;login:

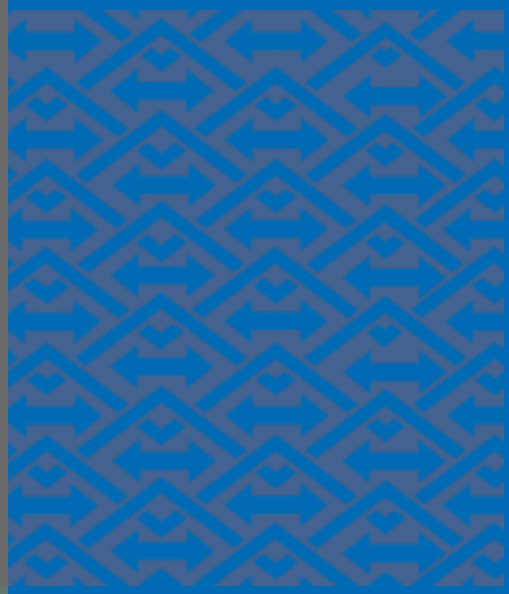
THE MAGAZINE OF USENIX & SAGE

December 2000 • volume 25 • number 8

inside:

PROGRAMMING:

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>

We don't often discuss subatomic physics and systems design in the same breath, but Heisenberg described a shared problem when he pointed out that you can't observe a system without affecting it.

For example, how heavily can you monitor your servers before you spend more cycles running monitors than you spend serving? Back in the dark ages, if more than five of us used the system monitor on our PDP-11/34, the compiles we were watching would never finish.

When I want to monitor a system, I like a graphical display. A picture is worth more than a thousand words when it's presenting numeric data. On the other hand, I think windowing systems and servers go together like kim-chee and ice cream. Windowing systems chew up too many resources that an overloaded server needs. (It never seems to matter how over-spec'ed the server was – it will end up overloaded.)

So, after some thought, I decided that I had more spare network bandwidth on the LAN than I had spare CPU cycles on the server, and the obvious solution was to run a small daemon on the server to spew raw data to another box where I had the cycles to do graphic displays and fancy analysis.

Since I have even fewer spare cycles than the overloaded server, I decided to write the client/server pair in Tcl. Writing a daemon in an interpreted language to save CPU cycles may seem strange, but the Tcl-based server chews up less than 0.1% of the CPU cycles on a 300Mhz K6, and even my server can afford that much overhead.

A simplified flow for a TCP/IP server is:

1. Initialize the server.
2. Wait for a connection from a client.
3. Validate the client.
4. Converse with the client.
5. Close the connection.
6. Return to wait state.

For these simple system monitors, the conversation is one-sided – the server sends data and never listens.

The key to this client/server pair is the socket connection. The BSD socket libraries are a well-designed set of subroutines, but they aren't always the simplest things to work with. When TCP/IP socket support was added to Tcl, the BSD libraries were hidden behind new Tcl commands and the socket interface was simplified.

The loss of functionality in this simplification is that the base Tcl interpreter supports TCP sockets, but not UDP sockets. If you need more complete socket support, look at the TclDP extension.

The command for creating a socket connection is socket. This command can open a client socket to a server, or open a server-style socket that will wait for a connection from a client.

The syntax for opening a client-side connection is very simple:

Syntax: *socket ?options? host port*

socket	Open a client socket connection.
?options?	Options to specify the behavior of the socket.

A server-side socket needs to run in an asynchronous mode, handling connection requests and other events as they occur.

-myaddr addr	Defines the address (as a name or number) of the client side of the socket. This is not necessary if the client machine has only one network interface.
-myport port	Defines the port number for the server side to open. If this is not supplied, then a port is assigned at random from the available ports.
-async	Causes the socket command to return immediately, whether the connection has been completed or not.
host	The host to open a connection to. May be a name or a numeric IP address.
port	The number of the port to open a connection to on the host machine.

For most applications, you can ignore the options, and the command to open a socket is just:

```
set channel [socket $address $port]
```

The socket command can accept addresses as a numeric IP address or as a system name. For instance, you can open a socket as:

```
set channel [socket 127.0.0.1 $port]
set channel [socket www.noucorp.com $port]
set channel [socket localhost $port]
```

Once this is done, you can read and write from the channel using the Tcl gets, read, and puts commands.

A server-side socket needs to run in an asynchronous mode, handling connection requests and other events as they occur. The Tcl interpreter commonly implements asynchronous processing with call-back procedures, and the socket command is no exception to this convention.

Syntax: *socket -server procedureName ?options? port*

socket -server	Open a socket to watch for connections from clients.
procedureName	A procedure to evaluate when a connection attempt occurs. This procedure will be called with three arguments: the channel to use for communication with the client the IP address of the client the port number used by the client.
?options?	Options to specify the behavior of the socket.
-myaddr addr	Defines the address (as a name or number) to be watched for connections. This is not necessary if the client machine has only one network interface.
port	The number of the port to watch for connections.

The code to establish a server-side socket looks like this:

```
socket -server initializeSocket $port
```

The procedure that gets called when a socket is opened (in this case, initializeSocket) does whatever setup is required. This might include client validation, opening connections to databases, configuring the socket for asynchronous read/write access, etc.

When tclsh opens a channel, the default is to open it as a buffered stream. Socket connections are also buffered. This can lead to some surprises when your script executes a puts but no data appears at the other end of the socket. What happened was that the data went into the buffer, but the buffer wasn't full. The socket code is waiting for the buffer to fill before any data is actually moved across the connection.

There are two ways of solving this problem:

- Flush the buffer after each write operation with a flush command.
- Change the style of buffering on the channel with the fconfigure command.

The simplest way to solve the problem is to follow each puts with a flush command. This works fine on small programs but gets cumbersome on larger projects.

Syntax: *flush channelId*

flush	Flush the output buffer of a buffered channel.
channelId	The channel to flush.

For example:

```
set smtpSocket [socket localhost 25]
puts $smtpSocket "helo clif@noucorp.com"
flush $smtpSocket
```

The better way to solve the buffered I/O problem is to figure out what style of buffering best suits your application and configure the channel to use that buffering.

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	The newValue argument may be set to:
full:	the channel will use buffered I/O
line:	the buffer will be flushed whenever a full line is received
none:	the channel will flush whenever characters are received.

By using fconfigure to set the buffering to line mode, we don't need the flush after each puts command.

```
set smtpSocket [socket localhost 25]
fconfigure $smtpSocket -buffering line
puts $smtpSocket "helo clif@noucorp.com"
```

We now know how to configure the socket, but there is one more trick to using Tcl to create a server. The normal tclsh script is read and evaluated from top to bottom, and then the program exits. This behavior is fine for a filter-type program, but not good for a server. We really need the server to sit in an event loop and wait for connections, process data, etc.

The vwait command causes the interpreter to wait until a variable is assigned a new value. While it's waiting, it processes events.

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.

Before we can build a simple server that will send the current disk usage to its clients, we need two more Tcl commands to collect the data and schedule reports.

The **exec** command extends Tcl's functionality by letting a script use any program on the system to generate data.

The two commands that collect data and schedule the distribution are the after command, which lets you schedule processing to happen in the future, and the exec command, which will execute a command outside the Tcl interpreter and return the results (similar to the shell back-tic operator).

The after command has two flavors. It can pause the script execution for a time period, or it can schedule processing to happen some time period in the future. This server uses the second flavor to schedule transmitting disk usage information after 1 second (1000 milliseconds), and every 2 seconds after that.

Syntax: *after milliseconds ?script?*

after Pause	Pause processing of the current script, or schedule a script to be processed in the future.
milliseconds	The number of milliseconds to pause the current processing, or the number of seconds in the future to evaluate another script.
?script?	If this argument is defined, this script will be evaluated after milliseconds time has elapsed.

The exec command extends Tcl's functionality by letting a script use any program on the system to generate data. For a UNIX disk-space monitor, we can run the df command. On a DOS/Windows platform, we can run command.com /C dir and just grab the last line.

Syntax: *exec ?-options? arg1 ?arg2...argn?*

exec		Execute arguments in a subproccess.
?-options?		The exec command supports two options:
	-keepnewline	Normally a trailing newline character is deleted from the program output returned by the exec command. If this argument is set, the trailing newline is retained.
	--	Denotes the last option. All subsequent arguments will be treated as subprocess program names or arguments.
arg		These arguments can be either a program name, a program argument, or a pipeline descriptor.

Finally, here is the code for a monitor program to report disk usage:

```
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
```

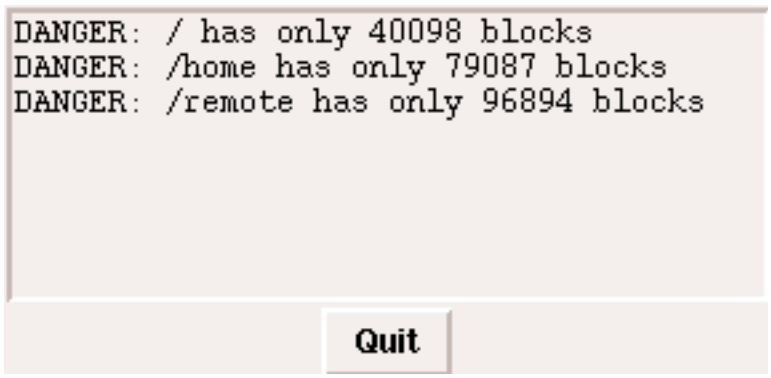
When this server is running, you can connect to it with Telnet and see the current disk usage reported on your screen every 2 seconds.

Telnet is a good way to check that a server is working, but we can do much better than scrolling lines of text across the screen.

One simple client for this application is a text widget with a small amount of smarts about what to display:

```
# Open a client socket on the local system (for testing purposes.)

set input [socket 127.0.0.1 55555]

# Create a small text window.

text .t -height 8 -width 40
pack .t

# And a quit button

button .b -text "Quit" -command "exit"
pack .b
while {[gets $input line] > 0} {

    # Parse the line into a set of variables.

    foreach {name size used avail pct mount} $line {}

    # A check to see if this is a new set of df data.
    # There will be a duplicate name when we start a new dataset.

    if {[info exists Data($name)]} {
    unset Data
    .t delete 0.0 end
    }

    # Get rid of the '%' symbol in the percent usage field

    regsub "%" $pct "" pct

    # Only report if disk is more than 95 percent full.

    if {$pct > 95} {
    .t insert end "DANGER: $mount has only $avail blocks\n"
    }
    set Data($name) $input
    update
}
```

Running this client/server pair will generate a display that looks like this:



This example ignores little details like confirming which clients are allowed to retrieve info about our disk usage; the display has no historical data, and shutting down the client generates an error in the server.

I'll start filling in those holes in the next Tclsh Spot article. Until then, this code and a more full-featured client/server skeleton are available at <*http://www.noucorp.com*>.