;login:

THE MAGAZINE OF USENIX & SAGE

October 2001 • Volume 26 • Number 6

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 telnet client that would report the initial configuration options. This article will expand the sniffer into a client that can interact with a server, maintain its internal state, and respond to various commands the server can send. In the course of this, I'll demonstrate some things we can do with the Tcl namespace.

The Tcl namespace command provides a private, named area in a Tcl script where data and procedures can exist without interfering with other data and procedures that might have the same names. A Tcl namespace can implement most of the capabilities of a Java or C++ class.

A Tcl script is most useful when it's merged into other Tcl code. You can merge one Tcl script into another with the source command which loads a script into a Tcl program, and evaluates the commands in that script before evaluating the next line of the original script.

Syntax: source *fileName*

This is similar to the C language #include or the C-shell source command. This is a simple technique, and it works well for many applications.

However, if you source two packages that have overlapping names for variables or procedures, the last package you load will overwrite the procedure body or data values set by the first package.

We can use the Tcl namespace command to create a private, named area in our telnet client where data and procedures can exist without interfering with other data and procedures that might have the same names.

A namespace is created with the namespace eval command:

Syntax: namespace eval *namespaceID arg ?args?*

namespace eval	Create a namespace, and evaluate the script arg in that scope. If more than one arg is present, the arguments are concatenated together into a single command to be evaluated.
namespaceID	The identifying name for this namespace.
arg ?args?	The script or scripts to evaluate within namespace namespaceID.

The variables defined within a namespace will last until the namespace is destroyed by the namespace delete command. This makes a namespace an ideal place to keep a package's internal state.

The state information for the telnet sniffer application was held in a global associative array. A telnet namespace with that array included in it can be created with code like this:

```
namespace eval telnet {
    variable Telnet
```

The variable command declares a Tcl variable within a namespace. When the variable command is used outside a procedure, it creates a variable within a namespace and initializes it to an optional value. When the variable command is used within a procedure, it maps a variable from that namespace scope into the procedure's local scope.

Syntax: variable *varName ?value? ?var2? ?val2?*

varName	The name of the variable to create, or map into local scope.
value	An optional value for this variable. If the variable already has a value, the new value will overwrite the old.

The sniffer program used two global arrays, the telnet array that held the state information, and the Lookups array that was used to map from hex values to human-friendly information strings. The Lookups array was created at run time by scanning the telnet.h include file and massaging the #define xx yy lines into Tcl array assignments in a simple procedure.

Any Tcl code can be evaluated within the namespace eval body. We can evaluate the readIncludeFile procedure to create the Lookups array within the telnet namespace, and can even use the source command to load the script that includes this procedure into the namespace.

```
namespace eval telnet {
    variable Telnet

    source readincl.tcl

    readIncludeFile Lookups /usr/include/arpa/telnet.h
    set Lookups(UNKNOWN.-1) "Unknown Option"
```

By sourcing the readincl.tcl script within the namespace, the readIncludeFile procedure is defined within the namespace. This procedure is available for use within the namespace but is not easily visible from the outside world.

Tcl namespaces are a tree-structured construct, like a file system or graphics windows. Where a POSIX-style file system uses a slash to separate parent from child directory, namespaces use a double colon to delimit parent and child namespaces.

When Tcl is started, the default, top level, namespace is ::. The namespace eval telnet {...} command creates a new namespace ::telnet.

Unlike Java or C++ classes, there is no privacy in a Tcl namespace. The Tcl philosophy is to make as much information as possible available to the programmer. Thus, you can always access a member of a Tcl namespace by its full namespace pathname.

We can define a procedure within the telnet namespace with a normal looking Tcl command like:

```
namespace eval telnet {
    ...
    proc openSocket {address} {
        variable Telnet
        set Telnet(socket) [socket $address 23]
    }
}
```

We could invoke the procedure from outside the namespace with a command like:

```
::telnet::openSocket 127.0.0.1
```

Java and C++ let the application programmer know the private and public API by restricting access to non-public methods. Since any procedure within a namespace can be invoked from outside the namespace, we need a mechanism to let the programmer know which are public and which are private methods.

One convention used within Tcl is that public methods start with a lowercase letter and private methods start with an uppercase letter. The rationale is that it takes an extra keystroke to type an uppercase letter. This forces programmers to recognize that they are violating the package's intended use by calling this procedure.

The Tcl namespace also includes a namespace export command to declare which procedures are part of the external API.

Including this line in the namespace eval body tells the application programmer that the external API for this namespace is the openSocket, binarySend, and telnetEvent procedures.

```
namespace export openSocket binarySend telnetEvent
```

Along with using a namespace to hide variables, we sometimes use a namespace to hold just procedures. This ensures that we don't have procedure-naming collisions when our application sources several files.

We could define a namespace that has functions to convert strings of binary data to hex digits like this:

```
namespace eval binaryStrings {
    namespace export bin2hex hex2bin

    proc bin2hex {binaryString} {
        binary scan $binaryString "H*" hexData
        regsub -all {..} $hexData {\0 } hexList
        return $hexList
    }

    proc hex2bin {string} {
        regsub -all " " $string "" string
        set line [binary format "H*" $string]
        return $line
    }
}
```

One of the strengths of the Tcl namespace is that they can be nested. This implements the OO composition (or "has-a") feature. For example, the binaryStrings namespace can be embedded in the telnet namespace with code like:

```
namespace eval telnet {
    source binaryStrings.tcl
    ...
}
```

Just as file systems support absolute naming by starting from the root file system (/usr/local/bin/tclsh) or relative naming, by starting from the current directory (subdir/file), Tcl namespaces can be accessed by either absolute or relative names.

The procedures in binaryStrings can be invoked from within the telnet namespace as: ::telnet::binaryStrings::hex2bin or binaryStrings::hex2bin.

Since the telnet namespace may be included in another namespace, it's best to use relative naming when embedding one namespace within another.

Alternatively, since the public API for the binaryStrings namespace is exported, we can use the namespace import command to import those procedures into the current namespace. This would let us invoke the procedures without any namespace prefix.

```
namespace eval telnet {
    source binaryStrings.tcl
    namespace import binaryStrings::*
    ...
    set hex [bin2hex $binaryValue]
}
```

Using the namespace import command undoes the protection from procedure-name collisions that we got from using namespaces, but within a controlled environment (like a namespace), the namespace import command can make code easier to read.

These techniques let us set up a telnet namespace for the sniffer that was developed in the last article. In order to handle real telnet client-server interactions, the script needs to be able to handle the negotiation commands and retain state information about the supported and unsupported options.

The telnet protocol includes a lot of subnegotiation options ranging from a need to echo characters to supporting defining data rates and terminal size.

Implementing these options follows a pattern – they have a current value and react to demands that the option be supported or not supported.

The reaction to a command varies depending on the state of that option. For example, if an option's status is to be supported, and the server sends a message to not use that option, the client should send a return message that the option won't be used. However, if the option was already turned off, no reply should be sent.

We could implement this with a set of objects that include the current state information and procedures to report the contents. We could use one object for each option, and use Tcl's ability to nest namespaces to hold all the option namespaces with the telnet namespace.

If we were writing this in a true object-oriented language like C++ or Java, we might have a base class for options and two derived classes for supported and unsupported options.

We can implement the inheritance (or "is-a") feature of true OO languages with the namespace command by using a base command to initialize a namespace and another set of commands to set the personality of the class.

In C++ terms, the script used to initialize the namespace is the base class, and the personality commands implement the methods in the derived class.

We can use the hex values of the commands to name the procedures in the option namespaces, just as we did in the sniffer program. This makes the parsing simpler (let Tcl do it). Because we have separate namespaces for the various procedures (fb, fd, 01, etc.), they don't conflict with each other or with the procedures sharing that name in the telnet namespace.

We can define the names of the namespaces on the fly. To make the code easier to write (if a bit more cryptic for reading), I'm using a naming convention of XX.TELOPT for the option namespaces, where XX is the hex value of the option number.

The code to define namespace objects for unsupported features looks like this:

```
set baseClass {variable clientValue %s supported %s;}

foreach cant {00 05 21 22 23 24 25 26 27 } {
    namespace eval $cant.TELOPT [format $baseClass "" 0]
    namespace eval $cant.TELOPT [format "proc fb {} {return fffe%s}" $cant]
    namespace eval $cant.TELOPT [format "proc fd {} {return fffc%s}" $cant]
    namespace eval $cant.TELOPT  "proc 01 {} {return {}}"
    namespace eval $cant.TELOPT [format \
        {proc fe {} {variable supported;
            if {$supported} {
                return fffc%s
            } else {
                set supported 0;
                return {}}
        }} $cant]
}
```

For supported options, it's a bit more complex, since some options have data that must be reported when requested. Here's code that will create objects for the supported options.

```
foreach {can initialVal}   {18 "dumb"
                            1f "0x00500018"
                            20 "57000,57000"
                            03 "01"
                            01 "01"} {
    namespace eval $can.TELOPT [format $baseClass $initialVal 1]
    namespace eval $can.TELOPT [format "proc fd {} {return fffb%s}" $can]
    namespace eval $can.TELOPT [format "proc fb {} {return fffd%s}" $can]

    if {[string first 0x $initialVal] == 0} {
        set hex [string range $initialVal 2 end]
    } else {
        binary scan $initialVal H* hex
    }

    proc $can.TELOPT::01 {dummy} [format "return fffa%s00%sfff0" $can $hex]

    namespace eval $can.TELOPT {proc 00 {val} {variable serverValue;
        set serverValue $val}}

    namespace eval $can.TELOPT [format \
        {proc fe {} {variable supported;
            if {$supported} {
                return fffc%s
            } else {
                set supported 0;
                return {}}
        }} $can]
}
```

The code to support the fb, fc, fd, and fe commands (WILL, WON'T, DO, and DON'T) in the telnet namespace are fairly simple. They look like this:

```
# WILL command fb : Reply DO(fd) or DONT(fe)
proc fb {subl textName oobName} {
    variable Telnet
    upvar $oobName oob
```

```
    set opt [lindex $subl 0]
    AddInfo $subl TELOPT_
    append oob [eval $opt.TELOPT::fb]

    return 1
}
```

The fa (SUBNEGOTIATION) command is a bit trickier, since the negotiation may be a request for the value, or a value being supplied.

The format for this command is either

ff (IAC) fa (SB) 1 (request data) ff (IAC) f0 (SE)

or

ff (IAC) fa (SB) 0 (provide data) DATAVALUES ff (IAC) f0 (SE).

The code to implement the fa command is:

```
proc fa {subl text oobName} {
    variable Telnet
    upvar $oobName oob
    # Starts with character after the 'fa'
    Debugputs "Dealing with: [lrange $subl 0 20]"
    set count 1
    set type [lindex $subl 0]
    set action [lindex $subl 1]
    foreach {p2 data} [FindFFF0 $subl] {}

    append oob [$type.TELOPT::$action $data]
    AddInfo $subl TELOPT_
    return $p2
}
```

That covers the interesting parts of this script. As usual, the full source code (about 450 lines) is available at *http://noucorp.com/cgi-bin/noucorp/generic.tcl?dir=/home/httpd/html/tcl/login.*