;login:
THE MAGAZINE OF USENIX & SAGE
April 2003 • volume 28 • number 2

inside:

PROGRAMMING
Flynt: The Tclsh Spot

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 two Tclsh Spot articles described building a Tcl extension to push packets onto a network, and using the Spirent AX-4000 to characterize the rate at which packets could be sent.

We generated the IP packet by using tcpdump to sniff a packet and by copying the data into the packet-generating program. This was adequate to test that the program could work but is not versatile enough for using the packet generator to test other systems.

Modern networks use a "Russian doll" paradigm to define packets of information. A small packet is embedded in a larger packet, which is embedded in a still larger packet. And, like the dolls, each packet looks much like the others, but with subtle differences.

For example, a DNS message contains an identification field, some flags, and a payload of a set of questions and answers. This packet is enclosed in a UDP packet that includes source and destination port identifiers and a payload that consists of the DNS packet. The UDP packet gets enclosed into an IP packet, with source and destination IP addresses as identifiers and a payload of the UDP packet. And the IP packet will finally get enclosed in an Ethernet or PPP packet, with machine identifiers and a payload that consists of the IP packet.

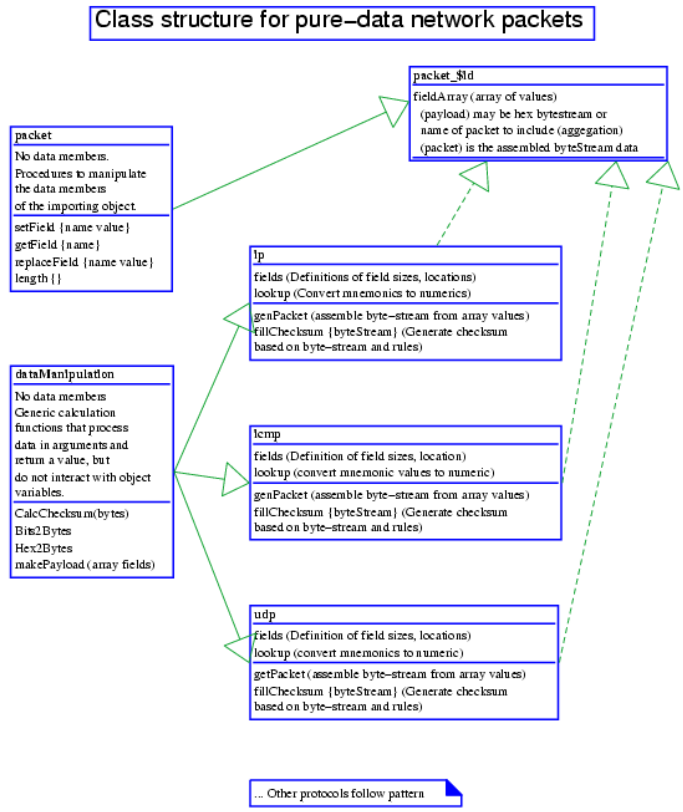
From a software design point of view, a set of similar entities with small differences is a classic situation for an object-oriented design. You can construct a base object with the core functionality and then derive special cases for the unique features.

The [incr Tcl] Tcl extension provides support for full-featured object-oriented programming, with classes, protection, inheritance, etc. For applications that require a rigorous OO implementation, this is an ideal solution. The standard Tcl distribution (ActiveTcl from *http://www.activestate.com*) includes [incr Tcl].

One of the Tcl features that makes [incr Tcl] possible is the namespace command (the [incr Tcl] extension introduced the namespace command to Tcl). The Tcl namespace provides hooks to do OO-style programming in Tcl. Using only the standard namespace command, you can implement inheritance, aggregation, and simple protection.

For this application, where I expect to have a relatively small number of object types (types of data packets – IP, ICMP, TCP, etc.) and a large number of objects (as many data packets as I need), I elected to use a very lightweight model in which the packet-type objects contain static data and methods, while the packet objects contain only data.

A Class/Object diagram of this design would resemble this:



The data objects include a single variable – an associative array indexed by the names of the fields within a packet using a particular protocol. The data is the value for that field. For example, an object named packet_1 might be an ICMP packet, which would have indices like type, code, and checksum (the minimal ICMP header fields).

One extra index is used to hold a bytestream representation of the packet, with the appropriate ordering and padding. This index is named packet.

The data objects always inherit a few basic methods from the packet class, which has no associated data, and inherit public methods and data from one *parent* class that describes the protocol.

The parent classes contain information to define the layout of fields within a packet of this type, a lookup table to convert from common mnemonics for this type of packet to a numeric value, and methods that implement the specific rules for building this type of packet. These classes invoke generic functions in the dataManipulation class as they need them.

This set of abstractions can be implemented with just the namespace command – no need for fancy OO extensions. While 700 lines of code to build ICMP, TCP, UDP, IP, and Ethernet packets is rather small, it's too much to completely discuss in this article. This article will introduce the interesting features of using the namespace command to implement this, and show how to test the output.

The Tcl namespace command provides a private area where static data and procedures can be kept. The same data and variable names can be used in multiple namespaces. Tcl namespaces are created and manipulated with the namespace command, which includes several subcommands. The namespace eval command creates new namespaces. It will evaluate a Tcl script within a namespace (and create the namespace if necessary.)

Syntax: namespace eval *namespaceID arg1 ?argN...?*

Create a namespace, and evaluate the script arg in that scope. If more than one arg is present, the arguments are concatenated into a single script to be evaluated.

namespaceID The identifying name for this namespace.

*arg** The script or scripts to evaluate within namespace *namespaceID*.

This code will create a namespace that holds an associative array, and includes two functions to set and retrieve array values.

```
namespace eval packet_1 {
    variable fieldArray

    proc setField {name value} {
        variable fieldArray
        set fieldArray($name) $value
    }

    proc getField {name } {
        variable fieldArray
        return $fieldArray($name)
    }
}
```

The variable command declares that a variable exists within a namespace and may initialize the variable's value. The variables declared with the variable command are persistent and will not be destroyed when the namespace scope is exited. These variables are easily accessed by procedures within their namespace, but not from other namespaces.

Note that the syntax for the variable command is different from the global command. The variable command supports setting an initial value for a variable, while the global command does not.

Syntax: variable *varName ?value? ?varNameN? ?valueN?*

Declare a variable to exist within the current namespace. The arguments are pairs of name and value combinations.

varName The name of a variable.

?value? An optional value for the variable.

The first variable fieldArray declares that the variable exists. The next variable fieldArray, inside the setField procedure, maps the fieldArray variable from the namespace scope into the local procedure scope.

The variable fieldArray is persistent, just like global variables, but exists in the packet_1 object.

Tcl namespaces are named in a tree fashion, similar to a directory tree. The separator for namespaces is a double-colon (::), and the top namespace (the default when you start up a Tcl shell) is also ::.

The first example creates a namespace named packet_1 in the scope where the command was evaluated. Assuming it is evaluated in the top-level scope, it creates the namespace ::packet_1, which contains a variable ::packet_1::fieldArray and two procedures, ::packet_1::getField and ::packet_1::setField.

Invoking the packet object's procedures by full namespace path resembles the C++/Java type of naming convention: packet_1::getField resembles packet_1->getField. This isn't really the best technique for accessing methods in Tcl. This style of invoking the procedures exposes the implementation of the object and makes the packet_1::getField invocation appear to be a unit, instead of the packet_1 being an object and getField being a method.

A Tcl-style object would be a command, and the methods would be subcommands.

Since Tcl has separate resolution tables for namespace names, variable names, and procedures, we can use the same name for a namespace, and the procedure to access it. This script will create a packet_1 command:

```
proc packet_1 {args} {namespace eval packet_1 $args}
```

When a variable named args is the last argument in a procedure definition, Tcl will assign any unassigned arguments to that variable. This allows you to define procedures that can be invoked with any number of arguments.

We can create namespaces with any script. For instance, this is equivalent to the first example:

```
set packetDef {
    variable fieldArray

    proc setField {name value} {
        variable fieldArray
        set fieldArray($name) $value
    }

    proc getField {name } {
        variable fieldArray
    return $fieldArray($name)
    }
}

namespace eval packet_1 $packetDef
proc packet_1 {args} {namespace eval packet_1 $args}
```

This makes it easy to create multiple packet namespaces, each of which includes its own copy of the fieldArray variable, which may contain unique values.

```
proc makePacket {name} {
    global packetDef
    namespace eval $name $packetDef
    proc $name {args} "namespace eval $name \$args"
}

makePacket packet_1
makePacket packet_2
makePacket packet_3
```

A downside to this technique is that the procedures are also duplicated in each namespace. While the variables in these namespaces are unique, the procedures are identical, and we don't need a new copy in each namespace. With three objects this doesn't matter, but if our application had several thousand objects, the overhead of duplicating the procedures could start to be a problem.

The namespace import and namespace export commands solve this problem (and several others we'll address later).

The namespace export command lists commands that are available to be imported. You would consider these public methods in a Java or C++ environment.

Syntax: namespace export *pattern1 ?patternN...?*

Export members of the current namespace that match the patterns. Exported procedure names can be imported into other scopes. The patterns follow glob rules.

*pattern** Patterns that represent procedure names and data names to be exported.

The flip side is the namespace import command, which will map a procedure that exists in one namespace into the current namespace.

Syntax: namespace import *?-force? ?pattern1 patternN...?*

Imports procedure names that match a pattern.

-force If this option is set, an import command will overwrite existing commands with new ones from the pattern namespace. Otherwise, namespace import will throw an error if a new command has the same name as an existing command.

*pattern** The patterns to import. The pattern must include thenamespaceID of the namespace from which items are being imported.

One trick to the import command is that it maps the procedure name into the current namespace, but when the procedure is evaluated, it evaluates within the namespace it was defined in. This makes the namespace import command work well for implementing inheritance, but makes a slight problem when we try to use it naïvely to avoid duplicating procedure bodies.

The code below doesn't work.

It sets values in ::BADprocs::fieldArray, rather than ::packet_1::fieldArray or ::packet_2:fieldArray.

```
namespace eval BADprocs {
    namespace export setField getField

    proc setField {name value} {
        variable fieldArray
        set fieldArray($name) $value
    }

    proc getField {name } {
        variable fieldArray
        return $fieldArray($name)
    }
}
```

```
set packetDef {
    variable fieldArray
    namespace import ::BADprocs::*
}

namespace eval packet_1 $packetDef
namespace eval packet_2 $packetDef

proc packet_1 {args} "namespace eval packet_1 \$args"
proc packet_2 {args} "namespace eval packet_2 \$args"

packet_1 setField foo bar1
packet_2 setField foo bar2
```

The Tcl upvar command is the solution for this. Upvar will map a variable from a higher level scope into the current scope.

Syntax: upvar *?level? varName1 localName1 ?varName2? ?localName2?*

	Maps a variable from a higher variable scope into the current variable scope.
?level?	An optional level to describe the level from which the variable should be linked. This value may be a number or the # symbol followed by a number. The *level* defaults to 1, the level of the script that invoked the current proc.
*varName**	The name of a variable in the higher scope to link to a local variable.
*localName**	The name of a variable in the local scope. This variable can be used in this script as a local variable. Setting a new value to this variable will change the value of the variable in the other scope.

Normally, the upvar command is used to implement call-by-name (instead of Tcl's normal call-by-value paradigm). For example, a procedure to print the contents of an array would resemble this:

```
proc printArray {arrayName} {
    upvar $arrayName a
    foreach index [array names a] {
        puts "$index: $a($index)"
    }
}

array set demo {index1 val1 index2 val2}
printArray demo
```

When doing object-style programming in Tcl, we can use the upvar to map the fieldArray variable from the packet_* namespaces into the procs namespace like this:

```
namespace eval procs {
    namespace export setField getField

    proc setField {name value} {
        upvar fieldArray localArray
        set localArray($name) $value
    }

    proc getField {name } {
        upvar fieldArray localArray
        return $localArray($name)
    }
}

set packetDef {
    variable fieldArray
    namespace import ::procs::*
}

proc makePacket {name} {
    global packetDef
    namespace eval ::$name $packetDef
    proc $name {args} "namespace eval $name \$args"
}

makePacket packet_1
makePacket packet_2

packet_1 setField foo bar1
packet_2 setField foo bar2
```

Note how the namespace eval command in makePacket prepends the :: to the namespace identifier. Like file-system paths, namespace identifiers may be absolute or relative. The namespace identifier ::packet_1 defines a namespace that is a child of the global scope. The namespace identifier packet_1 defines a namespace that is a child of the current scope, which could be anything.

Creating a procedure packet_1 that uses namespace eval packet_1 setField rather than directly invoking packet_1::setField creates an entry on the procedure stack for code evaluated in the packet_1 namespace. This allows the upvar command to map the fieldArray into the ::procs namespace. If you intend to invoke methods as namespace::method, you'll need to define the methods within the namespace rather than importing them.

This technique can be generalized further to create class and new commands, just like those in Java, C++, or [incr Tcl]. Several pure Tcl object-oriented programming packages such as stooop and SNIT use similar techniques. Since this application only creates one type of object (data packets), the simple makePacket command is sufficient.

Using this technique, we can define packets that contain named fields with numeric values with code like this:

```
makePacket icmp_packet_1
icmp_packet_1 setField sourcePort 9999
icmp_packet_1 setField destPort 25
icmp_packet_1 setField sequence 1234
...
```

These values can be extracted, and assembled in the proper order, with the proper padding to create a network packet. The next trick is to define the order of the fields, sizes, etc.

In C++, Java, or [incr Tcl] the packet class would be an abstract class, and we'd derive TCP, UDP, ICMP, etc. classes from this base class with the field definitions.

In pure Tcl, we can define a namespace that contains field definitions and methods for creating network packets and import those methods into our packet object.

We commonly think of a network packet as a series of bytes. However, some fields (header length, TCP flags) are less than 8 bits long. To generalize the algorithms, I decided to build the packets as bitstreams and define the fields as bit offsets and lengths. After the bitstream is completely assembled, it's converted to bytes.

The protocol definition namespace resembles:

```
namespace eval icmp {
    # name bit-offset bit-length
    set fields {
        type 0 8
        code 8 8
        checksum 16 16 }

    # Place a bytestream representation of the packet in
    # the fieldArray associative array.
    proc fillPacket {} {
        upvar fieldArray f
        variable fields

        foreach {name start len} $fields {
            lappend bitStream [makeElement $len $f($name)]
        }

        set f(packet) [Bits2Bytes $bitStream]
    }

    # Return a bitstream padded to the required length.
    proc makeElement {len value} {}

    # Convert a bitstream to a hexadecimal bytestream.
    proc Bits2Bytes {bitStream} {}
}
```

The definition of a packet object now looks like this:

```
namespace eval icmp_packet_1 {
    variable fieldArray     ;# Array of values for fields

    namespace import ::proc::*
    namespace import ::icmp::*
}
```

A bytestream packet can be constructed with code like:

```
icmp_packet_1 setField type 0
icmp_packet_1 setField code 0
icmp_packet_1 setField checksum 0
icmp_packet_1 fillPacket
```

The calls to setField will be evaluated in the ::procs namespace, which will use upvar to map the ::icmp_packet_1::fieldArray variable into the local procedure scope.

The call to fillPacket will be evaluated in the ::icmp namespace and will use upvar to map the ::icmp_packet_1::fieldArray variable into this scope. The ::icmp::fields variable is already in the correct scope.

By defining different values in the fields variable, we can define different protocols. These descriptions can be used to create different types of data packets by importing the appropriate namespace.

This leads to a more generalized makePacket procedure:

```
proc makePacket {name type} {
    global packetDef
    namespace eval ::$name $packetDef
    namespace eval ::$name "namespace import ::${type}::*"
    proc $name {args} "namespace eval $name \$args"
}

makePacket icmp_packet_1 icmp
```

The final useful tweak to the makePacket procedure is to support setting the fieldArray values when the object is created, instead of creating an empty object and requiring multiple setField calls.

The array set command assigns values to multiple associative array indices in a single command. For example, this command would assign 0s to fieldArray(type), fieldArray(code), and fieldArray(checksum):

```
array set fieldArray {type 0 code 0 checksum 0}
```

This makePacket procedure definition creates the new object, populates the fieldArray variable, inherits the field definition and packet-generating methods, and creates the procedure to use for further interaction with the new object.

```
proc makePacket {name type args} {
    global packetDef
    namespace eval ::$name $packetDef
    namespace eval ::$id [list array set fieldArray $args]
    namespace eval ::$name "namespace import ::${type}::*"
    proc $name {args} "namespace eval $name \$args"
}
```

```
# ...
makePacket icmp_packet_1 icmp type 0 code 0 checksum 0
```

In actual fact, this package is a bit larger and more complex than described. It allows values to be defined with mnemonics as well as numbers, translates from Internet and Ethernet style addresses into bytestreams, checks for a complete set of values, etc. The use of namespaces to implement an abstract class and lightweight objects, however, works as described.

In the actual packet-generating package, the makePacket command is named make and is contained in the ::packet:: namespace.

The first thing to do with the network packet package is to confirm that it generates the expected packets. There are more ways to test software than there are software writers. In this case, the tests break down into two categories:

1. Tests that compare the packet data to a known good bytestream
2. Tests that use external systems to analyze packets "off the wire"

The Tcl interpreter comes with a package for automating regression tests. This package is used to test the Tcl interpreter (and many other Tcl packages) and to invoke make tests. The two workhorses of this suite are:

Syntax: tcltest::test *name desc constraint script expectedAnswer*

	Run a test, and compare the results to the expected results.
name	The name of this test – to use when reporting pass/fail results.
desc	The description of this test – to use when reporting pass/fail results.
constraint	A set of constraints – to define when this test should be evaluated. (For example, only test on certain platforms, if other tests pass, etc.)
script	The test script to evaluate.
expectedAnswer	The expected result.

A simple test would resemble:

```
tcltest::test expr-1 "Confirm that expr will add 2+2" {} \
    {expr 2+2} 4
```

If the test fails (for instance, if we declare a wrong value for the expectedAnswer), Tcl generates output resembling this:

```
==== expr-1 Confirm that expr will add 2+2 FAILED
==== Contents of test case:
expr 2+2
---- Result was:
4
---- Result should have been (exact matching):
5
==== expr-1 FAILED
```

After running the tests, we can generate a report with the ::tcltest::cleanupTests command. The ::tcltest::cleanupTests command generates a report resembling this:

```
:     Total 2     Passed 1     Skipped 0     Failed 1
```

To test the packet-generating package, we can generate the expected packets by hand and compare these values to the output from the code.

A set of tests like this can exercise a package and confirm that it behaves as expected:

```
package require tcltest
package require packet

set p1 [packet::make ICMP type ICMP_ADDRESS code 0 \
    checksum 0 identifier 1 sequence 2 subnet 00]

set expected [list 0x11 0x00 0xee 0xfc 0x00 \
    0x01 0x00 0x02 0x00 0x00 0x00 0x00]

tcltest::test makeICMP-1 {ICMP_ADDRESS} {} \
    {$p1 getField packet} $expected

# ... more tests
::tcltest::cleanupTests
```

This produces a regression suite that runs quickly, provides a nice summary of results, and is easy to create. The downside is that it confirms that the package does what I expect, which may not be what's correct.

This is where an outside validation tool is useful. One easily available tool for this testing is tcpdump. We can generate packets, transmit them, and let tcpdump sniff the packets, do some analysis, and report problems.

For instance, the tcpdump command

```
tcpdump -s 15000 -l -x -n -v -i eth1
```

will generate output like this:

```
14:22:12.963048 192.168.9.2 &gt; 192.168.9.17: icmp:
    address mask request (ttl 32, id 2, len 32)
        4500 0020 0002 0000 2001 0778 c0a8 0902
        c0a8 0911 1100 eefc 0001 0002 0000 0000
14:22:12.963051 192.168.9.2 &gt; 192.168.9.17: icmp:
    address mask request (wrong icmp csum) (ttl 32, id 2, len 32)
        4500 0020 0002 0000 2001 0778 c0a8 0902
        c0a8 0911 1100 ee99 0001 0002 0000 0000
```

When testing packets generated with this code:

```
lappend auto_path .

package require packet
package require ip
package require icmp
package require ether
package require tcp
package require udp
package require dnetlib

load ./libdnet.so

set e [dnet::open eth1]

# Generate a good icmp Address Mask Request.
set icmp1 [packet::make ICMP type ICMP_ADDRESS code 0 checksum 0 \
    identifier 1 sequence 2 subnet 00]

# Generate a BAD CHECKSUM icmp Address Mask Request.
set icmp2 [packet::make ICMP type ICMP_ADDRESS code 0 checksum 99 \
    identifier 1 sequence 2 subnet 00]

# Embed the icmp packets into IP packets.
set ip1 [packet::make IP version 4 hdrlen 5 tos 0 length 32 id 2 flag 0 \
    offset 0 ttl 32 protocol ICMP checksum 0 source \
    192.168.9.2 dest 192.168.9.17 options {} payload [$icmp1 getField packet]]

set ip2 [packet::make IP version 4 hdrlen 5 tos 0 length 32 id 2 flag 0 \
    offset 0 ttl 32 protocol ICMP checksum 0 source \
    192.168.9.2 dest 192.168.9.17 options {} payload [$icmp2 getField packet]]

# Embed the IP packets into Ethernet packets.
set ep1 [packet::make ETHER dest 00:E0:4C:00:14:4D src 00:A0:CC:D1:B6:00 \
    type IP payload [$ip1 getField packet]]
set ep2 [packet::make ETHER dest 00:E0:4C:00:14:4D src 00:A0:CC:D1:B6:00 \
    type IP payload [$ip2 getField packet]]

# Convert the Ethernet packets into binary.
dnet::importPacket $ep1
dnet::importPacket $ep2

# Send the binary packets out into the world.
dnet::send $e $ep1
dnet::send $e $ep2
```

These two sets of tests (one internal, and one external) are probably adequate. But proper testing should use the most expensive and complex piece of equipment you can access. After all, how else can you justify nifty toys?

The Spirent AX-4000 was described briefly in the previous Tclsh Spot article. The next article will discuss using the AX-4000 to capture and analyze the output of the packet generator.