieVal=chocolateChip; mode=Frames}}

set statusLabelVar /booksearch.tcl?author=flynt

set resultString 0
vwait resultString
.t insert end "$resultString\n"
```

The script that converts the HTTP conversation to a Tcl script needs to substitute some values when the test script is generated (like the search parameters), while other substitutions are done when the test script is evaluated. For instance, if the square bracket substitution around the clock seconds command were done during test generation, the test script would measure the time from when the script was generated until the new page was returned, instead of the time from when the HTTP request was submitted until the new page was returned.

Controlling when substitutions will occur when generating new Tcl commands within a Tcl script can get tricky, especially if you want the script you are generating to perform some Tcl substitutions when you are generating the script, and others at runtime.

You can handle this by escaping the Tcl control symbols ($, [and]) with backslashes, but that gets confusing and hard to read very quickly.

For example, to generate this code:

```
set tmpVariable [expr $variable1 + 2]
puts $tmpVariable
set tmpVariable [expr $variable2 + 3]
puts $tmpVariable
```

the commands using backslash escapes and brackets might resemble this:

```
for {set i 1} {$i < 2} {incr i} {
    puts "set tmpVariable \[expr \$variable$i + [expr $i + 1]\]"
    puts {puts $tmpVariable}
}
```

A better solution is to generate the new lines using the Tcl format command.

When the *formatString* is placed in curly braces, the normal Tcl substitutions phase is disabled. This allows us to use otherwise special characters inside the *formatString* and merge in substituted values with %s and %d. Separating the characters you want to output as literals from those you wish to be substituted makes the code easier to maintain.

This script would generate the code above using format commands instead of backslash escapes:

```
for {set i 1} {$i < 2} {incr i} {
    puts [format {set tmpVariable [expr $variable%s + %d]} $i [expr $i+ 1]]
    puts {puts $tmpVariable}
}
```

The *formatString* can be hardcoded, as shown above, or saved in a Tcl variable like this:

```
set id 10
...
set fmt {set %s [http::geturl }
puts [format $fmt token$id]\\
```

There's no advantage to using variables instead of hardcoded strings in the format command, except that using the variable makes the code fit better on these pages.

This procedure will generate a test script from values extracted from the tcpdump output. I only used the format command for output that needed runtime substitutions.

```
set State(id) 0

proc writeCmd {} {
    global State

        set lastSlash [string last / $State(url)]
        set identifier [string range $State(url) $lastSlash end]

        puts "http::geturl $State(site)/$State(url) \\"
        if {[string match $State(type) post]} {
            puts "    -query [string trim $State(body)] \\"
        }
        set fmt {    -command "checkPage  %s [clock seconds] %s %s \\"}
        puts [format $fmt page$State(id) $State(id) $identifier]
```

```
        puts "    -headers {$State(headers)}"
        puts ""

        set lastSlash [string last / $State(url)]
        puts "set statusLabelVar $identifier"
        puts ""

        puts "set resultString 0"
        puts "vwait resultString"
        set fmt {.t insert end "$%s\n"}
        puts [format $fmt resultString]
        puts "\n"

        foreach index {url headers body} {
            catch {unset State($index)}
        }
        incr State(id)
}
```

Extracting the appropriate values from the tcpdump output can be done with a simple state engine. Anyone interested in that part of the code can find it at *http://www.noucorp.com*.