;login:
THE MAGAZINE OF USENIX & SAGE
April 2001 • volume 26 • number 2

inside:

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 two "Tclsh Spot" articles described building a client-server package to monitor disk and network usage. Being able to monitor the current state of a server is useful, but it's even more useful if you can use that data to watch for a trend. That means saving and displaying some historical data. This article will describe using the BLT vector and barchart commands to show the historical data, and will describe the technique of using a canvas widget to attach a scrollbar to widgets (like the barchart) that don't normally support scrolling.

The Tk canvas widget is one of the workhorse widgets in the wish application. You can draw vector style images on a canvas, display images on it, and even display other Tk widgets on it. If you have an image larger than your display, you can link the canvas to a scrollbar to view a window into a larger area.

The syntax for creating a canvas widget is the same as for creating other Tk widgets:

Syntax: canvas *canvasName* *?options?*

canvas	Create a canvas widget.	
canvasName	The name for this canvas.	
?options?	Some of the options supported by the canvas widget are:	
	-background *color*	The color to use for the background of this image. The default color is light gray.
	-scrollregion *boundingBox*	Defines the size of a canvas widget. The bounding box is a list: left top right bottom that defines the total area of this canvas, which may be larger than the displayed area. These coordinates define the area of a canvas widget that can scroll into view when the canvas is attached to a scrollbar widget. This defaults to 0 0 width height, the size of the displayed canvas widget.
	-height *size*	The height of the displayed portion of the canvas. If *-scrollregion* is declared larger than this, and scrollbars are attached to this canvas, this defines the height of the window into a larger canvas. The *size* parameter may be in pixels, inches, millimeters, etc.
	-width *size*	The width of this canvas widget. Again, this may define the size of a window into a larger canvas.

Once you've created a canvas widget, you can create items on the canvas with the $canvasWidget create subcommand. The syntax for the create subcommand is:

```
canvasName create itemType coordinates ?-flag value?
```

The code to create a canvas with a box in the upper left corner would resemble:

```
set cvs [canvas .c]
grid $cvs -row 0 -column 0
$cvs create rectangle 0 0 20 20
```

Creating graphic objects and images on a canvas is quite simple. When you use the canvas widget as a holder for other windows, life becomes a little more complicated.

Tk defines each widget as being part of a hierarchy of windows. Each window (except the top level) is the child of some parent window. The windows are named using a period as the name separator. The main window in a wish session is named ".", just as in a file system the root directory is "/".

In a simple GUI application all the widgets can be children of the main window. This leads to names like .button1, .canvas, etc.

For more complex applications, you may need to use the frame or canvas widgets to group your widgets. This leads to widget names like .buttonFrame.quitButton or .canvas.barchart.

Most windowing systems propagate parameters (like window size) from a parent window to the child windows. While Tk will let you display any window within a canvas or frame, it's best to make the window you intend to display in another window a child of that parent.

This code would create a label as a child of a canvas, and display the label on the parent canvas, instead of on the main window:

```
set cvs [canvas .c]
grid $cvs -row 0 -column 0
set l [label $cvs.label -text "I'm on a canvas"]
$cvs create window 20 20 -window $l
```

One problem with any drawing package is the need to show a drawing that's larger than your display. The standard solution to this problem is to create a viewing window into the larger object, and move the viewing window around the larger window with one or two scrollbars.

The canvas widget can be treated as a window into a larger drawing area by setting the -scrollregion option to be larger than the canvas size. A scrollbar widget can be created with the scrollbar command.

Syntax: scrollbar *scrollbarName ?options?*

scrollbar	Create a scrollbar widget.	
scrollbarName	The name for this scrollbar.	
options	This widget supports several options. The required *-command* option is required.	
	-command "*procName ?args?*"	This defines the command to invoke when the state of the scrollbar changes. Arguments that define the changed state will be appended to the arguments defined in this option.
	-orient *direction*	Defines the orientation for the scrollbar. The direction may be horizontal or vertical. Defaults to vertical.
	-troughcolor *color*	Defines the color for the trough below the slider. Defaults to the default background color of the frames.

The wish interpreter handles the interaction between the canvas and scrollbar by registering a callback procedure with the scrollbar and canvas widgets. Whenever one of these widgets changes state, it will evaluate the registered script to update the other widget.

The code to create and display a canvas and scrollbar would resemble:

```
set cvs [canvas .c -xscrollcommand {.sb set} -height 20 -width 40]
set scroll [scrollbar .sb -command {.c xview} -orient horizontal}
.c configure -scrollregion {0 0 20 400}
```

```
grid $cvs -row 0 -column 1
grid $scroll -row 1 -column 1 -sticky ew
```

With the canvas and scrollbar widgets, we can enhance the network monitor.

The server part of this application was described in the previous "Tclsh Spot" (February 2001 *;login:*, p. 49). It sends unformatted data to the clients from whatever system utility is being watched.

The client is responsible for parsing and displaying the data. The goal of this design is to be able to use compute-intensive graphics and intelligence to monitor a system without adding extra overhead to an already overloaded server. A fallout of this design is that we don't need to modify the server in order to add more information to the client display.

The client uses the BLT extension's barchart command to build a barchart, and the vector command to hold the data being displayed.

The syntax for creating a barchart widget looks like this:

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.

The barchart is created with code like this:

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

Once a barchart has been created, we can add elements to that barchart by creating a vector to hold the data with this code:

```
::blt::vector xvector(5)
::blt::vector yvector(5)
```

and then adding an element to the barchart with this command:

```
$Client(barChart) element create $name -label "$name" \
    -stipple [lindex $Client(stipple) $Client(count)] \
    -fg black -bg white \
    -xdata xvector -ydata yvector
```

As the client receives data, it parses the data, and sets the appropriate vector value, and the barchart automatically redraws itself to show bars of the appropriate height.

This is useful, but again, one goal for this client is to show historical data – like the data rates over the past few minutes.

If we were writing this type of an application in C, we might write something like this:

```
int dataValues[100];

...
addValue(int newValue) {
int i, j;
# Shift the values down, and add new value to the end of the array.

    for (i=0, j=1; i<99; i++, j++) {
        dataValues(i) = dataValues(j)
    }
    dataValues(99) = newValue
}
```

If we needed to, we could write similar code in Tcl to shift values in an associative array. However, if you are worried about performance, you should avoid large data-moving loops in interpreted languages.

One of the features of the BLT vector command is that you can delete an element at the beginning of the vector, and the rest of the values will shift to fill the empty space. You can add a new value to the end of a BLT vector with the special purpose ++end index. The Tcl code for sliding all the datavalues over and adding a new value resembles:

```
unset  dataValues(0)
set    dataValues(++end) $newValue
```

The previous example tracked the current values for number of bytes transmitted, received, and the number of collisions. To do this, we used a naming convention for the vectors we defined.

In order to get a list of the available vectors with the info globals command, we started each variable name with the characters DataVector, and then appended information about the type of data, and the device associated with this data. This leads to names like DataVector_x_eth0 and DataVector_y_eth0.

As a rule, it's better to organize data with an associative array, rather than a lot of individual variables. We can't quite do this when working with BLT vectors, since each vector must have a unique name.

What we can do is to use an associative array to hold the vector names. This means we don't need to worry about making sensible vector names. Instead, we can worry about making sensible array indices, which is actually an easier process.

When we create vectors, we need a unique name for each vector. The last version of this example used a clever naming convention involving the names of the devices and signals. This version uses a simple naming convention for the vectors (vector0, vector1, etc.) and lets the array index hold the information about the interface and type of data.

To generate the unique vector names, we need some way of creating unique numbers. The traditional technique is to use a variable and increment it after we create each vector. We can also use Tcl's ability to define procedures with a default argument and redefine procedures on the fly to create unique numbers when we need them.

This little gem was developed by Richard Suchenwirth. His original version of the code (along with many other clever little programming nuggets) is at *<http://mini.net/cgi-bin/wikit/526.html>*.

```
proc uniq {{val 0}} {
    incr val;
    proc uniq "{val $val}" [info body uniq]
    return $val
}
```

Using the associative array to hold the unique vector names, we can generate our list of vectors with code like this:

```
# Generate a set of unique vector names, and
# save those names in the DataVectors associative array

array set DataVectors [list \
    $name.y vector[uniq] \
    $name.y.rcv vector[uniq] \
    $name.y.xmt vector[uniq] ]

# Declare each of the new vectors to exist in global scope

foreach v [array names DataVectors $name*] {
    global $DataVectors($v)
}

# Create the new vectors

::blt::vector $DataVectors($name.y)(5)
::blt::vector $DataVectors($name.y.rcv)(200)
::blt::vector $DataVectors($name.y.xmt)(200)

# And initialize the history vectors

for {set i 0} {$i < 200} {incr i} {
    set $DataVectors($name.y.rcv)($i) 0
    set $DataVectors($name.y.xmt)($i) 0
}
```

The vector initialization lines may look strange if you know that Tcl does not support multidimensional arrays. The line set $DataVectors($name.y.rcv)($i) 0 looks like DataVectors is a two-dimensional array. What is actually happening is that the Tcl substitution phase is reading that line, and converting $DataVectors($name.y.rcv) to vector1, and actually invoking the set command as set vector1(0) 0; set vector1(1) 0, etc.

This set of code initializes a set of vectors for Y coordinates, but ignores the X coordinates for the bars.

By default, when two bars are defined to display at the same X location, the BLT widget will display the bars in the order they were defined, placing the latter bars on top of the earlier bars. This works when your data is scaled such that the latter sets of data always have smaller values than the earlier sets.

The barchart widget can also be configured to stack one set of data on top of the other, or to make the bars narrower and place them next to each other.

Using the -barmode aligned option allows us to create a single X vector for each barchart when the program starts, rather than creating a special X vector for each device we are watching.

Using these techniques, we can build the barchart for the current data with this code:

```
# Build a barchart for current data

set Client(barChart) [::blt::barchart .bcht -width 600 \
    "Network Activity" -barmode aligned]

::blt::vector dataXvector(5)
::blt::vector historyXvector(200)

for {set i 0} {$i < 5} {incr i} {
    set dataXvector($i) $i
}

for {set i 0} {$i < 200} {incr i} {
    set historyXvector($i) $i
}
```

Using five bars for the receive, transmit, receive errors, transmit errors and collisions is fine, but having just five sets of historical data is close to useless. We want to be able to scan several minutes of data, at least.

In order to display a few hundred bars, and make them more than a few pixels wide, we need a widget several thousand pixels wide. With the current monitor limit of 1600 pixels, this would limit us to 160 bars at 10 pixels wide. This is a bit under three minutes worth of history if we sample every two seconds.

While the BLT barchart widget cannot be directly linked to a scrollbar, it can be displayed within a canvas that has been attached to a scrollbar, as discussed at the beginning of this article.

This code will create a canvas 600 pixels wide, and pack an 8000-pixel-wide barchart into it.

```
# Build canvas, and then place the history barchart inside it

canvas .historyCvs -height 120 -width 600 -xscrollcommand {.histSB set}
grid .historyCvs -row 3 -column 0 -columnspan 4

scrollbar .histSB -orient horizontal -command {.historyCvs xview}
grid .histSB -row 4 -column 0 -columnspan 4 -sticky ew

set Client(historyChart) [::blt::barchart .historyCvs.bhist \
    -width 8000 \
    -height 150 \
    -barmode aligned \
    -title "" \
    -barmode aligned]
.historyCvs create window 1 1 -window .historyCvs.bhist -anchor nw
```

The BLT barchart widget supports drawing bars in various colors and stipples. For an application running on a color monitor, using colors to distinguish the bars is a good idea. Images in a magazine work better in black and white with stipple patterns. To display the transmit-and-receive data, I decided to invert the colors on a stipple pattern.

The Tcl associative array makes a handy lookup table to convert one color to the inverse. With this code, if the foreground color is white, then the inverse is $inverse(white) which is set to black.

This code creates two new barchart elements for a device, one for the transmit data and one for the receive data.

```
set inverse(black) white
set inverse(white) black
set color white

foreach dir {rcv xmt} {
    $Client(historyChart) element create ${dir}_$name -label "" \
        -stipple [lindex $Client(stipple) $Client(count)] \
        -fg $color -bg $inverse($color) -borderwidth 0 \
        -xdata historyXvector -ydata $DataVectors($name.y.$dir)
    set color $inverse($color)
}
```

These pieces of code, added to what was already in the client, will create the new bar-chart and new elements. The final piece is to put data into the history vector.

The Linux /proc/net/dev pseudo file reports data as running totals. What we want to display is the data for a single time period. We can handle this by saving the previous running total and subtract the current running total to get the per-interval value.

The associative array is a good way to save the previous values, and the Tcl info exists command will let us check to see if our program has a previous total to work with.

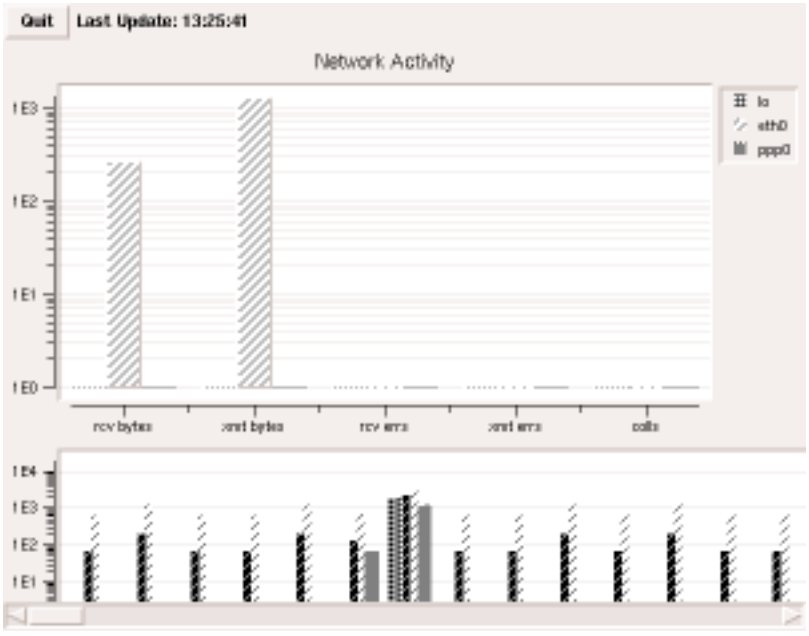
This code extracts the receive and transmit values from the raw data, converts the running total to the amount of data moved in the last interval and appends that value to the end of the appropriate history vector:

```
foreach direction {rcv xmt} \
            val [list [lindex $line 1] [lindex $line 9]] {
    if {[info exists Client($name.$direction)]} {
        set v2 [expr $val - $Client($name.$direction)]
        unset $DataVectors($name.y.$direction)(0)

        # Workaround to force vector to move values needed for some revisions
        #  of BLT
        set $DataVectors($name.y.$direction)(0)

        set
$DataVectors($name.y.$direction)(++end) $v2
    }
set Client($name.$direction) $val
}
```

When this is all done, the client generates images that look like this:



The code for this client and the code for the servers is available at <*http://www.noucorp.com*>.