The Net-SNMP Programming Guide

Ben Rockwood

Updated: Nov 17th, 2004

Contents

Chapter 1

Introduction to SNMP

1.1 General Overview

Simple Network Management Protocol is a *simple* method of interacting with networked devices. The standard was defined by IETF RFC 1157 in May of 1990. SNMP can often seem quite confusing and overly complicated, its available APIs tend to put alot of wrapping around what should be very simple. The available books on the topic tend to only complicate the subject, not demystify it.

SNMP is extremely easy for any programmer to understand. A gross over simplification can explain the system simply. A network device runs an SNMP *agent* as a daemon process which answers requests from the network. The agent provides a large number of *Object Identifiers* (OIDs). An OID is a unique key-value pair. The agent populates these values and makes them available. An SNMP *manager* (client) can then query the agents key-value pairs for specific information. From a programming standpoint it's not much different than importing a ton of global variables. SNMP OIDs can be read or written. While writing information to an SNMP device is fairly rare, it is a method used by several management applications to control devices (such as an administrative GUI for your switches). A basic authentication scheme exists in SNMP, allowing the manager to send a *community name* (think cleartext password) to authorize reading or writing of OIDs. Most devices use the insecure community name "public". SNMP communication is preformed via UDP on ports 161 and 162.

Notice that I didn't mention MIBs yet! The importance of MIBs are *greatly* overrated. MIBs look complicated at first, but they are extremely simple. OIDs are numerical and global. An OID looks similar to an IPv6 address and different vendors have different prefixes and so forth. The OIDs are long enough that it's complicated for a human to remember or make sense of them, so a method was devised for translating a numeric OID into a human readable form. This translation mapping is kept in a portable flat text file called a *Management Information Base* or *MIB*. You do *not* need a MIB to use SNMP or query

SNMP devices, however without a MIB you'll have to simply guess what the data your looking at means. In some cases this is easy, such as seeing host names, disk usage numbers, or port status information. Other times it can be more difficult and a MIB is more useful. It is not unusual for some applications to be written using strictly numeric IODs allowing the end user to avoid the hassles of properly installing a MIB. The action of "installing" a MIB is really just putting it in a place where your SNMP client application can find it to perform the translation.

SNMP can be used in 2 ways: polling and traps. *Polling* just means that you write an application that sets an SNMP GET request to an agent looking some value. This method is useful because if the device responds you get the information you want and if the device does not respond you know there is a problem. Polling is an active form of monitoring. On the other hand, SNMP *traps* can be used for passive monitoring by configuring an agent to contact another SNMP agent when some action occurs.

Looking at traps deeper, a network device such as a router can be configured to send SNMP traps for certain events. For instance, you can configure Cisco IOS to send traps either when an individual event occurs such as a linkDown (IOS: *snmp-server enable traps snmp linkdown*) or when any defined trap event happens (IOS: *snmp-server enable traps snmp*). When a trap event occurs, the agent on the device will send the trap to a pre-configured destination commonly called a *trap host*. The trap host will have it's own agent running which will accept and process the traps as they come in. The processing of these traps are done by *trap handlers*. Trap Handlers can be written in any language and are provided with information from the sent trap via STDIN. The handler can then do whatever is appropriate to respond to the trap, such as sending email or doing anything else you could want.

SNMP is most commonly used in conjunction with a *Network Management System* (NMS). Popular NMS's include BMC Patrol, CA Unicenter, Sun Management Console (formerly SyMon), IBM Tivoli NetView, and the world famous HP OpenView. Even an Open Source NMS is now available, the aptly named OpenNMS. The goal of a NMS is to provide a single point of monitoring and administration of all your SNMP enabled devices. By configuring your device agents to allow write access you can even manipulate your environment from a single application. When an environment is architected around a NMS solution you can be given unparalleled levels of control and visibility over your entire environment at a glance. While Net-SNMP provides all the tools you would need to build your own NMS we won't discuss the subject any further here. However, bear in mind that if you think the vendor of your SNMP enabled device isn't as forthcoming about details of their agent implementation as you'd like, it's most likely because they would like you to simply buy their NMS or a plug-in to use their device with one of the other popular NMS's.

1.2 Three Flavors of SNMP

Three different version of SNMP exist: SNMPv1 (RFC's 1155, 1157, and 1212), SNMPv2c (RFC's 1901 through 1908), and SNMPv3 (RFC's 3411 though 3418). The co-existence of all three versions are detailed in RFC 3584.

SNMPv1 is the original standard for community based management. SNMPv2 was derived from the SNMPv1 framework but had no message definition, which was later revamped aa SNMPv2c, a community based version of SNMPv2 with a message format similar to SNMPv1. SNMPv2 added several new datatypes (Counter32, Counter64, Gauge32, UInteger32, NsapAdress, and BIT STRING), as well as enhancements to OID tables and the setting of OID values. SNMPv3 is an extensable SNMPv2 framework with a new message format, ACL and security abilities, and remote configuration of SNMP parameters.

SNMP is based on several other standards including the *Abstract Syntax Notation 1 Basic Encoding Rules* (ASN.1 BER) which defines the SNMP used Datatypes and the *Structure of Management Information* (SMI) which details the grammar used by SNMP MIBs. SMI comes in two varieties: SMIv1 (RFC 1155) and SMIv2 (RFC 2578). SMIv1 is now obsolete and should not be used. If you choose to modify MIBs at some point you'll need to learn SMIv2 and ASN.1 syntax, but otherwise they are interesting but unnecessary to learn.

To this day, SNMPv1 and SNMPv2c are the most commonly used, however due to the insecurity inherent to these protocols read-only access is typical. In general, don't bother with SNMPv3 unless you really need the added security features.

1.3 What we won't discuss

There are several subject we will not be discussing in this paper. These topics include writing agents or sub-agents, writing MIB modules, trap generation and trap sending, synchronous vs asynchronous SNMP coding, and MIB parsing.

Something that scares new or inexperienced coders away from the Net-SNMP documentation is the seemingly constant reference to synchronous and asynchronous applications. Don't be afraid, thats referring to applications that can't afford to sit and wait for a response. If your application needs a non-blocking method of handling SNMP traffic, use the asynchronous interface (eg: GUIs, Threads, Forking, etc). Otherwise, just stick with the synchronous interfaces for typical use.

Lastly, this document addresses the use of Net-SNMP on UNIX systems only. Please refer to the Net-SNMP website for information regarding development on Win32.

Chapter 2

MIBs & OIDs

OIDs, or *Object Identifiers*, uniquely identify key values offered by an SNMP agent. MIBs, or *Management Information Bases*, provide a map between numeric OIDs and a textual human readable form.

2.1 OIDs

SNMP OIDs are laid out in a hierarchy forming unique addresses into a tree similar to the DNS hierarchy. Like many other forms of addressing, OIDs can be used in 2 forms: fully qualified and relative (sometimes called "relevant").

The fully qualified form starts from the root and moves outward to the individual value on a device. An example of a fully qualified address is:

.1.3.6.1.4.1.789.1.6.4.8.0

This OID could be rewritten in human readable form as:

.iso.org.dod.internet.private.enterprises.netapp.netapp1.raid.diskSummary.diskSpareCount.0

All fully qualified OIDs will begin with *.iso.org.dod.internet.private* represented numerically as *.1.3.6.1.4*. Almost all OIDs will then be followed by *enterprises* (.1) and a unique number for the vendor as assigned by the *Internet Assigned Numbers Authority* (IANA). In the example OID 789 represents the vendor ID for the Network Appliance Corporation (NetApp). Everything beyond the vendor ID is based on the vendors implementation and may vary between implementations. Please note the prefixing dot before *iso*. Similar to the trailing dot in DNS, properly qualified IODs begin with a dot representing the root.

The complete list of enterprise assignments can be found at the IANA website: *http://www.iana.org/assignments/enterprise-numbers*

The relative form of an OID, on the other hand, begins from the enterprises value and leaves all the implied addressing off. So we can use the relative form of

the above OID as *enterprises.netapp.netapp1.raid.diskSummary.diskSpareCount.0* or numerically as *.1.789.1.6.4.8.0* .

A common form of writing OIDs is by the name of the MIB and a unique key defined within the MIB. For instance, we could rewrite the above OID into the condensed form *NETWORK-APPLIANCE-MIB::diskSpareCount.0* . This condensed form follows the convention of MIB_Name::Unique_Key.instance. Some keys, while unique, can be represented by multiple instances of that key, and thus all OIDs end with an instance value. This is why you'll notice that most OIDs end with a *.0* .

2.2 MIBs

The structure of a MIBs internals are a little strange and foreign at first, but it's structured well enough that you can poke through it pretty intelligently without really knowing what your doing. The structure of a MIB comes from the *Structure of Management Information* (SMI) standard detailed in IETF RFC 1155 and 2578. If you choose to modify or write your own MIBs you'll benefit from understanding SMI before hacking much on MIBs.

Lets look at the header of a MIB to get a better idea of how they work:

```
-- PowerNet-MIB { iso org(3) dod(6) internet(1) private(4)
--    enterprises(1) apc(318) }

PowerNet-MIB DEFINITIONS ::= BEGIN

IMPORTS
   enterprises, IpAddress, Gauge, TimeTicks     FROM RFC1155-SMI
   DisplayString                                FROM RFC1213-MIB
   OBJECT-TYPE                                  FROM RFC-1212
   TRAP-TYPE                                    FROM RFC-1215;


apc                             OBJECT IDENTIFIER ::=  { enterprises 318 }

products                        OBJECT IDENTIFIER ::=  { apc 1 }
apcmgmt                         OBJECT IDENTIFIER ::=  { apc 2 }
```

Comments can be inserted into a MIB by prepending them with two dashes. In the header the declaration BEGIN starts off the MIB. Imports can be used to pull information from other MIBs, typically those mandated by the MIB-II standard.

The MIB lays out the structure of OID addresses starting from the enterprises value. Here the enterprise value 318 maps to "apc" (relative address .1.318). Typically then several categories are defined. Here we see 2 categories: products (.1.318.1) and apcmgmt (.1.318.2). Notice that in the curly braces two values are specified, its parent address followed by its address. So the *products*

identifier is parented by the *apc* identifier which is parented by the *enterprises* identifier, so on and so forth. This type of categorization and subcategorizing will typically continue on in the header of the MIB for awhile segmenting the available keys into tight subgroupings. By segmenting values out in this way it makes the available keys easier to navigate.

The real meat of the MIB is in the description of *object types*. Here's an example of a integer key:

```
upsBasicOutputStatus OBJECT-TYPE
   SYNTAX INTEGER  {
      unknown(1),
      onLine(2),
      onBattery(3),
      onSmartBoost(4),
      timedSleeping(5),
      softwareBypass(6),
      off(7),
      rebooting(8),
      switchedBypass(9),
      hardwareFailureBypass(10),
      sleepingUntilPowerReturn(11),
      onSmartTrim(12)
   }
   ACCESS read-only
   STATUS mandatory
   DESCRIPTION
      "The current state of the UPS.  If the UPS is unable
       to determine the state of the UPS this variable is set
       to unknown(1)."
   ::= { upsBasicOutput 1 }
```

Here is defined the upsBasicOutputStatus key with a return type of INTEGER. The returned integer maps to one of 12 different return values as listed. Notice that in the MIB a description of the key is provided. These descriptions can be extremely useful in determining which objects can best provide the data you want, especially if you don't have MIB documentation supplied by the vendor.

Notice also that the last line of the object type description includes the numeric value 1 with upsBasicOutput as the parent. If we follow this parenting backwards in the MIB we'd find that upsBasicOutput has the value 1 and is parented by upsOutput which has the value 4 and is parented by ups which has a value of 1, which is parented by hardware which has a value of 1, which is parented by products with a value of 1 which is parented by apc with a value of 318, which is parented by enterprises with a value of 1. So, if we put all that mapping together we get a relative address for the key upsBasicOutputStatus of *.1.318.1.1.1.4.1.1.0*. Remember that the trailing .0 represents the first instance

of the key. Applications called *MIB Browsers* can easily parse a MIB and make navigation much quicker than flipping through the file in vim, but don't be fooled into thinking it's difficult without such a tool.

So, whats really important to notice here is that the MIB is really just providing us with a road map of the OIDs available on the agent we wish to get values from. A MIB describes both where to find a value and what it returns. We can still interface with a device without the MIB, it's just much easier when you get a return of "Up" instead of "1". By leveraging the options of the Net-SNMP CLI tools you can decide just how you wish to return output which will be different if your just using the tool from the command line (where "Up" is preferable) or if your calling the tool from a script (where "1" is preferable).

2.3 OID DataTypes

SMI defines a fixed number of datatypes which are returned by OIDs. These datatypes include:

Integer Signed 32bit Integer (values between -2147483648 and 2147483647).

Integer32 Same as Integer.

UInteger32 Unsigned 32bit Integer (values between 0 and 4294967295).

Octet String Arbitrary binary or textual data, typically limited to 255 characters in length.

Object Identifier An OID.

Bit String Represents an enumeration of named bits. This is an unsigned datatype.

IpAddress An IP address.

Counter32 Represents a non-negative integer which monotonically increases until it reaches a maximum value of 32bits-1 (4294967295 dec), when it wraps around and starts increasing again from zero.

Counter64 Same as Counter32 but has a maximum value of 64bits-1.

Gauge32 Represents an unsigned integer, which may increase or decrease, but shall never exceed a maximum value.

TimeTicks Represents an unsigned integer which represents the time, modulo 2$\hat{}$32 (4294967296 dec), in hundredths of a second between two epochs.

Opaque Provided solely for backward-compatibility, its no longer used.

NsapAddress Represents an OSI address as a variable-length OCTET STRING.

Net-SNMP tools will report the datatype when returning an OID unless you otherwise disregard it. As an example of that you'll see:

```
SNMPv2-MIB::sysContact.0 = STRING: Ben Rockwood
IF-MIB::ifPhysAddress.1 = STRING: 0:c0:b7:63:ca:4c
SNMPv2-MIB::sysUpTime.0 = Timeticks: (47372422) 5 days, 11:35:24.22
IF-MIB::ifAdminStatus.1 = INTEGER: up(1)
SNMPv2-MIB::sysObjectID.0 = OID: SNMPv2-SMI::enterprises.318.1.3.7
RFC1213-MIB::atPhysAddress.1.1.10.10.1.1 = Hex-STRING: 00 50 73 28 47 A0
RFC1213-MIB::atNetAddress.1.1.10.10.1.1 = Network Address: 0A:0A:01:01
IF-MIB::ifSpeed.1 = Gauge32: 10000000
SNMPv2-MIB::snmpInPkts.0 = Counter32: 316
SNMPv2-MIB::snmpOutPkts.0 = Counter32: 314
```

This is a fairly typical spread of datatypes returned by Net-SNMP tools. Notice that some values are being automatically interpreted by Net-SNMP, such as the sysUpTime and ifAdminStatus. The MIB was used when these values were returned and Net-SNMP was nice enough to find the return value in the MIB and give us the textual representation of the value.

2.4 MIB-II

IETF RFC 1213 "defines the second version of the Management Information Base (MIB-II) for use with network management protocols in TCP/IP-based internets." All SNMP agent and tool distributions should include MIBs that will comply with MIB-II and all devices should at the very least return values that comply with the MIB-II standard.

Within the MIB-II standard several OID groups are defined, including:

The System Group Basic system identification and information OIDs such as sysDescr, sysContact, sysName, SysLocation, etc. (Reported by Net-SNMP in SNMPv2-MIB)

The Interfaces Group Network Interface information such as ifDescr, ifType, ifSpeed, ifAdminStatus, etc. (Reported by Net-SNMP in IF-MIB)

The Address Translation Group Address Translation (AT) information mapping Physical to Logical addressing such as atNetAddress, atPhysAddress, etc. (Reported by Net-SNMP in RFC1213-MIB)

The IP Group IP stats and settings such as ipInReceives, ipForwarding, ipInAddrErrors, etc. (Reported by Net-SNMP in IP-MIB)

The ICMP Group ICMP stats and settings such as icmpInMsgs, icmpInErrors, icmpInRedirects, etc. (Reported by Net-SNMP in IP-MIB)

The TCP Group TCP stats and settings such as tcpActiveOpens, tcpPassiveOpens, tcpInErrs, etc. (Reported by Net-SNMP in TCP-MIB)

The UDP Group UDP stats and settings such as udpInDatagrams, udpInErrors, etc. (Reported by Net-SNMP in UDP-MIB)

The EGP Group EGP stats and settings (if the device support EGP) such as egpNeighAs, egpNeighMode, etc (Reported by Net-SNMP in RFC1213-MIB)

The Transmission Group Device specific media transmition stats and settings (Reported by Net-SNMP in RFC1213-MIB or your vendor MIB)

The SNMP Group SNMP stats and settings such as snmpInPkts, snmpInASNParseErrs, snmpInTraps, etc. (Reported by Net-SNMP in SNMPv2-MIB)

If you do a default walk of an SNMP device the MIB-II data should be returned. All data with the exception of the EGP and Transmission groups are requirements of the standard. For most networking devices such as routers this information is usually sufficient to provide most of the information you could want.

2.5 Adding MIBs to Net-SNMP

Additional MIBs can be added to your Net-SNMP installation by simply copying them into the *$(PREFIX)/share/snmp/mibs* directory. MIBs should be (re)named to follow the convention (MIB_NAME).txt for clarity. You can find the MIBs proper name on the first uncommented line of the MIB (eg: *NameMIB DEFINITIONS ::= BEGIN*).

For example, if you downloaded MIB for the JetDirect Management Card found on HP LaserJet printers it might have been named something like "hpjetdirect.mib". The header of the MIB looks like the following:

```
-- (C)  COPYRIGHT HEWLETT-PACKARD COMPANY 1997.
-- LaserJet 5Si Printer Model Specific MIB.
--
LaserJet5Si-MIB DEFINITIONS ::= BEGIN
```

This MIB should be renamed to "LaserJet5Si-MIB.txt" and copied into the Net-SNMP *mibs/* directory.

By following this convention it assures greater clarity when utilizing the various MIBs and a consistency with all other installed MIBs.

MIBs can be specified by a command line tool using the -m argument or the *MIBS* environmental variable for libsnmp applications including the PERL module. MIBs can be referenced locally by supplying a proper path (ie: -m "./MY_MIB.txt") or globally by supplying the MIB name without the .txt suffix (ie: -m "MY_MIB") if it's located in the Net-SNMP MIBs directory.

Chapter 3

The Net-SNMP CLI

The Net-SNMP project provides a robust SNMP implementation with an easy to use CLI. The CLI is the best place to start when writing an SNMP application, especially if your writing an application thats specific to a device. Using the CLI we can preform *walks* and *gets* to probe the device and hone in on the data we want.

3.1 Probing a device: SNMP WALKs

When you decide that you want to write an SNMP monitoring application for a given device you need to start by understanding whats available to you. If you have a MIB you can read through it to get a feel. If you have MIB documentation you have an even better leg up. Whether you have a MIB or not, the best way to start out is to *walk* the device asking the device agent for every value it can supply you. Given this output you can then know exactly what is being reported where and how you want to tackle it in code.

The *snmpwalk* tool can walk the OID tree based on a starting OID or by default with no OID which just returns the MIB-II OIDs.

Lets walk a device (APC UPS) and look at whats output using the *snmpwalk* command.

```
$ snmpwalk -v1 -c public 10.10.1.224
SNMPv2-MIB::sysDescr.0 = STRING: APC Web/SNMP Management Card
SNMPv2-MIB::sysObjectID.0 = OID: SNMPv2-SMI::enterprises.318.1.3.7
SNMPv2-MIB::sysUpTime.0 = Timeticks: (47372422) 5 days, 11:35:24.22
SNMPv2-MIB::sysContact.0 = STRING: Ben Rockwood
SNMPv2-MIB::sysName.0 = STRING: APC-3425
SNMPv2-MIB::sysLocation.0 = STRING: 3425EDISON
SNMPv2-MIB::sysServices.0 = INTEGER: 72
IF-MIB::ifNumber.0 = INTEGER: 1
IF-MIB::ifIndex.1 = INTEGER: 1
IF-MIB::ifDescr.1 = STRING: veya
```

```
........
SNMPv2-MIB::snmpOutGetResponses.0 = Counter32: 338
SNMPv2-MIB::snmpOutTraps.0 = Counter32: 0
SNMPv2-MIB::snmpEnableAuthenTraps.0 = INTEGER: 0
$
```

A large amount of the output was removed for clarity, but you get the idea. We can see the System, Interfaces, and SNMP groups from the MIB-II standard in the example output.

While the MIB-II information is interesting, unless your probing a basic router you'll want a lot more information than is supplied. To get an extended idea of whats available we need to give the tool a starting OID from which to walk. To see everything we can supply the enterprise OID, since all OIDs will start from that root. This can be done both with and without a MIB. Without a MIB you'll be forced to use a "best guess" approach to interpreting values. However if you do have a MIB the output is generally self explanatory and can serve as a reference when writing your application.

Lets first walk without a MIB (there is *a lot* of output, I recommend redirecting output to a file):

```
$ snmpwalk -v1 -c public 10.10.1.224 .1.3.6.1.4.1.318
SNMPv2-SMI::enterprises.318.1.1.1.1.1.1.0 = STRING: "Silcon DP340E"
SNMPv2-SMI::enterprises.318.1.1.1.1.1.2.0 = STRING: "UPS_IDEN"
SNMPv2-SMI::enterprises.318.1.1.1.1.2.1.0 = STRING: "314.10.D"
.....
```

In this example we're asking the *snmpwalk* tool to walk all OIDs starting from our base OID *.1.3.6.1.4.1.318* (the fully qualified OID up to the vendor identification, which is 318 for APC). For this device I get a total of 308 OIDs returned. However, because all we have to judge the OIDs meaning by is the output this isn't entirely helpful unless the OID value is a string.

If we do have a vendor MIB we can use that to walk the device and get far more useful output:

```
$ snmpwalk -v1 -c public -m "./APC-POWERNET.txt" 10.10.1.224 apc
PowerNet-MIB::upsBasicIdentModel.0 = STRING: "Silcon DP340E"
PowerNet-MIB::upsBasicIdentName.0 = STRING: "UPS_IDEN"
PowerNet-MIB::upsAdvIdentFirmwareRevision.0 = STRING: "314.10.D"
.....
```

In this example we're performing the same operation as before but we're supplying a MIB to be used (the -m option) and instead of using the numeric representation of the APC enterprise prefix we can simply use the unique identifier "apc" instead to accomplish the same thing. In both cases I returned a total of 308 OIDs, but clearly the MIB assisted output is far more understandable.

SNMP walking is the SNMP application developers equivalent to a port scan, providing a sound base on which to continue exploring and developing

applications for your device. To avoid confusion it is recommended that all development begin by redirecting a full walk to a local file for reference.

3.2 Polling Individual OIDs: SNMP GETs

The Net-SNMP tool *snmpget* can retrieve the value of an OID and return it STDOUT in any form that you wish. It can be used for debugging applications or called directly from a scripting language to create "quick and dirty" SNMP monitoring applications.

Lets use the *snmpget* tool by passing an OID to it:

```
$ snmpget -v1 -c public -m "./APC-POWERNET.txt" 10.10.1.224 \
> PowerNet-MIB::upsAdvInputLineVoltage.0
PowerNet-MIB::upsAdvInputLineVoltage.0 = Gauge32: 118
$
```

In this example I used the APC MIB (*APC-POWERNET.txt*) for the UPS Input Line Voltage of my APC with the IP address of 10.10.1.224. I can see from the output that the current voltage is 118VAC.

I could have just as easily called for this OID directly using it's numeric OID and thus not needed the MIB. This is a useful way of writing SNMP applications that don't need the MIB. We can pass *snmpget* an output option that would display the fully qualified numeric OID which we could then use in our script, so that if we moved that script to another system we wouldn't have to worry about the MIB. Obviously, the only problem with doing this is that if the SNMP agent on the target device is ever updated and we're not using the MIB to retrieve values we might start probing the wrong OID and getting bad data. For this reason you should generally use the MIB if possible.

3.3 Net-SNMP CLI Tool Options

All the Net-SNMP tools use some similar arguments. A full list can be seen by calling the tool with no options. At a bare minimum, each command should be executed by supplying the version of SNMP that you wish to use (-v1, -v2c, or -v3), the community name of the target agent (-c *community_name*) and the IP or hostname of the target agent. All other arguments are optional.

Two of the most useful arguments include -m to specify a MIB to be used and -O to modify the output.

The -m argument can either be called with the proper name of the MIB (eg: "-m MY_MIB") when its located in the system MIB directory (typically /usr/local/share/snmp/mibs) or with the filename of the MIB including path (eg: "-m ./MY_MIB.txt").

I'll point out here something that you might have noticed already: the suffix for MIBs is typically *.txt* not *.mib*. I can't say why this is done but it is the accepted standard. Furthermore, usually the MIB name is in all caps and is

referred without the suffix. Thus when you place MY_MIB.txt in the system MIB directory you will call that MIB as simply MY_MIB.

The -O argument can reorganize the output of a given Net-SNMP command in several different ways depending on how we want it displayed. You can see a list of different formating options in the help information for the individual tool. Lets just look at some examples:

```
$  snmpget -v1 -c public -m "./APC-POWERNET.txt" \
> 10.10.1.224 PowerNet-MIB::upsAdvInputLineVoltage.0
PowerNet-MIB::upsAdvInputLineVoltage.0 = Gauge32: 119

$  snmpget -v1 -c public -m "./APC-POWERNET.txt" -On \
> 10.10.1.224 PowerNet-MIB::upsAdvInputLineVoltage.0
.1.3.6.1.4.1.318.1.1.1.3.2.1.0 = Gauge32: 119

$  snmpget -v1 -c public -m "./APC-POWERNET.txt" -Ov \
> 10.10.1.224 PowerNet-MIB::upsAdvInputLineVoltage.0
Gauge32: 119

$  snmpget -v1 -c public -m "./APC-POWERNET.txt" -Oq \
> 10.10.1.224 PowerNet-MIB::upsAdvInputLineVoltage.0
PowerNet-MIB::upsAdvInputLineVoltage.0 118

$  snmpget -v1 -c public -m "./APC-POWERNET.txt" -Ovq \
> 10.10.1.224 PowerNet-MIB::upsAdvInputLineVoltage.0
118
```

Notice that the later two forms output is much nicer and easily parsed. It is not unusual for scripted applications to be implemented solely using the last output form, like this:

```
#!/usr/local/bin/perl

$SNMP_GET_CMD = "snmpget -v1 -c public -Ovq";
$SNMP_TARGET = "10.10.1.224";

my $voltage = `${SNMP_GET_CMD} ${SNMP_TARGET} .1.3.6.1.4.1.318.1.1.1.3.2.1.0`;
chomp($voltage);

print("${SNMP_TARGET} as an Input Line Reading of ${voltage}VAC\n");
```

When I run that script I get the following tidy output:

```
$ ./lineinput.pl
10.10.1.224 as an Input Line Reading of 118VAC
```

This is the easiest way to write and prototype SNMP monitoring applications.

Chapter 4

Polling Applications

Polling applications are the most common type of SNMP monitoring applications written. They typically consist of several SNMP GETs or WALKs that deposit the returned values into string variables to be manipulated and processed however you see fit. In the case of a disk array or NetApp Filer that might mean getting the disk count OIDs and then sending mail if the "Failed" disk count is higher than 0. If you were monitoring a router you might simply retrieve the packet counts and error counters OIDs and then send those values to a graphing application such as RRD to be processed.

Writing polling applications is not difficult at all. With a basic understanding of your preferred programming language and some string parsing knowledge you can write almost any type of application you could require.

4.1 Simple Polling with PERL

In the last chapter we looked at a simple PERL script that would get an OID and then nicely format the output. We can use this method to string several gets together and build more useful monitoring applications.

Lets look at a "full featured" application that uses a simple polling method using the CLI interface to *snmpget*:

```
#!/usr/local/bin/perl

$SNMP_GET_CMD = "snmpget -v1 -c public -Ovq -m PowerNet-MIB";
$SNMP_TARGET = "10.10.1.224";

chomp($model = `${SNMP_GET_CMD} ${SNMP_TARGET} PowerNet-MIB::upsBasicIdentModel.0`);
chomp($serial = `${SNMP_GET_CMD} ${SNMP_TARGET} PowerNet-MIB::upsAdvIdentSerialNumber.0`
chomp($cap = `${SNMP_GET_CMD} ${SNMP_TARGET} PowerNet-MIB::upsAdvBatteryCapacity.0`);
chomp($temp = `${SNMP_GET_CMD} ${SNMP_TARGET} PowerNet-MIB::upsAdvBatteryTemperature.0`)
chomp($out_load = `${SNMP_GET_CMD} ${SNMP_TARGET} PowerNet-MIB::upsAdvOutputLoad.0`);
chomp($out_v = `${SNMP_GET_CMD} ${SNMP_TARGET} PowerNet-MIB::upsAdvOutputVoltage.0`);
```

```
chomp($out_f = `${SNMP_GET_CMD} ${SNMP_TARGET} PowerNet-MIB::upsAdvOutputFrequency.0`);
chomp($out_status = `${SNMP_GET_CMD} ${SNMP_TARGET} PowerNet-MIB::upsBasicOutputStatus.0

$model =~ s/\"//g;      # Ditch the quotes.
$serial =~ s/\"//g;

print <<END;
APC UPS                         ${SNMP_TARGET}
Model: ${model}         Serial No: ${serial}

Battery Capacity: ${cap}
Battery Temp(F): ${temp}

Output Status: ${out_status}
Output Load: ${out_load}
Output: ${out_v}VAC @ ${out_f}Hz
END
```

When we run that, we get a nice and pretty output with the data we want:

```
$ ./apc_status.pl
APC UPS                         10.10.1.224
Model: Silcon DP340E            Serial No: SE(removed)

Battery Capacity: 100
Battery Temp(F): 32

Output Status: onLine
Output Load: 53
Output: 118VAC @ 60Hz
```

This method, while not particularly pretty of efficient, is very quick and easy to both write and debug making this approach useful in a pinch.

4.2 The Net-SNMP PERL Module

The Net-SNMP distribution comes complete with a PERL module. You'll notice that several SNMP modules exist in CPAN but these are deprecated and should not be used.

The module is loaded in the usual way, with a "use" statement. It is recommended that you supply a module version number to avoid any possible problems.

Three main statements are used in a basic application: SNMP::Session(), SNMP::VarList, and SNMP::getnext(). The SNMP::Session method connects to the target agent and returns a session handle. Once a handle is open, a *varlist* can be populated by passing a number of OIDs (typically just the unique key

name). The VarList method builds a list and stores it using it's own internal format. We can use the GetNext method to actually fetch and then output all the returned values into a normal flat array to be used by other parts of our script.

Here is an example of our previous APC monitoring tool re-written using the Net-SNMP PERL module.

```
#!/usr/local/bin/perl

use SNMP '5.0.2.pre1' || die("Cannot load module\n");

$ENV{'MIBS'}="ALL";  #Load all available MIBs
$SNMP_TARGET = "10.10.1.224";
$SNMP_COMMUNITY = "public";

$SESSION = new SNMP::Session (DestHost => $SNMP_TARGET,
                              Community => $SNMP_COMMUNITY,
                              Version => 1);

# Populate a VarList with OID values.
$APC_VLIST =  new SNMP::VarList(['upsBasicIdentModel'],         #0
                              ['upsAdvIdentSerialNumber'],    #1
                              ['upsAdvBatteryCapacity'],      #2
                              ['upsAdvBatteryTemperature'],   #3
                              ['upsAdvOutputLoad'],           #4
                              ['upsAdvOutputVoltage'],        #5
                              ['upsAdvOutputFrequency'],      #6
                              ['upsBasicOutputStatus']);      #7

# Pass the VarList to getnext building an array of the output
@APC_INFO = $SESSION->getnext($APC_VLIST);

$APC_INFO[0] =~ s/\"//g;        # Ditch the quotes.
$APC_INFO[1] =~ s/\"//g;

# Output the results.
print <<END;
APC UPS                         ${SNMP_TARGET}
Model: ${APC_INFO[0]}       Serial No: ${APC_INFO[1]}

Battery Capacity: ${APC_INFO[2]}
Battery Temp(F): ${APC_INFO[3]}

Output Status: ${APC_INFO[7]}
Output Load: ${APC_INFO[4]}
```

```
Output: ${APC_INFO[5]}VAC @ ${APC_INFO[6]}Hz
END
```

Notice that in the header of our script we're defining the MIBS environmental variable to "ALL", which loads all of the MIBs in the Net-SNMP system MIBs directory.

Several varlists can be created from a single session to provide clear organization. When I create varlists I find it is helpful to put commented reference numbers next to each element of the list so that when I later use the populated array I can quickly check which OID is in which element.

The output looks the same as the previous versions did:

```
$ ./apc_status_mod.pl
APC UPS                          10.10.1.224
Model: Silcon DP340E        Serial No: SE0010000515

Battery Capacity: 100
Battery Temp(F): 33

Output Status: onLine
Output Load: 53

Output: 118VAC @ 60Hz
```

Chapter 5

Trap Handlers

Most SNMP enabled devices have the ability to send traps. The SNMP agent on the device will send a message to a predefined *trap host* when some event occurs triggering the trap. A trap host should have a *snmptrapd* trap daemon running thats waiting for the traps. The daemon has a simple configuration file (*snmptrapd.conf*) that outlines how each trap should be handled. Based on this configuration file the trap information is sent to one or more *trap handlers* which are custom applications or scripts that process the trap information and do something with it such as logging them or paging someone.

5.1 The Trap Daemon Configuration

In the *snmptrapd.conf* trap configuration file you define 2 pieces of information that tells *snmptrapd* how to direct the incoming traps: the OID of the incoming trap and the location of the traphandler to pass the incoming trap information to including any arguments you want passed to the trap handler.

The following is an example of a *snmptrapd.conf*:

```
### Format: traphandle (OID) (Trap Handler) (Trap Handler Args...)
#extremeModuleStateChanged
traphandle  .1.3.6.1.4.1.1916.0.15 /path/to/traphandle.pl statechange

#extremeHealthCheckFailed
traphandle  .1.3.6.1.4.1.1916.4.1.0.1 /path/to/traphandle.pl healthcheck

#extremeCpuHealthCheckFailed
traphandle  .1.3.6.1.4.1.1916.0.22 /path/to/traphandle.pl cpuhealthcheck

# Default Handler
traphandle  default /path/to/traphandle.pl
```

In this configuration file the same trap handler is used in all 4 lines uncommented line, but based on the trap OID different arguments are passed to allow the trap handler to react in different ways. If you supply the OID in numeric format it is wide to add a comment describing the traps MIB name. The final line uses the keyword "default" instead of an OID, which acts as a catch all to handle any traps that weren't processed by an earlier trap handler.

5.2 A Simple Trap Handler

Trap handlers can be written in any language you prefer, but for this document we'll use PERL.

At least 3 lines of input will be handed to your trap handler via STDIN, in addition to any arguments you passed to it in your *snmptrapd.conf*. The first line of input is the hostname (if it can not be resolved it will use the IP address). The second line of input is the IP address. All remaining lines of input will be the data passed by the trap.

Traps will pass in a list of OIDs with their values (elsewhere called a VARBIND or VarList), one per line. It is best to handle this list by simply pushing the values onto an array for later processing.

```
#!/usr/bin/perl
# A simple trap handler
my $TRAP_FILE = "/var/snmp/traps.all.log";

my $host = <STDIN>; # Read the Hostname - First line of input from STDIN
 chomp($host);
my $ip = <STDIN>; # Read the IP - Second line of input
 chomp($ip);

while(<STDIN>) {
        chomp($_);
        push(@vars,$_);
}

open(TRAPFILE, ">> $TRAP_FILE");
$date = `date`;
chomp($date);
print(TRAPFILE "New trap received: $date for $OID\n\nHOST: $host\nIP: $ip\n");
foreach(@vars) {
        print(TRAPFILE "TRAP: $_\n");
}
print(TRAPFILE "\n----------\n");
close(TRAPFILE);
```

Here is an example of a trap that was logged using the trap handler above:

```
New trap received: Wed Oct 27 16:32:18 PDT 2004 for

HOST: 10.100.2.248
IP: 10.100.2.248
TRAP: RFC1213-MIB::sysUpTime.0 48:17:24:09.31
TRAP: SNMPv2-MIB::snmpTrapOID.0 IF-MIB::linkUp
TRAP: RFC1213-MIB::ifIndex 5017
TRAP: RFC1213-MIB::ifAdminStatus up
TRAP: RFC1213-MIB::ifOperStatus up

----------
```

Basic trap handlers like this one are a useful starting point for building more complex trap handlers. By parsing the incoming traps and also using arguments passed to the handler based on the trap OID in the configuration file you can build complex trap handling routines to report problems, investigate them, or even respond to them if you have a strong stomach. Because of the large amount of parsing done when processing traps it is recommended that languages with strong string manipulation abilities such as PERL, Python or Ruby are used.

You can find an example of a more complicated trap handler here:
http://www.cuddletech.com/tools/extreme_traphandle.html

5.3 Starting the Trap Daemon

The trap daemon needs to be started explicitly in order to listen for traps. It can be started in whichever method your OS prefers (/etc/rc.local or an initscript). You can pass a variety of arguments to the daemon, such as a list of MIBs to be used (-m) or tell it to run in the foreground for debugging purposes (-f).

Here is an example init script:

```
#!/sbin/sh
#

case "$1" in
start)
        [ -f /usr/local/share/snmp/mibs/extreme620.mib ] || exit 0

        /usr/local/sbin/snmptrapd -m /usr/local/share/snmp/mibs/extreme620.mib -n
        ;;
stop)
        pkill snmptrapd
        ;;
*)
        echo "Usage: $0 { start | stop }"
        exit 1
        ;;
```

```
esac
exit 0
```

In this case the script ensures that the required MIB is present before starting the trap daemon. -m specifies the MIB (or list of MIBs) for the trap daemon to use and -n specifies that source addresses should not be translated into hostnames.

Chapter 6

The Net-SNMP C API

Developing SNMP applications with the C API is significantly more difficult and time consuming than using the PERL and CLI interfaces. However, the advantages of being able to integrate with other native C APIs to build complex applications are obvious.

In this chapter we will discuss several aspects of SNMP that have been hid from you in the other interfaces, namely PDUs. To properly harness the SNMP library you'll need a firm understanding of these concepts.

6.1 SNMP Internals

Within the SNMP C library API there are a large variety of different structures which are used to store information needed by different phases of a SNMP dialog. The basics steps are:

1. Initialize an SNMP session
2. Define attributes for the session
3. Add MIBs to the current MIB tree [Optional]
4. Create a PDU (Primary Data Unit)
5. Pack OIDs into the PDU
6. Send the request and wait for response
7. Do something with the returned values
8. Free the PDU
9. Close the session

When you initialize a new SNMP session a *snmp_session* structure will be initialized with default values. You can then modify elements of that structure to match your needs such as defining the SNMP version, community name, and target hostname. Once the session values are in order you can open the session which returns another *snmp_session* struct as a handle.

When a session is created the default system MIBs are loaded into the MIB tree used by the library, however you can also add (or remove) MIBs from the tree to use a MIB that isn't installed.

Once the session is in order and open, a *Primary Data Unit* (PDU) can be created. Richard Stevens' wrote in his book "TCP/IP Illustrated: Vol 1" that a PDU is just a fancy term for a packet. Each SNMP request packet includes a PDU. Each PDU can contain one or more OIDs. The type of request that is made is specified by the type of PDU. Therefore if you needed to read and write OIDs, you'd need 2 PDUS, however if you simply needed to read 2 or more OIDs you could just pack them into a single PDU. Adding an OID to a PDU is a two step process, first by reading the OID from the MIB and then by adding the OID to the PDU.

Once a PDU is populated and prepared it can be sent using the session handle returned earlier. The response is put into a new PDU structure with both the OIDs and the values.

6.2 Watching SNMP on the wire

To illustrate more accurately how SNMP actually works on the wire, lets look at sniffed packets captured during an SNMP GET. Only 2 packets are exchanged, a request by the manager and a response by the agent. The following was gathered using Ethereal:

```
No.     Time        Source      Destination Protocol Info
    112 1.936155    10.10.1.110 10.10.1.224 SNMP     GET SNMP...
Simple Network Management Protocol
    Version: 1 (0)
    Community: public
    PDU type: GET (0)
    Request Id: 0x2a7ee1af
    Error Status: NO ERROR (0)
    Error Index: 0
    Object identifier 1: 1.3.6.1.2.1.1.4.0 (SNMPv2-MIB::sysContact.0)
    Value: NULL

No.     Time        Source      Destination Protocol Info
    113 1.943425    10.10.1.224 10.10.1.110 SNMP     RESPONSE SN...
Simple Network Management Protocol
    Version: 1 (0)
    Community: public
```

```
PDU type: RESPONSE (2)
Request Id: 0x2a7ee1af
Error Status: NO ERROR (0)
Error Index: 0
Object identifier 1: 1.3.6.1.2.1.1.4.0 (SNMPv2-MIB::sysContact.0)
Value: STRING: Ben Rockwood
```

Each packet is referred to as a PDU in SNMP speak. The first packet (with the Ethernet, IP, and UDP headers removed) is a version 1 GET request with an OID to get and a community name of "public". Notice that the GET PDU has a *Value* field, but its NULL. The second packet is the response which you'll notice is identical to the GET PDU with 2 exceptions: the PDU type is RESPONSE instead of GET and the *Value* field is populated.

Given this example you can see that when you make a request you hand a PDU to an agent and say "Please fill this in".

Putting this into the context of the Net-SNMP C API, look at the *snmp_PDU* structure.

```
typedef struct snmp_pdu {

    long            version;
    int             command;
    long            reqid;
    ...
    long            errstat;
    long            errindex;
    ...
    netsnmp_variable_list *variables;
    ...
    u_char         *community;
    ...
```

The *version* value maps directly the value seen in the packet above. The *command* value is the PDU type as seen in the packet above. And you can also see the *reqid* in the packets above as the *Request ID*. The important point here is that when a PDU structure is manipulated using the API your actually preparing an SNMP packet for transmission.

Forming a request, such as a GET, can be done in several ways. One method is to create a single PDU per OID you wish to request. As we saw above this would mean sending one packet per OID and thus receiving one packet for each request. Another method, which is completely valid, is to *pack* a single GET PDU with multiple OIDs for retrieval. The following is a break capture of such an exchange as seen on the wire:

```
No.Time        Source      Destination Protocol Info
28 2.240789    10.10.1.110 10.10.1.224 SNMP     GET SNMPv...
Simple Network Management Protocol
```

```
    Version: 1 (0)
    Community: public
    PDU type: GET (0)
    Request Id: 0x30549f5c
    Error Status: NO ERROR (0)
    Error Index: 0
    Object identifier 1: 1.3.6.1.4.1.318.1.1.1.1.1.1.0 (SNMP...
    Value: NULL
    Object identifier 2: 1.3.6.1.4.1.318.1.1.1.1.2.3.0 (SNMP...
    Value: NULL

No.Time        Source      Destination Protocol Info
41 2.328751    10.10.1.224 10.10.1.110 SNMP     RESPONSE SNM...
Simple Network Management Protocol
    Version: 1 (0)
    Community: public
    PDU type: RESPONSE (2)
    Request Id: 0x30549f5c
    Error Status: NO ERROR (0)
    Error Index: 0
    Object identifier 1: 1.3.6.1.4.1.318.1.1.1.1.1.1.0 (SNMP...
    Value: STRING: "Silcon DP340E"
    Object identifier 2: 1.3.6.1.4.1.318.1.1.1.1.2.3.0 (SNMP...
    Value: STRING: "SE0XXXXXXX   "
```

This exchange looks identical to the previous exchange but this time we've packed multiple (8 total, but I only show 2 for brevity) OIDs into the PDU. Hence, we send one packet and we are returned one packet.

Looking back at the PDU structure, it should not be obvious why there is no fixed value for the OID but instead an included *netsnmp_variable_list* pointer. This variable list is otherwise referred to as a *VARLIST* or *VARBIND*.

```
typedef struct variable_list netsnmp_variable_list;

struct variable_list {
   struct variable_list *next_variable;
   oid            *name;
   size_t          name_length;
   u_char          type;
   netsnmp_vardata val;
   size_t          val_len;
   oid             name_loc[MAX_OID_LEN];
   u_char          buf[40];
   void           *data;
   void            (*dataFreeHook)(void *);
   int             index;
};
```

So looking at the variable list we immediately notice that its a linked list, hence the ability to pack multiple values into a single OID.

We can pull all this information together to get a much clearer idea of how the API really works. As seen in the captures what we really need to do is create, send, receive and then process one or more PDUs; thats the basics. In order to do this we need to initialize a PDU structure, append one or more variables into the variable list and then send that PDU out. What we'll get back is a fully populated (we hope) PDU, but this time each OID in the variable list will have a value referenced by the data pointer.

Next lets look at the actual code.

6.3 A simple example

The following is a very simple example taken from the Net-SNMP site. It initializes a session, adds a MIB in the current working directory to the MIB tree, creates a GET PDU, packs 2 OIDs into the PDU, then sends a synchronous request and returns the response and is finally processed.

```
#include <net-snmp/net-snmp-config.h>
#include <net-snmp/net-snmp-includes.h>
#include <string.h>

int main(int argc, char ** argv)
{
  struct snmp_session session;
  struct snmp_session *sess_handle;

  struct snmp_pdu *pdu;
  struct snmp_pdu *response;

  struct variable_list *vars;

  oid id_oid[MAX_OID_LEN];
  oid serial_oid[MAX_OID_LEN];

  size_t id_len = MAX_OID_LEN;
  size_t serial_len = MAX_OID_LEN;

  int status;

  struct tree * mib_tree;

  /**********************/

  if(argv[1] == NULL){
```

```
printf("Please supply a hostname\n");
exit(1);
  }

  init_snmp("APC Check");

  snmp_sess_init( &session );
   session.version = SNMP_VERSION_1;
   session.community = "public";
   session.community_len = strlen(session.community);
   session.peername = argv[1];
  sess_handle = snmp_open(&session);

  add_mibdir(".");
  mib_tree = read_mib("PowerNet-MIB.txt");

  pdu = snmp_pdu_create(SNMP_MSG_GET);

  read_objid("PowerNet-MIB::upsBasicIdentModel.0", id_oid, &id_len);
   snmp_add_null_var(pdu, id_oid, id_len);
  read_objid("PowerNet-MIB::upsAdvIdentSerialNumber.0", serial_oid, &serial_len);
   snmp_add_null_var(pdu, serial_oid, serial_len);

  status = snmp_synch_response(sess_handle, pdu, &response);

  for(vars = response->variables; vars; vars = vars->next_variable)
print_value(vars->name, vars->name_length, vars);

  snmp_free_pdu(response);
  snmp_close(sess_handle);

  return (0);
}
```

This example can be compiled like this and run like this:

```
$ gcc ‘net-snmp-config --cflags‘ ‘net-snmp-config --libs‘ \
> ‘net-snmp-config --external-libs‘ snmp_test.c -o snmp_test
$ ./snmp_test apc
STRING: "Silcon DP340E"
STRING: "SE00XXXXXX   "
$
```

You can see in the code that *init_snmp()* gets the ball rolling. *snmp_sess_init()* accepts a session structure address, this function will populate the bulk of the session structure with default values. Once the session structure is primed we

can modify it to meet our needs, such as defining the SNMP version, community name (not forgetting the length of the community name), and the hostname of agent referred to as the *peername*. When the session structure is ready for use, we can open the session with *snmp_open()* which will return a new session structure as a handle.

While you normally should refrain from using non-installed MIBs, in this example I've demonstrated how to add a directory to the internal MIB path and then read a MIB into the MIB tree using the *add_mibdir()* and *read_mib()* functions.

Now that the session is open and the MIBs we'll use are loaded into the tree we can create and populate the PDU(s) we'll be sending. *snmp_pdu_create()* will setup the base PDU information based on the supplied type of PDU (specified in macro form) and return the populated PDU structure. Now that we have the PDU we can use the *read_objid()* and *snmp_add_null_var()* functions to first read the OID from the MIB and then add that OID as a variable to the variable list used by the PDU. We can repeat this process several times to continue packing our PDU full of OIDs. Typically you'll see this type of operation wrapped in a loop, but I've illustrated it long hand for simplicity sake.

Once our session is open and our PDU is ready with all the OIDs we want crammed into the variable list we can actually send the request and await the response. This action is done in a single step using the *snmp_synch_response()* function. The function is passed the open session handle, the PDU to send, and an empty PDU structure to accept the response which includes the populated values for each OID in the variable list.

At this point you can extract and manipulate the returned data either by utilizing built in functions such as *print_value()* or by simply directly accessing the structures. Other convince functions for value output can be found in the *net-snmp/library/mib.h* header.

Finally, to properly clean up, the PDU(s) should be freed and the sessions closed using the *snmp_free_pdu()* and *snmp_close()* functions.

6.4 Closing Thoughts

This is just a small taste of what the C API is capable of, but it should be enough to get you moving in the right direction with a firm base to grow on. Components like SQLite, PostgreSQL, RRDtool, GNU Plot, EWL and other X toolkits, and many more can be combined easily to facilitate any type of management application you could desire to build. Net-SNMP provides both the power to get just what you want the way you want but also the flexibility to build your own tools that are just as good if not better than your vendors supplied application at little to no cost.