The following paper was originally presented at the
Third Annual Tcl/Tk Workshop
sponsored by Unisys, Inc. and USENIX Association
Toronto, Ontario, Canada, July 1995.

Tcl Extensions for Network Management Applications

J. Schoenwaelder and H. Langendoerfer
Department of Computer Science
Technical University of Braunschweig, Germany

Tcl Extensions for Network Management Applications

J. Schönwälder H. Langendörfer

Department of Computer Science
Technical University of Braunschweig, Germany

Abstract

This paper presents extensions to the Tool Command Language (Tcl) that have been designed to implement network management applications. Using Tcl, we were able to make our network management applications highly extensible. Experience has shown that many useful applications can be written with a few lines of Tcl code. Site specific adaptations are possible at very low cost. We have used our extensions to implement smart network management agents that can receive and execute management scripts provided by other management stations or agents.

1 Introduction

Network management is an area which has become very important over the last decade as todays networks have become critical for the success of many organizations. Network management software is designed to cope with problems from the size, complexity and heterogeneity of todays multi-protocol networks.

The Internet and the OSI networking communities have developed network management architectures, which define the structure of network management information and protocols to retrieve and manipulate management information provided by agents running on the network elements [16].

Network management applications are in most cases embedded into network management platforms, which provide a runtime and development environment for management applications. Although the platform approach has many advantages, we have seen a lot of network management applications that fail to work in some particular network configurations. This is not necessarily a fault of the application designer, as it is very difficult to write management applications that take into account the huge number of different network configurations in use today. Things get even worse due to all kinds of bugs present in many network devices.

As a consequence, many network management applications must be adapted to the target environment. This usually requires to write or to change existing C code. The C language programming interfaces to access network management services usually require a very good understanding of complex data structures before one can use them even for some trivial tasks. To overcome this problem, we started to design a network management platform which uses the Tool Command Language (Tcl) [10] to provide a higher level of abstraction.

This paper presents Tcl extensions to access various network services. Some examples will be given to show how these extensions can be used to implement simple but powerful management scripts. The next section starts with a short description of the underlying event-driven model and section 3 introduces extensions that provide access to standard services of the TCP/IP protocol suite.

In section 4, we present the interface to the management protocol of the Internet (SNMP) and section 5 describes an interface to the OSI management protocol (CMIP). A small example will demonstrate how both protocols can be used to solve a simple management problem. We will shortly review some more complex applications build with our extension in section 6.

Distributed network management using smart management agents is another interesting application area for our extension. We discuss the implementation of smart management agents in section 7 and we conclude with a brief comparison with related work and some ideas for further improvements in section 8.

2 Event-Driven Management

Our extensions named Scotty are based on the event-driven programming paradigm well known to every Tk programmer [10]. A typical Scotty application loads an initialization script which installs some basic event handlers. Once initialization is complete, the application enters an event loop. Tcl scripts are evaluated to process events that are created when a message is received from the network or

```
job create <command> <interval> [<repetitions>]
job info
job current
$j status [<value>]
$j command [<value>]
$j interval [<value>]
$j repetitions [<value>]
$j attribute <name> [<value>]
```

Figure 1: The job command.

if a timer expires. The event loop terminates if no timer events are left and no more messages are accepted from the network. We are using the generic event management library of Tk 4.0 to create an event-driven Tcl interpreter without the rest of Tk.

Many management applications require to perform tasks periodically. We have build a job scheduler on top of the event management library to simplify the implementation of periodic tasks. Figure 1 shows the `job` command. A new job is created with the `job create` command. Following the object-oriented approach, a new command is created that represents the job object. Operations on the job object allow to modify the job state (e.g. suspending a job), to change the Tcl command bound to a job or scheduling parameters.

The `attribute` command option allows to attach arbitrary attributes to a job. Attributes can store all information needed to execute a job and reduce the need for global variables. It is straight forward to implement a generic restart mechanism if all context information is attached to a job object.

The event-driven programming paradigm works quite well in most cases. However, some networking extensions do not fit well with the event-driven programming style. For example, some SUN RPCs block for about 20 seconds before they return control. In these cases, a thread based implementation would be a big win. However, a threaded implementation would reduce the portability of our extensions as there is still no common thread library available.

3 Basic TCP/IP Extensions

Internet network management often starts with tests to ensure the reachability of hosts using the Internet Control Message Protocol (ICMP) [11]. Scotty provides access to the ICMP protocol with the `icmp` command shown in figure 2.

The `icmp` command allows to send ICMP echo, mask or timestamp requests to a list of hosts. ICMP packets are send in a round-robin fashion while responses are collected. This allows to perform fast scans of entire IP address spaces. The return value of the `icmp` command is a list, wherein each element contains the host name and either the round trip time, the network mask or the time offset. The `ttl` and `trace` options send UDP packets to unused ports and catch the returned ICMP error message. This allows to create routing traces with a simple Tcl script as shown in figure 3.

```
icmp [<options>] echo <hosts>
icmp [<options>] mask <hosts>
icmp [<options>] timestamp <hosts>
icmp [<options>] ttl <num> <hosts>
icmp [<options>] trace <num> <hosts>
```

Figure 2: The icmp command.

```
proc traceroute {ip {max 32}} {
  set ttl 1
  set new ""
  while {$new != $ip && $ttl <= $max} {
    set old $new
    set hop [lindex [icmp trace $ttl $ip] 0]
    set new [lindex $hop 0]
    set rtt [lindex $hop 1]
    if {$old == $new} break
    puts [format "%3d %5d ms %s" $ttl $rtt $new]
    incr ttl
  }
}
```

Figure 3: A simple traceroute written in Tcl.

Access to many services above the transport layer is provided by the `tcp` and `udp` commands (figure 4). They implement much the same functionality as other TCP extensions for Tcl, e.g. tcl-dp [15]. Both commands manipulate standard Tcl file handles that are bound to socket file descriptors. The `info` option can be used to retrieve socket specific information like source and destination addresses.

Unlike other TCP extension, we preferred to distinguish between commands that manipulate TCP or UDP sockets. This makes Tcl code easier to understand, as there is a clear indication which kind of transport service is actually used.

The main purpose of the `tcp` and `udp` commands is to access standard test services at the transport layer (e.g. echo, discard or chargen). However, there are a number of other useful services that can be accessed by writing a few lines of Tcl code. Examples are the finger service [19] or the whois service [6].

```
tcp connect <host> <port>
tcp listen [<port>]
tcp accept <file>
tcp shutdown <file> <how>
tcp close <file>
tcp info [<file>]
udp open [<port>]
udp connect <host> <port>
udp send <file> [<host> <port>] <message>
udp receive <file>
udp close <file>
udp info [<file>]
```

Figure 4: The tcp and udp commands.

Another valuable source of information is the Domain Name System (DNS) [9]. A well maintained DNS does not only translate Internet names into IP addresses and back, it also offers information about machine types and operating systems. Access to the DNS is provided by the `dns` command shown in figure 5.

The `address` and `ptr` options convert Internet host names into IP addresses and back while the `hinfo` option retrieves the host information record. The mail exchanger record for a domain name can be read with the `mx` option and the `soa` option returns the server which provides authoritative answers for a domain.

```
dns [<options>] address <name>
dns [<options>] ptr <address>
dns [<options>] hinfo <name>
dns [<options>] mx <name>
dns [<options>] soa <name>
```

Figure 5: The dns command.

Access to local network databases is implemented with the `netdb` command (figure 6). If called without any optional arguments, the `netdb` command will return a list of all known records. For example, the command `netdb networks` lists all locally defined networks. Each element contains a network name and a network address.

You can lookup name or address information directly by supplying appropriate arguments. For example, `netdb protocols name 1` will return `icmp`.

Our extensions allow to access arbitrary documents on the Internet via the Hypertext Transfer Protocol (HTTP) [1]. HTTP is an application layer protocol that defines methods to retrieve (get, post), store (put) and delete arbitrary documents (which could be Tcl scripts) and is the protocol behind the popular World Wide Web.

```
netdb hosts     [<query> <key>]
netdb networks  [<query> <key>]
netdb protocols [<query> <key>]
netdb services  [<query> <key>]
netdb sunrpcs   [<query> <key>]
```

Figure 6: The netdb command.

Supporting HTTP is attractive because it is easy to implement and it provides access to a wide variety of transport mechanisms by using HTTP gateways. These gateways are often termed proxy server, as they retrieve documents on behalf of the requesting client. HTTP gateways are usually configured to retrieve documents using other well known Internet protocols like FTP, WAIS or GOPHER.

```
http proxy [<server>]
http head <url>
http get <url> <fileName>
http post <url> <doc> <fileName>
http put <url> <fileName>
http delete <url>
http server [<port>]
http bind <pattern> <method> <script>
http mime [<type> <extension>]
```

Figure 7: The http command.

Figure 7 shows the `http` command. The `proxy` option allows to define a proxy server that will be used as a gateway. The `head`, `get`, `post`, `put` and `delete` options invoke the corresponding HTTP method on the document given by the Uniform Resource Locator (URL) [2]. The destination of a retrieval method is always a local file. This indirection is necessary as documents may contain binary data which can not be dealt with in Tcl. The return value of a retrieval operation is a Tcl list containing the document header.

The `server` and `bind` options can be used to turn a Tcl interpreter into a HTTP server. A document is exported by binding a Tcl script to a URL pattern and a HTTP access method. If a HTTP request is received that matches a previously defined binding, the corresponding script is evaluated. The result of the script is either a file name of a document that will be sent to the client or an error code that is mapped into a HTTP error response message.

Figure 8 shows a HTTP server that exports documents stored in the local directory /usr/local/http. Documents stored under /usr/local/http/tubs are only exported to IP addresses of the class B network 134.169.*.*. The /version document is created on the fly and contains version information. A special mime type is defined to transfer *.tcl files as `application/x-tcl` mime type.

```
http bind /* get {
    return "/usr/local/http/%P"
}

http bind /tubs/* get {
    if [string match 134.169.* "%A"] {
        return "/usr/local/http/tubs/%P
    } else {
        error Forbidden
    }
}

http bind /version get {
    set name "/tmp/version-[pid]"
    set f [open $name w]
    puts $f "Tcl Version: [info tclversion]"
    puts $f "Scotty Version: $scotty_version"
    close $f
    return $name
}

http server 1701
http mime application/x-tcl tcl
```

Figure 8: A simple http server.

The `http bind` command uses % substitutions to provide access to parameters of the request. The example above uses the %P sequence to access the URL path contained in the get request and the %A sequence to check the IP address of the requesting client. Other % sequences can be used to access the search part of the URL or the authorization field of the HTTP request.

4 The SNMP Interface

The Simple Network Management Protocol (SNMP) [16] is the management protocol of the Internet and allows to retrieve and modify status and performance information. SNMP is based on the idea that every device on the network runs a SNMP agent which provides access to a virtual information store called the management information base (MIB). A manager application uses the SNMP protocol to read or alter variables in the MIB, which are related to real resources on the device. An agent may also send unsolicited messages to management applications if an important event occurs (e.g. a network interface is going down).

A programming interface for a network management protocol must provide commands to access the services provided by the protocol as well as commands to retrieve the definitions that describe the MIB supported by an agent. Our experience shows, that the latter is very important, as the SNMP protocol only transfers raw data which is not very useful without relating it to MIB definitions.

There are currently two versions of the SNMP protocol. SNMP version 1 (SNMPv1) is a standard protocol of the Internet and in widespread use today. The successor, SNMP version 2 (SNMPv2) is at the time of this writing a proposed standard but not in common use. SNMPv2 adds some protocol enhancements and an improved security model (authentication and privacy) [16].

```
snmp session [<options>]
$s configure [<options>]
$s cget <option>
$s get <vbl> [<callback>]
$s getnext <vbl> [<callback>]
$s getbulk <nr> <mr> <vbl> [<callback>]
$s set <vbl> [<callback>]
$s inform <trapOid> <vbl> [<callback>]
$s trap <trapOid> <vbl>
$s bind <notification> <script>
$s wait [<id>]
$s walk <var> <vbl> <body>
$s destroy
snmp wait
snmp info
```

Figure 9: The snmp command.

The Scotty SNMP extension to Tcl was written from scratch based on our experience with earlier SNMP extensions for Tcl. Our goals were

- to provide access to SNMPv1 and SNMPv2 using the convergence rules as defined in the SNMPv2 standard,
- to create an efficient and portable implementation,
- to allow fast access to MIB definitions,
- to define an easy to use Tcl interface,
- and to support asynchronous as well as synchronous operations.

```
proc DiscoverClassCNetwork {net} {
   for {set i 1} {$i < 255} {incr i} {
      set s [snmp session -address $net.$i]
      $s get sysDescr.0 {
         if {"%E" == "noError"} {
            set d [lindex [lindex "%V" 0] 2]
            puts "[%S cget -address] \t $d"
         }
         %S destroy
      }
      update
   }
   snmp wait
}
```

Figure 10: Script to discover SNMP devices.

Our Tcl interface to the SNMP protocol is shown in figure 9. The `snmp session` command creates a SNMP session and returns a handle which is used to invoke SNMP operations. Configuration options define parameters like the agent address and community name (SNMPv1) or the source/destination parties and the context (SNMPv2). The `configure` and `cget` options can be used to modify or retrieve the current configuration of a session.

All primitive SNMP operations (`get`, `getnext`, `getbulk`, `set`, `inform`, `trap`) can be invoked in synchronous or asynchronous mode. A synchronous operation returns either the list of requested SNMP variables or a SNMP error message. An asynchronous request is created when the optional callback parameter exists. In this case, a request identifier is returned immediately. The callback is evaluated when a response is received or if a request times out. The result of the SNMP operation as well as status information is substituted into the callback using % sequences. The `wait` option allows a script to wait until a single request, all requests sent over a session handle or all requests on all sessions have been processed.

All SNMP operations return a variable binding list, where each element contains the object identifier, the type and the value of a MIB variable. The object identifier is always returned in dotted notation and the value is always rendered according to the type or the textual convention defined for this variable[1]. In contrast, a variable binding list accepted by one of the SNMP commands is allowed to contain MIB labels instead of object identifiers in dotted notation and the value and type fields are optional. This strategy usually shortens the Tcl code and improves readability.

Figure 10 shows an example which discovers all SNMP devices on a class C network. The `for` loop is used to create SNMPv1 sessions for each IP address on the class C network. An asynchronous get request is send to each session to retrieve sysDescr.0. The callback first checks if there was an error (using the %E sequence) and writes the agent address and the value to the standard output, before the session (the %S sequence) is destroyed. The update command is used to clear the event queue inside the loop to make sure that incoming responses are processed as soon as possible.

```
mib name [-exact] <oid>
mib oid [-exact] <label>
mib syntax <label>
mib access <label>
mib index <label>
mib description <label>
mib tc <label>
mib format <label> <value>
mib scan <label> <value>
mib successor <label>
mib walk <var> <label> <body>
mib load <file>
```

Figure 11: The mib command.

SNMP MIB definitions can be accessed by the `mib` command (figure 11). The `mib name` command converts an object identifier from dotted notation into a label and the `mib oid` command converts a given label into the corresponding object identifier. The `syntax`, `access`, `index` and `description` options allow to retrieve the corresponding sections of the MIB definition. The `mib tc` command returns the formatting rule or enumerations defined for a MIB variable while the `scan` and `format` options can be used to explicitly apply a textual convention.

The `mib successor` command returns a list of all successors of a node in the MIB tree while the `mib walk` command allows to traverse a MIB subtree starting at the given label. The body of the walk command is evaluated for each MIB node in the subtree after the Tcl variable has been set to the object identifier of the node.

MIB definitions are added to the MIB tree using the `mib load` command. An integrated MIB parser converts the definitions contained in the MIB file into an internal MIB tree. A condensed format of the MIB file is written back to disk once a MIB has been parsed in order to reduce future loading times.

[1]Textual conventions are used in the SNMPv2 framework to define enumerations and formatting rules.

```
proc SnmpTcpUser {host port status} {
  set s [snmp session -address $host]
  set vbl [list tcpConnState tcpConnLocalPort \
              tcpConnRemAddress tcpConnRemPort]
  set result ""
  set code [catch { $s walk x $vbl {
       set state      [lindex [lindex $x 0] 2]
       set localPort  [lindex [lindex $x 1] 2]
       set remAddress [lindex [lindex $x 2] 2]
       set remPort    [lindex [lindex $x 3] 2]
       if {$state == $status
           && $localPort == $port} {
          lappend result "$remAddress $remPort"
       }
    }
  } msg]
  $s destroy
  if $code { error $msg }
  return $result
}
```

Figure 12: Script to count TCP connections.

Offsets into the MIB files are kept in memory to read MIB descriptions from the MIB file when evaluating the `mib description` command.

Figure 12 shows an example script that counts the number of TCP connections to a given port number. This script can be used for example to obtain the number of established connections to a news server. It uses the `snmp walk` command to walk through the SNMP tcpConnTable. The body of the `mib walk` command is evaluated for every row in the tcpConnTable after the variable named `x` has been set to the variable binding list. The values are extracted and the remote address and remote port number is added to the result list if the port and status attributes of a TCP connection match the procedure arguments. It is important to note that this solution usually requires multiple request/response interactions between our extension and the SNMP agent.

5 The CMIP Interface

We have done some initial work to implement a Tcl interface for the OSI management protocol CMIP [16]. The `cmip` command is shown in figure 13. Our implementation is based on the last freely available OSIMIS CMIP implementation [8].

The `cmip connect` command establishes an association to an OSI management agent. It returns a handle which is used to invoke CMIP operations (`get`, `set`, `create`, `delete`, `action`, `cancelGet`).

```
cmip connect <agent> <host>
$c get <class> <instance> [<options>]
$c set <class> <instance> [<options>]
$c create <class> [<options>]
$c delete <class> <instance> [<options>]
$c action <class> <instance> <action> [<options>]
$c cancelGet <request> [<callback>]
$c eventSink [<callback>]
$c wait
$c release
$c abort
cmip wait
cmip info
```

Figure 13: The cmip command.

The options allow to specify callback scripts for asynchronous operations or additional parameters like scopes and filters.

```
proc CmipTcpUser {host port status} {
  set instance sysName=$host@tcpId=""
  set attrs "tcpConnRemAddress tcpConnRemPort"
  set filter "((tcpConnState=$status) \
            & (tcpConnLocalPort=$port))"
  set c [cmip connect OIM-SMA $host]
  set code [catch {$c get tcp $instance \
    -attributes $attrs \
    -filter $filter -scope wholeSubtree} msg]
  $c release
  if $code { error $msg }
  set result ""
  foreach entry $msg {
    set elem ""
    foreach ava [lindex $entry 4] {
      lappend elem [lindex $ava 1]
    }
    lappend result $elem
  }
  return $result
}
```

Figure 14: Script to count TCP connections.

Figure 14 shows a CMIP version of the SNMP script to count TCP connections. It makes use of the fact that a CMIP agent is able to apply filter expressions. This results in exactly one request/response interaction apart from connection establishment and release. This makes the CMIP version faster in this particular case[2]. However, this is not true for arbi-

[2] It is possible to write a faster SNMP version as the one shown in figure 12 by taking advantage of the way the tcpConnTable is indexed.

trary management tasks.

The OSI management architecture uses object oriented concepts to define the management information base (MIB). MIB definitions are written in a format that is defined by the Guidelines for the Definitions of Managed Objects (GDMO) [16]. Definitions are organized in templates where each template consists of several constructs.

Access to GDMO definitions is provided by the `gdmo` command (figure 15). The `gdmo` command has an option for every GDMO template (class, package, parameter, attribute, group, action, behaviour, notification and namebinding). If called without arguments, a list of all known definitions is returned. Access to a specific construct of a definition requires to name the definition and the GDMO construct of interest.

```
gdmo class [<name> <construct>]
gdmo package [<name> <construct>]
gdmo parameter [<name> <construct>]
gdmo attribute [<name> <construct>]
gdmo group [<name> <construct>]
gdmo action [<name> <construct>]
gdmo behaviour [<name> <construct>]
gdmo notification [<name> <construct>]
gdmo namebinding [<name> <construct>]
gdmo load <file>
gdmo info <template>
```

Figure 15: The gdmo command.

The `gdmo load` command allows to load additional GDMO definitions. An embedded parser reads the GDMO file and creates an internal representation. The `gdmo info` command provides meta information about templates. It returns the constructs that are defined for a given template. This allows to write generic GDMO browsers without hard-coding names of GDMO constructs into Tcl code.

6 Applications

We have implemented a number of applications during the development of the Scotty extension. This prototyping approach allowed us to gain practical experience with our extensions, which produced positive feedback for the design process. Some of our applications are very small Tcl scripts e.g. to monitor the interface load via SNMP. More complex applications have been developed as part of our Tcl-based network management platform. This platform consists of three main parts:

1. The most visible part is a graphical user interface called Tkined (Tk-based interactive network editor) which allows applications to access and manipulate a graphical representation of the network. Tkined defines object types like nodes, networks or links and offers additional services like predefined dialogs or output windows. Stripchart or barchart objects may be used to display data gathered from the network. The network editor is a Tk-based implementation of an earlier version which was based on the InterViews/Unidraw toolkit [14].

2. The second component of our platform is a configuration database. We started with our own database server. However, since our server did not provide query processing facilities, we are now switching to the mSQL database backend [7] and the Tcl interface msqltcl. mSQL is a very fast implementation of a subset of the SQL standard. This allows to use a full-featured commercial database system like Oracle or Sybase by replacing the msqltcl interface with the appropriate Tcl extension (oratcl, sybtcl) with little changes to our Tcl code.

3. The third component is called Scotty and the subject of this paper. Scotty is responsible to provide access to various network services. Some additional extensions have been developed e.g. to integrate a network simulator.

Our management platform clearly separates management applications from the user interface. Access to the user interface is limited to the modification of Tkined objects and some predefined dialogs. This restricts the options available to management application programmers to control the user interface, but we feel that the win of getting a uniform interface for all our tools justifies this approach.

Access to Tkined objects by a Scotty process is implemented using pipes and is encapsulated in a Tcl command. This allows us to keep Tcl interpreter running management applications small as they do not need to contain any X11 code. Applications built with Scotty and Tkined include network monitoring applications, troubleshooting utilities and a network discovery tool [13].

Still the most frequently used network management tools are MIB browsers that allow to inspect MIB definitions and to retrieve data from management agents. We have written a SNMP MIB browser[3] and a GDMO MIB browser[4] that are integrated into the World Wide Web (WWW). The GDMO WWW browser creates cross reference information and the SNMP WWW browser allows to retrieve MIB variables by specifying target hosts. Other MIB browsers have been written for our management platform.

Another application area for the Scotty Tcl extensions is the development of special purpose MIB and GDMO compilers. The parser built into our extensions can be turned into a compiler by adding a few lines of Tcl code to create the desired output. The SNMP and GDMO WWW browsers can actually be thought of as compilers that convert MIB definitions into WWW documents.

7 Smart Agents written in Tcl

Management of large networks requires distributed management facilities, because centralized solutions do not scale very well. The Internet world has recognized the need of distributed network management and started to develop MIBs which support distributed management (e.g RMON [17], M2M [4]).

Although the RMON MIB does a good job in monitoring network traffic, there is an increasing need for agents which allow more flexibility. If a site is running special purpose software which should be controlled by the network management system, you need to write special purpose agents. We have therefore extended our SNMP interface with commands that make it possible to implement SNMP agents as Tcl scripts. Figure 16 shows the necessary extensions to the SNMP interface described in section 4.

```
$s agent
snmp instance <label> <var> [<default>]
snmp bind <label> <operation> [<script>]
snmp walk <var> <body>
```

Figure 16: More snmp commands.

The `agent` option turns a SNMP session into an agent session. The session parameters determine the agent configuration. MIB instances are created by linking global Tcl variables to MIB instances using the `snmp instance` command.

Incoming SNMP requests are answered automatically by the SNMP protocol engine. A reply packet is assembled by reading the values of the Tcl variables as requested by the received SNMP packet. The `snmp bind` command allows to bind scripts to MIB variables which are evaluated when a SNMP packet is processed. The operations currently supported are `get`, `set`, `create`, `commit` and `rollback`. The later two operations are required to implement the "as if simultaneous" semantics required by the SNMP set operation.

You can define bindings for any node in the MIB tree. Bindings are evaluated starting from the leaf nodes of the MIB tree up to the root. This mechanism allows to define bindings for an individual instance, a set of instances with a common prefix, a column of a conceptual table or a complete MIB subtree. Special attention is paid to return codes. If a script invokes the `break` command, then no other bindings will be evaluated.

Information about the received SNMP request is available by using % substitutions. Frequently used sequences are the instance identifier (%i) or the value contained in a set request (%v). Errors occurring during the evaluation of a binding are mapped to SNMP error messages. This allows to implement standard mechanisms to create and delete rows in conceptual tables where it is required to reject set requests with error messages like inconsistentValue, depending of the current state of the row.

```
[snmp session -port 1701] agent
snmp instance tclVersion.0 \
    tclVersion [info tclversion]
snmp instance tclCmdCount.0 tclCmdCount
snmp bind tclCmdCount.0 get {
    set tclCmdCount [info cmdcount]
}
```

Figure 17: A simple SNMP agent.

Figure 17 shows a simple SNMP agent which exports two scalars named tclVersion.0 and tclCmdCount.0. The MIB instance tclCmdCount.0 is linked to the Tcl variable tclCmdCount. The binding makes sure that every get operation on this variable returns the actual value of the Tcl command counter.

Running the Scotty extensions in manager and agent role allows to build very powerful Mid-Level-Manager. It is possible to submit a Tcl script to an agent which starts to perform management operations defined in the script without any further interaction with the manager. This approach to network management which is based on smart agents that are able to execute management scripts is termed man-

[3] http://www.cs.tu-bs.de/ibr/cgi_bin/sbrowser.tcl
[4] http://www.cs.tu-bs.de/ibr/cgi_bin/gbrowser.tcl

agement by delegation [18]. The key idea is to move management applications to the data they process. For example, the `SnmpTcpUser` script shown in figure 12 could be easily transmitted to a Scotty agent to perform the count locally.

Various SNMP MIBs have been proposed to transfer management scripts to smart agents (e.g. [5]). While it is possible to implement such a MIB using the Scotty extension, we feel that other protocols are more suited to this task. This is mainly due to the fact that message size limitations require that a script is split into single lines that are transmitted individually as table rows. The receiver has to concatenate the rows once the transfer is finished.

```
snmp instance scottyHttpProxy.0 scottyHttpProxy
snmp instance scottyHttpSource.0 scottyHttpSource
snmp instance scottyHttpError.0 scottyHttpError

snmp bind scottyHttpProxy.0 set {
    catch { http proxy "%v" }
    set scottyHttpProxy [http proxy]
}

snmp bind scottyHttpSource.0 set {
    set scottyHttpSource ""
    set scottyHttpError ""
    set file /tmp/http.[pid]
    set rc [catch { http get "%v" $file } msg]
    if {$rc} {
        set scottyHttpError $msg
        return
    }
    set rc [catch { source $file } msg]
    if {$rc} {
        set scottyHttpError $msg
        return
    }
    catch { exec rm -f $file }
    set scottyHttpSource "%v"
}
```

Figure 18: Retrieving scripts via HTTP.

A much simpler solution to this problem is shown in figure 18. This script implements three SNMP variables that allow to retrieve management scripts via HTTP. The scottyHttpSource.0 variable can be set to an URL that points to a file which contains a Tcl script to source. Errors are stored in the scottyHttpError.0 variable. The scottyHttpProxy.0 variable allows to define a HTTP proxy server which may act as a gateway to other transfer protocols like FTP.

8 Conclusions

In this paper, we presented extensions to Tcl that have been used to implement network management applications in a script language. Our decision to use Tcl was motivated by our needs to use a well tested and highly portable language that is easy to learn and provides a higher level of abstraction compared to languages like C or C++. Many people feel very familiar with command languages which was our reason not to use languages like scheme.

While Tcl was a great pleasure to use, we also identified a few problems. Some of them are well known and will be addressed in future Tcl versions (e.g. dynamic loading). The integration of the Safe-Tcl extension [3] into the core of Tcl would be a big win for us as our smart agents need such a mechanism to restrict the set of Tcl commands according to the authentication mechanism in use.

Another problem that will probably not be fixed very soon is the `expr` command of Tcl. The SNMP framework defines data types to hold 32 bit signed and unsigned integers as well as 64 bit unsigned integers. Even simple computations can get quite tricky if they should return correct results on 32 bit and 64 bit machines as Tcl does not define machine independent arithmetics. In some cases, we simply use floating point arithmetics to avoid this problem. However, this is not a good solution as we loose precision which could be very important in a network management application.

Our SNMP Tcl interface described in section 4 has been developed in parallel with the interface defined by M. Rose and K. McCloghrie [12]. Both interfaces provide similar services. However, our implementation supports more flexible callbacks and synchronous operations, which are very handy sometimes. The most important difference is our SNMP agent interface as described in section 7.

The implementation of our SNMP protocol engine does not fully conform to the standards. There is currently no way to define different views and there is no access to the SNMP party tables used in SNMP version 2. This will be added once the SNMP working group finishes its current work which will hopefully result in a standard configuration model.

The `cmip` command described in section 5 was implemented to experiment with the OSI management protocol. It is based on the last public version of OSIMIS and ISODE. These packages are not very portable and known to create big binaries. A more serious problem is caused by the fact that the OSIMIS implementation is only able to handle GDMO data types that are known at compile time. A solution to this problem would require to implement a generic interface that will convert ASN.1/BER encoded OSI messages into a string based representation. However, it is not clear if such a project would pay off, given the small acceptance of OSI protocols in the Internet world.

Acknowledgement

Many people have contributed to our extensions. We would like to thank Erik Schönfelder for his valuable contributions and his helping hand. Sven Schmidt wrote the initial version of the SNMP extension and Michael Kernchen contributed the CMIP interface and the GDMO parser. Also many thanks to all the people who reported bugs and shared their ideas with us. Their feedback has been a major force to improve and enhance our software.

Availability

The sources of the Scotty extensions are available via anonymous ftp from `ftp.ibr.cs.tu-bs.de` in the directory `/pub/local/tkined`. There is also a mailing list for the discussion of topics related to Tkined and Scotty. To subscribe, send a message to `tkined-request@ibr.cs.tu-bs.de`.

References

[1] T. Berners-Lee, R.T. Fielding, and H. Frystyk Nielsen. Hypertext Transfer Protocol – HTTP/1.0. Internet Draft 01, MIT, UCLA, CERN, December 1994.

[2] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738, CERN, Xerox Corporation, University of Minnesota, December 1994.

[3] N.S. Borenstein. EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail. Technical report, 1994.

[4] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Manager-to-Manager Management Information Base. RFC 1451, SNMP Research Inc., Hughes LAN Systems, Dover Beach Consulting Inc., Carnegie Mellon University, April 1993.

[5] J.D. Case and D.B. Levi. SNMP Mid-Level-Manager MIB. Internet Draft 00, SNMP Research, Inc., October 1993.

[6] K. Harrenstien, M. Stahl, and E. Feinler. NICNAME/WHOIS. RFC 954, SRI, October 1985.

[7] D.J. Hughes. Mini SQL: A Lightweight Database Engine. Technical report, Bond University, January 1995.

[8] G. Knight, G. Pavlou, and S. Walton. Experience of Implementing OSI Management Facilities. In *Proc. International Symposium on Integrated Network Management*, pages 259–270, April 1991.

[9] P. Mockapetris. Domain Names - Concepts and Facilities. RFC 1034, ISI, November 1987.

[10] J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, April 1994.

[11] J. Postel. Internet Control Message Protocol. RFC 792, ISI, September 1981.

[12] M. Rose and K. McCloghrie. *How to Manage Your Network Using SNMP.* Prentice Hall, 1995.

[13] J. Schönwälder and H. Langendörfer. How To Keep Track of Your Network Configuration. In *Proc. 7 th Conference on Large Installation System Administration (LISA VII)*, pages 189–193, Monterey (California), November 1993.

[14] J. Schönwälder and H. Langendörfer. INED – An Application Independent Network Editor. In *Proc. World Conference On Tools and Techniques for System Administration, Networking, and Security*, Arlington, (Virginia), April 1993.

[15] B.C. Smith, L.A. Rowe, and S. Yen. Tcl Distributed Programming. In *Proc. 1st Tcl/Tk Workshop*, pages 50–51, June 1993.

[16] W. Stallings. *SNMP, SNMPv2 and CMIP: The Practical Guide to Network Management Standards.* Addison-Wesley, 1993.

[17] S. Waldbusser. Remote Network Monitoring Management Information Base. RFC 1757, Carnegie Mellon University, February 1995.

[18] Y. Yemini, G. Goldszmidt, and S. Yemini. Network Management by Delegation. In *Proc. International Symposium on Integrated Network Management*, pages 95–107, April 1991.

[19] D.P. Zimmerman. The Finger User Information Protocol. RFC 1288, Rutgers University, December 1991.