Specific Simple Network Management Tools

Jürgen Schönwälder – Technical University of Braunschweig

ABSTRACT

The Simple Network Management Protocol (SNMP) has been around for more than a decade and is supported by most network devices and end systems. Despite this success, there is still a lack of simple to use network management applications.

This paper describes the design of an SNMP management tool called `scli` which provides an easy and efficient to use SNMP command line interface. The software architecture has been designed to make it easy for C programmers without any special SNMP programming skills to extend the functionality provided by `scli`.

Introduction

The Simple Network Management Protocol (SNMP) [1, 2] has been around for more than a decade and is supported by most network devices and end systems. Despite its success in the network devices, there is still a lack of simple to use network management applications.

High-end management platforms such as HP's OpenView [3] or Aprisma's Spectrum [4] are relatively expensive and require special training in order to use them effectively. Especially operators of smaller enterprise networks often cannot afford the purchase and training costs associated with these high-end management platforms. So they often revert to low-level SNMP tools such as `snmpwalk` in combination with shell scripting languages to get a certain job done quickly. Others use SNMP extensions of well known scripting languages such as Perl or Tcl to implement their own little management solution. While this seems to be a good approach in the short-term, these scripts tend to be loaded with site-specific details which often prevents people from sharing them. Furthermore, these scripts are often fragile and hard to maintain since error handling is usually poor and documentation is often missing.

This paper describes the design and implementation of an SNMP management tool called `scli` which provides an efficient to use command line interface to display, modify and monitor data retrieved from SNMP agents. It runs on simple ASCII terminals and does not require any graphical user interface capabilities. The tool provides command line editing, completion and history capabilities to make it easy for network operators and system administrators to use this tool, even if they cannot remember the precise command syntax.

The `scli` commands are organized in a logical command tree and hide the details of the SNMP interactions and the underlying MIB data structures. The default `scli` output format is optimized for human readability and in some cases resembles the output produced by existing Unix commands.

Optimizing the default output format for human readability has the disadvantage that it becomes harder to use `scli` in scripts since parsing the output is complicated and error prone. In order to use `scli` as a mechanism to collect data to be stored in data bases or displayed on web pages, a second XML-based output format has been implemented for many of the `scli` commands.

The `scli` software architecture has been designed to allow C programmers without knowledge of low-level SNMP APIs to extend the functionality provided by `scli`. This hopefully encourages a larger group of people to write and share extensions for specific device types, protocols or managed services.

This paper is organized as follows. The first section discusses the difference between generic and specific SNMP tools and why there is a need of specific rather than generic tools. The next section describes the overall software design of the simple command line management tool `scli`. The following section discusses some of the `scli` commands and presents examples how they can be used in practice. Subsequent sections describe the implementation of `scli` and how it can be extended. The paper concludes with some remarks on future work.

Generic vs. Specific Tools

The tool described in this paper was written because of the author's continued frustration how complex and inconvenient it is to configure, troubleshoot and monitor SNMP manageable devices. An analysis of the tools available shows that most of them fall into one of the following five categories:

Generic low-level SNMP Tools. The first category includes simple low-level SNMP command-line tools such as `snmpwalk` or `snmpset`. These tools are low-level since they just provide command-line interfaces to perform a single or a sequence of related SNMP protocol operations on MIB variables. The tools generally do not understand the semantics of the data they manipulate. Furthermore, they require a certain amount of SNMP and MIB knowledge to interpret the results correctly.

Generic low-level SNMP APIs. The second category includes tools and libraries that give programmers a relatively low-level programmatic interface to invoke SNMP protocol operations and to access MIB definitions. Examples of generic low-level scripting APIs are WinSNMP [5], SNMP++ [6], NET-SNMP or the Tnm extension for Tcl [7, 8]. Many network operators and system administrators seem to prefer APIs that are based on scripting languages such as Perl or Tcl over APIs for system programming languages such as C or C++ since they are much easier to deal with.

Generic MIB Browsers. The third category includes programs that allow network operators to browse through MIB data on SNMP-enabled devices. Some MIB browsers use Web technologies for their user interface while the majority provides graphical user interfaces.
MIB browsers are usually designed as generic tools that do not really understand the semantics of the data they display and manipulate. Many browser are able to load and interpret MIB module definitions at run-time and some of the more advanced browsers allow users to customize the displays to a large extent. However, many important semantics described in MIB description clauses are not machine readable. Therefore, generic MIB browsers generally require that users are familiar with the semantics of MIB variables or at least able to read and understand MIB module definitions.

Generic Monitoring Tools. The fourth category includes generic monitoring tools. MRTG [9] is an example of a generic monitoring tool which is used in many networks to gather statistics and to detect unusual system behavior. Some of these tools have limitations that cause them to produce erroneous results in some situations (handling of counter discontinuities) if the user configuring these tools does not pay attention to special MIB semantics. Some more specific tools such as Cricket [10] have been implemented on top of these generic monitoring tools to simplify the configuration for typical use cases.

Generic Management Platforms. The fifth category consists of management platforms which provide a generic infrastructure for the implementation of network management applications. Applications written on top of these platforms make use of platform specific interfaces and services, such as protocol APIs or database services. Platforms also often include generic tools for monitoring, event correlation or topology discovery.
The `tkined` [11] and `gxsnmp` packages are examples of openly available platforms. More complex examples are commercial management platforms such as HP's OpenView [3] or Aprisma's Spectrum [4].

There are many SNMP management tools available today which fall into one of the five categories described above. But the author still often feels uncomfortable when trying to use them, even though he has implemented some of these generic tools himself in the past. MIB browsers which display raw MIB data structures tend to be of little use for actual management because MIB data structures are usually designed to be read by programs rather than humans. Furthermore, MIB modules undergo revisions as the networking technologies evolve. Sometimes, the original MIB module design turns out to be problematic and the attempts to maintain backwards compatibility while supporting new features makes MIB module data structures often hard to understand.

Another problem are the naming schemes used in MIB modules, which are typically optimized for machines and which do not necessarily reflect what humans prefer to use. For example, humans prefer to identify network interfaces by names, such as "eth0". MIB modules, however, use numbers to identify network interfaces [12]. Furthermore, on some devices, these numbers can change with every re-initialization. A good specific tool should allow users to refer to interfaces by name and it should hide the SNMP and MIB specific naming details.

Generic tools often do not understand the relationships between MIB objects. For example, consider the speed of a network interface. There are two standardized objects in the IF-MIB [12] that report the speed of a network interface (`ifSpeed` and `ifHighSpeed`) and a good application should understand their semantics and relationship and display the correct value in a meaningful way. Generic MIB browsers are not able to do that and put the burden on the user to understand how `ifSpeed` and `ifHighSpeed` relate to each other and to pick the right value.

It seems that the approach to build generic tools that expose raw MIB data structures and which require that the network operator or system administrator has SNMP and MIB knowledge does not work very well in practice. There is a need for simple tools that are specific instead of generic in the sense that they understand the semantics of the data they manipulate and hide low-level SNMP and MIB details. Specific tools should be designed to do just one thing and they should attempt to do it right. Specific management tools must be written by programmers who do understand the semantics of the MIB objects as well as the conceptual model behind the relevant MIB modules [13].

The `scli` tool, which was born in January 2001 and which has been openly available since February 2001, has since its first release improved in many directions and it shows that the implementation of efficient specific network management tools such as `scli` can be easy and fun if one chooses a suitable software design.

Software Design

This section discusses the software design behind `scli`. The software design addresses five key requirements:

1. The first requirement is extensibility. The software design should make it relatively easy to add new features to `scli`. This requires that the internal APIs are as simple as possible. Furthermore, the code must be obvious so that people can easily derive extensions from the existing code base. In order to get many programmers involved, it is necessary to hide low-level SNMP communication details as much as possible.
2. The second requirement is robustness. The software design should ensure as much as possible that errors are detected and handled gracefully where possible. This implies to use facilities which help to avoid problems such as buffer overruns and to validate data received from the network before processing it. Furthermore, the program should abort if coders forget to check for possible error conditions as soon as possible so that bugs are noticed and fixed.
3. The third requirement is maintainability. It must be possible to evolve the software over time, which includes internal API changes. Furthermore, it is important to ensure that the documentation is available and in sync with the implementation since `scli` users are not expected to read MIB modules in order to use `scli`. Finally, it is important to stay focussed in the overall scope so that the resources available can be used effectively.
4. The fourth requirement is efficiency. The implementation should be efficient regarding the amount of resources needed to implement a given management operation. This is of special importance if the tool is used in scripts that perform more complex management tasks. Some of the well-known generic low-level SNMP tools consume noticeable resources while parsing MIB files during startup and are thus inefficient if they are called frequently in scripts.
5. The fifth requirement is portability (at least across Unix platforms). A port to Win32 platforms should be possible, although the author does not really have a need for such a port himself.

Implementation Language

The author's experience with the Tnm extension for Tcl, a generic low-level SNMP API [7], shows that only a few Tcl coders actually contribute scripts written on top of the Tnm API back to the package maintainer. And those who do so usually do not care too much about the overall code organization and the scripts often depend on side specific details. While the original motivation behind Tnm was to provide a solid SNMP scripting API which should make it easy for people to create a repository of useful management scripts, the overall success in reaching that goal is rather limited. In fact, many management scripts turn out to be rather fragile and trying to maintain them is relatively costly.

It is interesting to note that the author's experience with other open source projects that are coded in C is quite different. Contributed patches for C code are often of good quality and much easier to integrate and maintain. Furthermore, compiled languages greatly help to detect many potential problems and inconsistencies if internal APIs change. It was therefore decided to implement `scli` entirely in C.

C++ was also considered as a potential implementation language but was finally rejected since the benefits of C++ over C relative to the requirements stated above are limited and C is still more portable and efficient and the number of C programmers is still bigger than the number of C++ programmers. The Java language was considered but not selected since the resource consumption is noticeable.

Software Architecture

The overall software architecture is shown in Figure 1. The package uses the `glib` library to achieve portability and to reuse generic data structures such as lists and dynamic strings. The SNMP engine `gsnmp` has been derived from the `gxsnmp` package and was subsequently modified to fix bugs and to improve stability. The SNMP engine itself uses `glib`.

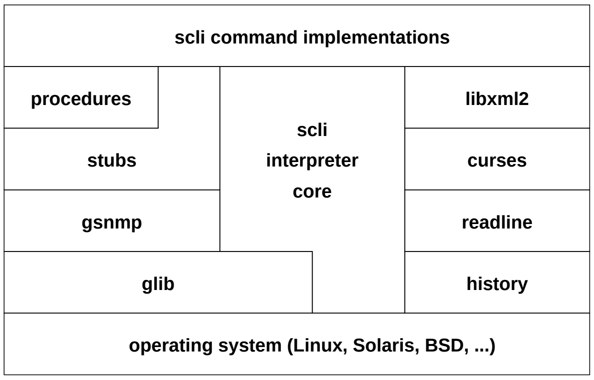


Figure 1: Software architecture.

The `gsnmp` library provides roughly the same low-level functionality as many other SNMP APIs. Since it was felt necessary to hide programmers from the low-level SNMP programming details, it was decided to use a MIB compiler to generate C stubs from MIB modules. The stubs consist primarily of C structures which represent MIB table rows or groups of scalars plus a set of stub functions which can be used to read/write these structures. The implementations of the stub functions serialize/deserialize the C structures into SNMP varbind lists. They also validate the data to ensure that the elements of the varbind lists have appropriate types and sizes. The stub code generator is further described below.

The `scli` command implementations either use the stubs directly or they use so called MIB procedures. MIB procedures extend the stub interface with specific functions for common operations like creating rows in certain MIB tables or iterating over certain MIB tables. MIB procedures are by definition MIB specific and implemented entirely by using the stub interface.

The `scli` interpreter core provides all the infrastructure needed to register commands, to tokenize the input stream, to locate and execute the function implementing a recognized command and to finally display the results on stdout or via a pager. The interpreter uses the GNU `readline` and `history` libraries for command line editing and the `curses` library for screen management. It also uses `glib` data types internally. All state information is bound to the `scli` interpreter. It is thus possible to have multiple `scli` interpreters in a single process – although this feature is currently not used.

The design decision to implement our own interpreter instead of using one of the available embeddable command interpreters was driven by the observation that the features needed by `scli` are very small and most interpreters such as Tcl are too heavy weight these days for simple tools like `scli`.

The interpreter core and some command implementations also use the `libxml2` library to create and manipulate XML documents. By using a dedicated XML library, it is possible to ensure that any generated XML output is well-formed.

User's View

This section describes `scli` from a user's point of view. It briefly introduces the basics about the structure of `scli` commands before presenting some examples how `scli` can be used.

Command Overview

All `scli` commands are hierarchically structured with a small set of top-level commands. The first five top-level commands (`open`, `close`, `exit`, `help`, `history`) are used to open and close SNMP sessions and to help with user interactions. The remaining top-level commands can be used to create/delete something (`create`, `delete`), to modify something (`set`), to display or monitor data (`show`, `monitor`), or to produce an `scli` script which restores the current configuration (`dump`). As an example, Figure 2 shows part of the `set` command hierarchy.

```
- set
  |- system
  |  |- contact
  |  |- name
  |  `- location
  |- ip
  |  |- forwarding
  |  `- ttl
  `- interface
     |- status
     |- alias
     |- notifications
     `- promiscuous
```

Figure 2: `scli set` command hierarchy.

Commands are also logically organized into `scli` modes. Figure 3 shows the syntax of the commands provided by the `interface` mode. Note that interfaces can be selected by regular expressions that are matched against the interface description. This allows users to perform a single operation on a set of interfaces.

```
set interface status <regexp> <status>
set interface alias <regexp> <string>
set interface notifications <regexp> <value>
set interface promiscuous <regexp> <bool>
show interface info [<regexp>]
show interface details [<regexp>]
show interface stack [<regexp>]
show interface stats [<regexp>]
monitor interface stats [<regexp>]
dump interface
```

Figure 3: `scli` interface mode.

`scli` supports the concept of recursive command evaluation, which is especially useful for the `show` and `dump` commands. The `show interface` command will retrieve and display all information about all network interfaces while the `show` command will retrieve and display all information available from a device.

Interactive Browsing and Monitoring

The `show` and `monitor` commands can be used to interactively inspect and monitor a device. The screenshot in Figure 4 shows how `scli` displays the containment hierarchy of the physical entities that make up a router. Additional `scli` commands can be used to get more detailed information about the physical entities.

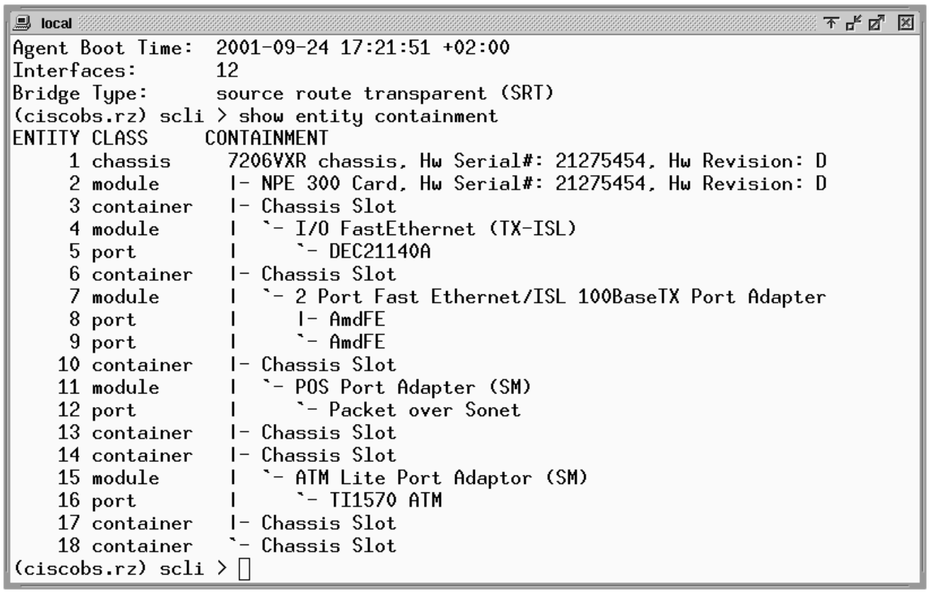
Sometimes it is necessary to quickly monitor some core statistics in order to track down a network problem. The `monitor` commands can be used for this purpose. Figure 5 shows a screenshot where `scli` shows basic statistics and status information for the network interfaces of a router. Note how `scli` handles data not available for some ATM layers.

The look & feel of the monitoring commands is similar to the well-known Unix `top` command. The top half of the screen displays basic summary information. Special keys can be used to customize the monitoring display.

Configuring Virtual LANs

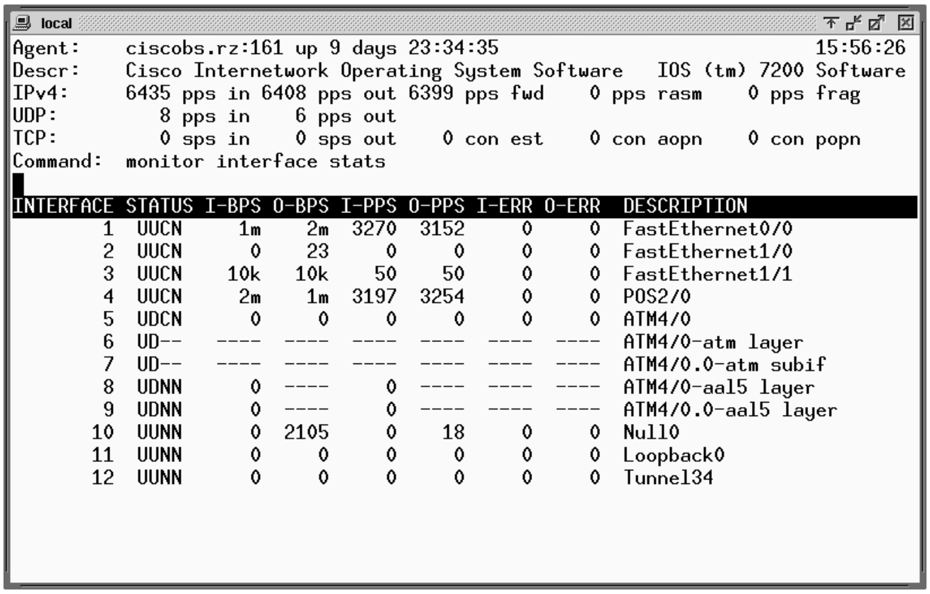
LAN bridges (sometimes called layer 2 switches) can often be configured via SNMP. The author recently had a need to configure virtual LANs across a number of bridges. The telnet interface provided by the bridges is menu-driven and not easy to handle for automated configuration. However, the bridges support vendor specific MIB objects to allow configuration via SNMP. (Unfortunately, the devices do not support the standard MIB for virtual LANs as defined in RFC 2674 [14].)

The approach to solve the configuration problem was to extend `scli` with commands that can create/delete virtual LANs and commands to assign ports to them. This allows to save the virtual LAN configuration for each bridge in a simple ASCII file. By using the `m4` macro processor, it is easy to import shared bridge configuration commands and to use symbolic names for port sets. Figure 6 shows the `scli` script to install the virtual LANs. Note the regular expression at the beginning to first remove all relevant virtual LANs. Figure 7 shows the `scli` script which configures the ports on a particular bridge.



```
Agent Boot Time:  2001-09-24 17:21:51 +02:00
Interfaces:       12
Bridge Type:      source route transparent (SRT)
(ciscobs.rz) scli > show entity containment
ENTITY CLASS     CONTAINMENT
     1 chassis     7206VXR chassis, Hw Serial#: 21275454, Hw Revision: D
     2 module      |- NPE 300 Card, Hw Serial#: 21275454, Hw Revision: D
     3 container   |- Chassis Slot
     4 module      |  `- I/O FastEthernet (TX-ISL)
     5 port        |     `- DEC21140A
     6 container   |- Chassis Slot
     7 module      |  `- 2 Port Fast Ethernet/ISL 100BaseTX Port Adapter
     8 port        |     |- AmdFE
     9 port        |     `- AmdFE
    10 container   |- Chassis Slot
    11 module      |  `- POS Port Adapter (SM)
    12 port        |     `- Packet over Sonet
    13 container   |- Chassis Slot
    14 container   |- Chassis Slot
    15 module      |  `- ATM Lite Port Adaptor (SM)
    16 port        |     `- TI1570 ATM
    17 container   |- Chassis Slot
    18 container   `- Chassis Slot
(ciscobs.rz) scli >
```

Figure 4: `scli` showing the containment structure of router components.



```
Agent:     ciscobs.rz:161 up 9 days 23:34:35                          15:56:26
Descr:     Cisco Internetwork Operating System Software   IOS (tm) 7200 Software
IPv4:      6435 pps in 6408 pps out 6399 pps fwd    0 pps rasm    0 pps frag
UDP:          8 pps in    6 pps out
TCP:          0 sps in    0 sps out    0 con est    0 con aopn    0 con popn
Command:  monitor interface stats

INTERFACE STATUS I-BPS O-BPS I-PPS O-PPS I-ERR O-ERR  DESCRIPTION
        1  UUCN     1m    2m  3270  3152     0     0  FastEthernet0/0
        2  UUCN      0    23     0     0     0     0  FastEthernet1/0
        3  UUCN    10k   10k    50    50     0     0  FastEthernet1/1
        4  UUCN     2m    1m  3197  3254     0     0  POS2/0
        5  UDCN      0     0     0     0     0     0  ATM4/0
        6  UD--   ----  ----  ----  ----  ----  ----  ATM4/0-atm layer
        7  UD--   ----  ----  ----  ----  ----  ----  ATM4/0.0-atm subif
        8  UDNN      0  ----     0  ----  ----  ----  ATM4/0-aal5 layer
        9  UDNN      0  ----     0  ----  ----  ----  ATM4/0.0-aal5 layer
       10  UUNN      0  2105     0    18     0     0  Null0
       11  UUNN      0     0     0     0     0     0  Loopback0
       12  UUNN      0     0     0     0     0     0  Tunnel34
```

Figure 5: `scli` monitoring network interface statistics.

```
# cleanup - regexps are cool :-)
delete nortel bridge vlan "^(134|ibr-)"

# IBR vlans (544-559)   134.169.34.*
create nortel bridge vlan 544 ibr-core
create nortel bridge vlan 545 ibr-cip
create nortel bridge vlan 546 ibr-test
create nortel bridge vlan 547 ibr-wlan
```

Figure 6: `scli` script for creating virtual LANs.

```
define(UP,‘25,185’)      # uplink ports
define(WLAN,‘2,56’)      # wireless vlan
define(CORE,‘1,3-24,33-55,65-88’)
                         # core vlan

# create the vlans:

include(vlan-all.scli)

# vlan port assignments:

set nortel bridge vlan ports \
                 ibr-core UP,CORE
set nortel bridge vlan default \
                  ibr-core CORE

set nortel bridge vlan ports \
                  ibr-wlan UP,WLAN
set nortel bridge vlan default \
                  ibr-wlan UP,WLAN
```

Figure 7: `scli` script which creates virtual LANs and assigns ports.

It is also useful to be able to dump the virtual LAN configuration via the `dump` command from the device in order to check whether it matches the configuration that is supposed to be on the device.

Generating HTML Status Pages

It is often convenient to generate HTML status pages for some devices (such as printers) which are linked to the Intranet. These status pages allow users to figure out why for example a print job does not progress in the print queue by looking at a virtual printer console. The `Printer-MIB` [15], which is supported by many printers, provides a simple way to read the console display and the status of the printer lights.

Generating HTML status pages is straight-forward since `scli` can generate XML output. XSL transformations can turn the `scli` XML output into a nice HTML page. Figure 8 shows the core of a transformation which shows console lights as an HTML table.

Programmer's View

This section introduces `scli` from a programmer's point of view. It first describes the stub code generator before explaining how a simple `scli` command to display the console lights of printers is implemented and registered.

Stub Generator

The stub code generator is a key component since it hides the low-level SNMP communication details. The stub code generator takes a MIB module as input and generates a pair of .h and .c files for the MIB module. The header file contains C type definitions for MIB table rows or groups of scalars. The SMI data types are mapped to `glib` data types according to the base data type model used by the SMIng proposal [16]. Some additional structure members whose names start with an underscore are introduced when dealing with variable size MIB variables.

The members of the generated C structures are usually pointers. This reflects the fact that SNMP agents are not required to return values for all variables, either due to implementation limitations or due

```
<xsl:template match="console">
  <table>
    <tr>
      <xsl:for-each select="light">
        <xsl:element name="td">
          <xsl:attribute name="width">60</xsl:attribute>
          <xsl:attribute name="align">center</xsl:attribute>
          <xsl:if test="status != 'off'">
            <xsl:attribute name="bgcolor">
              <xsl:value-of select="color"/>
            </xsl:attribute>
          </xsl:if>
          <xsl:if test="status = 'blink'">
            <xsl:element name="span">
              <xsl:attribute name="style">text-decoration:blink</xsl:attribute>
            </xsl:element>
          </xsl:if>
          <xsl:apply-templates select="description"/>
        </xsl:element>
      </xsl:for-each>
    </tr>
  </table>
</xsl:template>
```

Figure 8: XSL transformation for `show printer console` XML output.

to access control. Variables that are not accessible will be represented by a NULL pointer. The decision to use pointers requires that programmers check carefully whether the pointers are valid before using them. Failure to do so will result in segmentation faults – a clear indication that the program is buggy and must be fixed. An alternative option would have been to introduce special bit fields which indicate whether a given data member is valid or not. This option was rejected since programmers will likely forget to check these bit fields and programs will operate on invalid data without being noticed.

The generated header file also defines several stub functions that read/write the C structures from/to SNMP agents. Stub functions that retrieve complete MIB tables return the data to the application as an array of pointers to the C structures representing table rows. The read/write stub functions also have a mask argument which can be used to specify that only a subset of the members of the C structure should be read/written.

The implementation of the stub functions is contained in the .c files. The table retrieval stubs generate a sequence of suitable SNMP requests to read a table. Holes in tables are handled automatically and data which can be obtained by unpacking instance identifiers is not retrieved explicitly in order to save some bandwidth.

The data contained in response messages is first validated by doing some basic type and range/size checking. The instance identifier is unpacked and validated before C structures are filled with appropriate values. Detected errors are signaled using `glib` warnings and the values are ignored. This ensures that SNMP communication problems are noticed and that applications using the stubs only operate on validated values.

The stubs also provide mapping tables for enumerations. These tables can be used to map numbers or object identifier values to labels and vice versa. However, in many cases, the labels assigned in MIB modules are rather useless for direct display because they are either too cryptic or simply too long. It is thus not uncommon to implement more specific mapping tables in addition to the tables generated by the MIB compiler.

Figure 9 shows the stubs that are generated for the `prtConsoleLightEntry` of the Printer-MIB

```
/*
 * C type definitions for Printer-MIB::prtConsoleLightEntry.
 */

typedef struct {
    gint32   hrDeviceIndex;
    gint32   prtConsoleLightIndex;
    gint32   *prtConsoleOnTime;
    gint32   *prtConsoleOffTime;
    gint32   *prtConsoleColor;
    guchar   *prtConsoleDescription;
    gsize    _prtConsoleDescriptionLength;
} printer_mib_prtConsoleLightEntry_t;

extern void
printer_mib_get_prtConsoleLightTable(GSnmpSession *s,
        printer_mib_prtConsoleLightEntry_t ***prtConsoleLightEntry,
        gint mask);

extern void
printer_mib_free_prtConsoleLightTable(
        printer_mib_prtConsoleLightEntry_t **prtConsoleLightEntry);

extern printer_mib_prtConsoleLightEntry_t *
printer_mib_new_prtConsoleLightEntry(void);

extern void
printer_mib_get_prtConsoleLightEntry(GSnmpSession *s,
        printer_mib_prtConsoleLightEntry_t **prtConsoleLightEntry,
        gint32 hrDeviceIndex, gint32 prtConsoleLightIndex, gint mask);

extern void
printer_mib_set_prtConsoleLightEntry(GSnmpSession *s,
        printer_mib_prtConsoleLightEntry_t *prtConsoleLightEntry,
        gint mask);

extern void
printer_mib_free_prtConsoleLightEntry(
        printer_mib_prtConsoleLightEntry_t *prtConsoleLightEntry);
```

Figure 9: Stub interface for `prtConsoleLightEntry` of the `Printer-MIB` (RFC 1759).

[15] which describes the status of a printer console light. All type and function declarations are prefixed by the MIB module name in order to deal with potential name clashes. The first two stub functions operate on complete tables while the remaining stub functions operate on table rows. The stub code generator has been implemented as an output driver of the `smidump` MIB compiler.

Command Implementation

Figure 10 shows the implementation of the `show printer console lights` command. Commands are implemented as C functions which are called with a handle for the `scli` interpreter and the command arguments as input and return an `scli` return code. The commands usually first check the command arguments before retrieving the data they manipulate. If the retrieval was successful, they start manipulating the data. The last section of a command implementation is responsible to free any allocated resources.

The `show_printer_console_lights()` function shown in Figure 10 first iterates over the retrieved table to calculate the maximum length of the description strings. The second iteration calls an output formatting function for each table row, depending on the current state of the `scli` interpreter. The result is written into the interpreter, which provides either `glib` dynamic strings or a suitable pointer to an `libxml2` node.

The default output formatting function for the `show printer console lights` command is shown in Figure 11. It uses a utility function `fmt_enum()` to lookup the label for a color number and it does some computations to figure out whether the light is on, off or blinking.

Command Registration

Command implementations must be registered in the `scli` interpreter as shown in Figure 12. This is accomplished by creating an array of command descriptions. Each command description contains the command name, the description of the arguments accepted by the command, the documentation of the command and some command flags. Commands that are able to produce XML output also describe the XML path and the XML Schema definition (not shown in Figure 12).

Commands always belong to an `scli` mode. The structure which describes an `scli` mode contains the name of the mode, the documentation of the mode and the commands it provides.

```
static int
show_printer_console_lights(scli_interp_t *interp, int argc, char **argv)
{
    printer_mib_prtConsoleLightEntry_t **lightTable;
    int i;
    int light_width = 12;

    if (argc > 1) return SCLI_SYNTAX;

    printer_mib_get_prtConsoleLightTable(interp->peer, &lightTable, 0);
    if (interp->peer->error_status) return SCLI_SNMP;

    if (lightTable) {
        for (i = 0; lightTable[i]; i++) {
            if (lightTable[i]->_prtConsoleDescriptionLength > light_width) {
                light_width = lightTable[i]->_prtConsoleDescriptionLength;
            }
        }
        if (! scli_interp_xml(interp)) {
            g_string_sprintfa(interp->header,
                              "PRINTER LIGHT %-*s STATUS COLOR",
                              light_width, "DESCRIPTION");
        }
        for (i = 0; lightTable[i]; i++) {
            if (scli_interp_xml(interp)) {
                xml_printer_console_light(interp->xml_node, lightTable[i]);
            } else {
                fmt_printer_console_light(interp->result, lightTable[i],
                                          light_width);
            }
        }
    }

    if (lightTable) printer_mib_free_prtConsoleLightTable(lightTable);

    return SCLI_OK;
}
```

Figure 10: Implementation of the `show printer console lights` command.

All command and mode documentation is defined in the C code and registered within the interpreter. This encourages programmers to provide documentation when implementing new commands. The `scli` manual page and other documentation is generated automatically from the output produced by the `show scli modes` and the `show scli schema` commands.

The printer mode registration shown in Figure 12 registers the function `show_printer_console_lights()` twice. The second registration causes the command to be executed periodically since it sets the `SCLI_CMD_FLAG_MONITOR` flag.

Conclusions

This paper first motivated the need for specific rather than generic SNMP-based management tools. It then presented the overall software design for a specific management tool called `scli` before discussing `scli` in some more depth from the user's and the programmer's point of view.

The evolution of `scli` so far has shown that the approach of using compiler generated stubs to hide low-level SNMP communication details is feasible. Furthermore, the number of commands available is already large enough to inspect, monitor and configure devices using `scli`.

There are of course a number of areas where additional work can be done. The SNMP engine does not yet support SNMPv3 security [2]. SNMPv3 is slowly getting more widespread deployment and it would be nice to take advantage of strong security, especially for `set` and `create` commands.

The code generator can be improved in many ways. The biggest limitation right now is the restriction that stubs can only operate on table rows or groups of scalars. It is sometimes desirable to have atomic SNMP set operations on varbinds that include tabular data and scalars. The prominent example are spin-lock variables such as `snmpSetSerialNo` [17].

Many of the current MIB procedures follow similar patterns and it would be convenient to generate them automatically from formal MIB annotations. An annotation language would perhaps also enable us to automatically generate suitable caching strategies in order to reduce the amount of data retrieved from SNMP agents. However, some more experimentation is needed to better understand the requirements for such an annotation language.

Finally, it would be nice to have more command implementations. People who like the tool and its design are therefore encouraged to write new modes

```
static void
fmt_printer_console_light(GString *s,
                  printer_mib_prtConsoleLightEntry_t *lightEntry,
                          int light_width)
{
    const char *state = "off";
    const char *e;

    g_string_sprintfa(s, "%6d  ", lightEntry->hrDeviceIndex);
    g_string_sprintfa(s, "%4d  ", lightEntry->prtConsoleLightIndex);

    if (lightEntry->prtConsoleDescription) {
        g_string_sprintfa(s, "%-*.*s ", light_width,
                   (int) lightEntry->_prtConsoleDescriptionLength,
                         lightEntry->prtConsoleDescription);
    } else {
        g_string_sprintfa(s, "%*s", light_width, "");
    }

    if (*lightEntry->prtConsoleOnTime
        && !*lightEntry->prtConsoleOffTime) {
        state = "on";
    } else if (!*lightEntry->prtConsoleOnTime
               && *lightEntry->prtConsoleOffTime) {
        state = "off";
    } else if (*lightEntry->prtConsoleOnTime
               && *lightEntry->prtConsoleOffTime) {
        state = "blink";
    }
    g_string_sprintfa(s, " %-*s ", 5, state);

    e = fmt_enum(printer_mib_enums_prtConsoleColor,
                 lightEntry->prtConsoleColor);
    g_string_sprintfa(s, "%s\n", e ? e : "");
}
```

Figure 11: Implementation of the `show printer console lights` formatter.

for their favorite devices or protocols and to contribute them to the `scli` project.

Acknowledgments

The author likes to thank Frank Strauss for the many fruitful discussions which helped to shape the design of `scli` and his XSL transformations. The author is also grateful to the `scli` users who have contributed patches and who maintain packages for various Linux systems.

Availability

The SNMP command line interface `scli` has been released under the terms of the GNU General Public License version 2. The project home page is http://www.ibr.cs.tu-bs.de/projects/scli/ . Debian and RPM packages for Linux systems have been contributed by `scli` users.

The stub code generator has been integrated into the `libsmi` package which has been released under Berkeley copyright conditions. The project home page is http://www.ibr.cs.tu-bs.de/projects/libsmi/ .

Biography

Jürgen Schönwälder received his diploma in computer science in 1990 and his doctoral degree in 1996 from the Technical University of Braunschweig, Germany. His research interests are network management, distributed systems and network security. He has co-authored several network management related RFCs and is currently the chair of the Network Management Research Group (NMRG) of the Internet Research Task Force (IRTF).

References

[1] Stallings, W., *SNMP, SNMPv2, SNMPv3, and RMON1 and 2*, Addison-Wesley, Third edition, 1999.

[2] Case, J., R. Mundy, D. Partain, and B. Stewart, "Introduction to Version 3 of the Internet-standard Network Management Framework," *RFC 2570*, SNMP Research, TIS Labs at Network Associates, Ericsson, Cisco Systems, April, 1999.

[3] Blommers, J., *OpenView Network Node Manager: Designing and Implementing an Enterprise Solution*. Prentice Hall PTR, 2000.

[4] Lewis, L., *Managing Business and Service Networks*, Kluver Academic/Plenum Publishers, 2001.

[5] Natale, B., "WinSNMP v2.0 – Evolution of an Industry-standard API," *Simple Times*, Vol. 6, Num. 1, March, 1998.

[6] Mellquist, P. E., "SNMP++: An Object Oriented Approach to Network Management Programming," *Simple Times*, Vol. 7, Num. 1, March, 1999.

```
void
scli_init_printer_mode(scli_interp_t * interp)
{
    static scli_cmd_t cmds[] = {

        { "show printer console lights", NULL,
          "The show printer console lights command shows the current"
          "status of the printer's lights. [...]",
          SCLI_CMD_FLAG_NEED_PEER | SCLI_CMD_FLAG_XML,
          "printer console",
          "<xsd> <!-- ... --> </xsd>",
          show_printer_console_lights },

        { "monitor printer console lights", NULL,
          "The monitor printer console lights command shows the same"
          "information as the show printer console lights command. The"
          "information is updated periodically.",
          SCLI_CMD_FLAG_NEED_PEER | SCLI_CMD_FLAG_MONITOR,
          NULL, NULL,
          show_printer_console_lights },

        { NULL, NULL, NULL, 0, NULL, NULL, NULL }
    };

    static scli_mode_t printer_mode = {
        "printer",
        "The scli printer mode is based on the Printer-MIB as published"
        "in RFC 1759 and some updates currently being worked on in the"
        "IETF Printer MIB working group.",
        cmds
    };

    scli_register_mode(interp, &printer_mode);
}
```

Figure 12: Registration of the `show printer console lights` command.

[7] Schönwälder, J. and H. Langendörfer, ''Tcl Extensions for Network Management Applications,'' *Proc. Third Tcl/Tk Workshop*, pp. 279-288, Toronto, July, 1995.

[8] Zeltserman, D. and G. Puoplo, *Building Network Management Tools with Tcl/Tk*, Prentice Hall, 1998.

[9] Oetiker, T., ''MRTG – Multi Router Traffic Grapher,'' *Proc. Twelfth Conference on Large Installation System Administration (LISA XII)*, Boston, December, 1998.

[10] Allen, J. R., ''Driving by the Rear-View Mirror: Managing a Network with Cricket,'' *Usenix First Conference on Network Administration*, April, 1999.

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

[12] McCloghrie, K. and F. Kastenholz, ''The Interfaces Group MIB,'' *RFC 2863*, Cisco Systems, Argon Networks, June, 2000.

[13] Schönwälder, J. and A. Müller, ''Reverse Engineering Internet MIBs,'' *Proc. Seventh IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, May, 2001.

[14] Bell, E., A. Smith, P. Langille, A. Rijhsinghani, and K. McCloghrie, ''Definitions of Managed Objects for Bridges with Traffic Classes, Multicast Filtering and Virtual LAN Extensions,'' *RFC 2674*, 3Com, Extreme Networks, Newbridge Networks, Cabletron Systems, Cisco Systems, August, 1999.

[15] Smith, R., F. Wright, T. Hastings, S. Zilles, and J. Gyllenskog, ''Printer MIB,'' *RFC 1759,* Texas Instruments, Lexmark International, Xerox Corporation, Adobe Systems, Hewlett-Packard, March, 1995.

[16] Schönwälder, J. and F. Strauss, ''Next Generation Structure of Management Information for the Internet,'' *Proc. Tenth IFIP/IEEE Workshop on Distributed Systems: Operations and Management*, pp. 93-106, Springer Verlag, October, 1999.

[17] Case, J., K. McCloghrie, M. Rose, and S. Waldbusser, ''Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2),'' *RFC 1907*, SNMP Research, Cisco Systems, Dover Beach Consulting, International Network Services, January, 1996.