;login:

THE MAGAZINE OF USENIX & SAGE

April 2004 • volume 29 • number 2

inside:

PROGRAMMING

Flynt: The Tclsh Spot: Mobile Agents in Tcl

USENIX

The Advanced Computing Systems Association

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

Mobile Agents in Tcl

The previous "Tclsh Spot" article described how to build a client/server pair using the Secure Socket Layer extension (TLS) and a safe slave interpreter. This article extends that client/server pair to support transferring complete Tcl programs to the server for remote evaluation.

A mobile software agent is a set of code that can be sent from one trusted system to another to be evaluated on the remote system. A few years ago, this was considered a rather exotic style of program architecture, but it's become commonplace now. For example, when you download a Web page with a script component, your browser evaluates that code in a safe sandbox on your system. You are trusting that the code will not escape that sandbox and damage your system. Whether or not this trust is misplaced is left as an exercise for the reader. (If you hear a rant about IE and lack of proper sandboxes, you aren't mistaken.)

The protocol for the Client/Server pair described in this article is simple. The Server sends a Ready prompt when it's ready to accept input. The client sends Tcl commands to the server and waits for a Ready.

The simple client/server pair described in the previous article was able to implement most of this protocol, but had some shortcomings.

One of the least obvious problems is that using Tcl's basic interp create -safe command provides a bit more security than we need.

The safe interpreter created with the interp create -safe command is very limited in what it will do. Among the restrictions is that a safe interpreter has absolutely no access to the file system. This makes it impossible to load Tcl extensions or pure Tcl packages into the slave interpreter. We can extend a safe interpreter with the interp alias command that was described in the previous article to add support for loading packages, safe access to the file system, and such. In fact, there are enough requirements for extended functionality that Tcl provides a package with the common aliases already built.

The Tcl distribution comes with the ::safe:: package which extends the base safe interpreter. The interpreters created with the ::safe::interp create command still run in a safe sandbox, but also have hooks to allow a few more actions, including loading pure Tcl packages and extensions with a SafeInit entry point to be loaded into the safe slave.

There are several commands in the ::safe:: package (documented under man safe), but the three important ones are:

Syntax:::safe::interp create ?name? ?key value?

::safe::interp create	create a new safe interpreter. Returns the name of the new interpreter
?name?	An optional name for the new interpreter. The default name will be something boring like interp0.
?key value?	Optional keyword/value pairs to fine-tune the safe interpreter's environment. Some include:

-statics boolean
: True allows the slave interpreter to load statically linked packages (load {} Tk). False disables this ability. The default is True.

-nested boolean
: True allows the slave interpreter to load packages into sub-interpreters. False disables this ability. The default is True.

-deleteHook script
: A script to be evaluated before deleting an interpreter. This hook gives the slave interpreter a chance to do cleanup (perhaps log an exit message) before being destroyed.

Syntax:::safe::interpAddToAccessPath path

::safe::interpAddToAccessPath
: Adds directories to the list of directories the slave interpreter can search to find packages to load. This list is maintained in the parent interpreter. A safe slave cannot access the list, thus preventing a slave from leaking information about a filesystem to the outside world.

path
: The directory path to add.

Syntax:::safe::interpDelete interpName

::safe::interpDelete
: Delete the slave interpreter. All state for that interpreter will be destroyed.

interpName
: The name of the interpreter to be destroyed.

The following code uses the ::safe:: commands to create the interpreters. For a more restrictive environment, you can substitute the base interp create etc. for these.

The previous example server had only one set of state information and only one slave interpreter. For a single-purpose, stateless server, this is appropriate. However, if two agents tried to use such a server at the same time and each sent a script with the same name but a different body to be evaluated, one agent would be running the wrong code.

The agent server must maintain separate interpreters and sets of state information for each active connection, to keep the agents from interfering with each other. Fortunately, all the state for an active agent can be kept in that agent's interpreter. All we need to track in the server is the channel associated with an agent, the name of the interpreter for that agent, and incomplete input waiting to be evaluated.

In C, you might have an array of state structures to hold the necessary information. It might look like this:

```
struct agent {
    IO_channel *channel;
    Tcl_Interp *interp;
    char *input;
} activeAgents[10];
```

When data is read from a client, the server steps through the structures in the array activeAgents until it finds the appropriate element, identified by the IO_channel field.

Tcl does not support an array of structures the way you would define the data in C or Java. However, the associative array provides the same functionality for this requirement. In Tcl, we can initialize the equivalent data structure keyed by the channel identifier, like this:

```
set State(interp.$channel) [interp create -safe]
set State(input.$channel) ""
```

The procedure to establish a connection with a new client described in the previous "Tclsh Spot" article just waited for the handshake to be complete and established a fileevent to handle the input.

```
proc openConnection {channel clientaddr clientport } {
    global tlssignal

    # Wait until the handshake is complete
    fileevent $channel readable \
        [list handshakeHandler $channel $clientaddr]
    vwait tlssignal($channel)

    fileevent $channel readable\
        [list processMessage $channel]

    # fconfigure for line buffering.
    fconfigure $channel -buffering line
}
```

In the agent-based server, the procedure starts the same, but after the handshke is complete, it creates and initializes the interpreter for this agent. The interp alias for writeLog is a procedure that was defined in the previous article; it allows a safe slave interpreter to write to a predefined log file.

```
proc openConnection {channel clientaddr clientport } {
    global State
    global tlssignal

    # Wait until the handshake is complete
    fileevent $channel readable \
        [list handshakeHandler $channel $clientaddr]
    vwait tlssignal($channel)

    fileevent $channel readable [list readLine $channel]

    # fconfigure for line buffering.
    fconfigure $channel -buffering line

    # Create a safe interpreter
    set State(interp.$channel) [::safe::interp create]

    # Link the 'writeLog' procedure in this environment
    # to the 'log' procedure in the safe child interpreter.

    $State(interp.$channel) alias log writeLog
    puts $channel "Ready"
}
```

The next trick is to send the agent server a procedure to run. Since the server is running in a secure sandbox, we don't need a special protocol to support this: we can send normal Tcl com-

mands to the server. This even includes complex commands such as procedure and namespace definitions.

However, in order to download scripts, the agent server must be able to accept multiple line inputs. The previous server read a single line and evaluated it. If the line were something like "procedure foo {args} {," it would be incomplete, and the slave interpreter would throw an error. The server needs to know when a complete command has been received before it tries to evaluate that input.

This problem has been solved in many ways ranging from complex parsers to requiring a special pattern like "\n.\n" to mark the end of input.

Each of these techniques has some limitations:

1. An input parser must return the same results as the actual evaluation parser. Keeping these in sync and verifying that neither has unexpected exception conditions can cause headaches.
2. A special pattern must not be something that could exist in a real message.

Tcl has an elegant solution to this problem: use the actual parse engine to test input.

One of the powerful aspects of Tcl is how many of the interpreter's internal functions are exposed to the script writer. The script writer can use the complex parser that's already built into Tcl to test input data. Since the same parser is used both to test the input and then to evaluate it, this guarantees that the test parser and evaluation parser accept the same information,

The info command gives a Tcl script a sneak-peek into the state of the interpreter. You can get a list of known commands, global and local variables, the argument and body of a procedure, and a fair amount more.

The info complete command gives the script writer access to the Tcl interpreter's parsing logic. It returns TRUE (1) if the string it's presented is a complete Tcl command, and FALSE (0) if there are mismatched parentheses, quotes, braces, etc.

Syntax: info complete string

string	The string to check for matching braces, brackets, quotes, etc.

A server that only accepts single line commands from a client could look like this:

```
proc readLine {channel} {
    global State

    # Read a line of data
    set len [gets $channel line]

    # Close if we've recevied an EOF from the client
    if {($len <= 0) & [eof $channel]} {
        close $channel
        return
    }

    processMessage $channel $line
}
```

The agent server is just a little more complex. Once the server has received a complete command, it can eval it within the proper safe interp, but until then, it needs to save the data and check each time there's a new line.

The code below saves each line as it's read in an associative array indexed with the field name input and the channel identifier. It checks the saved text to see if it's a complete Tcl command, and if it is, processes it. If not, it appends a newline (the gets command strips off newlines), and continues.

One trick with the info complete command is that an empty string has no mismatched quotes, braces, etc., and is thus complete, even if it's meaningless. The script checks for empty lines just to avoid wasting time processing nothing.

```
proc readLine {channel} {
    global State

    # Read a line of data
    set len [gets $channel line]

    # Close if we've received an EOF from the client
    if {($len <= 0) & [eof $channel]} {
        close $channel
        ::safe::interpDelete $State(interp.$channel)
        return
    }

    # Put the data in a safe place
    append State(input.$channel) "$line"

    # And check to see if we've got a complete command

    if {(![string equal "" $State(input.$channel)]) &&
        ([info complete $State(input.$channel)])} {
        processMessage $channel
        set State(input.$channel) ""
    }   else {
        if {(![string equal "" $State(input.$channel)])} {
            append State(input.$channel) "\n"
        }
    }
}
```

The processMessage procedure is quite simple. All it needs to know is the channel identifier, which it can use to index into the State associative array to find the appropriate input and inter-

preter. It evaluates the script within the proper interpreter and sends the results to the client, along with a new Ready prompt.

```
proc processMessage {channel} {
    global State

    if {[string match "" [string trim $State(interp.$channel)]]} {
        return
    }
    set rply [$State(interp.$channel) eval $State(input.$channel)]
    puts $channel "$rply"
    puts $channel "Ready"
}
```

With the addition of a dozen lines of code (including some comments), we've changed the previous simple server into one that handles receiving scripts to be evaluated remotely, while maintaining the security of the system the server is running on.

The previous client was very simple. It sent a command to the server and waited for the reply. The reply was always a single line, and each command had a reply.

The protocol for communicating with this agent server is a little more complex. Input is requested with a Ready prompt, and other information is the response to the previous client command. We can add a procedure to watch for the Ready prompt. This procedure grabs a line of data and checks to see if it's the expected prompt. If it's not the prompt, the line of input is saved; when the prompt is received, the input is returned to the calling script.

```
proc wait4Prompt {channel prompt} {
    set rtn ""
    while {[string first $prompt [set line [gets $channel]]] != 0} {
        if {[eof $channel]} {error "CLOSED" "Channel Closed"}
        append rtn "$line\n"
    }
    return $rtn
}
```

With the addition of that procedure, a simple agent that requests a base64 encoding of some data might resemble this:

```
set setup {
package require base64
}

set serverSocket [tls::socket -password getPassword \
    -keyfile $certDir/clif@noucorp.com.key \
    -certfile $certDir/../1000.pem \
    -cafile $certDir/../rootca.pem \
    -request true -require true $host $port]

fconfigure $serverSocket -buffering line

wait4Prompt $serverSocket Ready

puts $serverSocket $setup
wait4Prompt $serverSocket Ready

puts $serverSocket [list base64::encode "This is a test
                                    message."]
set rply [wait4Prompt $serverSocket Ready]

puts "The reply is: $rply"
```

And that's a basic mobile agent client/server pair. In this implementation, only valid agents are allowed to use the server. The validation is done with the OpenSSL extension (TSL) which only allows connections from an agent that knows the passwords and keys to the SSL configuration files that exist on the server.

Including comments, this is 161 lines of code. Of course, interesting agents will have more application-specific instructions. As usual, this code will be available at *http://www.noucorp.com*.