;login:

THE MAGAZINE OF USENIX & SAGE

July 2001 • Volume 26 • Number 4

inside:

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 last several Tclsh Spot articles have shown ways to write simple client-server applications with Tcl. These pairs all used pure ASCII text to transmit data, in the tradition of SMTP, NNTP, POP, etc. This article will expand on the client-server articles and the previous article about the binary command, to develop a program to examine the behavior of telnet servers.

It's almost embarrassing to admit how many years I've used telnet without ever wondering what went on under the hood. The telnet protocol not only handles the printable ASCII text that we see on our screen, it also sends a number of unseen commands to negotiate how the session will be managed.

Peter Burden published a nice overview of the telnet protocol at *http://www.scit.wlv.ac.uk/jphb/comms/telnet.html*.

The telnet configuration commands all start with a binary 0xff byte, referred to as IAC (for Interpret-as-Command). The 0xff is followed by a binary command between 0xf0 and 0xfe, which is followed by the data required by the command.

Opening a connection to a telnet server is easy in Tcl: we just use the normal socket command.

```
set connection [socket $Client(ipAddress) 23]
```

The default behavior for the socket command is to open a buffered connection to the remote host and translate carriage-return/newline pairs to whatever the normal line termination characters are on the local host.

This works fine for printable ASCII text, but creates a problem with binary data, where a 0x0a might be not be a newline and adding a 0x0d will break a binary data stream.

The solution is to use the Tcl fconfigure command to modify the socket's behavior to something more acceptable for binary data.

The fconfigure command will modify several aspects of the channel including whether the channel should be buffered, the size of the buffer to use, whether to block on read/write and how to translate binary characters.

Syntax: fconfigure *channelId ? name? ?value?*

fconfigure	Configure the behavior of a channel	
channelId	The channel to modify	
name	The name of a configuration field which includes:	
	-blocking *boolean*	If set **true** (the default mode), a Tcl program will block on a gets, or read until data is available. If set **false**, gets, read, puts, flush, and close commands will not block.
	-buffering *newValue*	If *newValue* is set to **full**, the channel will use buffered I/O. If set to **line**, the buffer will be flushed whenever a full line is received. If set to **none**, the channel will flush whenever characters are received.
	-translation *{inMode outMode}*	Controls how end-of-line translations are performed. Acceptable values for *inMode* and *outMode* include:

auto	On input, treats any combination of cr and lf as line terminators. On output line termination is sent as whatever is native for the current platform.
binary	No end-of-line translations are performed.
cr	On input, all cr characters are converted to newline. On output, newlines are converted to a cr character.
crlf	On input, all crlf characters are converted to newline. On output, newlines are converted to a crlf sequence.

We can modify the behavior of our socket with this code:

```
fconfigure $connection -translation binary -buffering none -blocking 0
```

The previous client-server pairs used the Tcl gets command to read a line of text from the server. Again, looking for a newline doesn't work with binary data.

The Tcl read command handles this problem.

Syntax: read *channelID* *?numBytes?*

read	Read a certain number of characters from a channel.
channelID	The channel to read data from
numBytes	Optionally, the number of bytes to read. If this argument is left out, all available data is read.

These commands enable us to build a non-line-oriented client with commands like this:

```
set s [socket $argv 23]
fconfigure $s -translation binary -buffering none -blocking 0
fileevent $s readable "readData $s"
proc readData {s} {
    set n [read $s]
    if {[eof $s]} {
        close $s
    return
    }
    # Process data.
}
```

Whenever there is data available in the input buffer, the readData procedure will be evaluated to read and process the data. Since the socket is configured to be non-blocking, the read command will read whatever is in the buffer, whether it ends in a newline or not.

This brings us to the processing part of this example, which is to print out a human-friendly version of the server's requests.

Since Tcl is (like most of us) primarily text oriented, the first thing to do with the binary data is convert it into printable ASCII to make it easier to work with. Since the telnet protocol is byte oriented, it makes sense to convert the data into single byte hex values.

We can use the Tcl binary scan command to convert the binary data to a string of Hex values.

Syntax: binary scan *binaryString* *formatString* *arg1* *?varName1?* ... *?varNamen?*

binary scan	Converts a binary string to one or more printable ASCII strings
binaryString	The binary data
formatString	A string that describes the format of the ASCII data
*varName**	Names of variables to accept the printable representation of the binary data

For example to convert a string to hexadecimal values:

```
# Convert "abc" to "616263", save value in hexData
  binary scan "abc" "H*" hexData
```

That's better, but still hard to read and work with. If the string were a list of individual byte values, we could read the string more easily and use Tcl's list-processing commands to step through the data.

The Tcl split command will convert a string into a list, splitting on whatever character (or characters) you desire.

Syntax: split *data* *?splitChar?*

split	split *data* into a list.
data	The data to split.
?splitChar?	An optional character (or list of characters) to split the data at

A common use of the split command is converting some character-delimited string (like a spreadsheet export file) into a proper Tcl list.

```
# Convert "3/23:Cab from Airport:35.00"
# to
# {3/23 {Cab from Airport} 35.00}
set line "3/23:Cab from Airport:35.00"
set lst [split $line ":"]
```

By default, the splitChar is a whitespace. You can define any other character or characters to be the split character, as the previous example does. You can even declare the split character to be an empty string, which splits a string into a list where each character is a separate list element.

```
# convert "abcd" to {a b c d}
set lst [split "abcd" ""]
```

The next step is to convert the list of single hex values into a list of byte values. We could step through the list two elements at a time using Tcl's foreach command, but since this is a fairly simple pattern to convert, it's faster to use the regsub command.

The regsub command will modify patterns that match a regular expression to another pattern. The syntax looks like this:

Syntax: regsub *?options?* *expression* *string* *subSpec* *varName*

regsub	Copies *string* to the variable	
varName.	If *expression* matches a portion of *string* then that portion is replaced by *subSpec.*	
options	Options to fine-tune the behavior of regsub may be one of:	
	all	Replace all occurrences of the regular expression with the replacement string. By default only the first occurrence is replaced.
	-nocase	Ignores the case of letters when searching for match.
	—	Marks the end of options. Arguments which follow this will be treated as regular expressions, even if they start with a dash.
expression	A regular expression which will be compared to the target string.	
string	A target string to which the regular expression will be compared.	
subSpec	A string which will replace the regular expression in the target string.	
varName	A variable in which the modified target string will be placed.	

In the simple use of regsub, the *subSpec* value will be a simple string:

```
# Convert powerful to useful.
regsub "power" "Regular expressions are powerful"  "use" string2
```

The regexp command will sort parenthesized subsets of a regular expression pattern into separate variables. Similarly, the Tcl regsub command supports merging patterns from the original string in the subSpec value. These substitutions are specified by using a backslash-escaped number in the subSpec string to correspond to the parts of the pattern you wish to substitute. The number denotes which part of the pattern will be substituted for the backslash-escaped value.

The pattern \0 will substitute for the entire matched portion of the string. If sections of the regular expression are placed in parentheses, these sections can be addressed as \1 for the first parenthesized expression, and \2 for the second, etc.

```
# Invert the positions of "a" and "c"
# x will contain the string : "abc becomes: cba"
regsub "(a)(.)(c)" "abc" {\0 becomes: \3\2\1} backwards
```

Which, finally gets us to a simple three-line procedure to convert a binary string to a list of hexadecimal-coded byte values:

```
# Convert binary strings to lists of hex values
# "abc" converts to "61 62 63"

proc bin2hex {binaryString} {
    binary scan $binaryString "H*" hexData
    regsub -all {([^ ]) ([^ ])} [split $hexData ""] {\1\2} hexList
    return $hexList
}
```

The final process is to step through the list of binary data and print out something a simple human can easily read. The first byte of any command will be the IAC symbol, 0xFF. The value after that will be a byte that denotes which command this is.

In C, we might parse the second byte with a case command like this:

```
switch (i) {
    case FE:  {processFE; break;}
    case FD:  {processFD; break;}
}
```

In Tcl, we can let the interpreter handle the parsing, and instead of using a switch statement to parse our commands, we can name the procedures for the command name.

```
# Define procedures to process the command
proc FE {...} {
    # Process the FE command
    ...
}
proc FD {...} {
    # Process the FD command
    ...
}
...
# Select the command from the hexadecimal list
# (something like FA of FD)
set cmd [lindex $hexList $position]

# Evaluate the command
$cmd
```

The number of bytes after the command byte varies for each command. Some of the information commands (like terminal type) can even include free-form ASCII data.

In C, it would be nice to use a pointer, and advance the pointer as we process each command and data.

Unfortunately, this isn't an option in Tcl, so we'll have to use the lindex command to select each byte we want to look at from the list, and increment our position counter after we process each command.

To make life simpler, the procedures that process each command are defined to return the number of bytes to increment the pointer to get to the next IAC marker.

The telnet protocol allows interleaving of ASCII text and IAC commands in a data stream, so we have to check each byte to see if it's an IAC symbol and process the command if it is.

```
set lst [binaryStrings::bin2hex $string]
for {set i 0} {$i < [llength $lst]} {incr i} {
      # Get the current character
      set l [lindex $lst $i]
      # Loop for as long as the current character is
      #  an IAC marker
      while {[string match $l "ff"]} {
          # Get the command byte
incr i
set cmd [lindex $lst $i]
# Evaluate the command, and increment
#  the position marker
incr i [$cmd [lrange $lst $i end]]
# Get the next byte – if it's still an IAC we'll
#  loop.
set l [lindex $lst $i]
      }
}
```

The procedures that process the commands contain the logic for handling each telnet command and also save a comment about what they've done.

The basic format of the procedures is:

```
proc fd {sublist ...} {
      # Process the fd command
      # ...
      addInfo $subl TELOPT_
      return $commandLength
}
```

The addInfo procedure uses an associative array as a lookup table to get the human-readable information about each command and puts that string into a temporary comment buffer, which is combined by the main loop when a command has completed.

```
proc addInfo {subl prefix} {
      variable Telnet
      variable Lookups
      set m [lindex $subl 0]
      scan $m %x dec
      set id [array names Lookups ${prefix}*.$dec]
      if {[string match $id ""]} { set id UNKNOWN.-1 }
      append Telnet(comment-temp) " $Lookups($id)"
}
```

The lookup table has indices in the form mnemonic.decimalValue, and could be constructed with code like this:

```
set Lookup(IAC.255) {interpret as command:}
set Lookup(DONT.254) {you are not to use option}
set Lookup(DO.254) {please, you use option}
...
```

However, the telnet include file (telnet.h) already has a nice set of comments that would be useful for this application. Rather than making up my own messages, I decided to write a little script to read the include file, find the lines in the form

```
 #define mnemonic value /* Comment */
```

and use them to generate the lookup table to convert the binary values to a human-readable text. This uses the regexp command's support for sorting patterns into separate variables to recognize the lines with valid information and build the array index and value.

```
################################################################
# proc readIncludeFile {arrayName fileName}–
#    Read an include file, and make a lookup table for all the
#    lines that match a
#    #define xxx val  /* comment */
#    or
#    #define xxx val
#    format
#
# Arguments
#   arrayName: The name of an array in the calling procedures scope.
#   fileName: The full path for the include file to be read.
#
# Results
#   Adds new indices to the arrayName in the form:
#   set arrayName(mnemonic.value) "comment"
#
proc readIncludeFile {arrayName fileName} {
      upvar $arrayName localArray

      set if [open $fileName r]

      while {[set len [gets $if line]] = 0} {

      # Use the Advanced Regular Expression "\s"
      # Class Shorthand to encode for whitespace.
      set m [regexp \
          {#define[\s]+([^\s]+)[\s]+([^\s]+)[\s]+/\*([^*]+) \*/} \
              $line all mnemonic num comment]

      # If no match, it might be a line with no comment
      if {!$m} {
          set m [regexp {#define\s+([^\s]+)\s+([^\s]+)} \
              $line a mnemonic num]
      set comment "No Comment"
      }

      # If still no match, this line must note be a #define line
      #  skip it.
      if {!$m} {continue}

      # Holler if there's a collision.  I guess a human will
      #  need to deal with this file.
      if {[info exists localArray($mnemonic.$num)]} {
            puts "COLLISION: $num"
            exit
        }
        # Everyone's happy, assign the value
        set localArray($mnemonic.$num) $comment
    }
  }
```

The readIncludeFile requires Tcl 8.1 or newer to use the [\s] convention for describing whitespace.

This procedure is invoked as

```
        readIncludeFile Lookups /usr/include/arpa/telnet.h
```

That covers all the pieces in this application. When you run it, it can send some simple responses to the server, and will report all the IAC messages the server sent to the client. This can be useful if you are trying to figure out what optional protocols your server supports, or if you are writing a robot that needs to talk with a telnet server.

If you run the program with the telnet server distributed on RedHat 6.2 Linux, it will return:

```
  COMMENT:     please, you use option:      terminal type
  COMMENT:     please, you use option:      terminal speed
  COMMENT:     please, you use option:      X Display Location
  COMMENT:     please, you use option:      New – Environment variables
```

If you use the program to examine a BSD system it will return:

```
  COMMENT:     please, you use option:      Unknown Option
  COMMENT:     I will use option:           Encryption option
  COMMENT:     please, you use option:      terminal type
  COMMENT:     please, you use option:      terminal speed
  COMMENT:     please, you use option:      X Display Location
  COMMENT:     please, you use option:      New – Environment variables
  COMMENT:     please, you use option:      Old – Environment variables
```

As usual, this code for this example is available at *http://noucorp.com*.