;login:

THE MAGAZINE OF USENIX & SAGE

February 2001 • volume 26 • number 1

inside:

CLIF FLYNT:
THE TCLSH SPOT

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

the tclsh spot

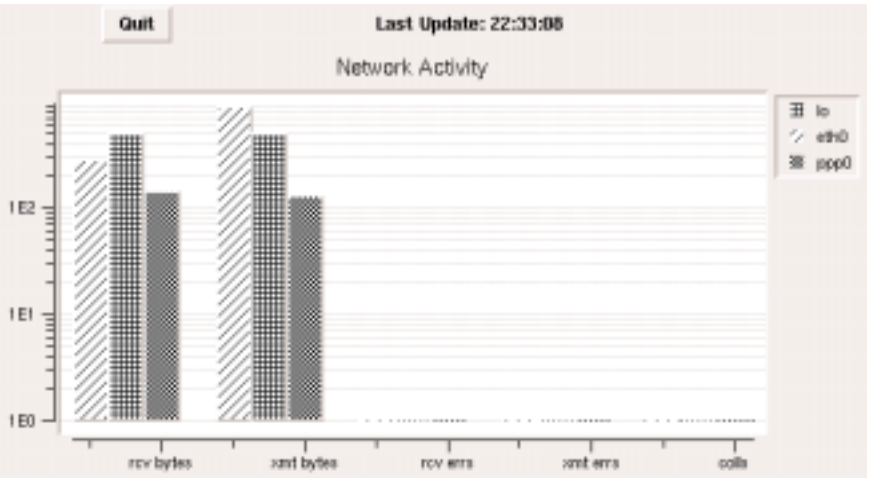
by Clif Flynt

Clif Flynt has been a professional programmer for almost twenty years, and a Tcl advocate for the past four. He consults on Tcl/Tk and Internet applications.

<clif@cflynt.com>

The previous Tclsh Spot article described how to use the Tcl socket command to build a simple client-server-based system monitor to watch disk space usage.

This article will expand on that idea to create a network activity monitor with a graphical display looking something like this:



The server in the previous article looked like this:

```
socket -server initializeSocket 55555

proc initializeSocket {channel addr port} {
    after 1000 sendDiskInfo $channel
}

proc sendDiskInfo {channel} {
    set info [exec df]
    puts $channel $info
    flush $channel
    after 2000 sendDiskInfo $channel
}

vwait done
```

This simple server has a few serious shortcomings. It throws an error when a client closes a connection, and it doesn't do any validity checks to confirm that a client is entitled to the information it's getting.

The error condition occurs when the server tries to flush the data out of a socket that was closed by the client. The simplest way to test if a channel is open is to try to send data, and see if it fails. Which is how the server generates those ugly error messages. If there were a way to run a command and find out if it worked without throwing an error, this would be perfect.

The catch command evaluates a script and returns the success or failure status without invoking the Tcl error handlers. The value that would otherwise be returned by the script is saved in an optional second variable.

Syntax: catch *script ?varName?*

catch	Catch an error condition and return the status and results rather than aborting the script.

script	The Tcl script to evaluate.
varName	Variable to receive the results of the script.

These two lines produce equivalent results, but the second one won't fail if x contains a non-numeric value:

```
set x2 [expr $x * 2]
set fail [catch {expr $x * 2} x2]

# Deal with the error
if {$fail} {puts "$x is not a number"}
```

One of the simplest validation checks is to confirm that the IP address a client is connecting from is on the list of allowed sites. Since the Tcl interpreter gives us the address of the client as one of the arguments to our initializeSocket procedure, it's easy to add this style of validation to the server. We could simply search a list of allowed IP addresses for this client address and close the channel if the search failed.

Unfortunately, while Tcl will search a list for patterns with wildcards, it won't search a list where some list elements have wildcards for a match to a specific string. So, if we wanted to use lsearch to search our string, we'd have to put each allowed address into the list. If you want to allow access to everyone on your class A subnet, this would get ugly.

However, the string match command will let us match a wildcard pattern against a fixed string, and we can use that to check for matches within a much smaller list.

Syntax: string match *pattern string*

string match	Returns 1 if pattern matches string, else returns 0.
pattern	The glob pattern to compare to string.
string	The string to match against the pattern.

This code compares the client IP address with patterns in a list, and only allows clients that match one of the patterns. A second set of patterns, and similar code, could check for addresses on a "forbidden" list.

```
set Server(allowed) {192.168.9.* 127.0.0.1}
...
proc initializeSocket {channel addr port} {
    global Server

    set reject 1
    foreach ip $Server(allowed) {
        if {[string match $ip $addr]} {
            set reject 0
            break;
        }
    }
    if {$reject} {
        close $channel
        return
    }
...
```

The previous server handles one type of service. It reports disk usage. Traditionally, we build a different server for each application, since most servers are complicated programs performing complicated tasks.

However, the system monitor server is pretty simpleminded. It leaves all the fancy analysis to the clients. So, rather than run multiple servers on this already overloaded machine, we can use a single server that listens on multiple ports and reports different information depending on which port was accessed.

The syntax for the socket command is:

```
socket -server command ?options? port
```

The command argument is generally thought of as the name of a procedure to invoke, but it's actually a script to which Tcl will append the three arguments and evaluate. You could have something as simple as the name of the procedure to evaluate, as we did in the previous server, or an arbitrarily complex command script.

In this case, we can pass a new argument to the initializeSocket procedure and have the initializeSocket procedure parse that value to decide which data reporting procedure to evaluate. That value could be some flag (1 for disk, 2 for network activity), but it's simpler to let the Tcl interpreter do the parsing for us by passing the name of the procedure to call to send data to the initializeSocket procedure like this:

```
socket -server {initializeSocket sendDiskInfo} 55555
socket -server {initializeSocket sendNetInfo} 55556

proc initializeSocket {proc channel addr port} {

  # Check validity.
    after 1000 $proc $channel
}
```

The sendNetInfo procedure looks a lot like the sendDiskInfo command, except that we collect some network statistics instead of disk usage.

On a Linux system, I can get a report of the number of bytes that have been transferred by reading the file /proc/net/dev. On a BSD system, you can get this information with the ifconfig command.

Here's the sendNetInfo procedure for a Linux system:

```
proc sendNetInfo {channel} {
    set if [open /proc/net/dev "r"]
    set data [read $if]
    close $if
    puts $channel $data
    set fail [catch {flush $channel} ]

    if {$fail} {
        close $channel
    } else {
        after 2000 sendNetInfo $channel
    }
}
```

Meanwhile, on the client end, we need to read that data.

The previous client looped on gets and hung until a line of data was available. This works fine for a simple client, but is a rather inelegant way of dealing with I/O.

Tcl supports both the linear type of program flow that we used in that block-until-data-is-ready model, and an event driven flow in which the interpreter waits in an event loop until something happens.

The fileevent command defines a script to evaluate when data becomes available. This guarantees that data will be available to read when the script is called, thus the application never blocks.

Syntax: fileevent *channel direction ?script?*

fileevent	Defines a script to evaluate when a channel readable or writable event occurs.
channel	The channel identifier returned by open or socket.
direction	Defines whether the script should be evaluated when data becomes available (readable) or when the channel can accept data (writable).
?script?	If provided, this is the script to evaluate when the channel event occurs. If this argument is not present, Tcl returns any previously defined script for this file event.

The lines in our client to implement this look like this:

```
set input [socket $Client(ip) $Client(port)]
fileevent $input readable "getNetInfo $input"
```

Once we've read a line of data we need to figure out if this line has any useful information in it. The output of /proc/net/dev includes two lines of column headers and some trailing blank lines that have no useful information (for this procedure).

The first word of the data lines from /proc/net/dev is the name of the device, but the second word will always be a number in the lines with data to process. The client can check to see if the second word is really a number, and if it's not go on to the next line.

The newer versions of Tcl have a string is command that will let you figure out if a string contains alphabetic, numeric, control characters, etc.

For older versions, we can use the catch and expr commands to figure out if a value is numeric.

If a value is numeric, you can multiply it. If the string has non-numeric characters in it, the exec command will fail, and catch will return an error.

Here's code to check that the second word in a line of data is numeric, and return immediately if it isn't.

```
if {[catch {expr [lindex $line 1] * 2} ]} {return}
```

Once we strip out the headers and blank lines, we are still getting a lot of numeric data, and we need to do something with it. This looks like another great application for the BLT widgets. The set of articles about the stock robots discussed using the BLT graph widget. This article will describe a bit about the BLT barchart widget.

You create a BLT barchart very much as you'd create a graph (or any other Tk widget).

Syntax: barchart *name ?option value?*

name	A name for this barchart widget, using the standard Tcl window-naming conventions.
?option value?	Option and value pairs to fine-tune the appearance of the barchart. The available options include:
	-background The color for the barchart background.
	-height The height of the barchart widget.
	-title A title for this barchart.

-width	The width of the barchart widget.
-barwidth	The width of each bar on the barchart.

This command will create a simple barchart, and save the widget name in an associative array variable. Note that the BLT widget commands exist within the ::blt:: namespace. The widgets created by these commands are created in the current namespace.

```
package require BLT
set Client(barChart) [::blt::barchart .bcht -width 600 -title "Network Activity"]
```

Like the graph widget, the barchart widget supports several options for configuring the axes. Two that we'll use in this application are:

-logscale boolean	Set the axis to use a logarithmic scale instead of linear.
-command script	Defines a script to invoke to get a value to use as a tic label.

The log scaling is particularly important with something like this network activity monitor. If the network is approaching saturation, we'll have a huge disparity between the number of bytes moved in two seconds and the number of collisions that occurred, but seeing the collision bar is what's important. If they get close enough to the same size that we can see the height of the collision bar on a linear scale, we've already lost.

Along with the graph and barchart widgets, BLT introduces a new primitive data type to Tcl – the vector.

From the script viewpoint, a BLT vector is an array of floating point values with the constraint that the indices must be integers within the range defined when you create the vector.

A vector can be created with the vector command like this:

```
::blt::vector myvector($size)
```

In this case, $size is a variable that contains the number of slots to allocate in this vector.

You can think of creating a BLT vector as a float myvector[size]; declaration, if it helps.

One neat thing about vectors is that you can use a vector to hold the X or Y data for a barchart element, and whenever a data value changes, the chart changes to reflect this without your code needing to do a redraw. For an application like this network activity barchart, where the height of the bars is constantly changing, this is very useful.

Barchart elements are created with the element create subcommand, just as graph elements are created. Like the graph element create command, we can supply several options to the element create command.

Useful options in a chart like this, where there are several sets of data, are the -foreground, -background, and -stipple options that let you control the color and texture of the bars to make them easily identified.

Which brings up the question of how to decide what color to make which bar. If we know the devices we'll have on a system, we could define a look-up table to convert from device name to color. However, this would mean a code rewrite when we change or add adapters.

Another thing we can do is initialize the client with a list of colors, and whenever a new device is seen, we create a new bar with the next color, and increment a pointer to the next color.

Playing games with variable names is not usually good style.

```
set Client(count) 0
set Client(colors) {red green blue purple orange}

...
# If new device name, create new bar
if {![info exists DataVector_x_${name}]} {
    vector DataVector_x_${name}(5)
    vector DataVector_y_${name}(5)

$Client(barChart) element create $name -label "$name" \
    -foreground  [lindex $Client(colors) $Client(count)] \
    -xdata DataVector_x_${name} -ydata DataVector_y_${name}
incr Client(count)
}
```

Note that we can use the info exists command to check if a vector has been defined, just as we'd use it to check for any other primitive Tcl data type like an array or a list.

The names used for the vectors in this code snippet look strange. The reason for this is that I'm playing some games with the variable names to create a common set of base identifiers for the vectors.

The vector is a linear structure, so we can't use the usual Tcl trick of making a multidimensional array with a naming convention for the index. However, we can create as many uniquely named vectors as we need, and can embed the name of the device in the name of the vector.

Playing games with variable names is not usually good style. Your code will be cleaner and easier to work with if you use an associative array. It's a bit too easy to confuse yourself with what parts of a variable name are being substituted, and what parts are the constant part of the name.

For example, you might write this code thinking you were creating two variables eth0_Bytes and eth0_Errors.

```
set id eth0

set $id_Bytes $byteCount
set $id_Errors $errorCount
```

The Tcl interpreter doesn't know that you intend to just use the characters $id as a variable substitution. The syntax rules say that a variable name is terminated by special character (usually a space). So, the Tcl interpreter throws an error that the variable id_Bytes hasn't been assigned.

The curly braces can be used to group a part of variable name into a single substitution unit. Thus, we could rewrite the above example like this to make it work.

```
set id eth0

set ${id}_Bytes $byteCount
set ${id}_Errors $errorCount
```

This works, but it's not pretty code. The better solution (when you can use the associative array) is:

```
set id eth0

set Bytes($id) $byteCount
set Errors($id) $errorCount
```

A clever way to design this client is to have it build bar elements as they are found to be needed, rather than starting out by building N sets of bar elements. After all, the client doesn't know (unless you put some hardcoded values into the code) how many devices are on the server until it starts to analyze the data the server sends. Letting the client configure itself to the environment makes it adaptable without the need to update code.

Letting the client configure itself to the environment makes it adaptable without the need to update code.

The BLT barchart widget supports a configure subcommand, and like other Tk widgets you can modify the appearance and behavior of an existing widget with this command.

Configuring the -barwidth option lets us make the bars narrower as we need more data sets, rather than expanding the widget until it scrolls off the screen.

We can fine-tune the location of the bars by changing the bar positions in the DataVector_x_* vectors, but that means we need to know the names of the DataVector_x_* vectors. We could save the names as we create the vectors, but Tcl has already saved all the names, so why duplicate the effort?

The Tcl info command can list the variables that have been defined in a local or a global scope. You can get a list of all the variables defined, or just the variables that match a particular glob pattern.

This is why I used the strange naming convention for the vector names, rather than simply defining them as:

```
vector $name(5)
```

The syntax for the info globals command is:

Syntax: info globals *pattern*

info globals	Returns a list of global variables that match the pattern.
pattern	A glob pattern to attempt to match.

So, putting these pieces together and wrapping it into a procedure, we get something like this to create a new element. Each element is the set of bars showing the number of bytes transferred, errors, and collisions.

```
proc makeNewBarSet {name} {
    global Client
        $Client(barChart) element create $name -label "$name" \
            -foreground [lindex $Client(colors) $Client(count)] \
            -xdata DataVector_x_${name} -ydata DataVector_y_${name}

        incr Client(count)

        # Make the bars 1/(n+1) wide -
        # this creates a one bar-width space
        # between the sets of data

        $Client(barChart) configure -barwidth \
            [expr 1.0 / ($Client(count) + 1)]

        # The DataVector_x_* vector holds the location
        #  for the bars.

        # Tic's are marked on integer boundaries, so start at
        # -.5 to get tic labels centered on the data sets

        set item 0
        foreach v [info globals DataVector_x_*] {
```

```
            global $v
            for {set i 0} {$i < [llength $Client(tics)]} {incr i} {
                set ${v}($i) [expr $i + $item / ($Client(count) + 1.0) -.5]
            }
            incr item
        }
}
```

Which gets us to parsing the data the server sends us. The output from /proc/net/dev is sets of lines that look like this:

```
eth0: 1535  429 0 0 0 0 0 0 320353952  956185  0 0 0 108688 0 0
```

The first field is the name of the device, then the number of bytes received, the number of packets received, errors received, etc. The BSD ifconfig output follows a similar pattern, except that it reports the quantities since the last invocation of ifconfig, rather than quantities since the system was booted.

We can treat each line as a list. The values we are interested in will always be at particular locations in the list. Thus, we can write some generic code to parse the list, and drive it with a pair of lists that describe the locations of the data we want, and a label for that data:

```
set Client(tics) {{rcv bytes} {xmt bytes} {rcv errs} {xmt errs} {colls}}
set Client(pos)  {1           9           3           11          14}
```

We need to save the values from the previous server report in order to calculate the number of bytes transferred. Which means we need to be able to find that data again when we need it.

This is another good place to simulate a 2-dimensional array with the Tcl associative array and a naming convention. Since we get the name of the device in position 0 of the list, and we know the positions of the fields we are collecting, we can parse the list with code that loops through the lists of positions and labels to collect and calculate the data. The results of the calculation are put into the DataVector_y_* vectors to cause the barchart to reflect the new values.

Again, we can use the Tcl info exists command to determine if a variable has had a value assigned to it yet. If the variable has had a value assigned to it, we can calculate a difference.

This code will grab values from the line of data, check to see if we've already saved one of them, and calculate the difference if we have.

```
        set vectorPos 0
        foreach pos $Client(pos) label $Client(tics) {
            set val [lindex $line $pos]
            if {[info exists Client($name.$label)]} {
                set DataVector_y_${name}($vectorPos) \
                    [expr $val - $Client($name.$label)]
                incr vectorPos
            }
            set Client($name.$label) $val
        }
    }
```

This gives us a nice little snapshot monitor. But, as they say, those who don't remember history are doomed for some ugly shocks.

In the next Tclsh Spot article I'll look at ways to save and present some historical data on the network activity.

Here's the complete code for this client/server pair. This code is also available at <*http://www.noucorp.com*>.

server.tcl

```
socket -server {initializeSocket sendDiskInfo} 55555
socket -server {initializeSocket sendNetInfo} 55556

set Server(allowed) {192.168.9.* 127.0.0.1}

proc bgerror {args} {
    global errorInfo
    puts "ERROR: $args"
    puts "$errorInfo"
}

proc initializeSocket {proc channel addr port} {
    global Server

    set reject 1
    foreach ip $Server(allowed) {
        if {[string match $ip $addr]} {
            set reject 0
        }
    }
    if {$reject} {
        close $channel
        return
    }
    after 1000 $proc $channel
}

proc sendDiskInfo {channel} {
    set info [exec df]
    puts $channel $info
    set fail [catch {flush $channel} out]
    if {$fail} {
        close $channel
    } else {
        after 2000 sendDiskInfo $channel
    }
}

proc sendNetInfo {channel} {
    set if [open /proc/net/dev "r"]
    set data [read $if]
    close $if
    puts $channel $data
    set fail [catch {flush $channel} out

    if {$fail} {
        close $channel
    } else {
        after 2000 sendNetInfo $channel
    }
}

vwait done
```

client.tcl

```
package require BLT

# Some defaults and constants
set Client(count) 0
set Client(colors) {red green blue purple orange}
set Client(tics) {{rcv bytes} {xmt bytes} {rcv errs} {xmt errs} {colls}}
set Client(pos) {1          9          3          11         14}

set Client(ip) 192.168.9.1
set Client(port) 55556

#########################################################################
# proc getNetInfo {channel}—
#    Retrieves a set of network information from the socket
#    Parses the info, and creates a set of 'diff' index arrays
#    that are the difference between this value and the previous
#    value for a field.
#
# Arguments
#   channel   The channel to read data from
#
# Results
#   Modifies the Client array.
#   Invokes processData to update the bar

proc getNetInfo {channel} {
    global Client

    gets $channel line

    # The first element is the name, but the second should
    #   be a number.  If it isn't (this is a line of column headers.)
    #   We'll skip out and wait for the next line of data.

    if {[catch {expr [lindex $line 1] * 2}]} {
        return
    }

    # Long integers may run into the ":" in the line label
    #  This gives us a space to parse on.
    regsub ":" $line " " line

    # The first entry is the device name.
    set name [lindex $line 0]

    global DataVector_x_${name}
    global DataVector_y_${name}
    if {![info exists DataVector_x_${name}]} {

        ::blt::vector DataVector_x_${name}(5)
        ::blt::vector DataVector_y_${name}(5)

        makeNewBarSet $name
    }

    # The DataVector_y vector holds the heights of the bars
    #   for a given data set.

    set vectorPos 0
    foreach pos $Client(pos) label $Client(tics) {
```

```
        set val [lindex $line $pos]
        if {[info exists Client($name.$label)]} {
            set DataVector_y_${name}($vectorPos) \
                [expr $val - $Client($name.$label)]
            incr vectorPos
        }

        set Client($name.$label) $val
    }
    set Client(update) "Last Update: [clock format [clock seconds]\
        -format %H:%M:%S]"
}

######################################################################
# proc makeNewBarSet {name}—
#  Creates a new set of bars, and reconfigures the barchart to hold them.
#
# Arguments
#  name       The name of the data associated with this set
#
# Results
#   Creates a new DataVector global.
#   Modifies the barchart and existing DataVector_x_* data.

proc makeNewBarSet {name} {
    global Client
    $Client(barChart) element create $name -label "$name" \
        -foreground [lindex $Client(colors) $Client(count)] \
        -xdata DataVector_x_${name} \
        -ydata DataVector_y_${name}

    incr Client(count)

    # Make the bars 1/(n+1) wide -
    # this creates a one bar-width space
    # between the sets of data

    $Client(barChart) configure -barwidth [expr 1.0 / ($Client(count) + 1)]

    # The DataVector_x_* vector holds the location
    #  for the bars.

    # Tic's are marked on integer boundaries, so start at
    # -.5 to get tic labels centered on the data sets

    set item 0
    foreach v [info globals DataVector_x_*] {
        global $v
        for {set i 0} {$i < [llength $Client(tics)]} {incr i} {
            set ${v}($i) [expr $i + $item / ($Client(count) + 1.0) -.5]
        }
        incr item
    }
}

######################################################################
# proc getTicLabel {chart tic}—
#    Returns a textual label for the barchart
# Arguments
#   chart  The chart associated with this request
#   tic  The position of the tic being requested.
```

```
proc getTicLabel {chart tic} {
      global Client
      return [lindex $Client(tics) $tic]
}

# Make a quit button
button .b -text "Quit" -command "exit"
grid .b -row 0 -column 0

# And the update time label
label .l -textvar Client(update)
grid .l -row 0 -column 1

# Build a barchart
set Client(barChart) [::blt::barchart .bcht -width 600 -title\
  "Network Activity"]
$Client(barChart) axis configure x -command getTicLabel
$Client(barChart) axis configure y -logscale 1

grid $Client(barChart) -row 2 -column 0 -columnspan 3

# Open a client socket on the local system
# (for testing purposes.)
set input [socket $Client(ip) $Client(port)]

# When data is available to be read, call getNetInfo
fileevent $input readable "getNetInfo $input"

# And wait for the fireworks to start.
```