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

Client Server Sockets

Previous Tclsh Spot articles described techniques for generating IP packets to simulate an attack on a firewall, while the last Tclsh Spot article described using the Expect extension to monitor a remote system's log files. This article will start to explore techniques for coordinating the attack-and-monitor activities for a firewall exerciser.

Several software architecture options exist for a system like this. The two obvious ones are a single application with attacking and monitoring subsections and a set of cooperating applications where each application provides a subset of the functionality.

A single application is conceptually simpler, since there's no need for interprocess communications. On the other hand, dealing with multiple sections that can require attention at undefined intervals is nearly as complex as interprocess communication. The real problem with a single application architecture in this case is that it limits the system to a single hardware platform. The validation application may need to run attacks and monitors from multiple sets of hardware.

Given that there will be multiple independent processes, the next question is whether they should be peers or operate in a master-slave relationship. If all the processes were identical, it would make sense to run a peer relationship. For a system where each child task has a different purpose, a peer relationship would mean that each child would need to know how to communicate with every different type of application. With a master-slave architecture, only the master needs to know how to talk to many types of applications, and the individual applications only need to know how to talk to the master. This allows the slave tasks to be simpler applications.

The last choice is whether to control the slave applications using command line arguments, pipes, or sockets. Again, the need to run on multiple sets of hardware drives the design to a socket-based client-server architecture.

Using the Tcl socket command to coordinate multiple tasks is fairly simple. The Tcl TCP socket implementation is possibly the easiest-to-use socket package available, and the callback mechanism used to service clients makes it easy for a server to interact with several active clients simultaneously.

Tcl uses a channel abstraction for I/O. A channel is a handle that references a source or destination for a stream of bytes. A Tcl channel is similar to the FILE pointer in C, abstracted a bit higher to include pipes and sockets.

We open either a client or server socket with Tcl's socket command. A client-side socket is slightly simpler, so we'll look at that first.

Syntax: socket *?options? host port*

socket	Open a client socket connection.	
?options?	Options to specify the behavior of the socket.	
	-myaddr *addr*	Defines the address (as a name or number) of the client side of the socket. This can be used to specify which of several Ethernet interfaces to use, and 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.	

The socket command will return a channel which can be used with the puts and gets commands to send and receive data from the channel.

As a quick test of a Tcl client, we might write the code shown below, expecting to see the beginnings of a Sendmail conversation.

```
set smtpSocket [socket 127.0.0.1 25]
set input [gets $smtpSocket]
puts "READ 1: $input"
puts $smtpSocket "helo foo@bar.baz"
set input [gets $smtpSocket]
puts "READ 2: $input"
```

Unfortunately, this won't quite work. This script generates a single line of output and then hangs:

```
READ 1: 220 vlad.cflynt.com ESMTP Sendmail
8.11.6/8.11.6; Tue, 5 Aug 2003 20:48:36 -0400
```

By default, Tcl channels use buffered I/O. The example above just hangs forever with the string "helo foo@bar.baz" sitting in the client socket's output buffer, while the Sendmail server waits for input.

Tcl provides two solutions to this dilemma: the flush command, which will flush a buffer, or the fconfigure command, which allows an application to modify the behavior of a channel.

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 127.0.0.1 25]
set input [gets $smtpSocket]
puts "READ 1: $input"

puts $smtpSocket "helo foo@bar.baz"
flush $smtpSocket

set input [gets $smtpSocket]
puts "READ 2: $input"
```

This generates the expected output of a simple conversation:

```
READ 1: 220 vlad.cflynt.com ESMTP Sendmail
8.11.6/8.11.6; Tue, 5 Aug 2003 20:48:36 -0400

READ 2: 501 5.0.0 Invalid domain name
```

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. For a challenge/response type interaction, this is probably line buffering; a character-based interactive application (like Telnet) would use no buffering, while an application moving lots of data (like an HTTP daemon) would use fully buffered I/O.

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 127.0.0.1 25]
fconfigure $smtpSocket -buffering line

set input [gets $smtpSocket]
puts "READ 1: $input"

puts $smtpSocket "helo example.com"

set input [gets $smtpSocket]
puts "READ 2: $input"
```

This script generates output resembling this:

```
READ 1: 220 vlad.cflynt.com ESMTP Sendmail
8.11.6/8.11.6; Tue, 5 Aug 2003 20:51:34 -0400

READ 2: 250 vlad.cflynt.com Hello localhost [127.0.0.1],
pleased to meet you
```

A server-side socket is a little different. Rather than opening a connection to another system, a server waits until a client requests a connection to a particular port. When a client requests a connection, a new port is assigned for the conversation, and a callback script defined in the socket -server command is evaluated.

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 openConnection $port
```

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

The script has three arguments appended to it before being evaluated: the handle for the new channel, the IP address of the client, and the port assigned to the client's socket.

A simple server to report the current time and close the connection looks like this:

```
#!/usr/local/bin/wish
socket -server openConnection 12345

proc openConnection {channel ip port} {
      puts $channel [clock format [clock seconds]]
      close $channel
}
```

This will open a connection, but doesn't do anything useful. A more useful server would interact with the client. The server could use the blocking gets command to wait for input, but while this paradigm works with single-client applications like Sendmail, it won't work with multiple clients, any of which might require service at any time.

Tcl supports both the linear-program flow used with a block-until-data-is-ready model, and an event-driven flow, which can be used with a multiple simultaneous session model.

The fileevent command defines a script to evaluate when data becomes available. Using fileevent 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.

Setting up a file event is commonly done on the server side within the openConnection script, and on a client side, immediately after opening the socket.

```
# Server side sample openConnection with fileevent

proc openConnection {channel ip port} {
      fileevent $channel readable [list processLine $channel]
      fconfigure $channel -buffering line
}

# Client sample open socket with fileevent

set Client(sock) [socket 127.0.0.1 12345]
fileevent $Client(sock) readable "processLine $Client(sock)"
```

The last "Tclsh Spot" article described using expect to automate examining a log file. We can use the challResp procedure from that example to build a client that will automate verifying an FTP server.

The challResp procedure provides a framework for challenge/response interactions:

```
################################################
# proc challResp {pattern response info}—
#     Hold a single interchange challenge/response conversation.
# Arguments
#     pattern:    The pattern to wait for as a challenge.
#     response:  The response to this pattern.
#     info:         Identifying information about this interac-
#                    tion for use with exception reporting.
#
# Results

proc challResp {pattern response info} {
      global spawn_id
      expect {
          $pattern {exp_send "$response\n"}
```

```
        timeout  {error "Timeout at $info" "Timeout at $info"}
        eof      {error "Eof at $info" "Eof at $info"}
    }
    return "OK"
}
```

We could automate this FTP login conversation:

```
$< ftp 192.168.90.222
  Connected to 192.168.90.222.
  220 vmware2.cflynt.com FTP server (Version
          wu-2.6.2-5) ready.
  Name (192.168.90.222:clif): anonymous
  331 Guest login ok, send your complete e-mail address
          as password.
  Password:
  230 Guest login ok, access restrictions apply.
```

with this code:

```
spawn ftp 192.168.99.99
challResp "Name" anonymous "Name prompt"
challResp "word:" foo@example.com "Password prompt"
challResp "Guest login ok" " " "FTP application prompt"
```

If there is no FTP server running on 192.168.99.99, a timeout error will be generated with the string Name prompt, and if anonymous logins are not supported, the error will include the string FTP application prompt. If anonymous logins are supported, no error will be generated.

This can be expanded and generalized into a procedure that keeps a list of arguments for challResp in a list, and iterates over them until the arguments are used up, or an error is thrown:

```
############################################
# proc runTest {}—
#    Run an FTP login test
# Arguments
#    NONE
# Results
#    Returns a list of result (Success/Fail) and optional
#    failure message.
#
proc runTest {} {
    global Client spawn_id errorInfo
    set errorInfo ""

    spawn ftp $Client(IP)

    set conversations {
        "Name" "$Client(User)" "Name prompt"
        "word:" "$Client(Passwd)" "Password prompt"
        "Guest login ok" {} "FTP application prompt"
    }

    foreach {challenge response msg} $conversations {
        set fail [catch {challResp $challenge [subst
                $response] $msg} result]
        if {$fail} {break;}
    }

    array set lookup {0 "Success" 1 "Fail"}

    puts $Client(output) [list RESULT: $lookup($fail)
                $result]
}
```

By using the associative array Client to hold the IP address, username, and password, it is easy to run multiple tests with code like this:

```
array set Client {IP 192.168.99.99 User badIP Passwd
        badPasswd}
runTest
array set Client {IP 192.168.90.222 User goodUser
        Passwd goodPasswd}
runTest
```

This set of code would create a stand-alone application with a hardcoded set of tests. The script would iterate through the tests and exit.

We can convert this into a client-server application by adding a procedure to process the data that's read from the server and a few lines to open and configure the socket. The problem with this is that the script would open the socket, send an initial "Hello," and then reach the end of the script and exit.

The vwait command is the solution to this problem. The vwait command causes a script to wait until a variable changes value. The interpreter pauses at the vwait command and enters the event loop, processing events until the variable is assigned a new value. After this the interpreter continues evaluating the commands in the script.

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.

The simplest way to process the data from the server is to have the server always send Tcl commands, which can be evaluated in the client using Tcl's eval command.

The eval command concatenates a set of strings into a single string, and passes it to the command evaluation section of the interpreter, just as lines in a script are evaluated.

Syntax: eval *string1 ?string2...?*

*string**	Strings that will compose a command.

```
proc processLine {channel} {
    global Client
```

```
        set len [gets $channel line]

        if {[eof $channel]} {
            close $channel
            return
        }
        eval $line
}

set Client(output) [socket 127.0.0.1 23456]
fileevent $Client(output) readable "processLine
                $Client(output)"

fconfigure $Client(output) -buffering line

# Let the server know we're open for business.

puts $Client(output) ready

set Client(done) 0
vwait Client(done)
```

The server will send the client data like this:

```
array set Client {IP 192.168.99.99 User badIP Passwd
                badPasswd}
runTest
```

When the client receives data, the fileevent command causes the processLine procedure to be evaluated, which reads a line from the socket and evaluates it.

Notice the eof test after the gets. This will catch the spurious read event generated by most TCP stacks when the other end of a socket closes.

The server end of this pair includes a list of tests to run and three procedures to coordinate the tests:

openConnection
: Accepts new client connections.

processLine
: Reads data from the client. Displays results and calls runTest to start the next test in the client.

runTest
: Sends the commands to the client.

The list of tests can simply be an identifier and a set of array indices and values to be sent to the client:

```
set Server(tests) {
    {{bad address } {IP 192.168.90.223 User badIP
                Passwd badPasswd}}
    {{bad username} {IP 192.168.90.222 User badUser
                Passwd badPasswd}}
    {{good username} {IP 192.168.90.222 User goodUser
                Passwd goodPasswd}}
    {{anonymous/badPasswd} {IP 192.168.90.222 User
                anonymous Passwd badPasswd}}
    {{anonymous/goodpwd} {IP 192.168.90.222 User
                anonymous Passwd foo@bar.com}}
}
```

The openConnection procedure registers the fileevent script to evaluate whenever the client sends data, configures the channel to be line buffered, and initializes a counter to step through the tests for this client.

Note how this procedure uses an associative array index with two fields to distinguish the test counts for connections from different clients. Using multiple fields in an array index provides the same functionality in Tcl as multiple dimensioned arrays provide in C and FORTRAN.

```
proc openConnection {channel ip port} {
    global Server
    fileevent $channel readable [list processLine $channel]
    fconfigure $channel -buffering line

    # initialize the test counter
    set Server($channel.testNum) 0
}
```

The processLine procedure starts the same as the client's processLine procedure by reading a line of data and checking for an EOF condition.

The server does a rudimentary parse, looking to see if a line starts with the phrase "RESULT". If the line starts with "RESULT", it's displayed. Once data is read, the runTest procedure is invoked.

```
proc processLine {channel} {
    global Server

    set len [gets $channel line]

    if {[eof $channel]} {
        close $channel
        unset Server($channel.testNum)
        return
    }
    if {[string first RESULT $line] == 0} {
        puts "$Server(descript): $line"
    }
    runTest $channel
}
```

Finally, the runTest procedure checks to see whether there are valid tests to be run. If there are, it sends the appropriate Tcl commands to the client and updates the counter.

```
proc runTest {channel} {
    global Server

    if {$Server($channel.testNum) = [llength
            $Server(tests)]} {
```

```
            puts $channel {set Client(done) 1}
        } else {
            foreach {Server(descript) params} \
                [lindex $Server(tests) $Server($channel.testNum)] {}
            puts $channel "array set Client [list $params]"
            puts $channel "runTest"
            incr Server($channel.testNum)
        }
  }
```

This pair of procedures implements a simple test framework that can be run with different sets of data to characterize an FTP server. It's not sufficient to handle characterizing a firewall, but it's getting closer.

Sending scripts to the client to evaluate is a technique used by agent-style applications. This technique supports a great deal of customization at runtime. Tcl's `eval` command creates safe sandboxed interpreters, which makes it an excellent choice for exploring agent style applications.

The next "Tclsh Spot" article will look at building a server-agent architecture to perform more tests. As usual, the complete code that was described in this article is available from *http://www.noucorp.com*.