The colors are used both on OpenView's maps and in the Event Categories. Parent objects, which represent the starting point for a network, are displayed in the color of the highest severity level associated with any object underneath them.[4] For example, if an object represents a network with 250 nodes and one of those nodes is down (a Critical severity), the object will be colored red, regardless of how many nodes are up and functioning normally. The term for how OpenView displays colors in relation to objects is *status source* ; it is explained in more detail in Chapter 6.

[4] Parent objects can show status (colors) in four ways: Symbol, Object, Compound, or Propagated.

10.2.2.4 Log messages, notifications, and automatic actions

Returning to Figure 10-1, the Event Log Message and Popup Notification fields are similar, but serve different purposes. The Event Log Message is displayed when you view the Event Categories and select a category from the drop-down list. The Popup Notification, which is optional, displays its message in a window that appears on any server running OpenView's NNM. Figure 10-2 shows a typical pop-up message. The event name, delme in this case, appears in the title bar. The time and date at which the event occurred are followed by the event message, "Popup Message Here." To create a pop-up message like this, insert "Popup Message Here" in the Popup Notification section of the Event Configurator. Every time the event is called, a pop-up will appear.

Figure 10-2. OpenView pop-up message

The last section of the Event Configurator is the Command for Automatic Action. The automatic action allows you to specify a Unix command or script to execute when OpenView receives an event. You can run multiple commands by separating them with a semicolon, much as you would in a Unix shell. When configuring an automatic action, remember that rsh can be very useful. I like to use *rsh sunserver1 audioplay -v50 /opt/local/sounds/siren.au*, which causes a siren audio file to play. The automatic action can range from touching a file to opening a trouble ticket.

In each Event Log Message, Popup Notification, and Command for Automatic Action, special variables can help you identify the values from your traps or events. These variables provide the

user with additional information about the event. Here are some of the variables you can use; the online help has a complete list:

$1

Print the first passed attribute (i.e., the value of the first variable binding) from the trap.

$2

Print the second passed attribute.

$n

Print the nth attribute as a value string. Must be in the range of 1-99.

*$**

Print all the attributes as *[seq] name (type)*.

> Before you start running scripts for an event, find out the average number of traps you are likely to receive for that event. This is especially true for OV_Node_Down. If you write a script that opens a trouble ticket whenever a node goes down, you could end up with hundreds of tickets by the end of the day. Monitoring your network will make you painfully aware of how much your network "flaps," or goes up and down. Even if the network goes down for a second, for whatever reason, you'll get a trap, which will in turn generate an event, which might register a new ticket, send you a page, etc. The last thing you want is "The Network That Cried Down!" You and other people on your staff will start ignoring all the false warnings and may miss any serious problems that arise. One way to estimate how frequently you will receive events is to log events in a file ("Log only"). After a week or so, inspect the log file to see how many events accumulated (i.e., the number of traps received). This is by no means scientific, but it will give you an idea of what you can expect.

10.2.3 Custom Event Categories

OpenView uses the default categories for all its default events. Look through the *$OV_CONF/C/trapd.conf* file to see how the default events are assigned to categories. You can add categories by going to "Event Configuration →Edit →Configure →Event Categories." Figure 10-3 shows this menu, with some custom categories added.

Figure 10-3. Adding event categories in OpenView

It's worth your while to spend time thinking about what categories are appropriate for your environment. If you plow everything into the default categories you will be bothered by the Critical "Printer Needs Paper" event, when you really want to be notified of the Critical "Production Server on Fire" event. Either event will turn Status Events red. The categories in Figure 10-3 are a good start, but think about the types of events and activities that will be useful in your network. The Scheduled and Unscheduled (S/U) Downtime category is a great example of a category that is more for human intervention than for reporting network errors. Printer Events is a nice destination for your "Printer Needs Paper" and "Printer Jammed" messages.

Even though none of the default categories are required (except for Error), we recommend that you don't delete them, precisely because they are used for all of the default events. Deleting the default categories without first reconfiguring all the default events will cause problems. Any event that does not have an event category available will be put into the default Error category. To edit the categories, copy the *trapd.conf* file into /tmp and modify */tmp/trapd.conf* with your favorite editor. The file has some large warnings telling you never to edit it by hand, but sometimes a few simple edits are the best way to reassign events. An entry in the portion of the file that defines event behavior looks like this:

```
EVENT RMON_Rise_Alarm .1.3.6.1.2.1.16.0.1 "Threshold Events" Warning
FORMAT RMON Rising Alarm: $2 exceeded threshold $5; value = $4. (Sample type = \
$3; alarm index = $1)
SDESC
This event is sent when an RMON device exceeds a preconfigured threshold.
EDESC
```

It's fairly obvious what these lines do: they map a particular RMON event into the Threshold Events category with a severity of Warning; they also specify what should happen when the event occurs. To map this event into another category, change Threshold Events to the appropriate category. Once you've edited the file, use the following command to merge in your updates:

```
$ $OV_BIN/xnmevents -l load /tmp/trapd.conf
```

10.2.4 The Event Categories Display

The Event Categories window (Figure 10-4) is displayed on the user's screen when NNM is started. It provides a very brief summary of what's happening on your network; if it is set up appropriately, you can tell at a glance whether there are any problems you should be worrying about.

Figure 10-4. OpenView Event Categories

If the window gets closed during an OpenView session, you can restart it using the "Fault →Events" menu item or by issuing the command *$OV_BIN/xnmevents*. The menu displays all the event categories, including any categories you have created. Two categories are special: the Error category is the default category used when an event is associated with a category that cannot be found; the All category is a placeholder for all events and cannot be configured by the Event Configurator. The window shows you the highest severity level of any event in each event category.

The box to the left of Status Events is cyan (a light blue), showing that the highest unacknowledged severity in the Status Events category is Warning. Clicking on that box displays an alarm browser that lists all the events received in the category. A nice feature of the Event Categories display is the ability to restore a browser's state or reload events from the *trapd.log* and *trapd.log.old* files. Reloading events is useful if you find that you need to restore messages you deleted in the past.

> Newer versions of OpenView extend the abilities of Event Categories by keeping a common database of acknowledged and unacknowledged events. Thus, when a user acknowledges an event, all other users see this event updated.

At the bottom of Figure 10-4, the phrase "[Read-Only]" means that you don't have write access to Event Categories. If this phrase isn't present, you have write access. OpenView keeps track of events on a per-user basis, using a special database

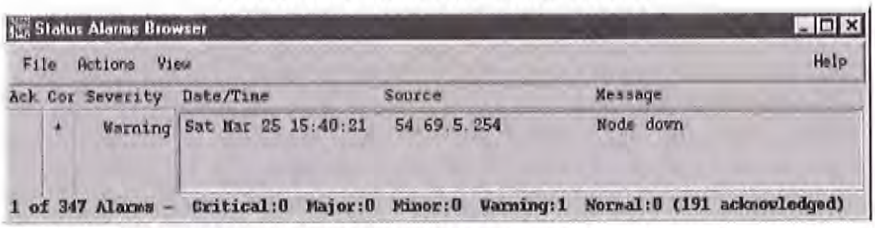
located in *$OV_LOG/xnmevents.username*.[5] With write access, you have the ability to update this file whenever you exit. By default, you have write access to your own event category database, unless someone has already started the database by starting a session with your username. There may be only one write-access Event Categories per user, with the first one getting write access and all others getting read-only privileges.

[5] Again, newer versions of OpenView have only one database that is common for all users.

10.2.5 The Alarm Browser

Figure 10-5 shows the alarm browser for the Status Events category. In it we see a single Warning event, which is causing the Status Events category to show cyan.

Figure 10-5. OpenView alarm browser



The color of the Status Events box is determined by the highest-precedence event in the category. Therefore, the color won't change until either you acknowledge the highest-precedence event or an event arrives with an even higher precedence. Clicking in the far left column (Ack) acknowledges the message[6] and sets the severity to 0.

[6] Newer versions of OpenView support Event Correlation, which has a column in this window as well.

The Actions menu in the alarm browser allows you to acknowledge, deacknowledge, or delete some or all events. You can even change the severity of an event. Keep in mind that this does not change the severity of the event on other Event Categories sessions that are running. For example, if one user changes the severity of an event from Critical to Normal, the event will remain Critical for other users. The View menu lets you define filters, which allow you to include or discard messages that match the filter.

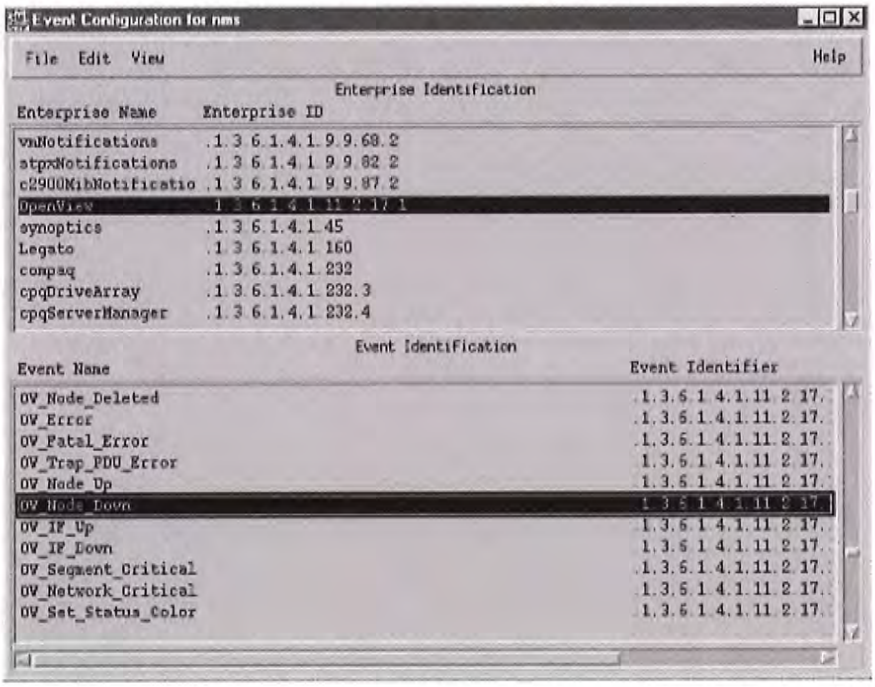
When configuring events, keep in mind that you may receive more traps than you want. When this happens, you have two choices. First, you can go to the agent and turn off trap generation, if the agent supports this. Second, you can configure your trap view to ignore these traps. We saw how to do this earlier: you can set the event to "Log only" or try excluding the device from the Event Sources list. If bandwidth is a concern, you

should investigate why the agent is sending out so many traps before trying to mask the problem.

10.2.6 Creating Events Within OpenView

OpenView gives you the option of creating additional (private) events. Private events are just like regular events, except that they belong to your private-enterprise subtree, rather than to a public MIB. To create your own events, launch the Event Configuration window from the Options menu of NNM. You will see a list of all currently loaded events (Figure 10-6).

Figure 10-6. OpenView's Event Configuration



The window is divided into two panes. The top pane displays the Enterprise Identification, which is the leftmost part of an OID. Clicking on an enterprise ID displays all the events belonging to that enterprise in the lower pane. To add your own enterprise ID, select "Edit →Add →Enterprise Identification" and insert your enterprise name and a registered enterprise ID.[7] Now you're ready to create private events. Click on the enterprise name you just created; the enterprise ID you've associated with this name will be used to form the OID for the new event. Click "Edit →Add →Event"; then type the Event Name for your new event, making sure to use Enterprise Specific (the default) for the event type. Insert an Event Object Identifier. This identifier can be any number that hasn't already been assigned to an event in the currently selected enterprise. Finally, click "OK" and save the event configuration (using "File →Save").

[7] Refer to Chapter 2 for information about obtaining your own enterprise ID.

To copy an existing event, click on the event you wish to copy and select "Edit →Copy Event"; you'll see a new window with the event you selected. From this point on, the process is the same.

Traps with "no format" are traps for which nothing has been defined in the Event Configuration window. There are two ways to solve this problem: you can either create the necessary events on your own or you can load a MIB that contains the necessary trap definitions, as discussed in Chapter 6. "No format" traps are frequently traps defined in a vendor-specific MIB that hasn't been loaded. Loading the appropriate MIB often fixes the problem by defining the vendor's traps and their associated names, IDs, comments, severity levels, etc.

Before loading a MIB, review the types of traps the MIB supports. You will find that most traps you load come, by default, in LOGONLY mode. This means that you will not be notified when the traps come in. After you load the MIB you may want to edit the events it defines, specifying the local configuration that best fits your site.

10.2.7 Monitoring Traps with Perl

If you can't afford an expensive package like OpenView, you can use the Perl language to write your own monitoring and logging utility. You get what you pay for, since you will have to write almost everything from scratch. But you'll learn a lot and probably get a better appreciation for the finer points of network management. One of the most elementary, but effective, programs to receive traps is in a distribution of SNMP Support for Perl 5, written by Simon Leinen. Here's a modified version of Simon's program:

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

use SNMP_Session "0.60";
use BER;
use Socket;
$session = SNMPv1_Session->open_trap_session (  );

while (($trap, $sender, $sender_port) = $session->receive_trap (  ))
{
    chomp ($DATE=`/bin/date \'+%a %b %e %T\'`);
    print STDERR "$DATE - " . inet_ntoa($sender) . " - port: $sender_port\n";
    print_trap ($session, $trap);
}
1;
```

```
sub print_trap ($$) {
    ($this, $trap) = @_;
    ($community, $ent, $agent, $gen, $spec, $dt, @bindings) = \
     $this->decode_trap_request ($trap);
    print "   Community:\t".$community."\n";
    print "   Enterprise:\t".BER::pretty_oid ($ent)."\n";
    print "   Agent addr:\t".inet_ntoa ($agent)."\n";
    print "   Generic ID:\t$gen\n";
    print "   Specific ID:\t$spec\n";
    print "   Uptime:\t".BER::pretty_uptime_value ($dt)."\n";
    $prefix = "   bindings:\t";
    foreach $encoded_pair (@bindings)
                {
        ($oid, $value) = decode_by_template ($encoded_pair, "%{%O%@");
        #next unless defined $oid;
        print $prefix.BER::pretty_oid ($oid)." => ".pretty_print ($value)."\n";
        $prefix = "  ";
                }
}
```

This program displays traps as they are received from different devices in the network. Here's some output, showing two traps:

```
Mon Apr 28 22:07:44 - 10.123.46.26 - port: 63968
   community:   public
   enterprise:  1.3.6.1.4.1.2789.2500
   agent addr:  10.123.46.26
   generic ID:  6
   specific ID: 5247
   uptime:      0:00:00
   bindings:    1.3.6.1.4.1.2789.2500.1234 => 14264026886

Mon Apr 28 22:09:46 - 172.16.51.25 - port: 63970
   community:   public
   enterprise:  1.3.6.1.4.1.2789.2500
   agent addr:  172.16.253.2
   generic ID:  6
   specific ID: 5247
   uptime:      0:00:00
   bindings:    1.3.6.1.4.1.2789.2500.2468 => Hot Swap Now In Sync
```

The output format is the same for both traps. The first line shows the date and time at which the trap occurred, together with the IP address of the device that sent the trap. Most of the remaining output items should be familiar to you. The **bindings** output item lists the variable bindings that were sent in the trap PDU. In the example above, each trap contained one variable binding. The object ID is in numeric form, which isn't particularly friendly. If a trap has more than one variable binding, this program displays each binding, one after another.

An ad hoc monitoring system can be fashioned by using this Perl script to collect traps and some other program to inspect the traps as they are received. Once the traps are parsed, the possibilities are endless. You can write user-defined rules

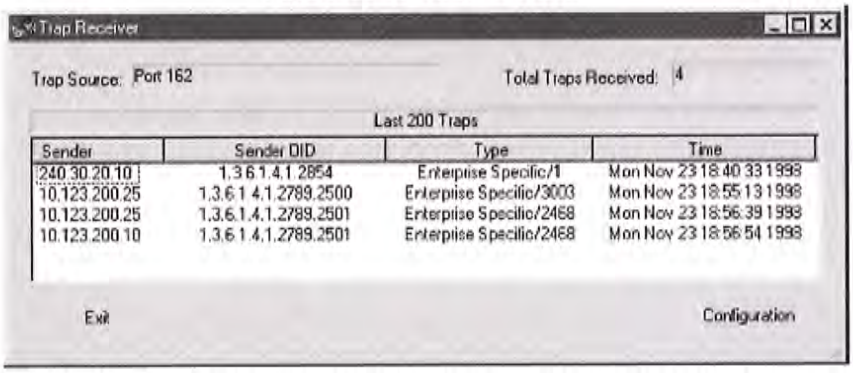
that watch for significant traps and, when triggered, send an email alert, update an event database, send a message to a pager, etc. These kinds of solutions work well if you're in a business with little or no budget for commercially available NMS software or if you're on a small network and don't need a heavyweight management tool.

10.2.8 Using the Network Computing Technologies Trap Receiver

The Trap Receiver by Network Computing Technologies is a freely available program that's worth trying.[8] This program, which currently runs only on Windows-based systems, displays trap information as it's received. It has a standard interface but can be configured to execute certain actions against traps, like OpenView's Command for Automatic Action function. Figure 10-7 shows Trap Receiver's user interface.

[8] This software can be found on their web page at http://www.ncomtech.com.

Figure 10-7. Trap Receiver



There are ways to log and forward messages and traps, send email or a page in response to a trap, as well as execute commands. By writing some code in C or C++, you can gain access to an internal trap stream. This program can be a great starting place for Windows administrators who want to use SNMP but lack the resources to implement something like OpenView. It's simple to use, extensible, and free.

10.2.9 Receiving Traps Using Net-SNMP

The last trap receiver we'll discuss is part of the Net-SNMP package, which is also freely available. *snmptrapd* allows you to send SNMP trap messages to facilities such as Unix syslog or stdout. For most applications the program works in the background, shipping messages to syslog(8). There are some configuration parameters for the syslog side of *snmptrapd;* these tell *snmptrapd* what facility level it should use for the syslog messages. The following command forwards traps to standard output *(-P)* rather than to syslog as they are received:

```
$ ./snmptrapd -P
2000-12-13 19:10:55 UCD-SNMP Version 4.1.2 Started.
2000-12-13 19:11:14 sunserver2.ora.com [12.1.45.26] enterprises.2789.2500:
        Enterprise Specific Trap (1224) Uptime: 5 days, 10:01:20.42
        enterprises.2789.2500.1224 = 123123

2000-12-13 19:11:53 sunserver2.ora.com [12.1.45.26] enterprises.2789.2500:
        Enterprise Specific Trap (1445) Uptime: 5 days, 10:01:21.20
        enterprises.2789.2500.1445 = "Fail Over Complete"
```

By now the output should look familiar; it's similar to the reports generated by the other programs we've seen in this chapter. The Net-SNMP trap daemon is another great tool for scriptwriters. A simple Perl script can watch the file in which *snmptrapd* logs its traps, looking for important events and reacting accordingly. It's easy to build a powerful and flexible monitoring system at little or no expense.

We have seen several packages that can receive traps and act on them, based on the traps' content. Keep in mind that all of these programs, whether they're free or cost tens of thousands of dollars, are basically doing the same thing: listening on some port (usually UDP port 162) and waiting for SNMP messages to arrive. What sets the various packages apart is their ability to do something constructive with the traps. Some let you program hooks that execute some other program when a certain trap is received. The simpler trap monitors just send a message logging the trap to one or more files or facilities. These packages are generally less expensive than the commercial trap monitors, but can be made to operate like full-fledged systems with some additional programming effort. Programs such as Perl give you the ability to extend these simpler packages.

10.3 Sending Traps

By now you should have a mechanism in place for receiving traps. In this section, we'll look at some different utilities that send traps and allow you to develop traps that are appropriate for your own environment. You'll notice that almost all trap utilities are command-line based. This allows you to execute the command from within a script, which is almost always what you want to do. For example, you can write a shell script that checks disk space every five minutes and sends a trap to the NMS if you're running low. You can also use these trap generators within existing programs and scripts. If you have a Perl script that accesses a database, you can use the Perl SNMP module to send a trap from within the script if a database insert fails. The possibilities are almost endless.

Although there are many different *snmptrap* programs, they are all fundamentally similar. In particular, though their command-line syntax may vary, they all expect roughly the same arguments:

Port

The UDP port to which to send the trap. The default port is 162.

SNMP version

The SNMP version appropriate to the trap you want to send. Many traps are defined only for Version 2. Note that many SNMP tools support only Version 1.

Hostname or IP address of NMS

The hostname or IP address of your NMS -- i.e., the trap's destination. It is better to use an IP address than a hostname in case you are sending traps during a Domain Name System (DNS) outage. Remember that SNMP is most valuable when your network is failing; therefore, try to avoid assuming that you have a fully functional network when you design traps.

Community name

The community name to be sent with the trap. Most management stations can be configured to ignore traps that don't have an appropriate community string.

Enterprise OID

The full enterprise OID for the trap you want to send: everything in the trap's OID from the initial .1 up to the enterprise number, including any subtrees within the enterprise but not the specific trap number. For example, if your enterprise number is 2789, you've further subdivided your enterprise to include a group of traps numbered 5000, and you want to send specific trap 1234, the enterprise OID would be .1.3.6.1.4.1.2789.5000.

If you have some reason to send a generic trap, you can set the enterprise ID to anything you want -- but it's probably best to set the enterprise ID to your own enterprise number, if you have one.

Now for the most confusing case. There are a few specific traps defined in various public MIBs. How do you send them? Basically, you construct something that looks like an enterprise OID. It's best to look at an example. One such trap is rdbmsOutOfSpace, which is defined in the RDBMS MIB. Its complete OID is .1.3.6.1.2.1.39.2.2 (.iso.org.dod.internet.mgmt.mib-2.rdbmsMIB.rdbmsTraps.rdbmsOutOfSpace). To send this trap, which is really an SNMPv2 notification, you would use everything up to rdbmsTraps as the enterprise OID, and the entire object ID as the specific trap number.

Hostname or IP address of sender

The IP address of the agent that is sending the trap. Although this may appear to be superfluous, it can be important if there is a proxy server between the agent and the NMS. This parameter allows you to record the actual address of the agent within the SNMP packet; in turn, the NMS will read the agent's address from the trap and ignore the packet's sender address. If you don't specify this

parameter, it will almost always default to the address of the machine sending the trap.

Generic trap number

A number in the range 0-6. The true generic traps have numbers 0-5; if you're sending an enterprise-specific trap, set this number to 6. Table 2-8 lists the generic traps.

Specific trap number

A number indicating the specific trap you want to send. If you're sending a generic trap, this parameter is ignored -- you're probably better off setting it to zero. If you're sending a specific trap, the trap number is up to you. For example, if you send a trap with the OID .1.3.6.1.4.1.2500.3003.0, 3003 is the specific trap number.

Timestamp

The time elapsed between the last initialization of the network entity and the generation of the trap.

OID_1, type_1, value_1

Data bindings to be included in the trap. Each data binding consists of an OID together with a datatype, followed by the value you want to send. Most programs let you include any number of data bindings in a trap. Note that the OIDs for these variable bindings are often specific to the trap and therefore "underneath" the specific OID for the trap. But this isn't a requirement, and it's often useful to send bindings that aren't defined as part of the trap.

Before we start to tackle this section, let's take a moment to review what we learned in Chapter 2 about the various datatypes:

- Each variable that we send has a particular datatype.
- Different datatypes are supported by different versions of SNMP.
- Some common datatypes are **INTEGER**, **OctetString**, **Null**, **Counter**, **Gauge**, and **TimeTicks**.

Be aware that not all programs support all datatypes. For example, the Perl SNMP module supports only the **OctetString**, **INTEGER**, and **OID** types, while the OpenView and Net_SNMP *snmptrap* commands support these three and many more. For each of the packages we use we will list, if applicable, each datatype the program supports.

In the next sections, we'll discuss *snmptrap* programs from OpenView, Network Computing Technologies, and Net-SNMP. We'll also include a script that uses a Perl module to send traps. If you are not using these particular programs in your environment, don't worry. You should still be able to relate these examples to your in-house programs.

10.3.1 Sending Traps with OpenView

OpenView has a command-line program for generating arbitrary traps called *snmptrap*. *snmptrap* supports the counter, counter32, counter64,[9] gauge, gauge32, integer, integer32, ipaddress, null, objectidentifier, octetstring, octetstringascii, octetstringhex, octetstringoctal, opaque, opaqueascii, opaquehex, opaqueoctal, timeticks, and unsigned32 datatypes. Its command-line structure looks like this:

[9] This type will work only on agents that support SNMPv2.

```
snmptrap -c community [-p port] node_addr enterprise_id agent-addr generic \
specific timestamp [OID type value] ...
```

Here's a typical *snmptrap* command. It sends one trap, with three ASCII-string variable bindings for values:

```
$ /opt/OV/bin/snmptrap -c public nms \
.1.3.6.1.4.1.2789.2500 "" 6 3003 "" \
.1.3.6.1.4.1.2789.2500.3003.1 octetstringascii "Oracle" \
.1.3.6.1.4.1.2789.2500.3003.2 octetstringascii "Backup Not Running" \
.1.3.6.1.4.1.2789.2500.3003.3 octetstringascii "Call the DBA Now for Help"
```

It's a complicated command, and it's hard to imagine that you would ever type it on the command line. Let's break it up into pieces. The first line specifies the community string (public) and the address to which the trap should be sent (nms, though in practice it would be better to use an IP address rather than a node name). The next line is in many respects the most complicated. It specifies the enterprise ID for the trap we're going to send (.1.3.5.1.6.1.2789.2500, which is a subtree of the enterprise-specific tree we've devoted to traps); the address of the agent sending the trap (in this case, the null string "", which defaults to the agent's address; if you're using a proxy server, it is useful to specify the agent's address explicitly); the generic trap number (6, which is used for all enterprise-specific traps); the specific trap number (3003, which we've assigned); and a timestamp ("", which defaults to the current time).

The remaining three lines specify three variable bindings to be included with the trap. For each binding, we have the variable's object ID, its datatype, and its value. The variables we're sending are defined in our private (enterprise-specific) MIB, so their OIDs all begin with .1.3.6.1.4.1.2789.2500. All the variables are strings, so their datatype is octetstringascii. The trap PDU will be packed with these three strings, among other things. The program that receives the trap will decode the trap PDU and realize that there are three variable bindings in the trap. These variable bindings, like the one that reads "Call the DBA Now for Help," can be used to alert the operator that something bad has happened.

10.3.2 Sending Traps with Perl

In Chapter 8 we learned how to use the *get* and *set* pieces of the SNMP Perl module. In this section we'll see how to use the snmptrap( ) routine to generate traps. Currently, SNMP_util supports only three types for traps: string, int, and oid. This can seem limiting, but it covers most needs. Here's how *snmptrap* is called:

```
snmptrap(communityname@host:port_number, enterpriseOID, host_name_from, \
generic_ID, specific_ID, OID, type, value, [OID, type, value ...])
```

One call to *snmptrap* can include any number of values; for each value, you must specify the object ID, the datatype, and the value you're reporting. The next script generates a trap with only one value:

```
#!/usr/local/bin/perl
# Filename: /opt/local/perl_scripts/snmptrap.pl

use SNMP_util "0.54";  # This will load the BER and SNMP_Session for us

snmptrap("public\@nms:162", ".1.3.6.1.4.1.2789", "sunserver1", 6, 1247, \
        ".1.3.6.1.4.1.2789.1247.1", "int", "2448816");
```

The call to snmptrap( ) sends a trap to port 162 on host nms. The trap is sent from host sunserver1; it contains a single variable binding, for the object .1.3.6.1.4.1.2789.1247.1. The OID's type is int and its value is 2448816.

Now let's try sending a trap with multiple values (multiple variable bindings). The first object we'll report is an integer, to which we give the arbitrary value 4278475. The second object has a string value and is a warning that our database has stopped. Because we're using OIDs that belong to our own enterprise, we can define these objects to be anything we want:

```
snmptrap("public\@nms:162", ".1.3.6.1.4.1.2789", "sunserver2", 6, 3301, \
        ".1.3.6.1.4.1.2789.3301.1", "int",    "4278475", \
        ".1.3.6.1.4.1.2789.3301.2", "string", "Sybase DB Stopped");
```

We can use the Net-SNMP *snmptrapd* program to monitor the traps coming in. We executed the preceding Perl code while running *snmptrapd* in stdout mode, and received:

```
$ ./snmptrapd -P
1999-10-12 09:45:08  [12.1.45.26] enterprises.2789.3000:
        Enterprise Specific Trap (3301) Uptime: 0:00:00
        enterprises.2789.3301.1 = 4278475
        enterprises.2789.3301.2 = "Sybase DB Stopped"
```

snmptrapd reported both of the values we sent in the trap: we see the integer value 4278475 and the notification that Sybase has stopped. Although this example is highly artificial, it's not all that different from what you would do when writing your own monitoring software. You would write whatever code is necessary to monitor vital systems such as your database and use the Perl SNMP module to send traps when significant events occur. You can then use any program capable of receiving traps to inform you when the traps arrive. If you want, you can add

logic that analyzes the values sent in the trap or takes other actions, such as notifying an operator via a pager.

10.3.3 Sending Traps with Network Computing Technologies Trap Generator

This Windows-based command-line utility gives us the same features as its Unix counterparts. It understands the String, Counter, Gauge, Integer, Address, OID, and TimeTicks datatypes. The command line for *nttrapgen* looks like this:

```
nttrapgen.exe -d DestinationIpAddress:port -c CommunityName
              -o senderOID -i senderIP -g GenericTrapType
              -s SpecificTrapType -t timestamp -v OID TYPE VALUE
```

Here's how to use *nttrapgen* to send a trap notifying us that the UPS battery is running low. We use the String datatype to send an informative message, and we use trap 4025.1 from our private enterprise ID, 2789:

```
C:\tools> nttrapgen.exe -d nms:162 -c public -o ^
1.3.6.1.4.1.2789.4025 -i 10.123.456.4 -g 6 -s 4025 -t 124501 ^
-v 1.3.6.1.4.1.2789.4025.1 STRING 5 Minutes Left On UPS Battery
```

This trap will be sent to our network-management station (which has the hostname nms) on port 162, which is the standard port for SNMP traps. Any management station should be able to receive the trap and act on it appropriately. You can use this command in batch scripts, which are essentially the same as Unix shell scripts. Therefore, you can use *nttrapgen* to generate traps as you need them: you can write scripts that monitor key processes and generate traps when any interesting events take place. As with the earlier Perl example, you can use this simple trap generator in your environment if you don't need a heavy-duty management system.

10.3.4 Sending Traps with Net-SNMP

This *snmptrap* program looks very similar to OpenView's *snmptrap*. This program uses a single letter to refer to datatypes, as shown in Table 10-2.

Table 10-2. Net-SNMP snmptrap Datatypes

Abbreviation	Datatype
a	IP address
c	Counter
d	Decimal string
i	Integer
n	Null

o	Object ID
s	String
t	Time ticks
u	Unsigned integer
x	Hexadecimal string

Here's how the Net-SNMP *snmptrap* program is invoked:

```
snmptrap hostname community enterprise-oid agent \
generic-trap specific-trap uptime [OID type value]...
```

If you use two single quotes ('') in place of the time, *snmptrap* inserts the current time into the trap. The following command generates a trap with a single value. The object ID is **2005.1**, within our private enterprise; the value is a string that tells us that the web server has been restarted:

```
$ snmptrap nms public .1.3.6.1.4.1.2789.2005 ntserver1 6 2476317 '' \
.1.3.6.1.4.1.2789.2005.1 s "WWW Server Has Been Restarted"
```

Here's how to send a Version 2 notification with Net-SNMP:[10]

[10] For information about sending Version 3 notifications with Net-SNMP, see Appendix F.

```
$ snmptrap -v2c nms public '' .1.3.6.1.6.3.1.1.5.3 \
ifIndex i 2 ifAdminStatus i 1 ifOperStatus i 1
```

The command is actually simpler than its Version 1 equivalent. There are no generic numbers, specific numbers, or vendor IDs. The "" argument defaults to the current system uptime. The OID specifies the linkDown notification, with three data bindings specifying the link's status. The definition of linkDown in the IF-MIB states that the linkDown notification must include the ifIndex, ifAdminStatus, and ifOperStatus objects, which report the index of the interface that went down, its administrative status, and its operational status, respectively. For ifAdminStatus and ifOperStatus, a value of 1 indicates that the link is up. So this notification reports that interface 2 has changed its state from "down" to "up."

Again, the *snmptrap* command-line tool lets you integrate SNMP monitoring into shell scripts and other programs.

10.3.5 Forcing Your Hardware to Generate Traps

When you install a new piece of equipment, you should verify that it generates traps correctly. Testing your equipment's ability to generate traps has the added benefit of testing the behavior of your NMS; you can ensure that it handles traps in the way you want. The best way to test new hardware is to read your vendor's MIB and look for all the **TRAP-TYPE**s they have

defined. This will give you a good feel for what sort of traps your vendor has implemented. For example, I read through our APC MIB and noticed that the unit will send a trap when it goes onto battery power if the AC power goes out. To test this feature, I secured the area in our datacenter and switched off the circuit breaker to simulate a power failure. The trap was generated, but it showed up in the Error event category because I did not have the correct MIB loaded in OpenView. I took the OID from the Error events and searched the APC MIBs for a match. When I found one, I loaded the MIB file into OpenView and repeated the test. This time, when the trap was received OpenView put an informative message in the Event Categories.

Most SNMP-compatible routers, switches, and network devices can generate linkDown traps. From RFC 1157, a linkDown trap is a "failure in one of the communication links represented in the agent's configuration." This means that if you start unplugging ports on your router you should receive traps, right? Yes, but first make sure you don't start disconnecting production database servers. Furthermore, make sure you don't disconnect the port by which your device would send the trap back to the NMS. Remember, SNMP is designed with the assumption that the network is unreliable -- if something sends a trap but there's no way for the trap to reach its destination, no one will find out. By default, a linkDown trap won't appear in OpenView's Event Categories, because the default setting for linkDown is "Log only"; watch the log file *$OV_LOG/trapd.log* to see these traps arrive. Once you have a mechanism for receiving traps, bringing the link up and down on your device should send some traps your way.

10.3.6 Using Hooks with Your Programs

A *hook* is a convenient interface that lets you integrate your own code into some other product. The Emacs text editor is a good example of a program that uses hooks, almost entirely, to allow its users to extend how it operates. Let's look at the following simple program to explain this concept further:

```
# Logical Sample Program NH1
# PROGRAM COMMENTS
# PROGRAM BEGINS

        PROGRAM ADDS            $VAR1 + $VAR2 = $VAR3
        PROGRAM SUBTRACTS       $VAR5 - $VAR6 = $VAR7
        PROGRAM PRINTS RESULTS $VAR3 $VAR7

# PROGRAM ENDS
```

This program simply ADDS, SUBTRACTS, and PRINTS RESULTS; it does not have any hooks. To add a feature, you have to modify the code. For a small program like this that is a trivial exercise, but it would be difficult in a program of any size. The next program contains some hooks that let you add extensions:

```
# Logical Sample Program H1
# PROGRAM COMMENTS
# PROGRAM BEGINS
   PROGRAM RUNS $PATH/start.sh

   PROGRAM ADDS            $VAR1 + $VAR2 = $VAR3
   PROGRAM SUBTRACTS       $VAR5 - $VAR6 = $VAR7
   PROGRAM PRINTS RESULTS $VAR3 $VAR7

   PROGRAM RUNS $PATH/end.sh
# PROGRAM ENDS
```

Notice the two additional **RUNS** statements. These hooks allow you to run anything you want at the start or end of the program. The first program, *start.sh*, might be as simple as the command *echo "I am starting"*, which sends a simple message to the system or management console. This script could also call one of the trap-generation programs to send a trap to the NMS stating that some program is starting. It would be even more useful to send a message when the program terminates, possibly including information about the program's status. Here's a slightly more complicated program that runs a script, providing a number of arguments so that the script can send useful information back to the NMS when it generates a trap:

```
# Logical Sample Program H2
# PROGRAM COMMENTS
# PROGRAM BEGINS
   PROGRAM RUNS $PATH/start.sh $PROGRAM_NAME

   PROGRAM ADDS            $VAR1 + $VAR2 = $VAR3
   PROGRAM SUBTRACTS       $VAR5 - $VAR6 = $VAR7
   PROGRAM PRINTS RESULTS $VAR3 $VAR7

   PROGRAM RUNS $PATH/end.sh $PROGRAM_NAME $VAR1 $VAR2 $VAR3 $VAR5 $VAR6 $VAR7
# PROGRAM ENDS
```

With the additional arguments available to the hook programs, we can generate messages like "The Program Widget has ended with sales at $4 and YTD at $7." If your hook programs are shell scripts, you can simply add *snmptrap* commands via a text editor. Once you finish adding the *snmptrap* code, you can test your hook program by running it on the command line.

On most systems, many scripts can benefit from *snmptrap* hooks. On Solaris or Linux machines, for example, some of your */etc/init.d* scripts can be retrofitted to make use of *snmptrap* commands. It might be useful to have some kind of notification when important processes such as your web server or DNS server start and stop. Having such information on hand might make life much easier for your helpdesk. (The Concord SystemEDGE SNMP agent provides more rigorous process-monitoring capabilities. See Chapter 11 for more information on this product.)

It's harder to add hooks to programs written in languages like C, because you need access to the source code as well as the ability to figure out where to place the hooks. Once you have identified where your hooks go and added them, you must recompile the source code. Some programs have hooks built in, allowing you to run external programs or RPCs. Check your program's documentation for the locations of these hooks. This is much more convenient than trying to build your own hooks into another program. Once you have established what these external programs are called, you can start writing your own traps or adding to existing ones.

Chapter 11. Extensible SNMP Agents

There will come a time when you want to extend an agent's functionality. Extending an agent usually means adding or changing the MIBs the agent supports. Many agents that claim to support SNMP cover only a minimal number of somewhat useless MIBs -- obviously a frustrating situation for someone who is planning on doing lots of automated network management. Upgrading your software to a newer version of SNMP, say Version 2 or 3, won't help; you won't get any more information out of a device than if you were using SNMPv1. The newer versions of SNMP add features to the protocol (such as additional security or more sophisticated options for retrieving and setting values), but the information that's available from any device is defined in the agent's MIBs, which are independent of the protocol itself.

When you are faced with an agent's limitations, you can turn to extensible agents.[1] These programs, or extensions to existing programs, allow you to extend a particular agent's MIB and retrieve values from an external source (a script, program, or file). In some cases, data can be returned as if it were coming from the agent itself. Most of the time you will not see a difference between the agent's native MIBs and your extensible ones. Many extensible agents give you the ability to read files, run programs, and return their results; they can even return tables of information. Some agents have configurable options that allow you to run external programs and have preset functions, such as disk-space checkers, built in.

> [1] We don't make a distinction between existing agents that can be extended and agents that exist purely to support extensions. We'll call them both "extensible agents."

The OpenView, Net-SNMP, and SystemEDGE agents are all examples of extensible agents. OpenView provides a separate extensible agent that allows you to extend the master agent (snmpdm); requests for the extensible agent won't work unless the master agent is running. You can start and stop the extensible agent

without disturbing the master agent. To customize the extensible agent you define new objects using the ASN.1 format, as specified by the SMI. The Net-SNMP agent takes an alternate approach. It doesn't make a distinction between the master agent and the extensible agent; there's only one agent to worry about. You can use ASN.1 to define new objects (as with the OpenView extensible agent), but there's also a facility for adding extensions without writing any ASN.1, making this agent significantly more accessible for the novice administrator. SystemEDGE is similar to Net-SNMP in that there is only one agent to worry about. Of the three agents discussed in this chapter, it is the easiest to extend. Figure 11-1 compares the design strategies of the OpenView, Net-SNMP, and SystemEDGE agents.

Figure 11-1. Architecture of extensible agents

All three agents have fairly comprehensive configuration options and all allow you to extend the local agent without heavy programming. You may need to write some scripts or a few short C programs, but with the sample programs here and the thousands more that are on the Internet,[2] nonprogrammers can still get a lot done.

[2] See Chapter 1 for a list of a few web sites that have links to commercial and free SNMP software.

We'll start with the Net-SNMP agent, since it is the simplest, then move to SystemEDGE. We'll round out the discussion with OpenView's extensible agent. Be sure to see Chapter 5 for information on where to obtain these agents.

11.1 Net-SNMP

When you install the Net-SNMP package, it creates a sample *snmpd.conf* configuration file called *EXAMPLE.conf* in the source directory. This file contains some great examples that demonstrate how to extend your agent. Read through it to see the types of things you can and can't do. We will touch on only a few of Net-SNMP's features: checking for any number of running processes (proc), executing a command that returns a

single line of output (exec), executing a command that returns multiple lines of output (exec), and checking disk-space utilization (disk).

The main Net-SNMP configuration file can be found at *$NET_SNMP_HOME/share/snmp/snmpd.conf*, where $NET_SNMP_HOME is the directory in which you installed Net-SNMP. Here is the configuration file that we will use for the remainder of this section:

```
# Filename: $NET_SNMP_HOME/share/snmp/snmpd.conf
# Check for processes running
# Items in here will appear in the ucdavis.procTable
proc sendmail 10 1
proc httpd

# Return the value from the executed program with a passed parm.
# Items in here will appear in the ucdavis.extTable
exec FileCheck /opt/local/shell_scripts/filecheck.sh /tmp/vxprint.error

# Multiline return from the command
# This needs its own OID
# I have used a subset of my registered enterprise ID (2789) within the OID
exec .1.3.6.1.4.1.2021.2789.51 FancyCheck /opt/local/shell_scripts/fancycheck.sh \
  /core

# Check disks for their mins
disk / 100000
```

Whenever you make changes to the Net-SNMP agent's configuration file, you can have it reread the configuration by sending the process an HUP signal:

```
$ ps -ef | grep snmpd
   root   12345     1  0   Nov 16 ?        2:35 /usr/local/bin/snmpd
$ kill -HUP 12345
```

Now let's look at the file itself. The first **proc** command says to check for the process **sendmail**. The numbers **10** and **1** define how many *sendmail* processes we want running at any given time (a maximum of 10 and a minimum of 1). The second **proc** command says that we want at least one **httpd** process running. To see what effect these commands have on our agent, let's look at an *snmpwalk* of ucdavis.procTable (.1.3.6.1.4.1.2021.2):

```
$ snmpwalk sunserver2 public .1.3.6.1.4.1.2021.2
enterprises.ucdavis.procTable.prEntry.prIndex.1 = 1
enterprises.ucdavis.procTable.prEntry.prIndex.2 = 2
enterprises.ucdavis.procTable.prEntry.prNames.1 = "sendmail"
enterprises.ucdavis.procTable.prEntry.prNames.2 = "httpd"
enterprises.ucdavis.procTable.prEntry.prMin.1 = 1
enterprises.ucdavis.procTable.prEntry.prMin.2 = 0
enterprises.ucdavis.procTable.prEntry.prMax.1 = 10
enterprises.ucdavis.procTable.prEntry.prMax.2 = 0
enterprises.ucdavis.procTable.prEntry.prCount.1 = 1
enterprises.ucdavis.procTable.prEntry.prCount.2 = 6
enterprises.ucdavis.procTable.prEntry.prErrorFlag.1 = 0
```

```
enterprises.ucdavis.procTable.prEntry.prErrorFlag.2 = 0
enterprises.ucdavis.procTable.prEntry.prErrMessage.1 = ""
enterprises.ucdavis.procTable.prEntry.prErrMessage.2 = ""
enterprises.ucdavis.procTable.prEntry.prErrFix.1 = 0
enterprises.ucdavis.procTable.prEntry.prErrFix.2 = 0
```

The agent returns the contents of the procTable. In this table, the *sendmail* and *httpd* process entries occupy instances 1 and 2. prMin and prMax are the minimum and maximum numbers we set for the *sendmail* and *httpd* processes.[3] The prCount value gives us the number of processes currently running: it looks like we have one *sendmail* process and six *httpd* processes. To see what happens when the number of processes falls outside the range we specified, let's kill all six *httpd* processes and look at the procTable again (instead of listing the whole table, we'll walk only instance 2, which describes the *httpd* process):

[3] When prMin and prMax are both 0, it says that we want at least one and a maximum of infinity processes running.

```
$ snmpwalk sunserver2 public .1.3.6.1.4.1.2021.2
enterprises.ucdavis.procTable.prEntry.prIndex.1 = 1
enterprises.ucdavis.procTable.prEntry.prNames.1 = "httpd"
enterprises.ucdavis.procTable.prEntry.prMin.1 = 0
enterprises.ucdavis.procTable.prEntry.prMax.1 = 0
enterprises.ucdavis.procTable.prEntry.prCount.1 = 0
enterprises.ucdavis.procTable.prEntry.prErrorFlag.1 = 1
enterprises.ucdavis.procTable.prEntry.prErrMessage.1 = "No httpd
process running."
enterprises.ucdavis.procTable.prEntry.prErrFix.1 = 0
```

We had six *httpd* processes running and now, per prCount, we have none. The prErrMessage reports the problem, and the prErrorFlag has changed from 0 to 1, indicating that something is wrong. This flag makes it easy to poll the agent, using the techniques discussed in Chapter 9, and see that the *httpd* processes have stopped. Let's try a variation on this theme. If we set prMin to indicate that we want more than six *httpd* processes running, then restart *httpd*, our prErrMessage is:

```
enterprises.ucdavis.procTable.prEntry.prErrMessage.1 = "Too few
httpd running (# = 0)"
```

The next command in the configuration file is exec; this command allows us to execute any program and return the program's results and exit value to the agent. This is helpful when you already have a program you would like to use in conjunction with the agent. We've written a simple shell script called *filecheck.sh* that checks whether the file that's passed to it on the command line exists. If the file exists, it returns a 0 (zero); otherwise, it returns a 1 (one):

```
#!/bin/sh
# FileName: /opt/local/shell_scripts/filecheck.sh

if [ -f $1 ]; then
   exit 0
```

```
fi
exit 1
```

Our configuration file uses *filecheck.sh* to check for the existence of the file */tmp/vxprint.error*. Once you have the *filecheck.sh* script in place, you can see the results it returns by walking ucdavis.extTable (.1.3.6.1.4.1.2021.8):

```
$ snmpwalk sunserver2 public .1.3.6.1.4.1.2021.8
enterprises.ucdavis.extTable.extEntry.extIndex.1 = 1
enterprises.ucdavis.extTable.extEntry.extNames.1 = "FileCheck"
enterprises.ucdavis.extTable.extEntry.extCommand.1 =
"/opt/local/shell_scripts/filecheck.sh /tmp/vxprint.error"
enterprises.ucdavis.extTable.extEntry.extResult.1 = 0
enterprises.ucdavis.extTable.extEntry.extOutput.1 = ""
enterprises.ucdavis.extTable.extEntry.extErrFix.1 = 0
```

The first argument to the **exec** command[4] in the configuration file is a label that identifies the command so we can easily recognize it in the extTable. In our case we used **FileCheck** -- that's not a particularly good name, because we might want to check the existence of several files, but we could have named it anything we deemed useful. Whatever name you choose is returned as the value of the extTable.extEntry.extNames.1 object. Because the file */tmp/vxprint.error* exists, *filecheck.sh* returns a 0, which appears in the table as the value of extTable.extEntry.extResult.1. You can also have the agent return a line of output from the program. Change *filecheck.sh* to perform an *ls -la* on the file if it exists:

[4] See the *EXAMPLE.conf* configuration file introduced at the beginning of this chapter.

```
#!/bin/sh
# FileName: /opt/local/shell_scripts/filecheck.sh

if [ -f $1 ]; then
   ls -la $1
   exit 0
fi

exit 1
```

When we poll the agent, we see the output from the script in the extOutput value the agent returns:

```
enterprises.ucdavis.extTable.extEntry.extOutput.1 = \
"  16 -rw-r--r--   1 root     other       2476 Feb 3 17:13 /tmp/vxprint.error."
```

This simple trick works only if the script returns a single line of output. If your script returns more than one line of output, insert an OID in front of the string name in the **exec** command.

Here's the next command from our *snmpd.conf* file:

```
exec .1.3.6.1.4.1.2021.2789.51 FancyCheck /opt/local/shell_scripts/fancycheck.sh \
/core
```

This command runs the program *fancycheck.sh*, with the identifying string **FancyCheck**. We won't bother to list *fancycheck.sh*; it's just like *filecheck.sh*, except that it adds a check to determine the file type. The OID identifies where in the MIB tree the agent will place the result of running the command. It needs to be in the ucdavis enterprise (.1.3.6.1.4.1.2021). We recommend that you follow the ucdavis enterprise ID with your own enterprise number, to prevent collisions with objects defined by other sources and avoid overwriting one of ucdavis's subtrees. Follow your enterprise number with another number to identify this particular command. In this case, our enterprise ID is 2789 and we assign the arbitrary number 51 to this command. Thus, the complete OID is .1.3.6.1.4.1.2021.2789.51.

Here are the results from walking the .1.3.6.1.4.1.2021.2789.51 subtree:

```
$ snmpwalk sunserver2 public .1.3.6.1.4.1.2021.2789.51
enterprises.ucdavis.2789.51.1.1 = 1
enterprises.ucdavis.2789.51.2.1 = "FancyCheck"
enterprises.ucdavis.2789.51.3.1 =
"/opt/local/shell_scripts/fancycheck.sh /core"
ucdavis.2789.51.100.1 = 0
ucdavis.2789.51.101.1 = "-rw-r--r--   1 root     other
346708 Feb 14 16:30 /core."
ucdavis.2789.51.101.2 = "/core:..ELF 32-bit MSB core file SPARC
Version 1, from 'httpd'."
ucdavis.2789.51.102.1 = 0
```

Notice that we have a few additional lines in our output. 2789.51.100.1 is the exit number, 2789.51.101.1 and 2789.51.101.2 are the output from the command, and 2789.51.102.1 is the errorFix value. These values can be useful when you are trying to debug your new extension. (Unfortunately, *snmpwalk* can give you only the numeric OID, not the human-readable name, because *snmpwalk* doesn't know what 2789.51.x is.)

The last task for Net-SNMP's extensible agent is to perform some disk-space monitoring. This is a great option that lets you check the availability of disk space and return multiple (useful) values. The **disk** option takes a filesystem mount point followed by a number. Here is what our entry looks like in *snmpd.conf*:

```
# Check disks for their mins
disk / 100000
```

The definition of the **disk** option from *UCD-SNMP-MIB.txt* is "Minimum space required on the disk (in kBytes) before the errors are triggered." Let's first take a look on sunserver2 to see what the common *df* program returns:

```
$ df -k /
Filesystem            kbytes    used   avail capacity  Mounted on
```

```
/dev/dsk/c0t0d0s0     432839   93449  296110    24%    /
```

To see what SNMP has to say about the disk space on our server, run *snmpwalk* against the ucdavis.diskTable object (.1.3.6.1.4.1.2021.9). This returns virtually the same information as the *df* command:

```
$ snmpwalk sunserver2 public .1.3.6.1.4.1.2021.9
enterprises.ucdavis.diskTable.dskEntry.dskIndex.1 = 1
enterprises.ucdavis.diskTable.dskEntry.dskPath.1 = "/" Hex: 2F
enterprises.ucdavis.diskTable.dskEntry.dskDevice.1 =
"/dev/dsk/c0t0d0s0"
enterprises.ucdavis.diskTable.dskEntry.dskMinimum.1 = 100000
enterprises.ucdavis.diskTable.dskEntry.dskMinPercent.1 = -1
enterprises.ucdavis.diskTable.dskEntry.dskTotal.1 = 432839
enterprises.ucdavis.diskTable.dskEntry.dskAvail.1 = 296110
enterprises.ucdavis.diskTable.dskEntry.dskUsed.1 = 93449
enterprises.ucdavis.diskTable.dskEntry.dskPercent.1 = 24
enterprises.ucdavis.diskTable.dskEntry.dskErrorFlag.1 = 0
enterprises.ucdavis.diskTable.dskEntry.dskErrorMsg.1 = ""
```

As you can see, the Net-SNMP agent has many customizable features that allow you to tailor your monitoring without having to write your own object definitions. Be sure to review *$NET_SNMP_HOME/share/snmp/mibs/UCD-SNMP-MIB.txt* for complete definitions of all Net-SNMP's variables. While we touched on only a few customizable options here, you will find many other useful options in the *EXAMPLE.conf* file that comes with the Net-SNMP package.

11.2 SystemEDGE

The SystemEDGE agent is also extensible. No other system processes need to be run in order to extend this agent. It comes with three predefined extended objects: Domain Name System (DNS) for Unix, Network Information System (NIS) for Unix, and Remote Pinger for Unix and Windows NT. The first object returns the domain name of the underlying operating system, the second returns the NIS domain name of the underlying operating system, and the third sends ICMP requests to a remote host from the system on which the agent is running. While these are nice scripts to have, what we want to focus on is how to add your own OIDs to the agent.

11.2.1 Extensibility for Unix and Windows NT

The SystemEDGE agent has a private MIB that defines a table called the extensionGroup. Its full OID is 1.3.6.1.4.1.546.14 (iso.org.dod.internet.private.enterprises.empire.extensionGroup). This is where you define your own objects. The first object you define has the OID extensionGroup.1.0 (1.3.6.1.4.1.546.14.1.0), where the .0 indicates that the object is scalar; the next has the OID extensionGroup.2.0, and so on. Note that all the objects defined this way must be scalar. For advanced users, Concord has developed a plug-in

architecture for SystemEDGE that allows you to develop complex extended objects (including tables) and full-blown MIBs.

To extend the agent, start by editing the *sysedge.cf* file. This file tells the agent to which extended OIDs it must respond. The format of a command in this file is:

```
extension LeafNumber Type Access 'Command'
```

The keyword **extension** tells the agent that this configuration entry is an extension that belongs to the extensionGroup. LeafNumber is the extension object number -- i.e., the number you assign to the object in the extensionGroup table. Type is the SNMP type for the OID. Valid types are **Integer**, **Counter**, **Gauge**, **Octetstring**, **TimeTicks**, **Objectid**, and **IPAddress**. Access is either **Read-Only** or **Read-Write**. And finally, Command is the script or program the agent will execute when this particular OID is queried by an NMS. We'll talk more about this shortly. Here are some examples of extension objects:

```
extension 1 Integer Read-Only '/usr/local/bin/Script.sh'
extension 2 Gauge Read-Only '/usr/local/bin/Script.pl'
extension 33 Counter Read-Write '/usr/local/bin/Program'
```

The first item defines a read-only OID of type **Integer**. The OID is 1.3.6.1.4.1.546.14.1.0. The agent will execute the command /usr/local/bin/exampleScript.sh when this OID is queried. The second entry is similar, except its type is **Gauge** and its numeric OID is 1.3.6.1.4.1.546.14.2.0. The third example simply shows that LeafNumber doesn't have to be sequential; you can use any number you want, provided that it is unique.

Extending the agent allows you to write your own scripts that do whatever you want: you can get information about devices or programs that are not SNMP-capable, as long as you can write a script that queries them for their status. In the example above, */usr/local/bin/Script.sh*, */usr/local/bin/Script.pl*, and */usr/local/bin/Program* are all examples of scripts the agent will execute when the OID assigned to each script is queried. Two requirements must be met by any script or program:

- All *set*, *get*, and *getnext* requests must generate output. For *get* and *getnext*, the output from the script should be the actual value of the object requested. This means that the script or program that fetches the required information must return a single value. For a *set* request, the script should return the object's new value. The request will fail if there is no output. (Note that for a *set* request, a script may succeed in changing the state of the device even if it produces no output and the agent considers the script to have failed.)

- The script or program should print whatever information it needs to return (based on the type of request), followed by a newline character. The agent parses only up to this character. If a newline is the first character the agent encounters, the agent generates an error and returns this to the NMS or SNMP application.

The agent sends three arguments to the script or program it executes: the LeafNumber, the request type (GET, GETNEXT, or SET, in capital letters), and a string that represents some value to be set (the third argument is used only for SET requests). The following skeletal Perl script, called *skel.pl*, shows how you can use all three arguments:

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

if ($ARGV[0] == 1) {
   # OID queried is 1.3.6.1.4.1.546.14.1.0
   if ($ARGV[1] eq "SET") {
         # use $ARGV[2] to set the value of something and return the set value,
         # followed by a newline character, to the agent
   } elsif (($ARGV[1] eq "GET") || ($ARGV[1] eq "GETNEXT")) {
         # get the information to which this OID pertains, then return it,
         # followed by a newline character, to the agent
   }
} else {
   return 0;
   # return 0, since I don't know what to do with this OID
}
```

All you have to do is add the logic that takes some action to retrieve (or set) the appropriate value and return the correct value to the agent. The corresponding entry in *sysedge.cf* might look something like this:

```
extension 1 Integer Read-Write '/usr/local/bin/skel.pl'
```

What we've done so far gives the agent the ability to respond to requests for a new kind of data. We still need to solve the other part of the puzzle: telling the management station that some new kind of data is available for it to retrieve. This requires creating an entry in a MIB file.[5] After adding this entry to the file, you must recompile the MIB into your NMS system so that the NMS will know the access and type of each of the extended objects in the MIB for which it is to perform queries. Here is a MIB entry that corresponds to the previous agent extension:

[5] Concord recommends that you keep all your extended MIB objects in a separate file, away from the SystemEDGE MIB file. This makes it easier for you to recompile it into your NMS.

```
skeletonVariable OBJECT-TYPE
    SYNTAX Integer
    ACCESS Read-Write
    DESCRIPTION
        "This is an example object."
::= { extensionGroup 1 }
```

Once this is compiled into the NMS, you can query the object by specifying its full name (iso.org.dod.internet.private.enterprises.empire.extensionGroup.skeletonVariable.0). Alternatively, you can use the numeric OID; for example:

```
$ snmpget server.ora.com public .1.3.6.1.4.1.546.14.1.0
```

Security can be a concern when writing your own extension scripts. On Unix systems, it's a good idea to create a separate user and group to execute your extensions, rather than allowing the root user to run your scripts.

11.2.2 Added Extensibility for Windows NT

While the extensionGroup is supported on all platforms, the Windows NT version of SystemEDGE allows you to extend SystemEDGE with objects taken from the registry and performance registry. You can gain access to configuration data and performance data, which are normally viewed using regedit and perfmon. The Windows NT extension group is defined as iso.org.dod.internet.private.enterprises.empire.nt.ntRegPerf (1.3.6.1.4.1.546.5.7). As with the Unix extensions, the NT extensions are defined in the *sysedge.cf* file.

To configure a registry extension, add a line with the following syntax to *sysedge.cf*:

```
ntregperf LeafNumber Type Registry 'Key' 'Value'
```

The keyword **ntregperf** defines this as an NT registry or performance extension object. LeafNumber and Type are the same as for Unix extensions. The keyword **Registry** identifies this entry as a registry extension. Registry extensions are read-only. Key is a quoted string that specifies the registry key to be accessed. Value is the value you want to read from the key. Here is an example:

```
ntregperf 1 OctetString Registry
'SYSTEM\CurrentControlSet\Control\CrashControl' 'DumpFile'
```

This creates a registry extension object that returns the path to the crash-control dump file. The OID is 1.3.6.1.4.1.546.5.7.1.0 (iso.org.dod.internet.private.enterprises.empire.nt.ntRegPerf.1.0).

To configure a performance extension, use the following syntax:

```
ntregperf LeafNumber Type Performance 'Object' 'Counter' 'PerfInstance'
```

Here again, **ntregperf** is the keyword that indicates this is an NT registry/performance extension object. LeafNumber and Type should be familiar to you. The keyword **Performance** indicates that we're reading a value from the performance registry; performance extensions are read-only. Object specifies the performance object to be accessed. Counter specifies the object's performance counter value to be accessed. Finally, PerfInstance specifies the performance counter instance to be accessed. This should be identical to what's listed with perfmon. Here's a typical performance extension:

```
ntregperf 2 Counter Performance 'TCP' 'Segments Sent/sec' '1'
```

You can use this extension to watch the total number of TCP segments transmitted by the system. Its OID is 1.3.6.1.4.1.546.5.7.2.0 (iso.org.dod.internet.private.enterprises.empire.nt.ntRegPerf.2.0). Keep in mind that you should create a MIB entry (in a MIB file) for any NT extensions you create, similar to the entry we defined above for skeletonVariable.

The examples in this section should be enough to get you up and running with an extended SystemEDGE agent. Be sure to read the SystemEDGE manual for a complete treatment of this topic.

11.3 OpenView's Extensible Agent

Before you start playing around with OpenView's extensible agent, make sure that you have its master agent (snmpdm) configured and running properly. You must also obtain an enterprise number, because extending the OpenView agent requires writing your own MIB definitions, and the objects you define must be part of the enterprises subtree.[6] Chapter 2 describes how to obtain an enterprise number.

[6] Do not use my enterprise number. Obtaining your own private enterprise number is easy and free. Using my number will only confuse you and others later in the game.

MIBs are written using the SMI, of which there are two versions: SMIv1, defined in RFCs 1155 and 1212; and SMIv2, defined in RFCs 2578, 2579, and 2580. RFC 1155 notes that "ASN.1 constructs are used to define the structure, although the full generality of ASN.1 is not permitted." While OpenView's extensible agent file *snmpd.extend* uses ASN.1 to define objects, it requires some additional entries to create a usable object. *snmpd.extend* also does not support some of the SNMPv2 SMI constructs. In this chapter, we will discuss only those constructs that are supported.

By default, the configuration file for the extensible agent in the Unix version of OpenView is */etc/SnmpAgent.d/snmp.extend*. To jump right in, copy the sample file to this location and then restart the agent:

```
$ cp /opt/OV/prg_samples/eagent/snmpd.extend /etc/SnmpAgent.d/
$ /etc/rc2.d/S98SnmpExtAgt stop
$ /etc/rc2.d/S98SnmpExtAgt start
```

You should see no errors and get an exit code of 0 (zero). If errors occur, check the *snmpd.log* file.[7] If the agent starts successfully, try walking one of the objects monitored by the extensible agent. The following command checks the status of the mail queue:

[7] On Solaris and HP-UX machines this file is located in */var/adm/snmpd.log*.

```
$ snmpwalk sunserver1  .1.3.6.1.4.1.4242.2.2.0
4242.2.2.0 : OCTET STRING- (ascii):     Mail queue is empty
```

We're off to a good start. We have successfully started and polled the extensible agent.

The key to OpenView's *snmpd.extend* file is the **DESCRIPTION**. If this seems a little weird, it is! Executing commands from within the **DESCRIPTION** section is peculiar to this agent, not part of the SNMP design. The **DESCRIPTION** tells the agent where to look to read, write, and run files. You can put a whole slew of parameters within the **DESCRIPTION**, but we'll tackle only a few of the more common ones. Here's the syntax for the *snmpd.extend* file:

```
your-label-here DEFINITIONS ::= BEGIN

-- insert your comments here

enterprise-name  OBJECT IDENTIFIER ::= { OID-label(1) OID-label{2) 3 }
subtree-name1    OBJECT IDENTIFIER ::= { OID-label(3) 4 }
subtree-name2    OBJECT IDENTIFIER ::= { OID-label(123) 56 }

data-Identifier[8] OBJECT-TYPE
    SYNTAX Integer | Counter | Gauge | DisplayString[9]
    ACCESS read-only | read-write
    STATUS mandatory | optional | obsolete | deprecated[10]
    DESCRIPTION
        "
         Enter Your Description Here
         READ-COMMAND: /your/command/here passed1 passed2
         READ-COMMAND-TIMEOUT: timeout_in_seconds (defaults to 3)
         FILE-COMMAND: /your/file-command/here passed1 passed2
         FILE-COMMAND-FREQUENCY: frequency_in_seconds (defaults to 10)
         FILE-NAME: /your/filename/here
        "
    ::= { parent-subtree-name subidentifier }

END
```

[8] This is sometimes called a leaf node, node, object, or MIB.

[9] These are just to name a few supported datatypes.

[10] For now we will always use mandatory as our STATUS.

We can glean some style guidelines from RFC 2578. While there are many guidelines, some more useful than others, one thing stands out: case does matter. Much of ASN.1 is case sensitive. All ASN.1 keywords and macros should be in uppercase: **OBJECT-TYPE**, **SYNTAX**, **DESCRIPTION**, etc. Your data-Identifiers (i.e., object names) should start in lowercase and contain no spaces. If you have read any of the RFC MIBs or done any polling, you should have noticed that all the object names obey this convention. Try to use descriptive names and keep your names well under the 64-character limit; RFC 2578 states that anything over 32

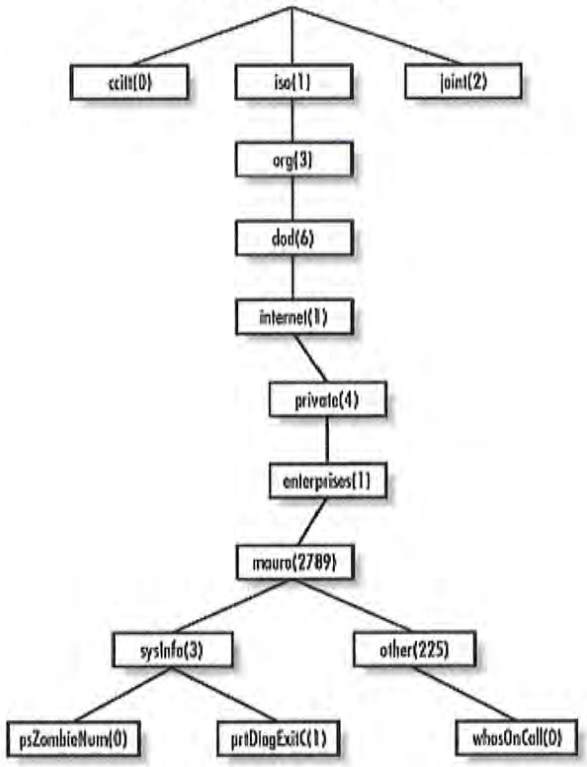
characters is not recommended. If you define an object under an existing subtree, you should use this subtree-name, or parent-name, before each new object-name you create. The ip subtree in mib-2 (RFC 1213) provides an example of good practice:

```
ip          OBJECT IDENTIFIER ::= { mib-2 4 }

ipForwarding OBJECT-TYPE
...
::= { ip 1 }

ipDefaultTTL OBJECT-TYPE
...
::= { ip 2 }
```

This file starts by defining the ip subtree. The names of objects within that subtree start with ip and use ip as the parent-subtree-name. As useful as this recommended practice is, there are times when it isn't appropriate. For example, this practice makes it difficult to move your objects to different parents while you are building a MIB file.

Here's a working *snmpd.extend* file that contains three definitions: psZombieNum, prtDiagExitC, and whosOnCall. I have placed all these objects within my own private enterprise (2789, which I have named mauro). Figure 11-2 shows this portion of my private subtree.

Figure 11-2. mauro subtree



You can now walk the tree and see what my new objects look like; my tree starts at the OID .1.3.6.1.4.1.2789, which is equivalent to .iso.org.dod.internet.private.enterprises.mauro. I can organize my own subtree any way I want, so I've split it into two branches beneath mauro: mauro.sysInfo (2789.3) will hold information about the status of the system itself (psZombieNum and prtDiagExitC), and mauro.other (2789.255) will hold additional information (whosOnCall). If you look further down, you can see the three leaf nodes we define in this file:

```
SampleExt DEFINITIONS ::= BEGIN

-- comments appear here behind the dashes

internet        OBJECT IDENTIFIER ::= { iso(1) org(3) dod(6) 1 }
enterprises     OBJECT IDENTIFIER ::= { internet(1) private(4) 1 }
mauro           OBJECT IDENTIFIER ::= { enterprises(1) 2789 }

-- Now that we have defined mauro, let's define some objects

sysInfo         OBJECT IDENTIFIER ::= { mauro 3 }
other           OBJECT IDENTIFIER ::= { mauro 255 }

psZombieNum OBJECT-TYPE
  SYNTAX  INTEGER
  ACCESS  read-only
  STATUS  mandatory
  DESCRIPTION
    "Search through ps and return the number of zombies.
    READ-COMMAND: VALUE=`ps -ef | grep -v grep | grep -c  \<defunct\>`; echo $VALUE
    "
 ::= { sysInfo 0 }

prtDiagExitC OBJECT-TYPE
    SYNTAX  INTEGER
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
        "On Solaris, prtdiag shows us system diagnostic information. The
         manpage states that if this command exits with a non-zero value,
         we have a problem. This is a great polling mechanism for some
         systems.
         READ-COMMAND: /usr/platform/`uname -m`/sbin/prtdiag > /dev/null; echo $?"
    ::= { sysInfo 1 }

whosOnCall OBJECT-TYPE
    SYNTAX  OctetString
    ACCESS  read-write
    STATUS  mandatory
    DESCRIPTION
        "This file contains the name of the person who will be on call
         today. The helpdesk uses this file. Only the helpdesk and
         managers should update this file. If you are sick or unable to
         be on call please contact your manager and/or the helpdesk.
```

```
        FILE-NAME: /opt/local/oncall/today.txt"
    ::= { other 0 }

END
```

The first two objects, psZombieNum and prtDiagExitC, both use the **READ-COMMAND** in the **DESCRIPTION**. This tells the agent to execute the named command and send any output the command produces to the NMS. By default, the program must complete within three seconds and have an exit value of 0 (zero). You can increase the timeout by adding a **READ-COMMAND-TIMEOUT**:

```
READ-COMMAND: /some/fs/somecommand.pl
READ-COMMAND-TIMEOUT: 10
```

This tells the agent to wait 10 seconds instead of 3 for a reply before killing the process and returning an error.

The last object, whosOnCall, uses a **FILE-NAME** in the **DESCRIPTION**. This tells the agent to return the first line of the file, program, script, etc. specified after **FILE-NAME**. Later we will learn how to manipulate this file.

Now that we've created a MIB file with our new definitions, we need to load the new MIB into OpenView. This step isn't strictly necessary, but it's much more convenient to work with textual names than to deal with numeric IDs. To do this, use *xnmloadmib*, discussed in Chapter 6. After we load the MIB file containing our three new objects, we should see their names in the MIB browser and be able to poll them by name.

Once you have copied the MIB file into the appropriate directory and forced the extensible agent, extsubagt, to reread its configuration (by using *kill -HUP*), try walking the new objects using OpenView's *snmpwalk* program:

```
$ snmpwalk sunserver2 -c public .1.3.6.1.4.1.2789
mauro.sysInfo.psZombieNum.0 : INTEGER: 0
mauro.sysInfo.prtDiagExitC.0 : INTEGER: 2
```

Notice anything strange about our return values? We didn't get anything for whosOnCall. Nothing was returned for this object because we haven't created the *oncall.txt* file whose contents we're trying to read. We must first create this file and insert some data into the file. There are two ways of doing this. Obviously, you can create the file with your favorite text editor. But the clever way is to use *snmpset*:

```
$ snmpset -c private sunserver2 \
.1.3.6.1.4.1.2789.255.0.0 octetstring "david jones"
mauro.Other.whosOnCall.0 : OCTET STRING- (ascii):        david jones
```

This command tells the SNMP agent to put **david jones** in the file */opt/local/oncall/today.txt*. The filename is defined by the **FILE-NAME: /opt/local/oncall/today.txt** command that we wrote in the extended MIB. The additional .0 at the end of the OID tells the agent we want the first (and only) instance of whosOnCall. (We could have used

.iso.org.dod.internet.private.enterprises.mauro.other.whosOnCal 1.0 instead of the numeric OID.) Furthermore, the *snmpset* command specifies the datatype **octetstring**, which matches the **OctetString** syntax we defined in the MIB. This datatype lets us insert string values into the file. Finally, we're allowed to set the value of this object with *snmpset* because we have **read-write** access to the object, as specified in the MIB.

If you choose to use an editor to create the file, keep in mind that anything after the first line of the file is ignored. If you want to read multiple lines you have to use a table; tables are covered in the next section.

Now let's add another object to the MIB for our extended agent. We'll use a modification of the example OpenView gives us. We'll create an object named fmailListMsgs (2) that summarizes the messages in the mail queue. This object will live in a new subtree, named fmail (4), under the private mauro subtree. So the name of our object will be mauro.fmail.fmailListMsgs or, in numeric form, .1.3.6.1.4.1.2789.4.2. First, we need to define the fmail branch under the mauro subtree. To do this, add the following line to *snmpd.extend*:

```
fmail           OBJECT IDENTIFIER ::= { mauro 4 }
```

We picked 4 for the branch number, but we could have chosen any number that doesn't conflict with our other branches (3 and 255). After we define fmail we can insert the definition for fmailListMsgs into *snmpd.extend*, placing it before the **END** statement:

```
fmailListMsgs OBJECT-TYPE
    SYNTAX DisplayString
    ACCESS read-only
    STATUS mandatory
    DESCRIPTION
        "List of messages on the mail queue.
         READ-COMMAND: /usr/lib/sendmail -bp
         READ-COMMAND-TIMEOUT: 10"
    ::= { fmail 2 }
```

When polled, fmailListMsgs runs the command *sendmail -bp*, which prints a summary of the mail queue. When all this is done, you can use your management station or a tool such as *snmpget* to read the value of mauro.fmail.fmailListMsgs and see the status of the outgoing mail queue.

11.3.1 Tables

Tables allow the agent to return multiple lines of output (or other sets of values) from the commands it executes. At its most elaborate, a table allows the agent to return something like a spreadsheet. We can retrieve this spreadsheet using snmpwalk -- a process that's significantly easier than issuing separate *get* operations to retrieve the data one value at a time. One table we've already seen is

.iso.org.dod.internet.mgmt.mib-2.interfaces.ifTable, which is defined in MIB-II and contains information about all of a device's interfaces.

Every table contains an *integer index*, which is a unique key that distinguishes the rows in the table. The index starts with 1, for the first row, and increases by one for each following row. The index is used as an instance identifier for the columns in the table; given any column, the index lets you select the data (i.e., the row) you want. Let's look at a small table, represented by the text file *animal.db*:

```
1      Tweety      Bird    Chirp   2
2      Madison     Dog     Bark    4
3      "Big Ben"   Bear    Grrr    5
```

Our goal is to make this table readable via SNMP, using OpenView's extensible agent. This file is already in the format required by the agent. Each column is delimited by whitespace; a newline marks the end of each row. Data that includes an internal space is surrounded by quotes. OpenView doesn't allow column headings in the table, but we will want to think about the names of the objects in each row. Logically, the column headings are nothing more than the names of the objects we will retrieve from the table. In other words, each row of our table consists of five objects:

animalIndex

An index that specifies the row in the table. The first row is 1, as you'd expect for SNMP tables. The SYNTAX for this object is therefore INTEGER.

animalName

The animal's name. This is a text string, so the SYNTAX of this object will be DisplayString.

animalSpecies

The animal's species (another text string, represented as a DisplayString).

animalNoise

The noise the animal makes (another DisplayString).

animalDanger

An indication of how dangerous the animal is. This is another INTEGER, whose value can be from 1 to 6. This is called an "enumerated integer"; we're allowed to assign textual mnemonics to the integer values.

At this point, we have just about everything we need to know to write the MIB that allows us to read the table. For example, we know that we want an object named animalNoise.2 to access the animalNoise object in the second row of the table; this object has the value Bark. It's easy to see how this notation can be used to locate any object in the table. Now let's write the MIB definition for the table.

```
TableExtExample DEFINITIONS ::= BEGIN

internet        OBJECT IDENTIFIER ::= { iso(1) org(3) dod(6) 1 }
enterprises     OBJECT IDENTIFIER ::= { internet(1) private(4) 1 }
mauro           OBJECT IDENTIFIER ::= { enterprises(1) 2789 }
other           OBJECT IDENTIFIER ::= { mauro 255 }

AnimalEntry ::=
    SEQUENCE {
    animalIndex INTEGER,
    animalName DisplayString,
    animalSpecies DisplayString,
    animalNoise DisplayString,
    animalDanger INTEGER
    }

animalTable OBJECT-TYPE
    SYNTAX SEQUENCE OF AnimalEntry
    ACCESS not-accessible
    STATUS mandatory
    DESCRIPTION
        "This is a table of animals that shows:
         Name
         Species
         Noise
         Danger Level
         FILE-NAME: /opt/local/animal.db"
    ::= { other 247 }

animalEntry OBJECT-TYPE
    SYNTAX AnimalEntry
    ACCESS not-accessible
    STATUS mandatory
    DESCRIPTION
        "List of animalNum"
    INDEX { animalIndex }
    ::= { animalTable 1 }

animalIndex OBJECT-TYPE
    SYNTAX INTEGER
    ACCESS read-only
    STATUS mandatory
    DESCRIPTION
        "The unique index number we will use for each row"
    ::= { animalEntry 1 }

animalName OBJECT-TYPE
    SYNTAX DisplayString
    ACCESS read-only
    STATUS mandatory
    DESCRIPTION
        "My pet name for each animal"
    ::= { animalEntry 2 }
```

```
animalSpecies OBJECT-TYPE
    SYNTAX DisplayString
    ACCESS read-only
    STATUS mandatory
    DESCRIPTION
        "The animal's species"
    ::= { animalEntry 3 }

animalNoise OBJECT-TYPE
    SYNTAX DisplayString
    ACCESS read-only
    STATUS mandatory
    DESCRIPTION
        "The noise or sound the animal makes"
    ::= { animalEntry 4 }

animalDanger OBJECT-TYPE
    SYNTAX INTEGER {
        no-Danger(1),
        can-Harm(2),
        some-Damage(3),
        will-Wound(4),
        severe-Pain(5),
        will-Kill(6)
    }
    ACCESS read-write
    STATUS mandatory
    DESCRIPTION
        "The level of danger that we may face with the particular animal"
    ::= { animalEntry 5 }

END
```

The table starts with a definition of the animalTable object, which gives us our DESCRIPTION and tells the agent where the *animal.db* file is located. The SYNTAX is SEQUENCE OF AnimalEntry. AnimalEntry (watch the case) gives us a quick view of all our columns. You can leave AnimalEntry out, but we recommend that you include it since it documents the structure of the table.

The table is actually built from animalEntry elements -- because object names are case sensitive, this object is different from AnimalEntry. animalEntry tells us what object we should use for our index or key; the object used as the key is in brackets after the INDEX keyword.

The definitions of the remaining objects are similar to the definitions we've already seen. The parent-subtree for all of these objects is animalEntry, which effectively builds a table row from each of these objects. The only object that's particularly interesting is animalDanger, which uses an extension of the INTEGER datatype. As we noted before, this object is an enumerated integer, which allows us to associate textual labels with integer values. The values you can use in an enumerated type should be a series of consecutive integers,

starting with 1.[11] For example, the animalDanger object defines six values, ranging from 1 to 6, with strings like **no-danger** associated with the values.

[11] Some SNMPv1 SMI-compliant MIB compilers will not allow an enumerated type of 0 (zero).

You can save this table definition in a file and use the *xnmloadmib* command to load it into OpenView. Once you've done that and created the *animal.db* file with a text editor, you can walk the table:

```
$ snmpwalk sunserver1 .1.3.6.1.4.1.mauro.other.animalTable
animalEntry.animalIndex.1 : INTEGER: 1
animalEntry.animalIndex.2 : INTEGER: 2
animalEntry.animalIndex.3 : INTEGER: 3
animalEntry.animalName.1 : DISPLAY STRING-(ascii): Tweety
animalEntry.animalName.2 : DISPLAY STRING-(ascii): Madison
animalEntry.animalName.3 : DISPLAY STRING-(ascii): Big Ben
animalEntry.animalSpecies.1 : DISPLAY STRING-(ascii): Bird
animalEntry.animalSpecies.2 : DISPLAY STRING-(ascii): Dog
animalEntry.animalSpecies.3 : DISPLAY STRING-(ascii): Bear
animalEntry.animalNoise.1 : DISPLAY STRING-(ascii): Chirp
animalEntry.animalNoise.2 : DISPLAY STRING-(ascii): Bark
animalEntry.animalNoise.3 : DISPLAY STRING-(ascii): Grrr
animalEntry.animalDanger.1 : INTEGER: can-Harm
animalEntry.animalDanger.2 : INTEGER: will-Wound
animalEntry.animalDanger.3 : INTEGER: severe-Pain
```

snmpwalk goes through the table a column at a time, reporting all the data in a column before proceeding to the next. This is confusing -- it would be easier if *snmpwalk* read the table a row at a time. As it is, you have to hop from line to line when you are trying to read a row; for example, to find out everything about Tweety, you need to look at every third line (all the .1 items) in the output.

Two more things are worth noticing in the *snmpwalk* output. The first set of values that *snmpwalk* reports are the index values (animalIndex). It then appends each index value to each OID to perform the rest of the walk. Second, the animalDanger output reports strings, such as **can-Harm**, rather than integers. The conversion from integers to strings takes place because we defined the animalDanger object as an enumerated integer, which associates a set of possible values with strings.

Of course, just reading a table doesn't do a whole lot of good. Let's say that we need to update this file periodically to reflect changes in the animals' behavior. The animalDanger object has an **ACCESS** of **read-write**, which allows us to set its value and update the database file using our SNMP tools. Imagine that the dog in row 2 turns very mean. We need to turn its danger level to 5 (**severe-Pain**). We could edit the file by hand, but it's easier to issue an *snmpset*:

```
$ snmpset -c private sunserver2 \
mauro.other.animalTable.animalEntry.animalDanger.2 integer "5"
mauro.other.animalTable.animalEntry.animalDanger.2 : INTEGER: severe-Pain
```

Now let's go back and verify that the variable has been updated:[12]

[12] We could already deduce that the *set* was successful when *snmpset* didn't give us an error. This example does, however, show how you can *snmpget* a single instance within a table.

```
$ snmpget sunserver2 \
mauro.other.animalTable.animalEntry.animalDanger.2
mauro.other.animalTable.animalEntry.animalDanger.2 : INTEGER: severe-Pain
```

Once the *snmpset* is complete, check the file to see how it has changed. In addition to changing the dog's danger level, it has enclosed all strings within quotes:

```
1 "Tweety" "Bird" "Chirp" 2
2 "Madison" "Dog" "Bark" 5
3 "Big Ben" "Bear" "Grrr" 5
```

There are even more possibilities for keeping the file up-to-date. For example, you could use a system program or application to edit this file. A *cron* job could kick off every hour or so and update the file. This strategy would let you generate the file using a SQL query to a database such as Oracle. You could then put the query's results in a file and poll the file with SNMP to read the results. One problem with this strategy is that you must ensure that your application and SNMP polling periods are in sync. Make sure you poll the file after Oracle has updated it, or you will be viewing old data.

An effective way to ensure that the file is up-to-date when you read it is to use **FILE-COMMAND** within the table's definition. This tells the agent to run a program that updates the table before returning any values. Let's assume that we've written a script named *get_animal_status.pl* that determines the status of the animals and updates the database accordingly. Here's how we'd integrate that script into our table definition:

```
animalTable OBJECT-TYPE
    SYNTAX   SEQUENCE OF AnimalEntry
    ACCESS   not-accessible
    STATUS   mandatory
    DESCRIPTION
        "This is a table of animals that shows:
         Name
         Species
         Noise
         Danger Level
         FILE-COMMAND: /opt/local/get_animal_status.pl
         FILE-NAME: /opt/local/animal.db"
    ::= { other 247 }
```

The command must finish within 10 seconds or the agent will kill the process and return the old values from the table. By

default, the agent runs the program specified by FILE-COMMAND only if it has not gotten a request in the last 10 seconds. For example, let's say you issue two *snmpget* commands, two seconds apart. For the first *snmpget*, the agent runs the program and returns the data from the table with any changes. The second time, the agent won't run the program to update the data -- it will return the old data, assuming that nothing has changed. This is effectively a form of caching. You can increase the amount of time the agent keeps its cache by specifying a value, in seconds, after FILE-COMMAND-FREQUENCY. For example, if you want to update the file only every 20 minutes (at most), include the following commands in your table definition:

```
FILE-COMMAND: /opt/local/get_animal_status.pl
FILE-COMMAND-FREQUENCY: 1200
FILE-NAME: /opt/local/animal.db"
```

This chapter has given you a brief introduction to three of the more popular extensible SNMP agents on the market. While a thorough treatment of every configurable option for each agent is beyond the scope of this chapter, it should help you to understand how to use extensible agents. With an extensible agent, the possibilities are almost endless.

Chapter 12. Adapting SNMP to Fit Your Environment

SNMP can make your life as a system administrator a lot easier by performing many of the tasks that you'd either have to do by hand or automate by writing some clever script. It's relatively easy to take care of most everyday system monitoring: SNMP can poll for disk-space utilization, notify you when mirrors are syncing, or record who is logging in or out of the system. The SNMP scripts in this chapter represent just a few of the things SNMP allows you to do; use them as a launching pad for your own ideas.

12.1 General Trap-Generation Program

Chapter 10 contained some scripts for collecting SNMP information using Perl, OpenView's *snmptrap* program, and some other tools. Here's how we used *snmptrap* to generate a trap giving us information about some problems with the database:

```
$ /opt/OV/bin/snmptrap -c public nms .1.3.6.1.4.1.2789.2500 "" 6 3003 "" \
.1.3.6.1.4.1.2500.3003.1 octetstringascii "Oracle" \
.1.3.6.1.4.1.2500.3003.2 octetstringascii "Backup Not Running" \
.1.3.6.1.4.1.2500.3003.3 octetstringascii "Call the DBA Now for Help"
```

The way you send a trap in Perl is a little more involved, but it's still easy to use:

```
#!/usr/local/bin/perl
# Filename: /opt/local/perl_scripts/snmptrap.pl

use SNMP_util "0.54";  # This will load the BER and SNMP_Session
```

```
snmptrap("public\@nms:162", ".1.3.6.1.4.1.2789", "sunserver1",
        6, 1247, ".1.3.6.1.4.1.2789.1247.1", "int", "2448816");
```

In this chapter, we won't look so much at how to write commands like these, but at how to use them in clever ways. We might want to include commands like these in startup scripts, or invoke them via hooks into other programs. We'll start by writing some code that records successful logins.

12.2 Who's Logging into My Machine? (I-Am-in)

When Unix users log in, the system automatically executes a profile; for users of the Bourne, Korn, or bash shells, the systemwide profile is named */etc/profile*. There's a similar file for users of csh and tcsh (*/etc/login*). We can use SNMP to record logins by adding a trap to these profiles. By itself this isn't all that interesting, because Unix already keeps a log of user logins. But let's say that you're monitoring a few dozen machines and don't want to check each machine's log. Adding a trap to the systemwide profile lets you monitor logins to all your systems from one place. It also makes your logging more secure. It's not too difficult for an intelligent user to delete the *wtmp* file that stores Unix login records. Using SNMP to do the logging stores the information on another host, over which you should have better control.[1]

> [1] Yes, a clever user could intercept and modify SNMP packets, or rewrite the shell profile, or do any number of things to defeat logging. We're not really interested in making it impossible to defeat logging; we just want to make it more difficult.

To generate the trap, invoke the external program */opt/local/mib_ programs/os/iamin* in */etc/profile* (you can call the same program within */etc/login*). Here is the code for *iamin*:

```
#!/usr/local/bin/perl
#
# Filename: /opt/local/mib_programs/os/iamin

chomp ($WHO = `/bin/who am i \| awk \{\'print \$1\'\}`);

exit 123 unless ($WHO ne '');

chomp ($WHOAMI = `/usr/ucb/whoami`);
chomp ($TTY = `/bin/tty`);
chomp ($FROM = `/bin/last \-1 $WHO \| /bin/awk \{\'print \$3\'\}`);

if ($FROM =~ /Sun|Mon|Tue|Wed|Thu|Fri|Sat/) { $FROM = "N/A"; }

# DEBUG BELOW
# print "WHO :$WHO:\n"; print "WHOAMI :$WHOAMI:\n"; print "FROM :$FROM:\n";
```

```
if ("$WHOAMI" ne "$WHO") { $WHO = "$WHO\-\>$WHOAMI"; }

# Sending a trap using Net-SNMP
#
system "/usr/local/bin/snmptrap nms public .1.3.6.1.4.1.2789.2500 '' 6 1502 ''
.1.3.6.1.4.1.2789.2500.1502.1 s \"$WHO\"
.1.3.6.1.4.1.2789.2500.1502.2 s \"$FROM\"
.1.3.6.1.4.1.2789.2500.1502.3 s \"$TTY\"";

# Sending a trap using Perl
#
#use SNMP_util "0.54";  # This will load the BER and SNMP_Session for us
#snmptrap("public\@nms:162", ".1.3.6.1.4.1.2789.2500", mylocalhostname, 6, 1502,
#".1.3.6.1.4.1.2789.2500.1502.1", "string", "$WHO",
#".1.3.6.1.4.1.2789.2500.1502.2", "string", "$FROM",
#".1.3.6.1.4.1.2789.2500.1502.3", "string", "$TTY");

# Sending a trap using OpenView's snmptrap
#
#system "/opt/OV/bin/snmptrap -c public nms .1.3.6.1.4.1.2789.2500 \"\" 6 1502 \"\"
#.1.3.6.1.4.1.2789.2500.1502.1 octetstringascii \"$WHO\"
#.1.3.6.1.4.1.2789.2500.1502.2 octetstringascii \"$FROM\"
#.1.3.6.1.4.1.2789.2500.1502.3 octetstringascii \"$TTY\"";
#

#
print "\n##############\n";
print "#   NOTICE   \# - You have been logged: :$WHO: :$FROM: :$TTY: \n"; #
print "##############\n\n";
```

This script is a bit meatier than expected because we need to weed out a number of bogus entries. For instance, many programs run within a shell and hence invoke the same shell profiles. Therefore, we have to figure out whether the profile is being invoked by a human user; if not, we quit.[2] The next step is to figure out more about the user's identity; i.e., where she is logging in from and what her real identity is -- we don't want to be confused by someone who uses *su* to switch to another identity. The third part of the program sends the trap with all the newly found information (who the user is, the host from which she is logging in, and what TTY she is on). We've included trap-generation code using the Net-SNMP utilities, the native Perl module, and OpenView's utilities. Take your pick and use the version with which you're most comfortable. The last portion of this program tells the user that she has been logged.

[2] This will also fail if the user is *su*'ing to another user. In a well-designed environment, users really shouldn't have to *su* all that often -- using *sudo* or designing appropriate groups should greatly reduce the need to *su*.

This script isn't without its problems. The user can always break out of the script before it is done, bypassing logging. You can counter this attempt by using trap(1), which responds to different signals. This forces the user to complete this program, not letting her stop in midstream. This strategy creates its own problems, since the root user doesn't have any way to bypass the check. In a sense, this is good: we want to be particularly careful about root logins. But what happens if you're trying to investigate a network failure or DNS problem? In this case, the script will hang while DNS tries to look up the host from which you're logging in. This can be very frustrating. Before implementing a script like this, look at your environment and decide which profiles you should lock.

Any of the packages for receiving traps can be used to listen for the traps generated by this program.

12.3 Throw Core

Programs frequently leave core dumps behind. A core file contains all the process information pertinent to debugging. It usually gets written when a program dies abnormally. While there are ways to limit the size of a dump or prevent core dumps entirely, there are still times when they're needed temporarily. Therefore, most Unix systems have some sort of *cron* script that automatically searches for core files and deletes them. Let's add some intelligence to these scripts to let us track what files are found, their sizes, and the names of the processes that created them.

The following Perl program is divided into four parts: it searches for a file with a given name (defaults to the name *core*), gets the file's statistics, deletes the file,[3] and then sends a trap. Most of the processing is performed natively by Perl, but we use the command *ls -l $FILENAME* to include the pertinent core file information within the SNMP trap. This command allows our operators to see information about the file in a format that's easy to recognize. We also use the *file* command, which determines a file's type and its creator. Unless you know who created the file, you won't have the chance to fix the real problem.

[3] Before you start deleting core files, you should figure out who or what is dropping them and see if the owner wants these files. In some cases this core file may be their only means of debugging.

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

# Finds and deletes core files. It sends traps upon completion and
# errors. Arguments are:
# -path directory   : search directory (and subdirectories); default /
```

```
# -lookfor filename : filename to search for; default core
# -debug value      : debug level

while ($ARGV[0] =~ /^-/)
{
    if    ($ARGV[0] eq "-path")    { shift; $PATH    = $ARGV[0]; }
    elsif ($ARGV[0] eq "-lookfor") { shift; $LOOKFOR = $ARGV[0]; }
    elsif ($ARGV[0] eq "-debug")   { shift; $DEBUG   = $ARGV[0]; }
    shift;
}


################################################################
#########################  Begin Main  ########################
################################################################

require "find.pl";     # This gives us the find function.

$LOOKFOR = "core" unless ($LOOKFOR); # If we don't have something
                                     # in $LOOKFOR, default to core

$PATH    = "/"    unless ($PATH);    # Let's use / if we don't get
                                     # one on the command line

(-d $PATH) || die "$PATH is NOT a valid dir!";    # We can search
                                                  # only valid
                                                  # directories

&find("$PATH");

################################################################
#####################  Begin SubRoutines  #####################
################################################################

sub wanted
{
    if (/^$LOOKFOR$/)
        {
            if (!(-d $name)) # Skip the directories named core
            {
               &get_stats;
               &can_file;
               &send_trap;
            }
        }
}

sub can_file
{
    print "Deleting :$_: :$name:\n" unless (!($DEBUG));
    $RES = unlink "$name";
    if ($RES != 1) { $ERROR = 1; }
}
```

```
sub get_stats
{
    chop ($STATS = `ls -l $name`);
    chop ($FILE_STATS = `/bin/file $name`);

    $STATS =~ s/\s+/ /g;
    $FILE_STATS =~ s/\s+/ /g;
}

sub send_trap
{
    if ($ERROR == 0) { $SPEC = 1535; }
    else             { $SPEC = 1536; }
    print "STATS: $STATS\n" unless (!($DEBUG));
    print "FILE_STATS: $FILE_STATS\n" unless (!($DEBUG));

# Sending a trap using Net-SNMP
#
#system "/usr/local/bin/snmptrap nms public .1.3.6.1.4.1.2789.2500 '' 6 $SPEC ''
#.1.3.6.1.4.1.2789.2500.1535.1 s \"$name\"
#.1.3.6.1.4.1.2789.2500.1535.2 s \"$STATS\"
#.1.3.6.1.4.1.2789.2500.1535.3 s \"$FILE_STATS\"";

# Sending a trap using Perl
#
use SNMP_util "0.54";  # This will load the BER and SNMP_Session for us
snmptrap("public\@nms:162", ".1.3.6.1.4.1.2789.2500", mylocalhostname, 6, $SPEC,
".1.3.6.1.4.1.2789.2500.1535.1", "string", "$name",
".1.3.6.1.4.1.2789.2500.1535.2", "string", "$STATS",
".1.3.6.1.4.1.2789.2500.1535.3", "string", "$FILE_STATS");

# Sending a trap using OpenView's snmptrap
#
#system "/opt/OV/bin/snmptrap -c public nms
#.1.3.6.1.4.1.2789.2500 \"\" 6 $SPEC \"\"
#.1.3.6.1.4.1.2789.2500.1535.1 octetstringascii \"$name\"
#.1.3.6.1.4.1.2789.2500.1535.2 octetstringascii \"$STATS\"
#.1.3.6.1.4.1.2789.2500.1535.3 octetstringascii \"$FILE_STATS\"";
}
```

The logic is simple, though it's somewhat hard to see since most of it happens implicitly. The key is the call to **find( )**, which sets up lots of things. It descends into every directory underneath the directory specified by **$PATH** and automatically sets $_ (so the **if** statement at the beginning of the **wanted()** subroutine works). Furthermore, it defines the variable **name** to be the full pathname to the current file; this allows us to test whether or not the current file is really a directory, which we wouldn't want to delete.

Therefore, we loop through all the files, looking for files with the name specified on the comand line (or named *core*, if no *-lookfor* option is specified). When we find one we store its statistics, delete the file, and send a trap to the NMS reporting the file's name and other information. We use the

variable **SPEC** to store the specific trap ID. We use two specific IDs: 1535 if the file was deleted successfully and 1536 if we tried to delete the file but couldn't. Again, we wrote the trap code to use either native Perl, Net-SNMP, or OpenView. Uncomment the version of your choice. We pack the trap with three variable bindings, which contain the name of the file, the results of *ls -l* on the file, and the results of running */bin/file*. Together, these give us a fair amount of information about the file we deleted. Note that we had to define object IDs for all three of these variables; furthermore, although we placed these object IDs under 1535, nothing prevents us from using the same objects when we send specific trap 1536.

Now we have a program to delete core files and send traps telling us about what was deleted; the next step is to tell our trap receiver what to do with these incoming traps. Let's assume that we're using OpenView. To inform it about these traps, we have to add two entries to *trapd.conf*, mapping these traps to events. Here they are:

```
EVENT foundNDelCore .1.3.6.1.4.1.2789.2500.0.1535 "Status Alarms" Warning
FORMAT Core File Found :$1: File Has Been Deleted - LS :$2: FILE :$3:
SDESC
This event is called when a server using cronjob looks for core
files and deletes them.

$1 - octetstringascii   - Name of file
$2 - octetstringascii   - ls -l listing on the file
$3 - octetstringascii   - file $name
EDESC
#
#
#
EVENT foundNNotDelCore .1.3.6.1.4.1.2789.2500.0.1536 "Status Alarms" Minor
FORMAT Core File Found :$1:
File Has Not Been Deleted For Some Reason - LS :$2: FILE :$3:
SDESC
This event is called when a server using cronjob looks for core
files and then CANNOT delete them for some reason.

$1 - octetstringascii   - Name of file
$2 - octetstringascii   - ls -l listing on the file
$3 - octetstringascii   - file $name
EDESC
#
#
#
```

For each trap, we have an **EVENT** statement specifying an event name, the trap's specific ID, the category into which the event will be sorted, and the severity. The **FORMAT** statement defines a message to be used when we receive the trap; it can be spread over several lines and can use the parameters $1, $2, etc. to refer to the variable bindings that are included in the trap.

Although it would be a good idea, we don't need to add our variable bindings to our private MIB file; *trapd.conf* contains enough information for OpenView to interpret the contents of the trap.

Here are some sample traps[4] generated by the *throwcore* script:

[4] We've removed most of the host and date/time information.

```
Core File Found :/usr/sap/HQD/DVEBMGS00/work/core: File Has Been \
Deleted - LS :-rw-rw---- 1 hqdadm sapsys 355042304 Apr 27 17:04 \
/usr/sap/HQD/DVEBMGS00/work/core: \
FILE :/usr/sap/HQD/DVEBMGS00/work/core: ELF 32-bit MSB core file \
SPARC Version 1, from 'disp+work':

Core File Found :/usr/sap/HQI/DVEBMGS10/work/core: File Has Been \
Deleted - LS :-rw-r--r-- 1 hqiadm sapsys 421499988 Apr 28 14:29 \
/usr/sap/HQI/DVEBMGS10/work/core: \
FILE :/usr/sap/HQI/DVEBMGS10/work/core: ELF 32-bit MSB core file \
SPARC Version 1, from 'disp+work':
```

Here is root's *crontab*, which runs the *throwcore* script at specific intervals. Notice that we use the *-path* switch, which allows us to check the development area every hour:

```
# Check for core files every night and every hour on special dirs
27 * * * * /opt/local/mib_programs/scripts/throwcore.pl -path /usr/sap
23 2 * * * /opt/local/mib_programs/scripts/throwcore.pl
```

12.4 Veritas Disk Check

The Veritas Volume Manager is a package that allows you to manipulate disks and their partitions. It gives you the ability to add and remove mirrors, work with RAID arrays, and resize partitions, to name a few things. Although Veritas is a specialized and expensive package that is usually found at large data centers, don't assume that you can skip this section. The point isn't to show you how to monitor Veritas, but to show you how you can provide meaningful traps using a typical status program. You should be able to extract the ideas from the script we present here and use them within your own context.

Veritas Volume Manager (*vxvm*) comes with a utility called *vxprint*. This program displays records from the Volume Manager configuration and shows the status of each of your local disks. If there is an error, such as a bad disk or broken mirror, this command will report it. A healthy *vxprint* on the rootvol (/) looks like this:

```
$ vxprint -h rootvol
Disk group: rootdg

TY NAME         ASSOC       KSTATE   LENGTH   PLOFFS   STATE    TUTILO  PUTILO
v  rootvol      root        ENABLED  922320   -        ACTIVE   -       -
pl rootvol-01   rootvol     ENABLED  922320   -        ACTIVE   -       -
```

```
sd rootdisk-B0  rootvol-01   ENABLED  1       0       -       -       Block0
sd rootdisk-02  rootvol-01   ENABLED  922319  1       -       -       -
pl rootvol-02   rootvol      ENABLED  922320  -       ACTIVE  -       -
sd disk01-01    rootvol-02   ENABLED  922320  0       -       -       -
```

The KSTATE (kernel state) and STATE columns give us a behind-the-scenes look at our disks, mirrors, etc. Without explaining the output in detail, a KSTATE of ENABLED is a good sign; a STATE of ACTIVE or - indicates that there are no problems. We can take this output and pipe it into a script that sends SNMP traps when errors are encountered. We can send different traps of an appropriate severity, based on the type of error that *vxprint* reported. Here's a script that runs *vxprint* and analyzes the results:

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

$VXPRINT_LOC    = "/usr/sbin/vxprint";
$HOSTNAME       = `/bin/uname -n`; chop $HOSTNAME;

while ($ARGV[0] =~ /^-/)
{
    if    ($ARGV[0] eq "-debug")        { shift; $DEBUG = $ARGV[0]; }
    elsif ($ARGV[0] eq "-state_active") { $SHOW_STATE_ACTIVE = 1; }
    shift;
}

#################################################################
##########################  Begin Main  ##########################
#################################################################

&get_vxprint;  # Get it, process it, and send traps if errors found!

#################################################################
#######################  Begin SubRoutines  ######################
#################################################################

sub get_vxprint
{

    open(VXPRINT,"$VXPRINT_LOC |") || die "Can't Open $VXPRINT_LOC";
    while($VXLINE=<VXPRINT>)
    {
        print $VXLINE unless ($DEBUG < 2);
        if ($VXLINE ne "\n")
        {
            &is_a_disk_group_name;
            &split_vxprint_output;

            if (($TY ne "TY")   &&
                ($TY ne "Disk") &&
                ($TY ne "dg")   &&
                ($TY ne "dm"))
            {
                if (($SHOW_STATE_ACTIVE) && ($STATE eq "ACTIVE"))
```

```
                {
                    print "ACTIVE: $VXLINE";
                }
                if (($STATE ne "ACTIVE") &&
                    ($STATE ne "DISABLED") &&
                    ($STATE ne "SYNC") &&
                    ($STATE ne "CLEAN") &&
                    ($STATE ne "SPARE") &&
                    ($STATE ne "-")      &&
                    ($STATE ne ""))
                {
                    &send_error_msgs;
                }
                elsif (($KSTATE ne "ENABLED") &&
                       ($KSTATE ne "DISABLED") &&
                       ($KSTATE ne "-")       &&
                       ($KSTATE ne ""))
                {
                    &send_error_msgs;
                }
            } # end if (($TY
        }     # end if ($VXLINE
    }         # end while($VXLINE
}             # end sub get_vxprint

sub is_a_disk_group_name
{
    if ($VXLINE =~ /^Disk\sgroup\:\s(\w+)\n/)
    {
        $DISK_GROUP = $1;
        print "Found Disk Group :$1:\n" unless (!($DEBUG));
        return 1;
    }
}

sub split_vxprint_output
{
($TY, $NAME, $ASSOC, $KSTATE,
    $LENGTH, $PLOFFS, $STATE, $TUTILO,
    $PUTILO) = split(/\s+/,$VXLINE);

    if ($DEBUG) {
        print "SPLIT: $TY $NAME $ASSOC $KSTATE ";
        print "$LENGTH $PLOFFS $STATE $TUTILO $PUTILO:\n";
            }
}

sub send_snmp_trap
{
    $SNMP_TRAP_LOC          = "/opt/OV/bin/snmptrap";
    $SNMP_COMM_NAME         = "public";
    $SNMP_TRAP_HOST         = "nms";

    $SNMP_ENTERPRISE_ID     = ".1.3.6.1.4.1.2789.2500";
```

```
    $SNMP_GEN_TRAP          = "6";
    $SNMP_SPECIFIC_TRAP     = "1000";

    chop($SNMP_TIME_STAMP        = "1" . `date +%H%S`);
    $SNMP_EVENT_IDENT_ONE   = ".1.3.6.1.4.1.2789.2500.1000.1";
    $SNMP_EVENT_VTYPE_ONE   = "octetstringascii";
    $SNMP_EVENT_VAR_ONE     = "$HOSTNAME";

    $SNMP_EVENT_IDENT_TWO   = ".1.3.6.1.4.1.2789.2500.1000.2";
    $SNMP_EVENT_VTYPE_TWO   = "octetstringascii";
    $SNMP_EVENT_VAR_TWO     = "$NAME";

    $SNMP_EVENT_IDENT_THREE = ".1.3.6.1.4.1.2789.2500.1000.3";
    $SNMP_EVENT_VTYPE_THREE = "octetstringascii";
    $SNMP_EVENT_VAR_THREE   = "$STATE";

    $SNMP_EVENT_IDENT_FOUR  = ".1.3.6.1.4.1.2789.2500.1000.4";
    $SNMP_EVENT_VTYPE_FOUR  = "octetstringascii";
    $SNMP_EVENT_VAR_FOUR    = "$DISK_GROUP";

    $SNMP_TRAP = "$SNMP_TRAP_LOC \-c $SNMP_COMM_NAME $SNMP_TRAP_HOST
    $SNMP_ENTERPRISE_ID \"\" $SNMP_GEN_TRAP $SNMP_SPECIFIC_TRAP $SNMP_TIME_STAMP
    $SNMP_EVENT_IDENT_ONE   $SNMP_EVENT_VTYPE_ONE   \"$SNMP_EVENT_VAR_ONE\"
    $SNMP_EVENT_IDENT_TWO   $SNMP_EVENT_VTYPE_TWO   \"$SNMP_EVENT_VAR_TWO\"
    $SNMP_EVENT_IDENT_THREE $SNMP_EVENT_VTYPE_THREE \"$SNMP_EVENT_VAR_THREE\"
    $SNMP_EVENT_IDENT_FOUR  $SNMP_EVENT_VTYPE_FOUR  \"$SNMP_EVENT_VAR_FOUR\"";

    # Sending a trap using Net-SNMP
    #
    #system "/usr/local/bin/snmptrap $SNMP_TRAP_HOST $SNMP_COMM_NAME
    #$SNMP_ENTERPRISE_ID '' $SNMP_GEN_TRAP $SNMP_SPECIFIC_TRAP ''
    #$SNMP_EVENT_IDENT_ONE s \"$SNMP_EVENT_VAR_ONE\"
    #$SNMP_EVENT_IDENT_TWO s \"$SNMP_EVENT_VAR_TWO\"
    #$SNMP_EVENT_IDENT_THREE s \"$SNMP_EVENT_VAR_THREE\"
    #$SNMP_EVENT_IDENT_FOUR s \"$SNMP_EVENT_VAR_FOUR\"";

    # Sending a trap using Perl
    #
    #use SNMP_util "0.54";  # This will load the BER and SNMP_Session for us
    #snmptrap("$SNMP_COMM_NAME\@$SNMP_TRAP_HOST:162", "$SNMP_ENTERPRISE_ID",
    #mylocalhostname, $SNMP_GEN_TRAP, $SNMP_SPECIFIC_TRAP,
    #"$SNMP_EVENT_IDENT_ONE", "string", "$SNMP_EVENT_VAR_ONE",
    #"$SNMP_EVENT_IDENT_TWO", "string", "$SNMP_EVENT_VAR_TWO",
    #"$SNMP_EVENT_IDENT_THREE", "string", "$SNMP_EVENT_VAR_THREE",
    #"$SNMP_EVENT_IDENT_FOUR", "string", "$SNMP_EVENT_VAR_FOUR");

    # Sending a trap using OpenView's snmptrap (using VARs from above)
    #
    if($SEND_SNMP_TRAP) {
         print "Problem Running SnmpTrap with Result ";
         print ":$SEND_SNMP_TRAP: :$SNMP_TRAP:\n";
    }

sub send_error_msgs
```

```
{
    $TY =~ s/^v/Volume/;
    $TY =~ s/^pl/Plex/;
    $TY =~ s/^sd/SubDisk/;

    print "VXfs Problem: Host:[$HOSTNAME] State:[$STATE] DiskGroup:[$DISK_GROUP]
        Type:[$TY] FileSystem:[$NAME] Assoc:[$ASSOC] Kstate:[$KSTATE]\n"
        unless (!($DEBUG));

    &send_snmp_trap;
}
```

Knowing what the output of *vxprint* should look like, we can formulate Perl statements that figure out when to generate a trap. That task makes up most of the **get_vxprint** subroutine. We also know what types of error messages will be produced. Our script tries to ignore all the information from the healthy disks and sort the error messages. For example, if the **STATE** field contains **NEEDSYNC**, the disk mirrors are probably not synchronized and the volume needs some sort of attention. The script doesn't handle this particular case explicitly, but it is caught with the default entry.

The actual mechanism for sending the trap is tied up in a large number of variables. Basically, though, we use any of the trap utilities we've discussed; the enterprise ID is .1.3.6.1.4.1.2789.2500 ; the specific trap ID is 1000 ; and we include four variable bindings, which report the hostname, the volume name, the volume's state, and the disk group.

As with the previous script, it's a simple matter to run this script periodically and watch the results on whatever network-management software you're using. It's also easy to see how you could develop similar scripts that generate reports from other status programs.

12.5 Disk-Space Checker

OpenView's agent has a fileSystemTable object that contains statistics about disk utilization and other filesystem parameters. At first glance, it looks extremely useful: you can use it to find out filesystem names, blocks free, etc. But it has some quirks, and we'll need to play a few tricks to use this table effectively. Walking fileSystemTable.fileSystemEntry.fileSystemDir (.1.3.6.1.4.1.11.2.3.1.2.2.1.10) lists the filesystems that are currently mounted:[5]

[5] We've truncated the leading .iso.org.dod.internet.private.enterprises.hp.nm.system.general to the *walk* results for space reasons.

```
[root][nms] /opt/OV/local/bin/disk_space> snmpwalk spruce \
.1.3.6.1.4.1.11.2.3.1.2.2.1.10
```

```
fileSystem.fileSystemTable.fileSystemEntry.fileSystemDir.14680064.1
: DISPLAY STRING- (ascii):  /
fileSystem.fileSystemTable.fileSystemEntry.fileSystemDir.14680067.1
: DISPLAY STRING- (ascii):  /var
fileSystem.fileSystemTable.fileSystemEntry.fileSystemDir.14680068.1
: DISPLAY STRING- (ascii):  /export
fileSystem.fileSystemTable.fileSystemEntry.fileSystemDir.14680069.1
: DISPLAY STRING- (ascii):  /opt
fileSystem.fileSystemTable.fileSystemEntry.fileSystemDir.14680070.1
: DISPLAY STRING- (ascii):  /usr
fileSystem.fileSystemTable.fileSystemEntry.fileSystemDir.41156608.1
: DISPLAY STRING- (ascii):  /proc
fileSystem.fileSystemTable.fileSystemEntry.fileSystemDir.41680896.1
: DISPLAY STRING- (ascii):  /dev/fd
fileSystem.fileSystemTable.fileSystemEntry.fileSystemDir.42991617.1
: DISPLAY STRING- (ascii):  /net
fileSystem.fileSystemTable.fileSystemEntry.fileSystemDir.42991618.1
: DISPLAY STRING- (ascii):  /home
fileSystem.fileSystemTable.fileSystemEntry.fileSystemDir.42991619.1
: DISPLAY STRING- (ascii):  /xfn
```

Let's think about how we'd write a program that checks for available disk space. At first glance, it looks like this will be easy. But this table contains a number of objects that aren't filesystems in the normal sense; */proc*, for example, provides access to the processes running on the system and doesn't represent storage. This raises problems if we start polling for free blocks: */proc* isn't going to have any free blocks, and */dev/fd*, which represents a floppy disk, will have free blocks only if a disk happens to be in the drive. You'd expect */home* to behave like a normal filesystem, but on this server it's automounted, which means that its behavior is unpredictable; if it's not in use, it might not be mounted. Therefore, if we polled for free blocks using the fileSystem.fileSystemTable.fileSystemEntry.fileSystemBavail object, the last five instances might return 0 under normal conditions. So the results we'd get from polling all the entries in the filesystem table aren't meaningful without further interpretation. At a minimum, we need to figure out which filesystems are important to us and which aren't. This is probably going to require being clever about the instance numbers.

When I discovered this problem, I noticed that all the filesystems I wanted to check happened to have instance numbers with the same leading digits; i.e., fileSystemDir.14680064.1, fileSystemDir.14680067.1, fileSystemDir.14680068.1, etc. That observation proved to be less useful than it seemed -- with time, I learned that not only do other servers have different leading instance numbers, but that on any server the instance numbers could change. Even if the instance number changes, though, the leading instance digits seem to stay the same for all disks or filesystems of the same type. For example, disk arrays might have instance numbers like

fileSystemDir.12312310.1, fileSystemDir.12312311.1, fileSystemDir.12312312.1, and so on. Your internal disks might have instance numbers like fileSystemDir.12388817.1, fileSystemDir.12388818.1, fileSystemDir.12388819.1, and so on.

So, working with the instance numbers is possible, but painful -- there is still nothing static that can be easily polled. There's no easy way to say "Give me the statistics for all the local filesystems," or even "Give me the statistics for */usr*." I was forced to write a program that would do a fair amount of instance-number processing, making guesses based on the behavior I observed. I had to use snmpwalk to figure out the instance numbers for the filesystems I cared about before doing anything more interesting. By comparing the initial digits of the instance numbers, I was able to figure out which filesystems were local, which were networked, and which were "special purpose" (like */proc*). Here's the result:

```
#!/usr/local/bin/perl
# filename: polling.pl
# options:
#    -min n    : send trap if less than n 1024-byte blocks free
#    -table f  : table of servers to watch (defaults to ./default)
#    -server s : specifies a single server to poll
#    -inst n   : number of leading instance-number digits to compare
#    -debug n  : debug level

$|++;

$SNMPWALK_LOC  = "/opt/OV/bin/snmpwalk -r 5";
$SNMPGET_LOC   = "/opt/OV/bin/snmpget";
$HOME_LOC      = "/opt/OV/local/bin/disk_space";
$LOCK_FILE_LOC = "$HOME_LOC/lock_files";
$GREP_LOC      = "/bin/grep";
$TOUCH_LOC     = "/bin/touch";
$PING_LOC      = "/usr/sbin/ping";       # Ping Location
$PING_TIMEOUT  = 7;                      # Seconds to wait for a ping

$MIB_C = ".1.3.6.1.4.1.11.2.3.1.2.2.1.6";      # fileSystemBavail
$MIB_BSIZE = ".1.3.6.1.4.1.11.2.3.1.2.2.1.7";  # fileSystemBsize
$MIB_DIR = ".1.3.6.1.4.1.11.2.3.1.2.2.1.10";   # fileSystemDir

while ($ARGV[0] =~ /^-/)
{
    if    ($ARGV[0] eq "-min")    { shift; $MIN = $ARGV[0]; }   # In 1024 blocks
    elsif ($ARGV[0] eq "-table")  { shift; $TABLE = $ARGV[0]; }
    elsif ($ARGV[0] eq "-server") { shift; $SERVER = $ARGV[0]; }
    elsif ($ARGV[0] eq "-inst")   { shift; $INST_LENGTH = $ARGV[0]; }
    elsif ($ARGV[0] eq "-debug")  { shift; $DEBUG = $ARGV[0]; }
    shift;
}

##################################################################
#########################  Begin Main  ########################
##################################################################
```

```
$ALLSERVERS  = 1 unless ($SERVER);
$INST_LENGTH = 5 unless ($INST_LENGTH);

$TABLE = "default" unless ($TABLE);

open(TABLE,"$HOME_LOC/$TABLE") || die "Can't Open File $TABLE";
while($LINE=<TABLE>)
{
    if ($LINE ne "\n")
    {
    chop $LINE;
    ($HOST,$IGNORE1,$IGNORE2,$IGNORE3) = split(/\:/,$LINE);

    if (&ping_server_bad("$HOST")) { warn "Can't Ping Server
       :$HOST:" unless (!($DEBUG)); }
    else
    {
        &find_inst;

        if ($DEBUG > 99)
        {
        print "HOST:$HOST: IGNORE1 :$IGNORE1: IGNORE2 :$IGNORE2:
              IGNORE3 :$IGNORE3:\n";
        print "Running :$SNMPWALK_LOC $HOST $MIB_C \| $GREP_LOC
              \.$GINST:\n";
        }

        $IGNORE1 = "C1ANT5MAT9CHT4HIS"
                 unless ($IGNORE1); # If we don't have anything then let's set
        $IGNORE2 = "CA2N4T6M8A1T3C5H7THIS"
                 unless ($IGNORE2); # to something that we can never match.
        $IGNORE3 = "CAN3TMA7TCH2THI6S" unless ($IGNORE3);

        if (($SERVER eq "$HOST") || ($ALLSERVERS))
        {
          open(WALKER,"$SNMPWALK_LOC $HOST $MIB_C \| $GREP_LOC
             \.$GINST |") || die "Can't Walk $HOST $MIB_C\n";
          while($WLINE=<WALKER>)
          {
              chop $WLINE;
              ($MIB,$TYPE,$VALUE) = split(/\:/,$WLINE);
              $MIB =~ s/\s+//g;
              $MIB =~ /(\d+\.\d+)$/;

              $INST = $1;

              open(SNMPGET,"$SNMPGET_LOC $HOST $MIB_DIR.$INST |");
              while($DLINE=<SNMPGET>)
              {
                  ($NULL,$NULL,$DNAME) = split(/\:/,$DLINE);
              }

              $DNAME =~ s/\s+//g;
```

```
            close SNMPGET;

            open(SNMPGET,"$SNMPGET_LOC $HOST $MIB_BSIZE.$INST |");
            while($BLINE=<SNMPGET>)
            {
            ($NULL,$NULL,$BSIZE) = split(/\:/,$BLINE);
            }

            close SNMPGET;

            $BSIZE =~ s/\s+//g;

            $LOCK_RES = &inst_found; $LOCK_RES = "\[ $LOCK_RES \]";

            print "LOCK_RES :$LOCK_RES:\n" unless ($DEBUG < 99);

            $VALUE = $VALUE * $BSIZE / 1024; # Put it in 1024 blocks

            if (($DNAME =~ /.*$IGNORE1.*/) ||
               ($DNAME =~ /.*$IGNORE2.*/) ||
               ($DNAME =~ /.*$IGNORE3.*/))
            {
               $DNAME = "$DNAME "ignored"";
            }

            else
            {
                if (($VALUE <= $MIN) && ($LOCK_RES eq "\[ 0 \]"))
                {
                   &write_lock;
                   &send_snmp_trap(0);
                }

                elsif (($VALUE > $MIN) && ($LOCK_RES eq "\[ 1 \]"))
                {
                   &remove_lock;
                   &send_snmp_trap(1);
                }
            }

            $VALUE = $VALUE / $BSIZE * 1024; # Display it as the
                                             # original block size

            write unless (!($DEBUG));

         } # end while($WLINE=<WALKER>)
      }     # end if (($SERVER eq "$HOST") || ($ALLSERVERS))
   }        # end else from if (&ping_server_bad("$HOST"))

    }            # end if ($LINE ne "\n")
}                # end while($LINE=<TABLE>)

##############################################################
```

```
#####################  Begin SubRoutines  #####################
#############################################################

format STDOUT_TOP =
Server     MountPoint       BlocksLeft    BlockSize    MIB        LockFile
---------  ---------------  ------------  -----------  ---------  ----------
.

format STDOUT =
@<<<<<<<<  @<<<<<<<<<<<<<<  @<<<<<<<<<<<  @<<<<<<<<<<  @<<<<<<<<  @<<<<<<<<<
$HOST,     $DNAME,          $VALUE,       $BSIZE,      $INST,     $LOCK_RES
.

sub inst_found
{
    if (-e "$LOCK_FILE_LOC/$HOST\.$INST") { return 1; }
    else { return 0; }
}

sub remove_lock
{
    if ($DEBUG > 99) { print "Removing Lockfile $LOCK_FILE_LOC/$HOST\.$INST\n"; }
    unlink "$LOCK_FILE_LOC/$HOST\.$INST";
}

sub write_lock
{
    if ($DEBUG > 99) { print "Writing Lockfile
         $TOUCH_LOC $LOCK_FILE_LOC/$HOST\.$INST\n"; }
    system "$TOUCH_LOC $LOCK_FILE_LOC/$HOST\.$INST";
}

#################################################################
## send_snmp_trap ##
####################
##
# This subroutine allows you to send diff traps depending on the
#  passed parm and gives you a chance to send both good and bad
#  traps.
#
# $1 - integer - This will be added to the specific event ID.
#
# If we created two traps:
#  2789.2500.0.1000 = Major
#  2789.2500.0.1001 = Good
#
# If we declare:
#  $SNMP_SPECIFIC_TRAP     = "1000";
#
# We could send the 1st by using:
#  send_snmp_trap(0);  # Here is the math (1000 + 0 = 1000)
#  And to send the second one:
#  send_snmp_trap(1);  # Here is the math (1000 + 1 = 1001)
#
```

```
# This way you could set up multiple traps with diff errors using
#  the same function for all.
#
##
#################################################################

sub send_snmp_trap
{
    $TOTAL_TRAPS_CREATED    = 2;  # Let's do some checking/reminding
                                  # here. This number should be the
                                  # total number of traps that you
                                  # created on the nms.

    $SNMP_ENTERPRISE_ID     = ".1.3.6.1.4.1.2789.2500";
    $SNMP_SPECIFIC_TRAP     = "1500";

    $PASSED_PARM            = $_[0];
    $SNMP_SPECIFIC_TRAP    += $PASSED_PARM;

    $SNMP_TRAP_LOC          = "/opt/OV/bin/snmptrap";
    $SNMP_COMM_NAME         = "public";
    $SNMP_TRAP_HOST         = "nms";

    $SNMP_GEN_TRAP          = "6";

    chop($SNMP_TIME_STAMP        = "1" . `date +%H%S`);

    $SNMP_EVENT_IDENT_ONE   = ".1.3.6.1.4.1.2789.2500.$SNMP_SPECIFIC_TRAP.1";
    $SNMP_EVENT_VTYPE_ONE   = "octetstringascii";
    $SNMP_EVENT_VAR_ONE     = "$DNAME";

    $SNMP_EVENT_IDENT_TWO   = ".1.3.6.1.4.1.2789.2500.$SNMP_SPECIFIC_TRAP.2";
    $SNMP_EVENT_VTYPE_TWO   = "integer";
    $SNMP_EVENT_VAR_TWO     = "$VALUE";

    $SNMP_EVENT_IDENT_THREE = ".1.3.6.1.4.1.2789.2500.$SNMP_SPECIFIC_TRAP.3";
    $SNMP_EVENT_VTYPE_THREE = "integer";
    $SNMP_EVENT_VAR_THREE   = "$BSIZE";

    $SNMP_EVENT_IDENT_FOUR  = ".1.3.6.1.4.1.2789.2500.$SNMP_SPECIFIC_TRAP.4";
    $SNMP_EVENT_VTYPE_FOUR  = "octetstringascii";
    $SNMP_EVENT_VAR_FOUR    = "$INST";

    $SNMP_EVENT_IDENT_FIVE  = ".1.3.6.1.4.1.2789.2500.$SNMP_SPECIFIC_TRAP.5";
    $SNMP_EVENT_VTYPE_FIVE  = "integer";
    $SNMP_EVENT_VAR_FIVE    = "$MIN";

    $SNMP_TRAP = "$SNMP_TRAP_LOC \-c $SNMP_COMM_NAME $SNMP_TRAP_HOST
      $SNMP_ENTERPRISE_ID \"$HOST\" $SNMP_GEN_TRAP $SNMP_SPECIFIC_TRAP
      $SNMP_TIME_STAMP
      $SNMP_EVENT_IDENT_ONE   $SNMP_EVENT_VTYPE_ONE   \"$SNMP_EVENT_VAR_ONE\"
      $SNMP_EVENT_IDENT_TWO   $SNMP_EVENT_VTYPE_TWO   \"$SNMP_EVENT_VAR_TWO\"
      $SNMP_EVENT_IDENT_THREE $SNMP_EVENT_VTYPE_THREE \"$SNMP_EVENT_VAR_THREE\"
      $SNMP_EVENT_IDENT_FOUR  $SNMP_EVENT_VTYPE_FOUR  \"$SNMP_EVENT_VAR_FOUR\"
```

```
    $SNMP_EVENT_IDENT_FIVE  $SNMP_EVENT_VTYPE_FIVE  \"$SNMP_EVENT_VAR_FIVE\"";

  if (!($PASSED_PARM < $TOTAL_TRAPS_CREATED))
  {
     die "ERROR SNMPTrap with a Specific Number \>
         $TOTAL_TRAPS_CREATED\nSNMP_TRAP:$SNMP_TRAP:\n";
  }

  # Sending a trap using Net-SNMP
  #
  #system "/usr/local/bin/snmptrap $SNMP_TRAP_HOST $SNMP_COMM_NAME
  #$SNMP_ENTERPRISE_ID '' $SNMP_GEN_TRAP $SNMP_SPECIFIC_TRAP ''
  #$SNMP_EVENT_IDENT_ONE s \"$SNMP_EVENT_VAR_ONE\"
  #$SNMP_EVENT_IDENT_TWO i \"$SNMP_EVENT_VAR_TWO\"
  #$SNMP_EVENT_IDENT_THREE i \"$SNMP_EVENT_VAR_THREE\"
  #$SNMP_EVENT_IDENT_FOUR s \"$SNMP_EVENT_VAR_FOUR\"";
  #$SNMP_EVENT_IDENT_FIVE i \"$SNMP_EVENT_VAR_FIVE\"";

  # Sending a trap using Perl
  #
  #use SNMP_util "0.54";  # This will load the BER and SNMP_Session for us
  #snmptrap("$SNMP_COMM_NAME\@$SNMP_TRAP_HOST:162", "$SNMP_ENTERPRISE_ID",
  #mylocalhostname, $SNMP_GEN_TRAP, $SNMP_SPECIFIC_TRAP,
  #"$SNMP_EVENT_IDENT_ONE", "string", "$SNMP_EVENT_VAR_ONE",
  #"$SNMP_EVENT_IDENT_TWO", "int", "$SNMP_EVENT_VAR_TWO",
  #"$SNMP_EVENT_IDENT_THREE", "int", "$SNMP_EVENT_VAR_THREE",
  #"$SNMP_EVENT_IDENT_FOUR", "string", "$SNMP_EVENT_VAR_FOUR",
  #"$SNMP_EVENT_IDENT_FIVE", "int", "$SNMP_EVENT_VAR_FIVE");

  # Sending a trap using OpenView's snmptrap (using VARs from above)
  #
  if($SEND_SNMP_TRAP) {
      print "ERROR Running SnmpTrap Result ";
      print ":$SEND_SNMP_TRAP: :$SNMP_TRAP:\n"
  }

sub find_inst
{
  open(SNMPWALK2,"$SNMPWALK_LOC $HOST $MIB_DIR |") ||
                              die "Can't Find Inst for $HOST\n";
  while($DLINE=<SNMPWALK2>)
  {
    chomp $DLINE;
    ($DIRTY_INST,$NULL,$DIRTY_NAME) = split(/\:/,$DLINE);
    $DIRTY_NAME =~ s/\s+//g;  # Lose the whitespace, folks!
    print "DIRTY_INST :$DIRTY_INST:\nDIRTY_NAME :$DIRTY_NAME:\n"
                          unless (!($DEBUG>99));
      if ($DIRTY_NAME eq "/")
      {
          $DIRTY_INST =~ /fileSystemDir\.(\d*)\.1/;
          $GINST = $1;
          $LENGTH = (length($GINST) - $INST_LENGTH);
          while ($LENGTH--) { chop $GINST; }
          close SNMPWALK;
```

```
            print "Found Inst DIRTY_INST :$DIRTY_INST: DIRTY_NAME\
                :$DIRTY_NAME: GINST :$GINST:\n"
                        unless (!($DEBUG > 99));
            return 0;
        }
    }

    close SNMPWALK2;
    die "Can't Find Inst for HOST :$HOST:";
}

sub ping_server_bad
{
    local $SERVER  = $_[0];
    $RES = system "$PING_LOC $SERVER $PING_TIMEOUT \> /dev/null";
    print "Res from Ping :$RES: \- :$PING_LOC $SERVER:\n"
                                    unless (!($DEBUG));
    return $RES;
}
```

The script contains a handful of useful features:

- We use an external ASCII file for a list of servers to poll. We specify the file by using the switch *-table FILENAME*. If no *-table* switch is given, the file named *default* in the current directory is used.
- We can specify a single server name (which must appear in the file above) to poll using the switch *-server SERVER_NAME*.
- We can ignore up to three filesystems per server. For example, we might want to ignore filesystems that are being used for software development.
- The script polls only servers that respond to a *ping*. We don't want to get filesystem traps from a server that is down or not on the network.
- We can set the minimum threshold for each list of servers in 1024-byte blocks using the *-min blocks* option.
- The script sends a trap when a server's threshold has been met and sends another trap when the state goes back to normal.
- We use lockfiles to prevent the server from sending out too many redundant traps.[6] When a threshold has been met, a file named *hostname.instance* is created. We send a trap only if the lockfile doesn't exist. When the space frees up, we delete the lockfile, allowing us to generate a trap the next time free storage falls below the threshold.

[6] There have been a few times that we have missed the fact that a system has filled up because a trap was lost during transmission. Using *cron*, we frequently delete everything in the *lock* directory. This resubmits the entries, if any, at that time.

- We can set the number of leading instance digits used to grab the appropriate filesystem with the *-inst* switch. Unfortunately, the number of instance digits you can safely use to isolate a local filesystem varies from installation to installation. The default is five, but a lower value may be appropriate.
- The script displays a useful table when we invoke it with the *-debug* flag.

The script starts by reading the table of servers in which we're interested. It *ping*s the servers and ignores those that don't respond. It then calls the subroutine **find_inst**, which incorporates most of the instance-number logic. This subroutine walks the filesystem table to find a list of all the filesystems and their instance numbers. It extracts the entry for the root filesystem (/), which we know exists, and which we assume is a local disk. (We can't assume that the root filesystem will be listed first; we do assume that you won't use a script like this to monitor diskless workstations). We then store the first **INST_LENGTH** digits of the instance number in the variable **GINST**, and return.

Back in the main program, we ask for the number of blocks available for each filesystem; we compare the instance number to **GINST**, which selects the local filesystems (i.e., the filesystems with an instance number whose initial digits match the instance number for /). We then ask for the total number of blocks, which allows us to compare the space available against our threshholds. If the value is less then our minimum we send one of the two enterprise-specific traps we've defined for this program, 1500, which indicates that the filesystem's free space is below the threshold. If the free space has returned to a safe level we send trap 1501, which is an "out of danger" notification. Some additional logic uses a lockfile to prevent the script from bombarding the NMS with repeated notifications; we send at most one warning a day and send an "out of danger" only if we've previously sent a warning. In either case, we stuff the trap with useful information: a number of variable bindings specifying the filesystem, the available space, its total capacity, its instance number, and the threshold we've set. Later, we'll see how to map these traps into OpenView categories.

Let's put the program to work by creating a table called *default* that lists the servers we are interested in watching:

```
db_serv0
db_serv1
db_serv2
```

Now we can run the script with the *-debug* option to show us a table of the results. The following command asks for all filesystems on the server db_serv0 with fewer than 50,000 blocks (50 MB) free:

```
$ /opt/OV/local/bin/disk_space/polling.pl -min 50000 -server db_serv0 -debug 1
Res from Ping :0: - :/usr/sbin/ping db_serv0:
Server      MountPoint        BlocksLeft  BlockSize  MIB            LockFile
----------  ----------------  ----------  ---------  -------------- --------
db_serv0    /                 207766      1024       38010880.1     [ 0 ]
db_serv0    /usr              334091      1024       38010886.1     [ 0 ]
db_serv0    /opt              937538      1024       38010887.1     [ 0 ]
db_serv0    /var              414964      1024       38010888.1     [ 0 ]
db_serv0    /db1              324954      1024       38010889.1     [ 0 ]
```

Notice that we didn't need to specify a table explicitly; because we omitted the *-table* option, the *polling.pl* script used the default file we put in the current directory. The *-server* switch let us limit the test to the server named db_serv0; if we had omitted this option the script would have checked all servers within the default table. If the free space on any of the filesystems falls under 50,000 1024-byte blocks, the program sends a trap and writes a lockfile with the instance number.

Because SNMP traps use UDP, they are unreliable. This means that some traps may never reach their destination. This could spell disaster -- in our situation, we're sending traps to notify a manager that a filesystem is full. We don't want those traps to disappear, especially since we've designed our program so that it doesn't send duplicate notifications. One workaround is to have *cron* delete some or all of the files in the *lock* directory. We like to delete everything in the *lock* directory every hour; this means that we'll get a notification every hour until some free storage appears in the filesystem. Another plausible policy is to delete only the production-server lockfiles. With this policy, we'll get hourly notification about filesystem capacity problems on the server we care about most; on other machines (e.g., development machines, test machines), we will get only a single notification.

Let's say that the filesystem /db1 is a test system and we don't care if it fills up. We can ignore this filesystem by specifying it in our table. We can list up to three filesystems we would like to ignore after the server name (which must be followed by a ":"):

```
db_serv0:db1
```

Running the *polling.pl* script again gives these results:

```
$ /opt/OV/local/bin/disk_space/polling.pl -min 50000 -server db_serv0 -debug 1
Res from Ping :0: - :/usr/sbin/ping db_serv0:
Server      MountPoint        BlocksLeft  BlockSize  MIB            LockFile
----------  ----------------  ----------  ---------  -------------- --------
db_serv0    /                 207766      1024       38010880.1     [ 0 ]
db_serv0    /usr              334091      1024       38010886.1     [ 0 ]
db_serv0    /opt              937538      1024       38010887.1     [ 0 ]
db_serv0    /var              414964      1024       38010888.1     [ 0 ]
db_serv0    /db1 (ignored)    324954      1024       38010889.1     [ 0 ]
```

When the /db1 filesystem drops below the minimum disk space, the script will not send any traps or create any lockfiles.

Now let's go beyond experimentation. The following *crontab* entries run our program twice every hour:

```
4,34 * * * * /opt/OV/bin/polling.pl -min 50000
5,35 * * * * /opt/OV/bin/polling.pl -min 17000 -table stocks_table
7,37 * * * * /opt/OV/bin/polling.pl -min 25000 -table bonds_table -inst 3
```

Next we need to define how the traps *polling.pl* generates should be handled when they arrive at the NMS. Here's the entry in OpenView's *trapd.conf* file that shows how to handle these traps:

```
EVENT DiskSpaceLow .1.3.6.1.4.1.2789.2500.0.1500 "Threshold Alarms" Major
FORMAT Disk Space For FileSystem :$1: Is Low With :$2:
1024 Blocks Left - Current FS Block Size :$3: - Min Threshold
:$5: - Inst :$4:
SDESC
$1 - octetstringascii   - FileSystem
$2 - integer            - Current Size
$3 - integer            - Block Size
$4 - octetstringascii   - INST
$5 - integer            - Min Threshold Size
EDESC
#
#
#
EVENT DiskSpaceNormal .1.3.6.1.4.1.2789.2500.0.1501 "Threshold Alarms" Normal
FORMAT Disk Space For FileSystem :$1: Is Normal With :$2:
1024 Blocks Left - Current FS Block Size :$3: - Min Threshold
:$5: - Inst :$4:
SDESC
$1 - octetstringascii   - FileSystem
$2 - integer            - Current Size
$3 - integer            - Block Size
$4 - octetstringascii   - INST
$5 - integer            - Min Threshold size
EDESC
```

These entries define two OpenView events: a DiskSpaceLow event that is used when a filesystem's capacity is below the threshold, and a DiskSpaceNormal event. We place both of these in the Threshold Alarms category; the low disk space event has a severity of Major, while the "normal" event has a severity of Normal. If you're using some other package to listen for traps, you'll have to configure it accordingly.

12.6 Port Monitor

Most TCP/IP services use static ports to listen for incoming requests. Monitoring these ports allows you to see whether particular servers or services are responding or not. For example, you can tell whether your mail server is alive by periodically poking port 25, which is the port on which an SMTP

server listens for requests. Some other ports to monitor are FTP (23), HTTP (80) and POP3 (110).[7] A freely available program called netcat can connect to and interact with a specific port on any device. We can write a wrapper for this program to watch a given port or service; if something happens outside of its normal operation, then we can send a trap. In this section, we'll develop a wrapper that checks the SMTP port (25) on our mail server. The program is very simple, but the results are outstanding!

[7] Check your *services* file for a listing of port numbers and their corresponding services. On Unix systems, this file is usually in the directory */etc*; on Windows it is usually in a directory such as *C:\WINNT \System32\drivers\etc*, though its location may vary depending on the version of Windows you are using.

Before we start to write the program, let's establish what we want to do. Telnet to port 25 of your SMTP server. Once you're connected, you can issue the command *HELO mydomain.com*. This should give you a response of 250. After you get a response from the mail server, issue the *QUIT* command, which tells the server you are done. Your session should look something like this:

```
$ telnet mail.ora.com 25
220 smtp.oreilly.com ESMTP O'Reilly & Associates Sendmail 8.11.2 ready
HELO mydomain.com
250 OK
QUIT
221 closing connection
```

The netcat program needs to know what commands you want to send to the port you are monitoring. We will be sending only two commands to our mail server, so we'll create a file called *input.txt* that looks like this:

```
HELO mydomain.com
QUIT
```

Next, we should test this file and see what output we get from the server. The actual netcat executable is named nc; to test the file, run it like this:

```
$ /opt/OV/local/bin/netcat/nc -i 1 mailserver 25 < input.txt
```

This command produces the same results as the telnet session. You won't see the commands in your *input.txt* file echoed, but you should see the server's responses. Once you have verified that netcat works and gives the same response each time, save a copy of its output to the file *mail_good*. This file will be used to determine what a normal response from your mail server looks like. You can save the output to a file with the following command:

```
$ /opt/OV/local/bin/netcat/nc -i 1 mailserver 25 < input.txt > mail_good
```

An alternate approach is to search for the line numbered 250 in the mail server's output. This code indicates that the server is up and running, though not necessarily processing mail correctly. In any case, searching for 250 shields you from variations in the server's response to your connection.

Here's a script called *mail_poller.pl* that automates the process. Edit the appropriate lines in this script to reflect your local environment. Once you have customized the script, you should be ready to go. There are no command-line arguments. The script generates an output file called *mail_status* that contains a 0 (zero) if the server is okay (i.e., if the output of netcat matches $GOOD_FILE); any number other than 0 indicates that an error has occurred:

```
#!/usr/local/bin/perl
# filename: mail_poller.pl

$HOME_LOC     = "/opt/OV/local/bin/netcat";
$NC_LOC       = "/opt/netcat/nc";
$DIFF_LOC     = "/bin/diff";
$ECHO_LOC     = "/bin/echo";

$MAIL_SERVER  = "mail.exampledomain.com";
$MAIL_PORT    =  25;
$INPUT_FILE   = "$HOME_LOC\/input.txt";
$GOOD_FILE    = "$HOME_LOC\/mail_good";
$CURRENT_FILE = "$HOME_LOC\/mail_current";
$EXIT_FILE    = "$HOME_LOC\/mail_status";

$DEBUG = 0;

print "$NC_LOC -i 1 -w 3 $MAIL_SERVER $MAIL_PORT
    \< $INPUT_FILE \> $CURRENT_FILE\n" unless (!($DEBUG));

$NETCAT_RES = system "$NC_LOC -i 1 -w 3 $MAIL_SERVER $MAIL_PORT
    \< $INPUT_FILE \> $CURRENT_FILE";
$NETCAT_RES = $NETCAT_RES / 256;

if ($NETCAT_RES)
{
    # We had a problem with netcat... maybe a timeout?
    system "$ECHO_LOC $NETCAT_RES > $EXIT_FILE";
    &cleanup;
}

$DIFF_RES = system "$DIFF_LOC $GOOD_FILE $CURRENT_FILE";
$DIFF_RES = $DIFF_RES / 256;

if ($DIFF_RES)
{
    # looks like things are different!
    system "$ECHO_LOC $DIFF_RES > $EXIT_FILE";
    &cleanup;
}
```

```
else
{
    # All systems go!
    system "$ECHO_LOC 0 > $EXIT_FILE";
    &cleanup;
}

sub cleanup
{
   unlink "$CURRENT_FILE";
   exit 0;
}
```

After you run the program, review the results in *mail_status*. If you can, try shutting down the mail server and running the script again. Your file should now contain a nonzero error status.

Once you have made sure the script works in your environment, you can insert an entry in *crontab* to execute this program at whatever interval you would like. In our environment, we use a 10-minute interval:

```
# Check the mail server and create a file that we can poll via OpenView
1,11,21,31,41,51 * * * * /opt/OV/local/bin/netcat/mail_poller.pl
```

Notice we staggered the polling so that we don't check on the hour, half hour, or quarter hour. Once *cron* has started updating *mail_status* regularly, you can use tools such as the extensible OpenView agent to check the file's contents. You can configure the agent to poll the file regularly and send the results to your management console. The entry in my */etc/SnmpAgent.d/snmpd.extend* looks like this:

```
serviceInfo     OBJECT IDENTIFIER ::= { mauro 5 }

-- BEGIN - serviceInfo
--

serMailPort  OBJECT-TYPE
    SYNTAX  INTEGER
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
        "This file is updated via crontab. It uses netcat to check the
         port and push a value into this file.
         FILE-NAME: /opt/OV/local/bin/netcat/mail_status"
    ::= { serviceInfo 0 }
```

We discuss the syntax of this file in Chapter 11. Basically, this entry just defines a MIB object in the serviceInfo tree, which is node 5 under my private-enterprise tree. In other words, this object's OID is mauro.serviceInfo.serMailPort (2789.5.0). The object can be read by any program that can issue an SNMP *get* operation. The **DESCRIPTION**, as we saw in Chapter 11, specifies a filename from which the agent will read an integer value to use as the value of this object. This program

can easily be modified to monitor any port on any number of machines. If you're ambitious, you might want to think about turning the serMailPort object into an array that reports the status of all your mail servers.

Our goal in this chapter hasn't been to provide you with scripts you can immediately place in your environment. More to the point, we have wanted to show you what's possible, and get you thinking about how you might be able to write scripts that provide elaborate custom monitoring features. If you're thinking creatively about what you can do with SNMP, we've succeeded.

Chapter 13. MRTG

The *Multi Router Traffic Grapher* (MRTG) is a freely available and fully configurable trend-analysis tool that's easy to configure and use. It's a surprisingly small, lightweight package because it doesn't implement a heavyweight user interface. Instead, it generates graphs in the form of GIF or PNG images; these graphs are embedded in standard HTML pages. Therefore, you can view MRTG's output using any graphical web browser and even make its reports visible across your network by using a web server.

Although MRTG is best at displaying usage graphs for router interfaces, it can be configured to graph things like memory usage, load average, and disk usage on server equipment. MRTG is particularly useful for determining when something "peaks out" for an extended period of time, which indicates that you have a capacity problem and need to upgrade. For example, you might find that your T1 interface is maxed out during your peak business hours and you need to upgrade to a bigger circuit, or you might find that you need to add more memory to a server. Likewise, MRTG may let you know that your network connections are operating at a fraction of the available bandwidth and that you can therefore eliminate a few T1 circuits and reduce your telecommunications costs.

Many sites that use MRTG use its default graphing capabilities for capacity planning and provisioning. MRTG doesn't provide the fine-grained statistical tools you need to calculate baseline information or project when your network will need to be upgraded. However, it can be a very useful tool for businesses that don't have the resources necessary to purchase a full-fledged trend-analysis package. Baselines and projections are invaluable, but MRTG's graphs can give you similar behavior at a glance; your eyes are very good at spotting typical behavior and trends, even if they can't give you the statistical analysis that your management might like.

MRTG has many options that allow you to customize how it operates. It is beyond the scope of this chapter to discuss every option; instead, we will discuss how to install MRTG and use its default graphing capabilities. We'll also outline how you can configure MRTG to gather system information from a server.

It's important to understand that MRTG is not an NMS solution. Although its graphing capabilities make it look superficially like an NMS, it's really a simple polling engine that's very clever about the output it generates. It performs the same *get* functions that an NMS would, but its job isn't problem detection and resolution. It doesn't have a facility for generating alarms or processing traps, nor does it have the ability to set objects. It's simply designed to provide a graphical view of how your network is performing. If you're interested in an open source NMS package, you should investigate Bluebird (http://www.opennms.org).

13.1 Using MRTG

Before using MRTG, you have to download and install the software. The primary MRTG web site is http://www.mrtg.org. The download link takes you to a directory maintained by MRTG's inventor and primary developer, Tobias Oetiker (http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/pub/). This directory contains some older MRTG releases, as well as the current one. We downloaded the file *mrtg-2.9.10.tar.gz* (the Unix version) from the list. We will focus on that version in this chapter.

MRTG requires four third-party packages in order to run: Perl Version 5.004_5 (at least), and the gd, libpng, and zlib libraries. MRTG comes with a Perl-based implementation of SNMP, so you don't have to worry about getting and installing any SNMP libraries. You can determine what version of Perl you have (and whether it's installed) by typing the command *perl -v*. This may or may not spit out a bunch of information. If it does, the first line will be the version of Perl you have installed. If you get some sort of "command not found" error, Perl may not be installed. In any event, go to http://www.perl.com to get the latest version of Perl.

The gd library is used to generate the GIF images that MRTG displays. You can download it from http://www.boutell.com/gd/. The other two packages, libpng and zlib, are also used for various aspects of graphic image creation. They are available from http://www.libpng.org/pub/png/.

Once you have ensured that Perl, gd, libpng, and zlib are installed on your machine, download and unpack the Unix version of MRTG with the following commands:

```
[root][linuxserver] > cd /usr/local
[root][linuxserver] > tar -zxvf mrtg-2.9.10.tar.gz
```

Once it's unpacked, *cd* into the directory it created (which should be *mrtg-2.9.10*) and read the installation hints from the *README* file. To build MRTG, you execute three commands:

```
[root][linuxserver] ~/mrtg-2.9.10> ./configure
[root][linuxserver] ~/mrtg-2.9.10> make
[root][linuxserver] ~/mrtg-2.9.10> make install
```

All three of these commands produce a lot of output, which we have omitted. The *configure* command inspects your system for tools it needs to build MRTG. It will tell you which items are missing and where to go to get them. Running *make* builds MRTG, but don't bother running this if the *configure* command failed; MRTG will not build unless everything has been installed and configured properly. Finally, *make install* installs MRTG and its associated files in the appropriate places. Again, don't bother running *make install* if the previous *make* command terminated with errors. The default location for the MRTG executables is */usr/local/mrtg-2/bin*. You may want to add this directory to your search path.

Once you've built MRTG, you need to decide where to put the graphs it generates. Since MRTG's graphs are designed to be viewed by a web browser, they're often stored in a directory that's visible to a web server. However, it really doesn't matter where they go. What's more important is who you want to view the graphs. You probably don't want the world to see your network statistics. On a small network, you can place the graphs in a directory that is out of view of the web server and then use a web browser to view the HTML reports in the local filesystem. In a larger network, other people (e.g., other network staff or management) may need to access the reports; to allow access without publishing your network statistics to the rest of the world, you may want to set up some kind of a secure web server. At any rate, the next set of commands you'll want to execute is something like this:

```
[root][linuxserver] ~/mrtg-2.9.10> mkdir /mrtg/images
[root][linuxserver] ~/mrtg-2.9.10> cp ./images/mrtg*.gif /mrtg/images/
```

The first command creates a directory for storing the graphs MRTG creates. The second command copies some MRTG images into the newly created directory for later use in HTML files. For the remainder of this chapter, we will assume that graphs are stored in */mrtg/images*.

You're now ready to set up your first device to poll, which is called a *target* in MRTG. MRTG uses a configuration file to tell it what devices to poll, what options to apply to the creation of the graphs it will generate, etc. The syntax of the configuration file is complex, but MRTG provides a tool called *cfgmaker* to help you build it. You'll probably need to edit the file by hand, but it's much easier to start with a working template. Here's how to execute *cfgmaker*:

```
[root][linuxserver] ~/mrtg-2.9.10> setenv PATH /usr/local/mrtg-2/bin:$PATH
```

```
[root][linuxserver] ~/mrtg-2.9.10> cfgmaker --global 'WorkDir: /mrtg/images' \
--output /mrtg/run/mrtg.cfg public@router
```

The first argument to *cfgmaker* sets the **WorkDir** variable in the configuration file. This tells MRTG where to store any data it gathers from the devices it's going to poll. The second argument specifies where we want *cfgmaker* 's output sent; in this case it's */mrtg/run/mrtg.cfg*. The last argument specifies the device we want to poll and the community string to use when polling that device; its format is community_string@device.

The output from *cfgmaker* is a mix of commands and HTML. It performs *get-next* commands on the device you specified on the command line, in order to get an idea of how many interfaces your device has, which ones are up, which are down, etc. It walks the iso.org.dod.internet.mgmt.mib-2.interfaces (1.3.6.1.2.1.2) tree to discover the total number of interfaces in this table. It then creates logical entries that represent a list of devices to poll, except the list of devices is actually one device with each interface number specified as a target. For example, Ethernet0 is in the fourth row of the interfaces table on our Cisco router, so *cfgmaker* created a **Target** entry called cisco.4. If this interface occupied the second row in the interfaces table, the **Target** entry would be called cisco.2.

Here's a shortened version of our *mrtg.cfg* file:

```
WorkDir: /mrtg/images/

Target[cisco.4]: 4:public@cisco
MaxBytes[cisco.4]: 1250000
Title[cisco.4]: cisco (cisco): Ethernet0
PageTop[cisco.4]: <H1>Traffic Analysis for Ethernet0
 </H1>
 <TABLE>
   <TR><TD>System:</TD><TD>cisco in Atlanta, Ga</TD></TR>
   <TR><TD>Maintainer:</TD><TD></TD></TR>
   <TR><TD>Interface:</TD><TD>Ethernet0 (4)</TD></TR>
   <TR><TD>IP:</TD><TD>cisco (  )</TD></TR>
   <TR><TD>Max Speed:</TD>
       <TD>1250.0 kBytes/s (ethernetCsmacd)</TD></TR>
  </TABLE>
```

It's worth learning a bit about the format of the configuration file. Comment lines begin with #; in a real configuration file, you'll see many of them. Most of the lines in the file are either commands or snippets of HTML that will be used in MRTG's output files. MRTG commands take the form of command[key]: options. For example, the command for the third line is **Target**, the key is **cisco.4**, and the options are **4:public@cisco**. The key is an identifying string that groups entries in the configuration file and provides a base filename for MRTG to use when generating graphs and HTML files. At a complex site, MRTG might be used to monitor dozens of pieces of equipment, with hundreds of interfaces; the key keeps the configuration file in some

semblance of order. The options provide the actual parameters to the command.

This should help you understand the configuration file. The first line specifies the working directory in which MRTG will place its graphs and HTML files. This is a global command, so no key is needed. The working directory is typically somewhere under a web server tree, so that MRTG's reports can be visible from a web browser. We've set ours to **/mrtg/images/**. The third line (**Target**) tells MRTG which device it should poll. The format for this option is interface:community_string@device, or in our case **4:public@cisco**. The device is specified by its hostname or IP address; we already know about community strings. Since MRTG is only a data-collection tool, the read-only community string will suffice. **Interface** specifies which interface on the device to poll, according to the device's ifTable. In this case, we're polling interface 4 in the ifTable.

The **MaxBytes** line sets up the maximum value for the parameters MRTG is going to read from this interface. By default, MRTG reads ifInOctets and ifOutOctets. It tries to pick a reasonable maximum value depending on the interface's type, which it should be able to read from the device itself. Since this is an Ethernet interface, MRTG sets **MaxBytes** to **1250000**. The **Title** specifies the title for the HTML page generated for the graph. Finally, **PageTop** and the following lines tell MRTG what kind of information to place at the top of the HTML page containing the usage graphs. The command contains actual HTML code, which was generated by *cfgmaker*.

Altogether, this entry tells MRTG to poll for the default objects (ifInOctets and ifOutOctets) on entry 4 in the interface table for the device cisco. Therefore, MRTG will issue *get* commands for the OIDs .1.3.6.1.2.1.2.2.1.10.4 (iso.org.dod.internet.mgmt.mib-2.interfaces.ifTable.ifEntry.ifInOctets.4) and .1.3.6.1.2.1.2.2.1.16.4 (iso.org.dod.internet.mgmt.mib-2.interfaces.ifTable.ifEntry.ifOutOctets.4). By default, MRTG will generate the following graphs:

- Daily graph with 5-minute averages
- Weekly graph with 30-minute averages
- Monthly graph with 2-hour averages
- Yearly graph with 1-day averages

Once you've finished, try running MRTG by hand to see if there are any problems with the configuration script:

```
[root][linuxserver] ~/mrtg-2.9.10> mrtg /mrtg/run/mrtg.cfg
```

If MRTG has no problems with your configuration file, it will run with no configuration-file errors. If it does have problems, it will give you a fairly verbose description of the

problem. The first time you run MRTG, it will complain about not being able to find any log files. If you run MRTG three times you'll see messages similar to these:

```
[root][linuxserver] ~/mrtg-2.9.10> mrtg /mrtg/run/mrtg.cfg
Rateup WARNING: /mrtg/run//rateup could not read the primary log file for cisco.4
Rateup WARNING: /mrtg/run//rateup The backup log file for cisco.4 was invalid as
well
Rateup WARNING: /mrtg/run//rateup Can't remove cisco.4.old updating log file
Rateup WARNING: /mrtg/run//rateup Can't rename cisco.4.log to cisco.4.old
updating log file

[root][linuxserver] ~/mrtg-2.9.10> mrtg /mrtg/run/mrtg.cfg
Rateup WARNING: /mrtg/run//rateup Can't remove cisco.4.old updating log file

[root][linuxserver] ~/mrtg-2.9.10> mrtg /mrtg/run/mrtg.cfg
[root][linuxserver] ~/mrtg-2.9.10>
```

As you can see, the first time we ran the program it spat out some errors. The second run produced only one error, and the last time it ran with no errors. These errors are normal when you run MRTG for the first time; don't worry about them.

The next step is to make sure MRTG runs every five minutes. There's no need for MRTG to be run by root; any user will do. Add a line like the following to the *crontab* entry for the appropriate user:

```
*/5 * * * * /usr/local/mrtg-2/bin/mrtg /mrtg/run/mrtg.cfg
```

This runs MRTG every five minutes of every day. Note that the `*/5` notation is Linux-specific; on other Unix systems you'll have to specify the times explicitly `(0,5,10,15,20,25,30,35,40,45,50,55)`. If your network is fairly large, you might run into problems if MRTG does not finish all its polling duties before the next polling cycle starts. If this is the case, setting a five-minute poll interval may not be a good idea. You may have to experiment to determine a good interval for your environment.

13.2 Viewing Graphs

Once you've generated some graphs, you will want to look at them to see the results. To make it easier to view the graphs, MRTG comes with an *indexmaker* script that generates HTML index pages. Here's how to run *indexmaker* for a typical set of graphs:

```
[root][linuxserver] ~/mrtg-2.9.10> indexmaker --title "Cisco to Internet" \
--filter name=~'cisco' --output /mrtg/images/cisco.html /mrtg/run/mrtg.cfg
```

This command creates one index page with the five-minute average graph for each target you've specified in your *mrtg.cfg* file. Keep in mind that the target is the interface from which you're gathering data. If you have four targets for your router, there will be four graphs in the index file, all pointing to the daily, weekly, monthly, and yearly summary graphs for that target. The *--title* option tells *indexmaker*

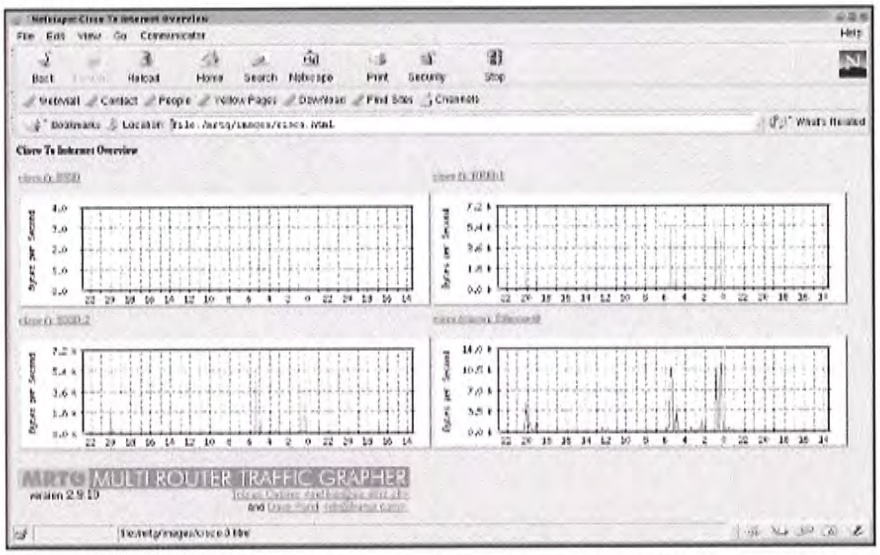
what title to use for the index file. *--filter name=~cisco* allows you to select some of the targets in the *mrtg.cfg* file by using a regular expression: we told *indexmaker* to find all targets that include the string cisco. The *--output* option is the name of the index file. The final argument on the command line is the full path to the configuration file. Table 13-1 gives a synopsis of these options as well as some other useful options to *indexmaker*.

Table 13-1. Command-Line Options to indexmaker

Option	Description
--title	Specify a title for the HTML page.
--filter	Specify the regular expression that will be used to find a specific target from the *mrtg.cfg* file. These matched targets are used to create the HTML report files.
--output	Indicate the full pathname for the HTML file that is to be generated. The default is standard output.
--sort	Sort how the graphs show up on the index page.
--columns	Arrange the graphs on the index page by x columns. The default is 2.
--width	Set the width of the graphs. This is not set by default.
--height	Set the height of the graphs. This is not set by default.
--show	Pick which graph to show on the index page. The default is day. Other options include week, month, year, and none.

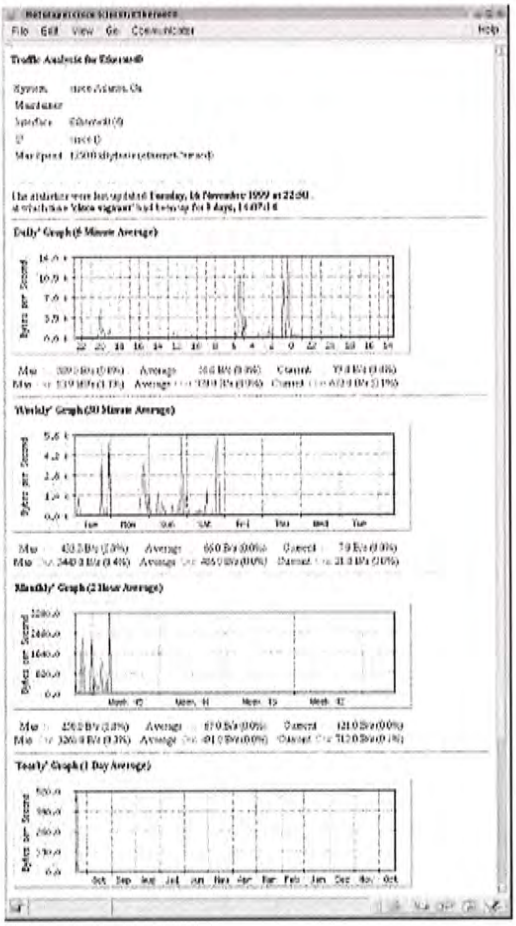
To display the entire list of options to *indexmaker*, run the command without any options. Figure 13-1 shows how the *cisco.html* file generated by *indexmaker* looks when it's loaded into a web browser.

Figure 13-1. Cisco graph overview



There are four graphs on the page, one for each of the operational interfaces (interfaces that were up and running when we ran *cfgmaker*) on our router. This page includes links to other pages that have more detailed information about individual interfaces; Figure 13-2 shows the daily, weekly, monthly, and yearly traffic graphs for the Ethernet0 interface.

Figure 13-2. Daily, weekly, monthly, and yearly graphs for Etherenet0



The daily graph (which actually represents a 32-hour period) is the one that most people are interested in viewing. It shows the five-minute average of the traffic on this particular interface. Incoming traffic (ifInOctets) is represented by a green line; outgoing traffic (IfOutOctets) is represented by a blue line. If we had clicked on one of the other interfaces on the Cisco index page (Figure 13-1), we would have seen a similar graph.

That's all there is to viewing the graphs. MRTG stores the raw data it collects in flat-text-file format but, due to its intelligent log rolling capabilities, the log files don't grow out of control; their sizes remain quite manageable even if you use MRTG extensively.

13.3 Graphing Other Objects

MRTG polls and graphs the MIB variables ifInOctets and ifOutOctets by default, but it is possible to poll and graph the values of other objects, in addition to polling different kinds of devices. Let's first get MRTG collecting input and output octets from a server. To do this, run the following command:

```
[root][linuxserver] ~/mrtg-2.9.10> cfgmaker public@linuxserver >> \
/mrtg2/run/mrtg.cfg
```

This is almost identical to the command we ran earlier in the chapter, except for the community string and target[1] (public@linuxserver). We appended the output to the *mrtg.cfg* file, as opposed to specifying an output file with the - -*output* option; this lets us add a new host to the existing configuration file, rather than starting a new file. Because the existing file already specifies a working directory, we also omitted the working directory option (- -*global 'WorkDir: .. '*). This *cfgmaker* command adds a number of lines like the following to the configuration file:

[1] Make sure that your target is running an SNMP agent. See Chapter 7 for a discussion of how to configure several SNMP agents for Unix and Windows NT.

```
Target[linuxserver]: 2:public@localhost
MaxBytes[linuxserver]: 1250000
Title[linuxserver]: linuxserver(linuxserver): eth0
PageTop[linuxserver]: <H1>Traffic Analysis for eth0
 </H1>
 <TABLE>
   <TR><TD>System:</TD><TD>linuxserver</TD></TR>
   <TR><TD>Maintainer:</TD><TD></TD></TR>
   <TR><TD>Interface:</TD><TD>eth0 (2)</TD></TR>
   <TR><TD>IP:</TD><TD>linuxserver(  )</TD></TR>
   <TR><TD>Max Speed:</TD>
       <TD>1250.0 kBytes/s (ethernetCsmacd)</TD></TR>
  </TABLE>
```

These lines tell MRTG how to poll the server's Ethernet interface. The key used for this interface is linuxserver, and the target number is 2. Why 2? Remember that *cfgmaker* walks the interface table to determine what entries to add to the configuration file. Therefore, you'll see a set of lines like this for each interface on the device, including the loopback interface. The target numbers are actually indexes into the interface table; on this server, the loopback interface has the index 1.

Now let's create an entry to graph the number of users logged onto the server and the total number of processes running. MRTG is capable of graphing these parameters, but you have to specify explicitly which MIB variables to graph. Furthermore, you have to specify two variables -- MRTG won't graph just one. (This is a rather strange limitation, but at least it's consistent: remember that the default graphs show both input and output octets.)

First, let's look at the MIB variables we plan to graph. The two variables, hrSystemNumUsers and hrSystemProcesses, are defined as OIDs 1.3.6.1.2.1.25.1.5.6.0 and 1.3.6.1.2.1.25.1.6.0, respectively. The .0 at the end of each OID indicates that these two objects are both scalar variables, not part of a table. Both come from the Host Resources MIB (RFC 2790), which defines a set of managed objects for system administration. (Some agents that run on server systems implement this MIB but, unfortunately, the Microsoft and Solaris agents do not.) The definitions for these objects are:

```
hrSystemNumUsers OBJECT-TYPE
    SYNTAX Gauge
    ACCESS read-only
    STATUS mandatory
    DESCRIPTION
        "The number of user sessions for which this host is storing state
         information. A session is a collection of processes requiring a
         single act of user authentication and possibly subject to collective
         job control."
    ::= { hrSystem 5 }

hrSystemProcesses OBJECT-TYPE
    SYNTAX Gauge
    ACCESS read-only
    STATUS mandatory
    DESCRIPTION
        "The number of process contexts currently loaded or running on
         this system."
    ::= { hrSystem 6 }
```

The entry we added to our configuration file looks like this:

```
Target[linuxserver.users]:1.3.6.1.2.1.25.1.5.0&1.3.6.1.2.1.25.1.6.0:public@linuxserver
MaxBytes[linuxserver.users]: 512
Options[linuxserver.users]: gauge
```

```
Title[linuxserver.users]: linuxserver (linuxserver): Number of users and processes
YLegend[linuxserver.users]: Users/Processes
Legend1[linuxserver.users]:  Users:
Legend0[linuxserver.users]:  Processes:
PageTop[linuxserver.users]: <H1>Number of users and processes</H1>
 <TABLE>
   <TR><TD>System:</TD><TD>linuxserver<TD></TR>
   <TR><TD>Maintainer:</TD><TD></TD></TR>
   <TR><TD>IP:</TD><TD>linuxserver(  )</TD></TR>
  </TABLE>
```

We've highlighted the changes and additions to the configuration file in bold. The first line specifies the device we want MRTG to poll, along with the two OIDs (hrSystemNumUsers and hrSystemProcessess) we want to graph. This statement is obviously more complex than the **Target** statement we looked at earlier; its syntax is OID1&OID2:community_string@device. The OIDs must be separated by an ampersand character (&). Using this syntax, you can convince MRTG to graph any two scalar-valued MIB variables.

In the next line, we set **MaxBytes** to **512**. This is the maximum value for the graph; values greater than 512 are set to **512**. (Forget about bytes; **MaxBytes** simply defines a maximum value.) For the number of users logged in, this is a high number; there should never be this many people logged onto our system at once. The same goes for the total number of processes running on the system. You can choose values that make sense for your particular environment. If you need separate maximum values for each object, replace **MaxBytes** with two lines setting **MaxBytes1** and **MaxBytes2**.

The **Options** command is a new one; it allows you to change how MRTG treats the data it gathers. The only option we have specified is **gauge**. This instructs MRTG to treat the gathered data as **Gauge** data, not **Counter** data. Recall that **Counter** data is monotonically increasing, while **Gauge** data is not. Since the MIB definitions for both objects specify the **Gauge** datatype, this option makes sense.

The **YLegend**, **Legend1**, and **Legend0** options are also new. **YLegend** simply changes the label that is placed on the Y-axis of the graph itself. Since we're graphing the number of users and processes, we set the legend to **Users/Processes**. It's important for the legend to be short; if it's too long, MRTG silently ignores it and doesn't print anything for the label. **Legend1** changes the legend used below the graph for the so-called "input variable" (in this case the number of users logged into the system -- remember that MRTG expects to be graphing input and output octets). **Legend0** changes the legend for the "output variable" (the total number of processes running on the system). The terminology is unfortunate; just remember that MRTG always graphs a pair of objects and that the input legend always

refers to the first object, while the output legend refers to the second.

Once you have added this entry to your configuration file and saved it, MRTG will start gathering data from the device every time it runs. If you have added the appropriate entry in your *crontab* file, you're all set. Now we'll use *indexmaker* to create intuitive index files for the server graphs, just as we did for the router graphs. The command to create a new index file is similar to the one we used to create the Cisco index file:

```
[root][linuxserver] ~/mrtg-2.9.10>  indexmaker --title "Linux Server" \
--filter name=~'linuxserver' --output /mrtg/images/linux.html /mrtg/run/mrtg.cfg
```

Figure 13-3 shows the index page for the server graphs. It contains only two graphs: one shows traffic on the Ethernet interface and the other shows the number of running processes versus the number of users logged onto the system.

Figure 13-3. Linux Server overview graphs

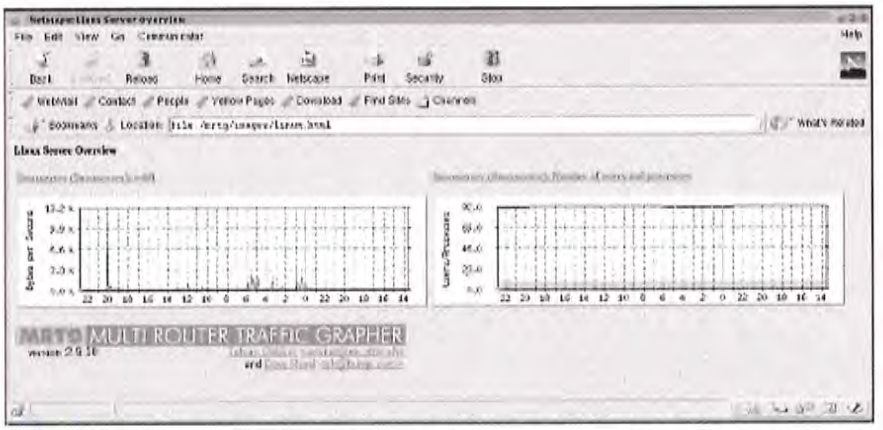
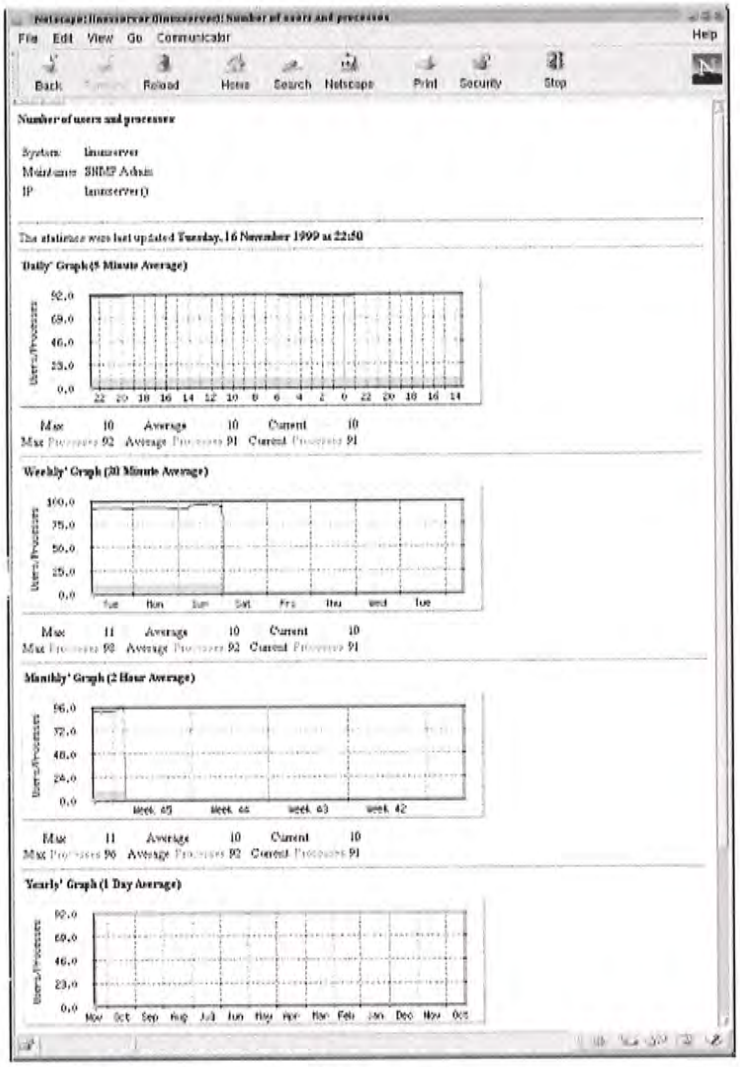


Figure 13-4 shows the daily, weekly, monthly, and yearly graphs for the number of users and processes logged into the system.

Figure 13-4. Daily, monthly, weekly, and yearly graphs for number of users and processes



13.4 Other Data-Gathering Applications

What if you need to monitor devices on your network that don't support SNMP? MRTG is up to the task. For example, you may have a Perl script that gathers usage statistics from some device that doesn't support SNMP. How can you collect and graph this data? Let's make this more concrete. Assume that you have the following script, */usr/local/scripts/hostinfo.pl*, which reports the number of users and the number of processes on the system:

```
#!/usr/bin/perl

$who = "/usr/bin/who | wc -l";
$ps = "/bin/ps -ef | wc -l";

chomp($numUsers = int(`$who`));
# We subtract two because ps generates a header and the ps process
# is counted as running.
chomp($numProcesses = int(`$ps`) - 2);
print "$numUsers\n";
print "$numProcesses\n";
```

```
#
# The following code prints the system uptime and the hostname. These two
# items need to be included in every script that you write and should be the
# very last thing that is printed.
#
chomp($uptime = `/usr/bin/uptime`);
print "$uptime\n";

chomp($hostname = `/bin/hostname`);
print "$hostname\n";
```

This script prints four variables: the number of users and the number of processes (the data we want MRTG to collect) and the system uptime and hostname (required by MRTG). To get MRTG to run this script, we'll have to edit *mrtg.cfg* by hand. The modification is actually simpler than our previous example. Here's the new entry to *mrtg.cfg*, with the changes shown in bold:

```
Target[linuxserver.users]: `/usr/bin/perl /usr/local/bin/hostinfo.pl`
MaxBytes[linuxserver.users]: 512
Options[linuxserver.users]: gauge
Title[linuxserver.users]: linuxserver (linuxserver): Number of
users and processes
YLegend[linuxserver.users]: Users/Processes
Legend1[linuxserver.users]:  Users:
Legend0[linuxserver.users]:  Processes:
PageTop[linuxserver.users]: <H1>Number of users and processes
 </H1>
 <TABLE>
   <TR><TD>System:</TD><TD>linuxserver<TD></TR>
   <TR><TD>Maintainer:</TD><TD></TD></TR>
   <TR><TD>IP:</TD><TD>linuxserver(  )</TD></TR>
  </TABLE>
```

Note the addition of `/usr/bin/perl /usr/local/bin/hostinfo.pl` to the Target command. This line tells MRTG to run the script or program between the backticks. The rest should be familiar. MRTG interprets the first value that the script prints (the number of users) as its input data; the second value (the number of processes) is the output data. When it generates graphs, it applies the appropriate input and output legends (Legend1 and Legend0).

13.5 Pitfalls

Many SNMP-capable devices change the order of interfaces in the interfaces table whenever a new interface card is inserted or an old one is removed. If you run a fairly static router environment (i.e., you hardly ever add or remove cards from your routers), the configuration examples we've shown should work well for you. But in today's fast-paced network environments, stability is rare. MRTG's *cfgmaker* command provides a command-line option, *--ifref*, to help with this

problem. It doesn't solve the problem, but it does allow you to generate graphs in which interfaces are labeled with their addresses, descriptions, or names; with this information, you don't have to remember whether interface 1 is your local network interface or your T1 connection. Table 13-2 summarizes the usage of - -*ifref*.

Table 13-2. Summary of --ifref Options

Option	Description
--*ifref=ip*	Identify each interface by its IP address.
--*ifref=eth*	Use the Ethernet address to identify the interface.
--*ifref=descr*	Use the interface description to identify the interface.
--*ifref=name*	Use the interface name to identify the interface.

Thus, to label interfaces with their IP addresses, run *cfgmaker* like so:

```
[root][linuxserver] ~/mrtg-2.9.10> cfgmaker --global 'WorkDir: /mrtg/images' \
--output /mrtg/run/mrtg.cfg --ifref=ip public@router
```

Be sure to read the *cfgmaker* manual that comes with the MRTG documentation.

13.6 Getting Help

The MRTG web site, http://www.mrtg.org, offers a great deal of information and help. You can subscribe to the MRTG mailing list from this page. MRTG is also discussed frequently in the Usenet newsgroup comp.dcom.net-management. Finally, don't ignore MRTG's documentation, which is located in the *doc* subdirectory of the MRTG distribution. The documentation is included in both text and HTML form and is fairly complete and comprehensive.

Appendix A. Using Input and Output Octets

To be SNMP-compatible, an IP device must support MIB-II (iso.org.dod.internet.mgmt.mib-2) objects. MIB-II contains the interfaces table (mib-2.interfaces.ifTable.ifEntry), which is one of the most useful objects for network monitoring. This table contains information about the system's network interfaces. Some of its objects are:

ifDescr

A user-provided description of the interface

ifType

The interface's type (token ring, Ethernet, etc.)

ifOperStatus

Whether the interface is up, down, or in some kind of test mode

ifMtu

The size of the largest packet that can be sent over the interface

ifSpeed

The maximum bandwidth of the interface

ifPhysAddress

The low-level (hardware) address of the interface

ifInOctets

The number of octets received by the interface

ifOutOctets

The number of octets sent by the interface

We explored various parts of this table in other chapters, but avoided saying too much about ifInOctets and ifOutOctets. RFC 1213 states that ifOutOctets and ifInOctets are the total number of octets sent and received on an interface, including framing characters.

In many environments, this information is crucial. Companies such as Internet service providers (ISPs) make their livelihoods by providing usable bandwidth to their customers, and thus spend huge amounts of time and money monitoring and measuring their interfaces, circuits, etc. When these pipes fill up or get clogged, customers get upset. So the big question is, how can you monitor bandwidth effectively? Being able to answer this question is often a life and death issue.

The information you need to answer this question comes in a few parts. First, you must know what type of line you are trying to monitor. Without this information, the numbers don't mean much. Then you must find the line's maximum speed and determine whether it is used in full- or half-duplex mode. In most cases, you can find both of these pieces of information using SNMP. The ifSpeed object defined in MIB-II's interfaces table provides "an estimate of the interface's current bandwidth in bits per second." You can poll this object to find the line's maximum speed, or at least what the agent thinks the line's maximum speed should be. Note, though, that you must watch for some pitfalls. For example, Cisco routers have default maximum bandwidths for various types of links, but these defaults may not have much to do with reality: for instance, the default bandwidth for a serial line is 1.544 Mbps, regardless of the actual line speed. To get meaningful data, you must configure the router to report the maximum bandwidth correctly. (Sometimes, network administrators intentionally set the

interface bandwidth to an incorrect number to nudge routing paths a different way. If this is the case, you're going to have trouble getting meaningful data out of SNMP.)

It's easier to get reliable information about the line's duplex mode. Serial lines operate in full-duplex mode. This means they can send and receive information at the same time (e.g., a 56 Kbps serial line can upload and download at 56 Kbps simultaneously, for a total of 112 Kbps). Other types of lines, such as 10BaseT Ethernet, can handle only half duplex. In a typical 10BaseT environment, the distinction between uploading and downloading data is meaningless; total bandwidth through the line is limited to 10 Mbps of input and output combined. Some devices have 10/100 cards in them, which makes identification even harder.

Many vendors have private MIBs that return the duplex state. For example, the following Cisco object returns the duplex state for an interface on the model 2900 switch:
iso.org.dod.internet.private.enterprises.cisco.ciscoMgmt.ciscoC2900MIB
.c2900MIBObjects.c2900Port.c2900PortTable.c2900PortEntry.c2900PortDuplexStatus.

The table to which this object belongs also contains an object that can be used to switch an interface's duplex state. This object is useful if you have a device that is incorrectly negotiating half duplex instead of full duplex; you can use it to force the port into the correct duplex state.

Once you find the line's maximum speed and duplex mode, you can calculate its utilization percentage. Many NMS products let you create expressions, which are named formulas that use MIB objects as variables. OpenView allows you to define expressions in the file *$OV_CONF/mibExpr.conf*. The syntax used in this file is complicated. Expressions are written in postfix notation.[A] The file contains some entries by default; these expressions are often useful, and may not need any tweaking[A] to work for your environment. Here is the default definition of the expression **lf%util**:

[A] Also referred to as "reverse Polish notation." Instead of writing "1 + 2", you would write "1 2 +".

[A] The recommended way to modify *$OV_CONF/mibExpr.conf* is to use *xnmcollect* with the *-delExpr* or *-loadExpr* switch.

```
lf%util \
"Percent of available bandwidth utilized on an interface\n\
Computed by:\n\
    (Received byte rate + transmitted byte rate) * 8\n\
    ----------------------------------------------\n\
                    interface link speed\n\
then converted to a percentage."\
```

```
.1.3.6.1.2.1.2.2.1.10. \
.1.3.6.1.2.1.2.2.1.16. \
+ \
8 \
* \
.1.3.6.1.2.1.2.2.1.5. \
/ \
100 \
*
```

This expression is broken up into three parts: an expression name, comments, and the expression itself. We will use the expression name within *xnmgraph* for our data-collection definitions. The comments will help us understand what this expression really does. The syntax of the expression is defined in the mibExpr.conf (4) manpage. In short, it adds the values of two MIB objects (ifInOctets and ifOutOctets), multiplies by 8 to get the number of bits traveling through the interface, divides by the interface speed (ifSpeed), and converts the result to a percentage. As you can see here, you can break expressions into several lines by using the familiar Unix backslash-escape at the end of each line.

Once we have defined `lf%util`, we can use it to plot utilization with *xnmgraph*:

```
$ /opt/OV/bin/xnmgraph -monochrome -c public -poll 5 -title lfutil_Formula -mib \
lf%util:CiscoRouter1a::::.1.3.6.1.2.1.2.2.1.2:::" CiscoRouter14a
```

This displays a graph of the percent utilization for every interface on the device CiscoRouter14a. Note that you can use an expression name as the first of the colon-separated arguments in the *xnmgraph* command.

Before you start using `lf%util` to measure your entire organization, notice that this expression measures only half-duplex lines -- that is, it compares the sum of the input and output octets to the line's capacity. Any full-duplex line graphed with this calculation will look wrong. To prove this point, consider a full-duplex serial line with a maximum speed of 500 Kbps in each direction that is currently sending 125 Kbps and receiving 125 Kbps. The formula for `lf%util` gives us a utilization of 50%, which is incorrect: the line is really at 25% of capacity. For a full-duplex line, it makes more sense to make separate computations for incoming and outgoing data. This gives you a better representation of what your network is doing, since in full-duplex mode the incoming data rate isn't affected by the outgoing data. Here are revised expressions for send utilization (`WANIF%SendUtil`) and receive utilization (`WANIF%RecvUtil`):

```
WANIf%SendUtil \
"% interface utilization from (ifOutOctets * 8 * 100) / ifSpeed"\
.1.3.6.1.2.1.2.2.1.16. \
8 \
* \
```

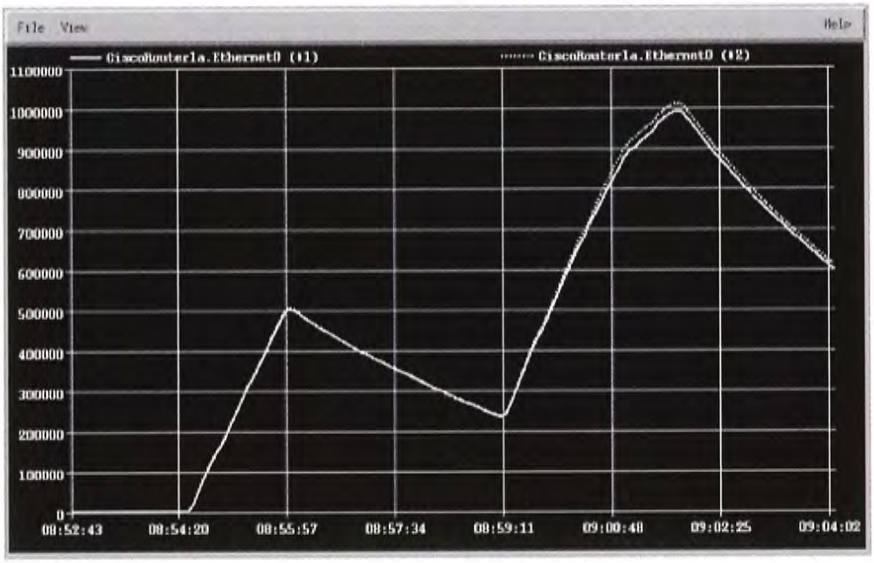
```
100 \
* \
.1.3.6.1.2.1.2.2.1.5. \
/

WANIf%RecvUtil \
"% interface utilization from (ifInOctets * 8 * 100) / ifSpeed"\
.1.3.6.1.2.1.2.2.1.10. \
8 \
* \
100 \
* \
.1.3.6.1.2.1.2.2.1.5. \
/
```

Now let's take a look at some actual graphs. We graphed different expressions and MIB objects at the same time for a 10BaseT (half-duplex) Ethernet interface. We then created some traffic on the interface and captured the results. Here is the script that generates the graphs:

```
/opt/OV/bin/xnmgraph -monochrome -c public -poll 5 -title \
Cisco_Private_Local_Mib -mib \
".1.3.6.1.4.1.9.2.2.1.1.6:CiscoRouter1a:4:::.1.3.6.1.2.1.2.2.1.2:::,\
.1.3.6.1.4.1.9.2.2.1.1.8:CiscoRouter1a:4:::.1.3.6.1.2.1.2.2.1.2:::" \
CiscoRouter1a &

/opt/OV/bin/xnmgraph -monochrome -c public -poll 5 -title Ifutil_Formula \
-mib "If%util:CiscoRouter1a:4:::.1.3.6.1.2.1.2.2.1.2:::" CiscoRouter1a &

/opt/OV/bin/xnmgraph -monochrome -c public -poll 5 -title \
WANIfRecvUtil_Formula -mib \
"WANIf%RecvUtil:CiscoRouter1a:4:::.1.3.6.1.2.1.2.2.1.2:::" CiscoRouter1a &

/opt/OV/bin/xnmgraph -monochrome -c public -poll 5 -title
WANIfSendUtil_Formula -mib \
"WANIf%SendUtil:CiscoRouter1a:4:::.1.3.6.1.2.1.2.2.1.2:::" CiscoRouter1a &

/opt/OV/bin/xnmgraph -monochrome -c public -poll 5 -title ifInOctets -mib \
".1.3.6.1.2.1.2.2.1.10:CiscoRouter1a:4:::.1.3.6.1.2.1.2.2.1.2:::" \
CiscoRouter1a &

/opt/OV/bin/xnmgraph -monochrome -c public -poll 5 -title ifOutOctets -mib \
".1.3.6.1.2.1.2.2.1.16:CiscoRouter1a:4:::.1.3.6.1.2.1.2.2.1.2:::" \
CiscoRouter1a &
```

Figure A-1 shows the MIB objects .iso.org.dod.internet.private.enterprises.cisco.local.linterfaces.lifTable.lifEntry.locIfInBitsSec and .iso.org.dod.internet.private.enterprises.cisco.local.linterfaces.lifTable.lifEntry.locIfOutBitsSec. These are private Cisco MIB objects that report the data rate in and out of an interface, in bits per second.

Figure A-1. Graph of Cisco private MIB objects



The next graph, shown in Figure A-2, shows the expression **If%util**. It's surprisingly different. The difference arises because Cisco uses a five-minute decaying average for these two objects. This can be both good and bad. The decaying average can prevent you from seeing local peaks and valleys in usage. In this example, we see two usage peaks, which the decaying average smears over a longer period of time. When using vendors' private MIBs, be sure to find out how they calculate their numbers.

Figure A-2. Graph of If%util

Figure A-3. Graph of WANIf%RecvUtil

Figures A-3 and A-4 show the **WANIf%RecvUtil** and **WANIf%SendUtil** expressions. Since this is a half-duplex interface we don't need to look at each direction (in and out) separately, but it may help to verify whether the receive path or the send path is maxed out. Comparing Figure A-3 with Figure A-4 shows that we are sending a bit more traffic than we are receiving.

Figure A-4. Graph of WANIf%SendUtil

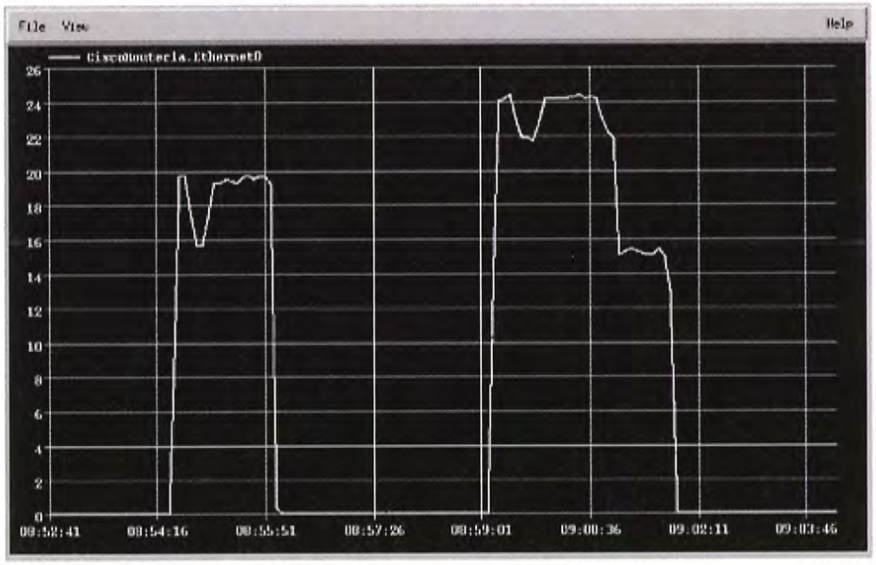


Figure A-5. Graph of ifInOctets

The standard MIB-II objects ifInOctets and ifOutOctets are graphed in Figure A-5 and Figure A-6. Remember that these do not show bits per second. Again, these graphs show that we are sending more traffic than we are receiving. The octet graphs in Figures A-5 and A-6 show a real-time picture, like the WAN expressions but unlike Cisco's private MIB objects.

Figure A-6. Graph of ifOutOctets

Try to get a feel for what you are looking for before you start writing expressions. Are you trying to find someone who is flooding the network, or just looking for a weekly average? No matter what you are graphing, be sure to research the device's MIB objects before you start generating graphs that may look good but contain meaningless data. Recheck the variables each time you create new graphs.

Keep in mind that some devices have the ability to switch from full to half duplex automatically. You should be aware of your

interface's saturation point, which is the point at which no more traffic can be sent or received. This saturation point is indicated in your graphs by a sustained horizontal ceiling line and can really be seen only over extended periods of time. Thus, while there are some horizontal lines in the graphs in this appendix, we are obviously not close to the interface's capacity.

If you plan to use graphs like these, be sure to plan for the average and not for the exceptions (peaks). All networks have traffic spikes here and there; unless you like spending a lot more on telecommunications than you need to, you should plan your network so that it is geared toward servicing your average day-to-day activities, not the occasional peak.

Appendix B. More on OpenView's NNM

By now you should be familiar with OpenView's NNM and its supporting utilities. Even though many network administrators can get by with the basic OpenView information provided in this book, there is much more to learn. Configuring NNM with your own custom tools makes using it that much better.

While we can't cover all the features of NNM in this appendix, we'll discuss each of the following:

- Using external data with *xnmgraph*
- Inserting additional menu items into NNM's menu
- Creating NNM profiles for different users
- Using NNM as a centralized communication device

B.1 Using External Data

Chapter 9 introduced the *xnmgraph* command, but only touched on its features. One particularly useful feature is the ability to graph data from external sources. To see how you might graph external data, first generate a graph of any type -- one of the graphs we created in Chapter 9 will do -- and save the data to a file. Then examine the contents of the file. Each output file contains a short tutorial showing how to reshow the graph. Be sure to look at *$APP_DEFS/Xnmgraph*, which contains *xnmgraph*'s default settings.

Here's a table we created by hand, copying the format of a standard *xnmgraph* datafile. The data points are organized into streams. A *stream* is a set of data that will be plotted as a single curve on the graph. All the streams in the file will be combined into a single graph with multiple curves. The **StartTime** is ignored. The **StopTime** provides the value for the X

(horizontal) axis and the Value provides the value for the Y (vertical) axis:

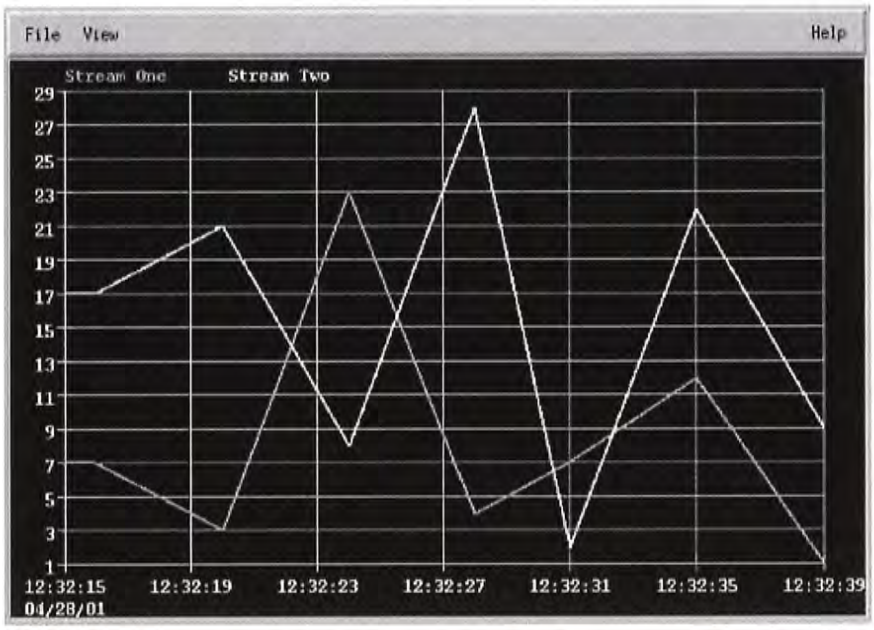
```
# /tmp/data1
#
# Stream Number StartTime      StopTime                  Value
# ------------- ---------      ------------------        -----
#
# Start of Stream 1
#
  1             0              04.28.2001-12:32:16       7
  1             0              04.28.2001-12:32:20       3
  1             0              04.28.2001-12:32:24       23
  1             0              04.28.2001-12:32:28       4
  1             0              04.28.2001-12:32:31       7
  1             0              04.28.2001-12:32:35       12
  1             0              04.28.2001-12:32:39       1
#
# Start of Stream 2
#
  2             0              04.28.2001-12:32:16       17
  2             0              04.28.2001-12:32:20       21
  2             0              04.28.2001-12:32:24       8
  2             0              04.28.2001-12:32:28       28
  2             0              04.28.2001-12:32:31       2
  2             0              04.28.2001-12:32:35       22
  2             0              04.28.2001-12:32:39       9
```

The following *xnmgraph* command displays our datafile. Notice that we use stream numbers, preceded by minus signs, instead of object IDs. The minus sign indicates that the stream can take on negative values. If the stream number is preceded by a + or = sign, *xnmgraph* will take the absolute value of all negative numbers in the datafile.

```
cat /tmp/data1 | xnmgraph -mib "-1:Stream One:::::::,-2:Stream Two:::::::"
```

Figure B-1 shows the result of this command. If your graph looks squished, right-click on it and then left-click on "Show All." An option under the View menu lets you generate a black-and-white graph, which is often more effective if you have only a small number of streams.

Figure B-1. Sample OpenView graph



Now that we can get data into a format that *xnmgraph* can display, let's see if we can generate some graphs from the output of the Unix *vmstat* utility. *vmstat* should be familiar to all Unix administrators; it provides a lot of information about your memory system, in a cumbersome format. Here's the kind of output *vmstat* produces:

```
procs     memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr s6 s2 s2 sd   in   sy   cs us sy id
 0 4 0 5431056 33672  1 2371 0  8  8  0  0  0 18 18  2 2161 5583 4490 17 14 69
 0 2 0 5430912 33576  1 2499 0 20 20  0  0  0  1  1  0 2997 8374 7030 25 18 58
 0 2 0 5431296 33824  0 179  4  0  0  0  0  0  0  0  1 2587 3990 6379 18  8 74
 0 0 0 5431240 33792  1 2460 4  8  8  0  0  0  1  1  0 2909 7768 7080 25 18 57
 0 3 0 5431216 33768  1 2359 0 12 12  0  0  0  2  2  0 1934 5057 3818 18 13 70
 0 0 0 5431288 33824  0 136  0  0  0  0  0  0  0  0  1 1842 2190 3803 13  5 82
 0 2 0 5431216 32920  2 1189 0 3196 3176 0 0 0 0  0  4 2734 9980 5642 24 11 65
 0 4 0 5431032 32352  8 1571 0 3100 3044 0 0 0 2  2  5 2763 7767 5817 22 15 63
```

Imagine taking 10,000 lines of this output and trying to figure out the trends (min/avg/max) in any given parameter. It's not easy. But with some help from a Perl script, we can massage this data into an *xnmgraph* input file. Here is what our Perl script looks like:

```
#!/usr/local/bin/perl
# Filename: /usr/local/bin/perl_scripts/cputimes

$|++; # Unbuffer the output!

open(VMSTAT,"/bin/vmstat 2 |") || die "Can't Open VMStat";
while($CLINE=<VMSTAT>)
{
    ($null,$r,$b,$w,$swap,$free,$re,$mf,$pi,$po,$fr,$de,$sr,$aa,$dd1,\
$dd2,$f0,$in,$sy,$cs,$us,$sycpu,$id) = split(/\s+/,$CLINE);

    if (($id) && ($id ne "id"))
```

```
    {
        $DATE = `date +%m.%d.%y-%H:%M:%S`;
        chomp $DATE;
        print "1 0 $DATE $us \n";
        print "2 0 $DATE $sycpu \n";
        print "3 0 $DATE $id \n";
    }
    sleep 2;
}
```

This script prints the current CPU usage, as a percentage, in the User ($us), System ($sycpu), and Idle ($ide) states; stream 1 is the User percentage, stream 2 is the System percentage, and stream 3 is the Idle percentage. The first item on each line is the stream number; note that we can interleave the data from the three streams:

```
[root][nms] /> /usr/local/bin/perl_scripts/cputimes
1 0 8.14.99-21:00:22 6
2 0 8.14.99-21:00:22 3
3 0 8.14.99-21:00:22 92
1 0 8.14.99-21:00:24 0
2 0 8.14.99-21:00:24 0
3 0 8.14.99-21:00:24 100
1 0 8.14.99-21:00:26 1
2 0 8.14.99-21:00:26 0
3 0 8.14.99-21:00:26 98
1 0 8.14.99-21:00:28 1
2 0 8.14.99-21:00:28 0
3 0 8.14.99-21:00:28 99
```

The following command generates a graph from the script's output:

```
/usr/local/bin/perl_scripts/cputimes | xnmgraph -title "CPU Time"  -mib \
"+1:User:::::::,+2:System:::::::,+3:Idle::::::"
```

While this graph is based on live data, it's trivial to save data in an appropriate format and write a script that pulls historical data from your logs and plots it with *xnmgraph*.

B.2 Adding a Menu to NNM

Once you have a toolbox of scripts, adding them to an NNM menu makes them easier to access and execute. This trick can be especially useful if you prefer to use NNM's graphical interface.

The key to adding custom menus is the directory *$OV_REGISTRATION/C*. (*$OV_REGISTRATION* contains directories for all the languages available on your system; *C* is the directory for the default language and is probably where you should start.) The *C* directory contains all the files that make up the menu system you see when you run NNM. For example, the file *ovw*

contains the familiar options from the main window (New, Open, Refresh, etc.).

Let's look at the *$OV_REGISTRATION/C/ovsnmp/xnmloadmib* file. It's fairly easy to see how to hook an external command into a menu. Let's jump right in and create a menu that is two levels deep with two menu choices:

```
Application "Graph Menu"
{
        Menubar <100> "Local_Graphs" _p
        {
          <100> "Network"       _N f.menu "network_menu";
        }

        Menu "network_menu"
        {
          <90> "5 Minute CPU"   _M f.action "5mincpu";
          <90> "Bits In and Out For All Up Interfaces"  \
                                _B f.action "bit_for_all_up";
        }

        Action "5mincpu" {
          Command "/opt/OV/local/scripts/Cisco_5min_cpu \
                                        \"${OVwSelections}\"";
          MinSelected     1;
          MaxSelected     7;
          SelectionRule   (isSNMPSupported || isSNMPProxied) ;
        }

        Action "bit_for_all_up" {
          Command "/opt/OV/local/scripts/Cisco_Line_Up_Bits \
                                        \"${OVwSelections}\"";
          MinSelected     1;
          MaxSelected     3;
          SelectionRule   (isSNMPSupported || isSNMPProxied) ;
        }
}
```

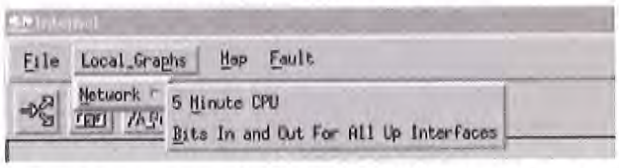
Create a file within *$OV_REGISTRATION/C* and insert the previous code listing. Once this is done, run *ovw* with the *-verify* switch, which checks for errors.[B] You may see errors or warnings about your new menu item but, if you're successful, you'll see an item that looks like the menu in Figure B-2.

[B] Do not leave any backup files within any of the directories, because NNM takes each file seriously. Backup or redundant files will produce warnings when you run *ovw*.

NNM can be picky with registration files. If you can't see your menu, try the *ovw -verify* trick. If it reveals no errors, take some entries out and restart *ovw*. Keep doing this until your items appear. You should also break up your menu

items into multiple files. Do not put all your menus and actions into one file. The more files you have, the easier it will be to diagnose and troubleshoot your new menu items.

Figure B-2. A new menu



Let's talk about some commonalties within our registration file:

- Each menu and menu item is associated with a keyboard shortcut that allows the user to access it. The trigger character is preceded by an underscore. For example, from the "Local_Graphs →Network" menu, you can hit "M" to go to the "5 Minute CPU" item.
- Each menu item has a precedence number within angle brackets. This allows you to control the order in which items appear. Items with the highest precedence appear first in a menu; items with the same precedence are listed in the order in which they appear in the file. For example, if we reduce the precedence of "5 Minute CPU" from <90> to <80> it will appear after the "Bits In and Out" menu item, because the higher-precedence item comes first.

The Menubar entry contains the menus that will appear in the top NNM menu bar. We used the function f.menu to call a submenu. The following code shows how we could have used f.action to call an action directly:

```
Menubar <precedence> "menubar Label" _MnemonicChar
  {
     <precedence> "SubMenu Label" _MnemonicChar f.menu "menu-name"
     <precedence> "Action Name" _MnemonicChar f.action "action-name"
  }
```

A Menu looks and behaves like the menu bar (or menu) that contains it, with a few differences. Menus don't declare mnemonic characters or precedence; these are defined by the containing menu or menu bar. The menu-name is the linking name that appears after f.menu.

```
Menu "menu-name"
  {
     <precedence> "SubMenu Label" _MnemonicChar f.menu  "menu-name"
     <precedence> "Action Name" _MnemonicChar f.action "action-name"
  }
```

Actions are called just like Menus. The action-name is the linking name of an action that gets called when selected from a previous item (either a Menu or a Menubar):

```
Action "action-name"
  {
    Command "/opt/OV/local/scripts/Cisco_5min_cpu \"${OVwSelections}\"";
    MinSelected     1;
    MaxSelected     7;
    SelectionRule   (isSNMPSupported || isSNMPProxied) ;
  }
```

There are a few additional parameters in our **Action** declaration:

- **Command** specifies which program or script should be executed. The \"${OVwSelections}\" at the end of the command string passes all currently selected objects to the program as arguments.
- **MinSelected** declares how many nodes must be selected before this item becomes available. If nothing is selected, the corresponding menu choice will be grayed out and unclickable.
- **MaxSelected** works the same way, but declares the maximum number of objects that can be selected.
- **SelectionRule** uses capability fields[B] within a logical statement. These rules declare what is necessary for the selection to be deemed a "good selection."

  [B] Check out $OV_FIELDS for more definitions of capability fields.

Action declarations can contain many additional parameters, as can registration files. The examples we've given should be enough to get you going in the right direction. The OVwRegIntro (5) manpage defines the syntax of the registration files in detail; read this page carefully if you're serious about adding custom menu items.

B.3 Profiles for Different Users

Some users may have specific ways in which they want to use NNM. For example, an operator who is watching the network for problems may need a fairly limited set of menus and tools; a senior network engineer might want a substantially larger set of options. You can use the *$OV_REGISTRATION* directory and the $OVwRegDir environment variable to customize NNM on a per-user basis.

The previous section shows how to add menus by modifying files in the *$OV_REGISTRATION/C* directory. By default, this is the directory NNM uses when it starts. However, you can create as many profiles as you need under the *$OV_REGISTRATION* directory. Once you have created another profile directory, you can change the $OVwRegDir environment variable to point to that new directory. Then, when NNM starts, it will use the new profile.

One way to set up user-specific profiles is to create an account that anyone can use for starting an NNM session. With this account, the network map is opened read-only[B] and has only the minimal menus ("File →Exit," "Map →Refresh," "Fault →Alarms," etc.). Create a new profile for this account in the directory *$OV_REGISTRATION/skel* by copying all the files in the default profile *$OV_REGISTRATION/C* to the new *skel* directory. Then modify this profile by removing most of the menu choices, thus preventing the operator from being able run any external commands.[B] To start NNM using this profile, you must point the $OVwRegDir environment variable to the new profile directory. To test the new profile, give the following Bourne shell commands:

> [B] When starting NNM via the command line, use *$OV_BIN/ovw -ro* to open the default map in read-only mode. This will prevent the user from making any map changes (moves, add, deletes, etc.).
>
> [B] Just because a map is opened read-only does not mean that users cannot make changes to the backend of NNM. A user who has the ability to launch the menu items can make changes just like the superuser can. The best way to prevent these changes is to take out any/all configuration menu options.

```
[root][nms] /> OVwRegDir=/etc/opt/OV/share/registration/skel
[root][nms] /> export OVwRegDir
[root][nms] /> $OV_BIN/ovw
```

Once you're confident that this new profile works, create an account for running NNM with minimal permissions and, in the startup script for that account, set $OVwRegDir appropriately (i.e., to point to your skeleton configuration). Then make sure that users can't run NNM from their normal accounts -- perhaps by limiting execute access for NNM to a particular group, which will force users not in that group to use the special account when they want to run NNM. You should also make sure that the users you don't trust can't modify the *$OV_REGISTRATION* directory or its subdirectories.

B.4 Using NNM for Communications

One of the more exotic ways to use SNMP is as a tool for passing messages back and forth. For example, it's certainly useful to know that the Oracle database has gone down, but it's even more useful to send messages to key users notifying them that the database has crashed or that it's going down for maintenance at the end of the day. In a small environment, it's easy to come up with hacks that provide various kinds of notification. But in a large company with many offices, it's useful to have a standard way for communicating with other

departments. NNM's Event Categories is the perfect tool to use as a centralized communication device.

Imagine a web interface that allows you to send traps to Event Categories. Filling out a simple form in a browser automatically generates a trap that is posted to the appropriate categories. Figure B-3 shows such an interface.

Figure B-3. SNMP web interface

Select Your Name	Other:
Select Desc Of Action	Other:
Select Server(s) Affected	Other:
Select Services Affected	Other:
Select Time Of Event	Seconds / From Now
Select Est Time Of Completion	Seconds
Select Severity	* See Below For Severity Descriptions
Additional Comments	
	Submit Event - Clear Form

What types of questions does everyone (you, managers, users, etc.) ask when there's a problem? The most typical ones are:

Who is in charge? Name, phone, pager
What is going on? Reboot, upgrade, failure
What servers are affected? Production, test, development
What services are affected? Mail, news, database, web server
When did this happen? E.g., 10 minutes ago, 4 days from now
When will this be fixed? E.g., immediately, tomorrow
What is the severity? Normal, Warning, Minor, Major, Critical

All these questions can be answered using the HTML form in Figure B-3. The CGI script or Java servlet that processes the form can refuse to accept the form until the user has filled in all the fields, guaranteeing that you have complete and consistent information.

Setting up a reporting system like this is not very difficult. You can use any standard web server,[B] a little HTML, and your favorite language for processing the form. Once you parse the output from the form, you can use any of the trap-generation programs we've discussed to send the trap. This trap will then show up in one of NNM's Event Categories. (If you're not using NNM, we've discussed other trap daemons that can be used to receive the trap and notify users. However, NNM is convenient because it will do everything for you.)

[B] Check out http://www.apache.org for more information on a free Unix or NT web server.

The key to this whole setup is getting people to use and watch NNM. If it isn't used by everyone, this mechanism really doesn't accomplish anything. Training users in nontechnical departments to watch NNM for important notifications may not be easy, but if you succeed you'll have created an elegant mechanism for getting important information to users.

Appendix C. Net-SNMP Tools

This appendix provides brief summaries of the command-line tools included in Version 4.2 of the Net-SNMP package (available from http://net-snmp.sourceforge.net).

Rather than trying to describe all the options to all the commands, we've focused on those that are most important and useful. We have also pointed out a few cases in which the behavior of the commands differs from the behavior that's described in the manual pages. Unfortunately, there are many discrepancies. The current situation is obviously far from ideal, but hopefully either the documentation or the commands will be fixed in some later release.

C.1 Net-SNMP and MIB Files

By default, Net-SNMP reads the MIB files in the directory */usr/local/share/snmp/mibs*. When you install Net-SNMP it populates this directory with a few dozen MIB files, including the UCD MIB (Net-SNMP used to be called UCD-SNMP) and RFC 1213 MIB (MIB-II). Net-SNMP uses the MIB files to translate between numeric object IDs and their textual representations. The MIB files also give the tools access to information about each object (its syntax, the type of access allowed, its description, etc.). Adding a vendor-specific MIB file to Net-SNMP is as simple as placing it in the *mibs* directory and setting the environment variable $MIBS to ALL, as discussed in the next section.

C.2 Common Command-Line Arguments

For the most part, the Net-SNMP commands follow a similar command structure; they share many options and use roughly the same syntax. For example, in the abstract, an *snmpget* command looks like this:

```
snmpget options hostname community objectID...
```

In other words, the command name is followed by a series of options, the hostname of the system you want to poll, the community string, and one or more object IDs. (Note that you can use the *-c* community option instead of placing the community string after the hostname. You can also provide a default hostname in your *snmp.conf* file.) The syntax of *snmpset* is only slightly different; because *snmpset* changes object

values, it requires you to specify the object's datatype and the new value:

```
snmpset options hostname community objectID type value...
```

Table C-1 summarizes some of the most useful options that are common to all Net-SNMP commands. See the *snmpcmd*(1) manpage for a complete list.

Table C-1. Summary of Command-Line Options

Option	Description
-m	Specifies which MIB modules you would like the command to load. If you want the command to parse the MIB file for a particular vendor, copy the MIB file to */usr/local/share/snmp/mibs* and invoke the command with the option *-m ALL*. The argument ALL forces the command to read all the MIB files in the directory. Setting the environment variable $MIBS to ALL achieves the same thing. If you don't want the command to read all the MIB files, you can follow the *-m* option with a colon-separated list of the MIB files you want parsed.
-M	Allows you to specify a colon-separated list of directories to search for MIB files. This option is useful if you don't want to copy MIB files into the default MIB location. Setting the shell variable $MIBDIRS has the same effect.
-IR	Performs a random-access search through the MIB database for an OID label. By default, the commands assume that you specify an object ID relative to .iso.org.dod.internet.mgmt.mib-2. In practice, this option allows you to avoid typing long OIDs for objects that aren't under the mib-2 subtree. For example, there's a group of objects in the Cisco MIB named lcpu. If you use the *-IR* option, you can retrieve objects in this group without typing the entire OID; the following command is sufficient: `snmpget -IR hostname community lcpu.2` If there is more than one object with the given name, the Net-SNMP tools will access the first object they find. Since this feature is billed as a random-access search, there's no way to predict which object the tools will find first. Within the standard MIBs, objects rarely (if ever) have the same name, but there's no guarantee that any name will be unique, particularly if you're using vendor-specific MIBs.
-On	Prints OIDs numerically (e.g., .1.3.6.1.2.1.1.3.0). Note that the *-O* options can be combined, as long as the

	combination makes sense.
-Of	Prints the entire OID (i.e., starting with .1).
-Os	Displays only the final part of the OID, in symbolic form (e.g., sysUpTime.0).
-OS	Same as *-Os*, but prefixes the object name with the name of the MIB file from which the object is taken (e.g., SNMPv2-MIB::sysUpTime.0).
-T	Specifies whether the command should use TCP or UDP as the transport-layer protocol. UDP is the default; *-T tcp* uses TCP.
-v	Specifies which version of SNMP to use. By default, the commands use Version 1. Valid options are -v 1, -v 2c, and -v 3. Note that some commands, such as *snmpbulkget*, are available only for Versions 2c and 3.
-h	Displays help information for the command.
-c	Specifies the community string for the command. Alternately, you can place the community string after the hostname and omit the *-c* option.

C.3 Net-SNMP Command-Line Tools

This section briefly describes each of the Net-SNMP tools. By default, installing Net-SNMP places all these commands in */usr/local/bin*. All the examples in this section assume that */usr/local/bin* is in your path.

C.3.1 snmpwalk

snmpwalk performs the get-next operation. We've used it throughout the book, so it should be familiar; in this section, we'll use it to demonstrate some of the options introduced in Table C-1.

Let's say you want to perform an *snmpwalk* against a Cisco router. If you don't have any Cisco MIBs installed, here's what you will see:

```
$ snmpwalk cisco.ora.com public .1.3.6.1.4.1.9
enterprises.9.2.1.1.0 = "..System Bootstrap, Version 11.2(17)GS2, [htseng 180]
EARLY DEPLOYMENT RELEASE SOFTWARE (fc1)..Copyright (c) 1999 by Cisco Systems,
Inc..."
enterprises.9.2.1.2.0 = "reload"
enterprises.9.2.1.3.0 = "cisco"
enterprises.9.2.1.4.0 = "ora.com"
enterprises.9.2.1.5.0 = IpAddress: 127.45.23.1
enterprises.9.2.1.6.0 = IpAddress: 0.0.0.0
enterprises.9.2.1.8.0 = 131890952
```

```
enterprises.9.2.1.9.0 = 456
enterprises.9.2.1.10.0 = 500
enterprises.9.2.1.11.0 = 17767568
enterprises.9.2.1.12.0 = 0
enterprises.9.2.1.13.0 = 0
enterprises.9.2.1.14.0 = 104
enterprises.9.2.1.15.0 = 600
...
```

Recall that .1.3.6.1.4.1 is .iso.org.dod.internet.private.enterprises, and 9 is Cisco's private enterprise number. Therefore, the previous command is walking the entire Cisco subtree, which is very large; we've deleted most of its output. The output you see isn't very readable because we haven't yet installed the Cisco MIBs, so the *snmpwalk* command has no way of providing human-readable object names. We just have to guess what these objects are.

This problem is easy to solve. Copy the Cisco MIBs[c] to the main Net-SNMP repository (*/usr/local/share/snmp/mibs*) and use the *-m ALL* command-line option. With this option, *snmpwalk* parses all the files in the MIB repository. As a result we get the object IDs in string (human-readable) form, and we can walk the cisco subtree by name rather than specifying its complete numeric object ID (.1.3.6.1.4.1.9):

[c] You can find many Cisco MIBs at ftp://ftp.cisco.com/pub/mibs/.

```
$ snmpwalk -m ALL cisco.ora.com public cisco
enterprises.cisco.local.lcpu.1.0 = "..System Bootstrap, Version 11.2(17)GS2,
[htseng 180] EARLY DEPLOYMENT RELEASE SOFTWARE (fc1)..Copyright (c) 1999 by Cisco
Systems, Inc..."
enterprises.cisco.local.lcpu.2.0 = "reload"
enterprises.cisco.local.lcpu.3.0 = "cisco"
enterprises.cisco.local.lcpu.4.0 = "ora.com"
enterprises.cisco.local.lcpu.5.0 = IpAddress: 127.45.23.1
enterprises.cisco.local.lcpu.6.0 = IpAddress: 0.0.0.0
enterprises.cisco.local.lcpu.8.0 = 131888844
enterprises.cisco.local.lcpu.9.0 = 456
enterprises.cisco.local.lcpu.10.0 = 500
enterprises.cisco.local.lcpu.11.0 = 17767568
enterprises.cisco.local.lcpu.12.0 = 0
enterprises.cisco.local.lcpu.13.0 = 0
enterprises.cisco.local.lcpu.14.0 = 104
enterprises.cisco.local.lcpu.15.0 = 600
...
```

Now let's trim the output by adding the *-Os* option, which omits the initial part of each OID:

```
$ snmpwalk -m ALL -Os cisco.ora.com public cisco
lcpu.1.0 = "..System Bootstrap, Version 11.2(17)GS2, [htseng 180] EARLY
DEPLOYMENT RELEASE SOFTWARE (fc1)..Copyright (c) 1999 by Cisco Systems, Inc..."
lcpu.2.0 = "reload"
lcpu.3.0 = "cisco"
```

```
lcpu.4.0 = "ora.com"
lcpu.5.0 = IpAddress: 127.45.23.1
lcpu.6.0 = IpAddress: 0.0.0.0
lcpu.8.0 = 131888844
lcpu.9.0 = 456
lcpu.10.0 = 500
lcpu.11.0 = 17767568
lcpu.12.0 = 0
lcpu.13.0 = 0
lcpu.14.0 = 104
lcpu.15.0 = 600
...
```

This output is a little easier to read, since it cuts off the redundant part of each OID. Let's take this command one step further:

```
$ snmpwalk -OsS cisco.ora.com public system
RFC1213-MIB::sysDescr.0 = "Cisco Internetwork Operating System Software ..IOS (tm)
GS Software (GSR-K4P-M), Version 12.0(15)S, EARLY DEPLOYMENT RELEASE SOFTWARE
(fc1)..TAC Support: http://www.cisco.com/cgi-bin/ibld/view.pl?i=support..
Copyright (c) 1986-2001 by Cisco Systems, Inc..."
RFC1213-MIB::sysObjectID.0 = OID: DTRConcentratorMIB::catProd.182
EXPRESSION-MIB::sysUpTimeInstance = Timeticks: (344626986) 39 days, 21:17:49.86
RFC1213-MIB::sysContact.0 = "O'Reilly Data Center"
RFC1213-MIB::sysName.0 = "cisco.ora.com"
RFC1213-MIB::sysLocation.0 = "Atlanta, GA"
RFC1213-MIB::sysServices.0 = 6
RFC1213-MIB::system.8.0 = Timeticks: (0) 0:00:00.00
```

This command walks the system subtree. Since the system group falls under mib-2, there is no need to use *-m ALL*; mib-2 is one of the MIBs the Net-SNMP tools load automatically. Adding *S* to the *-O* option instructs the command to prefix each line of output with the name of the MIB file; we see that each line begins with RFC1213-MIB, which is the name of the file that defines mib-2.

C.3.2 snmpget

The *snmpget* command issues a single get operation. Its syntax is:

snmpget *options hostname community objectID...*

C.3.3 snmpbulkget

SNMPv2 provides an operation called get-bulk, which is implemented by the *snmpbulkget* command. get-bulk allows you to retrieve a chunk of information in one operation, as opposed to a single get or sequence of get-next operations. The syntax of *snmpbulkget* is:

snmpbulkget -v 2c *options hostname community objectID*

-v 2c is required because get-bulk is defined by SNMP Version 2.

There is one command-specific option, *-B nonrep rep*. nonrep is the number of scalar objects that this command will return; rep is the number of instances of each nonscalar object that the command will return. If you omit this option the default values of nonrep and rep, 1 and 100, respectively, will be used.

C.3.4 snmpbulkwalk

The *snmpbulkwalk* command uses the get-bulk command sequence to retrieve parts of a MIB. This command differs from *snmpbulkget* in that it does not need the *-B* option set; it walks the entire tree until it reaches the end or retrieves all the requested objects. Its syntax is:

```
snmpbulkwalk -v 2c options hostname community objectID
```

C.3.5 snmpset

The *snmpset* command is used to change, or set, the value of a MIB object. The command looks like this:

```
snmpset options hostname community objectID type value...
```

You can provide any number of objectID/type/value triples; the command will execute set operations for all the objects you give it. type is a single-character abbreviation that indicates the datatype of the object you're setting. Table C-2 lists the valid types.

Table C-2. snmpset Object Types

Abbreviation	Type
a	IP address
b[C]	Bits
d	Decimal string
D	Double
F	Float
i	Integer
I	Signed int64
n	Null
o	Object ID
s	String
t	Time ticks

u	Unsigned integer
U	Unsigned int64
x	Hexadecimal string

[c] While the manpages show this as a valid datatype, the help output from the command does not.

C.3.6 snmptrap

To send a trap, use the *snmptrap* command. The syntax for this command is:

snmptrap *options hostname community trap parameters...*

For Version 1, the following trap parameters are required:

enterprise-oid agent trap-type specific-type uptime objectID type value...

This command is discussed in detail in Chapter 10. Each object ID/type/value triplet specifies a variable binding to be included with the trap; you may include any number of variable bindings. Note that the agent and the uptime are not optional; however, if you provide an empty string ("") as a placeholder they default to the IP address of the system sending the trap and the system's current uptime.

The parameters are simpler for Version 2 traps, largely because traps (now called notifications) are full-fledged MIB objects in their own right. The following parameters are required:

snmptrap -v 2c *options hostname community uptime trapoid objectID type value...*

C.3.7 snmpdelta

The *snmpdelta* command monitors OIDs and tracks changes in OID values over time. Its syntax is:

snmpdelta *options hostname community objectID...*

snmpdelta requires you to specify the OID of an integer-valued scalar object -- it can't monitor tables. For example, if you want to want to watch the octets arriving on an interface, you can't just specify ifInOctets; you must specify the interface number in addition to the object name (e.g., ifInOctets.3). By default, *snmpdelta* polls the given object every second.

Table C-3 lists some of the *snmpdelta*-specific options. There are many problems with the documentation for this command, but if you stick to the options listed below you should be on firm ground.

Table C-3. snmpdelta Options

Option	Description

-t	The documentation says "Determine time interval from the monitored entity." It's not clear what this means, but you seem to need this entry to get nonzero readings.
-s	Display a timestamp with every set of results.
-m	Print the maximum value obtained.
-l	Write the output to a file. The filename is in the form hostname-OID. For example, if you want to monitor the variables ifInOctets.3 and ifOutOctets.3 on the host router, the *-l* option will create two files, hostname-ifInOctets.3 and hostname-ifOutOctets.3, where the output of *snmpdelta* will be written. (Note that this output has no apparent connection to the configuration, as the documentation claims.)
-p	Specify the polling interval (the default is 1 second).
-T	Print output in tabular format.

C.3.8 snmpdf

snmpdf works exactly like the Unix *df* command, except it uses SNMP to query hosts on a network. Its syntax is:

snmpdf -Cu *options... hostname community*

The *-Cu* option tells the command to consult the Net-SNMP private MIB. The Host Resources MIB is used by default.

C.3.9 snmpgetnext

The *snmpgetnext* command uses the get-next operation to retrieve the next object from a host. For example, if you ask it to perform a get-next for ifOutOctets.4 it will retrieve the next object in the MIB tree, which will probably be ifOutOctets.5. (If the machine you're polling has only four interfaces, you'll get the next object in the MIB, whatever that happens to be. You should also be aware that there are some obscure situations that create a "hole" in the interface table, so the interface following .4 might be .6 or .7.) You can use this command to implement your own version of *snmpwalk*. The syntax is:

snmpgetnext *options... hostname community objectID...*

There are no options specific to *snmpgetnext*.

C.3.10 snmpstatus

The *snmpstatus* command retrieves status information from a host. It prints the following information:

- The IP address of the entity
- A textual description of the entity (sysDescr.0)

- The uptime of the entity (sysUpTime.0)
- The sum of received packets on all interfaces (ifInUcastPkts.* + ifInNUcastPkts.*)
- The sum of transmitted packets on all interfaces (ifOutUcastPkts.* + ifOutNUcastPkts.*)
- The number of IP input packets (ipInReceives.0)
- The number of IP output packets (ipOutRequests.0)

The syntax of *snmpstatus* is straightforward, and there are no command-specific options:

snmpstatus *options... hostname community*

C.3.11 snmptable

The *snmptable* command uses get-next commands to print the contents of a table in tabular form. Its syntax is:

snmptable *options... hostname community objectID*

The objectID must be the ID of a table (e.g., ifTable), not of an object within a table. Table C-4 lists some of the *snmptable*-specific options.

Table C-4. snmptable Options

Option	Description
-Cf F	Separate table columns with the string F. For example, *-Cf :* separates columns with a colon, which might make it easier to import the output from *snmptable* into another program.
-Cw W	Set the maximum width of the table to W. If the lines are longer than W, the table is split into sections. Since tables can have many columns, you almost certainly want to use this option.
-Ci	Prepend the index of the entry to all printed lines.
-Cb	Display a brief heading.
-Ch	Print only column headers.
-CH	Suppress column headers.

C.3.12 snmpusm

The *snmpusm* command provides simple access to the agent's *User-based Security Model* (USM) table. This is primarily used for configuring the agent's SNMPv3 features (managing users, setting and changing passphrases, etc.). This command is discussed in Appendix F.

C.3.13 snmpconf

This command is an interactive Perl script used to create and maintain the Net-SNMP configuration files, *snmp.conf* and *snmpd.conf*. Its syntax is:

```
snmpconf filename
```

filename must be either *snmp.conf* or *snmpd.conf*.

C.3.14 snmpinform

This command can be used to send an SNMPv2 trap. If you send a trap with *snmpinform*, it will wait for a response from the recipient. Note that you can send an *inform* using the *snmptrap* command if you specify *-Ci*. The options to *snmpinform* are identical to those for *snmptrap*.

C.3.15 snmptranslate

The Net-SNMP package comes with a handy tool called *snmptranslate* that translates between numerical and human-readable object names. More generally, it can be used to look up information from MIB files. Its syntax is:

```
snmptranslate options objectID
```

snmptranslate does not perform queries against any device, so it doesn't need the hostname or community parameters. Its sole purpose is to read MIB files and produce output about specific objects. Before looking at examples, it's worth noting that *snmptranslate*'s interpretations of the *-O* options are, to be kind, interesting. To speak more plainly, they're just plain wrong. The following examples show what actually happens when you use these options -- we'll leave the rationalization to you. We expect these problems to be fixed in some later version of Net-SNMP.

Let's say you want to know the enterprise OID for Cisco Systems. The following command does the trick:

```
$ snmptranslate -m ALL -IR -Of cisco
.1.3.6.1.4.1.9
```

This tells us that Cisco's enterprise OID is .1.3.6.1.4.9. Note the use of the *-IR* option, which tells *snmptranslate* to do a random-access search for an object named cisco. If you leave this option out, *snmptranslate* will fail because it will try to locate cisco under the mib-2 tree.

Let's say you want to take .1.3.6.1.4.1.9 and convert it to its full symbolic name. That's easy:

```
$ snmptranslate -m ALL -Ofn .1.3.6.1.4.1.9
.iso.org.dod.internet.private.enterprises.cisco
```

In this case, *-IR* isn't needed because we're not performing a random-access search. *-Ofn* ensures that we print the full

object ID, in symbolic (text) form. Here's what happens if we use *-Of* by itself:

```
$ snmptranslate -m ALL -Of .1.3.6.1.4.1.9
enterprises.cisco
```

As we said earlier, this is not how you'd expect *-Ofn* and *-Of* to behave. If you're writing scripts, you shouldn't count on this behavior staying the same in future versions.

Now, let's say you want to know a little bit more information about a particular object. The *-Td* option displays the object's definition as it appears in the MIB file:

```
$ snmptranslate -Td system.sysLocation
.1.3.6.1.2.1.1.6
sysLocation OBJECT-TYPE
  -- FROM               SNMPv2-MIB, RFC1213-MIB
  -- TEXTUAL CONVENTION DisplayString
  SYNTAX                OCTET STRING (0..255)
  DISPLAY-HINT          "255a"
  MAX-ACCESS            read-write
  STATUS                current
  DESCRIPTION           "The physical location of this node (e.g., 'telephone
                         closet, 3rd floor'). If the location is unknown, the
                         value is the zero-length string."
::= { iso(1) org(3) dod(6) internet(1) mgmt(2) mib-2(1) system(1) 6 }
```

-Td can save you a lot of work poking through MIB files to find an appropriate definition, particularly when combined with *-IR*. Furthermore, the last line shows you the entire object ID in both numeric and string forms, not just the object's parent. Note that the other Net-SNMP commands have an unrelated *-T* option; don't get confused. *-T* is meaningless for this command, because *snmptranslate* only looks up a local file and doesn't need to access the network.

The *-Tp* option prints an entire OID tree. The best way to understand this is to see it:

```
$ snmptranslate -Tp system
+--system(1)
   |
   +-- -R-- String    sysDescr(1)
   |        Textual Convention: DisplayString
   |        Size: 0..255
   +-- -R-- ObjID     sysObjectID(2)
   +-- -R-- TimeTicks sysUpTime(3)
   +-- -RW- String    sysContact(4)
   |        Textual Convention: DisplayString
   |        Size: 0..255
   +-- -RW- String    sysName(5)
   |        Textual Convention: DisplayString
   |        Size: 0..255
   +-- -RW- String    sysLocation(6)
   |        Textual Convention: DisplayString
   |        Size: 0..255
```

```
+-- -R-- Integer   sysServices(7)
+-- -R-- TimeTicks sysORLastChange(8)
|        Textual Convention: TimeStamp
|
+--sysORTable(9)
   |
   +--sysOREntry(1)
      |
      +-- ---- Integer   sysORIndex(1)
      +-- -R-- ObjID     sysORID(2)
      +-- -R-- String    sysORDescr(3)
      |        Textual Convention: DisplayString
      |        Size: 0..255
      +-- -R-- TimeTicks sysORUpTime(4)
               Textual Convention: TimeStamp
```

We displayed the system subtree because it's fairly short. From this output it's relatively easy to see all the objects underneath system, together with their types and textual conventions. This is a great way to see what objects are defined in a MIB, as well as their relationships to other objects. The output can be voluminous, but it's still a convenient way to get a map and figure out what objects are likely to be useful.

Appendix D. SNMP RFCs

This appendix provides a brief list of all the SNMP RFCs, along with the status of each RFC. This list (often referred to as the Standards Summary) was taken from The Simple Times, an online publication that should be familiar to anyone working with SNMP. It is used with their permission and can be found in each quarterly edition of the magazine. Please go to http://www.simple-times.org for information on how to subscribe to this free publication.

D.1 SMIv1 Data Definition Language

Full Standards:

RFC 1155 -- Structure of Management Information
RFC 1212 -- Concise MIB Definitions

Informational:

RFC 1215 -- A Convention for Defining Traps

D.2 SMIv2 Data Definition Language

Full Standards:

RFC 2578 -- Structure of Management Information
RFC 2579 -- Textual Conventions
RFC 2580 -- Conformance Statements

D.3 SNMPv1 Protocol

Full Standards:

RFC 1157 -- Simple Network Management Protocol

Proposed Standards:

RFC 1418 -- SNMP over OSI
RFC 1419 -- SNMP over AppleTalk
RFC 1420 -- SNMP over IPX

D.4 SNMPv2 Protocol

Draft Standards:

RFC 1905 -- Protocol Operations for SNMPv2
RFC 1906 -- Transport Mappings for SNMPv2
RFC 1907 -- MIB for SNMPv2

Experimental:

RFC 1901 -- Community-based SNMPv2
RFC 1909 -- Administrative Infrastructure
RFC 1910 -- User-based Security Model

D.5 SNMPv3 Protocol

Draft Standards:

RFC 2571 -- Architecture for SNMP Frameworks
RFC 2572 -- Message Processing and Dispatching
RFC 2573 -- SNMP Applications
RFC 2574 -- User-based Security Model
RFC 2575 -- View-based Access Control Model
RFC 1905 -- Protocol Operations for SNMPv2
RFC 1906 -- Transport Mappings for SNMPv2
RFC 1907 -- MIB for SNMPv2

Proposed Standards:

RFC 2576 -- Coexistence between SNMP Versions

Informational:

RFC 2570 -- Introduction to SNMPv3

Experimental:

RFC 2786 -- Diffie-Hellman USM Key Management

D.6 SNMP Agent Extensibility

Proposed Standards:

RFC 2741 -- AgentX Protocol Version 1
RFC 2742 -- AgentX MIB

D.7 SMIv1 MIB Modules

Full Standards:

RFC 1213 -- Management Information Base II
RFC 1643 -- Ethernet-Like Interface Types MIB

Draft Standards:

RFC 1493 -- Bridge MIB
RFC 1559 -- DECnet phase IV MIB

Proposed Standards:

RFC 1285 -- FDDI Interface Type (SMT 6.2) MIB
RFC 1381 -- X.25 LAPB MIB
RFC 1382 -- X.25 Packet Layer MIB
RFC 1414 -- Identification MIB
RFC 1461 -- X.25 Multiprotocol Interconnect MIB
RFC 1471 -- PPP Link Control Protocol MIB
RFC 1472 -- PPP Security Protocols MIB
RFC 1473 -- PPP IP NCP MIB
RFC 1474 -- PPP Bridge NCP MIB
RFC 1512 -- FDDI Interface Type (SMT 7.3) MIB
RFC 1513 -- RMON Token Ring Extensions MIB
RFC 1515 -- IEEE 802.3 MAU MIB
RFC 1525 -- Source Routing Bridge MIB
RFC 1742 -- AppleTalk MIB

D.8 SMIv2 MIB Modules

Full Standards:

RFC 2819 -- Remote Network Monitoring MIB

Draft Standards:

RFC 1657 -- BGP Version 4 MIB
RFC 1658 -- Character Device MIB
RFC 1659 -- RS-232 Interface Type MIB
RFC 1660 -- Parallel Printer Interface Type MIB
RFC 1694 -- SMDS Interface Type MIB
RFC 1724 -- RIP Version 2 MIB
RFC 1748 -- IEEE 802.5 Interface Type MIB
RFC 1850 -- OSPF Version 2 MIB
RFC 1907 -- SNMPv2 MIB
RFC 2115 -- Frame Relay DTE Interface Type MIB
RFC 2571 -- SNMP Framework MIB
RFC 2572 -- SNMPv3 MPD MIB
RFC 2573 -- SNMP Applications MIBs
RFC 2574 -- SNMPv3 USM MIB
RFC 2575 -- SNMP VACM MIB
RFC 2790 -- Host Resources MIB
RFC 2863 -- Interfaces Group MIB

Proposed Standards:

RFC 1611 -- DNS Server MIB
RFC 1612 -- DNS Resolver MIB
RFC 1666 -- SNA NAU MIB
RFC 1696 -- Modem MIB
RFC 1697 -- RDBMS MIB
RFC 1747 -- SNA Data Link Control MIB

RFC 1749 -- 802.5 Station Source Routing MIB
RFC 1759 -- Printer MIB
RFC 2006 -- Internet Protocol Mobility MIB
RFC 2011 -- Internet Protocol MIB
RFC 2012 -- Transmission Control Protocol MIB
RFC 2013 -- User Datagram Protocol MIB
RFC 2020 -- IEEE 802.12 Interfaces MIB
RFC 2021 -- RMON Version 2 MIB
RFC 2024 -- Data Link Switching MIB
RFC 2051 -- APPC MIB
RFC 2096 -- IP Forwarding Table MIB
RFC 2108 -- IEEE 802.3 Repeater MIB
RFC 2127 -- ISDN MIB
RFC 2128 -- Dial Control MIB
RFC 2206 -- Resource Reservation Protocol MIB
RFC 2213 -- Integrated Services MIB
RFC 2214 -- Guaranteed Service MIB
RFC 2232 -- Dependent LU Requester MIB
RFC 2238 -- High Performance Routing MIB
RFC 2266 -- IEEE 802.12 Repeater MIB
RFC 2287 -- System-Level Application Mgmt MIB
RFC 2320 -- Classical IP and ARP over ATM MIB
RFC 2417 -- Multicast over UNI 3.0/3.1 / ATM MIB
RFC 2452 -- IPv6 UDP MIB
RFC 2454 -- IPv6 TCP MIB
RFC 2455 -- APPN MIB
RFC 2456 -- APPN Trap MIB
RFC 2457 -- APPN Extended Border Node MIB
RFC 2465 -- IPv6 Textual Conventions and General Group MIB
RFC 2466 -- ICMPv6 MIB
RFC 2493 -- 15 Minute Performance History TCs
RFC 2494 -- DS0, DS0 Bundle Interface Type MIB
RFC 2495 -- DS1, E1, DS2, E2 Interface Type MIB
RFC 2496 -- DS3/E3 Interface Type MIB
RFC 2512 -- Accounting MIB for ATM Networks
RFC 2513 -- Accounting Control MIB
RFC 2514 -- ATM Textual Conventions and OIDs
RFC 2515 -- ATM MIB
RFC 2558 -- SONET/SDH Interface Type MIB
RFC 2561 -- TN3270E MIB
RFC 2562 -- TN3270E Response Time MIB
RFC 2564 -- Application Management MIB
RFC 2576 -- SNMP Community MIB
RFC 2584 -- APPN/HPR in IP Networks
RFC 2591 -- Scheduling MIB
RFC 2592 -- Scripting MIB
RFC 2594 -- WWW Services MIB
RFC 2605 -- Directory Server MIB
RFC 2613 -- RMON for Switched Networks MIB
RFC 2618 -- RADIUS Authentication Client MIB
RFC 2619 -- RADIUS Authentication Server MIB

RFC 2667 -- IP Tunnel MIB
RFC 2662 -- ADSL Line MIB
RFC 2665 -- Ethernet-Like Interface Types MIB
RFC 2668 -- IEEE 802.3 MAU MIB
RFC 2669 -- DOCSIS Cable Device MIB
RFC 2670 -- DOCSIS RF Interface MIB
RFC 2677 -- Next Hop Resolution Protocol MIB
RFC 2720 -- Traffic Flow Measurement Meter MIB
RFC 2737 -- Entity MIB
RFC 2742 -- AgentX MIB
RFC 2787 -- Virtual Router Redundancy Protocol MIB
RFC 2788 -- Network Services Monitoring MIB
RFC 2789 -- Mail Monitoring MIB
RFC 2837 -- Fibre Channel Fabric Element MIB
RFC 2851 -- Internet Network Address TCs
RFC 2856 -- High Capacity Data Type TCs
RFC 2864 -- Interfaces Group Inverted Stack MIB
RFC 2895 -- RMON Protocol Identifier Reference
RFC 2925 -- Ping, Traceroute, Lookup MIBs
RFC 2932 -- IPv4 Multicast Routing MIB
RFC 2933 -- IGMP MIB
RFC 2940 -- COPS Client MIB
RFC 2954 -- Frame Relay Service MIB
RFC 2955 -- Frame Relay/ATM PVC MIB
RFC 2959 -- Real-Time Transport Protocol MIB

Informational:

RFC 1628 -- Uninterruptible Power Supply MIB
RFC 2620 -- RADIUS Accounting Client MIB
RFC 2621 -- RADIUS Accounting Server MIB
RFC 2666 -- Ethernet Chip Set Identifiers
RFC 2707 -- Print Job Monitoring MIB
RFC 2896 -- RMON Protocol Identifier Macros
RFC 2922 -- Physical Topology MIB

Experimental:

RFC 2758 -- SLA Performance Monitoring MIB
RFC 2786 -- Diffie-Hellman USM Key MIB
RFC 2934 -- IPv4 PIM MIB

D.9 IANA-Maintained MIB Modules

Interface Type Textual Convention

ftp://ftp.iana.org/mib/iana.mib/ianaiftype.mib

Address Family Numbers Textual Convention

ftp://ftp.iana.org/mib/iana.mib/ianaaddressfamilynumbers.mib

TN3270E Textual Conventions

ftp://ftp.iana.org/mib/iana.mib/ianatn3270etc.mib

Language Identifiers

ftp://ftp.iana.org/mib/iana.mib/ianalanguage.mib

IP Routing Protocol Textual Conventions

ftp://ftp.iana.org/mib/iana.mib/ianaiprouteprotocol.mib

D.10 Related Documents

Informational:

RFC 1270 -- SNMP Communication Services
RFC 1321 -- MD5 Message-Digest Algorithm
RFC 1470 -- Network Management Tool Catalog
RFC 2039 -- Applicability of Standard MIBs to WWW Server Management
RFC 2962 -- SNMP Application Level Gateway for Payload Address Translation

Experimental:

RFC 1187 -- Bulk Table Retrieval with the SNMP
RFC 1224 -- Techniques for Managing Asynchronously Generated Alerts
RFC 1238 -- CLNS MIB
RFC 1592 -- SNMP Distributed Program Interface
RFC 1792 -- TCP/IPX Connection MIB Specification
RFC 2593 -- Script MIB Extensibility Protocol

Appendix E. SNMP Support for Perl

This appendix summarizes Mike Mitchell's SNMP_util module, which we have used in our Perl scripts throughout this book. This module is distributed with Simon Leinen's SNMP Perl module; Mike's module, together with Simon's, can make SNMP programming a snap. You can get these modules from http://www.switch.ch/misc/leinen/snmp/perl/ or http://www.cpan.org.

Perl scripts need two use statements to take advantage of the SNMP Perl module:

```
use BER;
use SNMP_Session;
```

The BER and SNMP_Session modules make up the core of Simon's package. The SNMP_util module discussed in this appendix makes using this package a little easier. It requires only one use statement:

```
use SNMP_util;
```

Mike's package uses the other two modules, so it's not necessary to include all three in your scripts.

E.1 MIB Management Routines

The following sections describe a set of routines for working with MIBs.

E.1.1 snmpmapOID( )

The MIB objects in RFC 1213 (MIB-II) and RFC 2955 (Frame Relay) are preloaded by the routines in this package. This means that you can refer to a symbolic name like sysLocation.0 rather than to its numeric OID (.1.3.6.1.2.1.1.6). The **snmpmapOID()** routine allows you to add name-OID pairs to this map. The routine is used as follows:

snmpmapOID(*text, OID, [text, OID...]*)

All the parameters are strings. text is the textual (or symbolic) name that you want to use and OID is the numeric object ID of the object to which the name refers. A single call to this routine may specify any number of name-OID pairs.

If **snmpmapOID()** fails it returns **undef**, so you can test for errors like this:

```
@return = snmpmapOID(..);
if(!@return) {
   # error
}
```

E.1.2 snmpMIB_to_OID( )

This routine takes the filename of a MIB as an argument. It reads and parses the MIB file and associates the object IDs defined by the MIB with their textual names. It returns the number of mappings it created. A return value of zero means that no mappings were created; -1 means an error occurred (i.e., it was unable to open the file). The routine is used as follows:

snmpMIB_to_OID(*filename*)

E.1.3 snmpLoad_OID_Cache( )

This routine allows you to map textual names to object IDs using a file. The file should consist of a number of lines in the form:

textual_name OID

This is much faster than calling **snmpMIB_to_OID()** because it doesn't require parsing a MIB file. The only argument to this routine is the name of the file that contains the preparsed data:

snmpLoad_OID_Cache(*filename*)

snmpLoad_OID_Cache() returns -1 if it can't open the file; a return value of 0 indicates success.

E.1.4 snmpQueue_MIB_File( )

This routine specifies a list of MIB files that will be used for mapping textual names to object IDs. If a name or OID can't be found in the internal map, each MIB file is parsed in turn until a match is found. The routine is used as follows:

snmpQueue_MIB_File(*filename, [filename]*)

E.2 SNMP Operations

The routines for performing SNMP operations correspond to the standard SNMP Version 1 operations[E] and have the following parameters in common:

[E] Simon Leinen's package supports both SNMP v1 and v2; Mike Mitchell's SNMP_util module supports only v1.

community (optional)

The community string. If no community string is specified, public is used.

host (required)

The hostname or IP address of the device you want to query.

port (optional)

The port number to which to send the query or trap. The default for all routines except snmptrap() is 161. The default for snmptrap() is 162.

timeout (optional)

The timeout in seconds; if no response is received within this period, the operation is considered to have failed and is retried. The default is 2 seconds.

retries (optional)

The number of retries before the routine returns failure. The default is 5.

backoff (optional)

The backoff value; for each successive retry, the new timeout period is obtained by multiplying the current timeout with the backoff. The default is 1.

OID (required)

The object ID or textual name of the object you are querying.

E.2.1 snmpget( )

The syntax of the snmpget() routine is:

```
snmpget(community@host:port:timeout:retries:backoff, OID, [OID...])
```

If snmpget() fails, it returns undef.

Recall that all the MIB-II objects are preloaded into this Perl module, so the following code is legal:

```
@sysDescr = snmpget("public\@cisco.ora.com", "sysDescr");
```

We did not specify any of the optional parameters (timeout, backoff, etc.); the default values will be used. This routine lets us request "sysDescr" as shorthand for sysDescr.0. When the Perl module builds its mappings of names to object IDs, it

automatically appends the trailing .0 to any scalar objects it finds. Because sysDescr is a scalar object defined by MIB-2, and because the MIB-2 objects are pre-loaded, sysDescr is mapped to .1.3.6.1.2.1.1.1.0. If you request a scalar object from a private MIB, you must append .0 to the OID.

Since one call to snmpget() can retrieve many objects, the return values are stored in an array. For example:

```
@oids = snmpget("public\@cisco.ora.com", "sysDescr", "sysName");
```

When this function call executes, the value for sysDescr will be stored in $oids[0]; the value for sysName will be stored in $oids[1]. All the routines in this package share this behavior.

E.2.2 snmpgetnext( )

The snmpgetnext() routine performs a get-next operation to retrieve the value of the MIB object that follows the object you pass to it. Its syntax is:

snmpgetnext(*community@host:port:timeout:retries:backoff, OID, [OID...]*)

If snmpgetnext() fails, it returns undef.

As with snmpget(), you can request many OIDs; the return value from snmpgetnext() is an array, with the result of each get-next operation in each successive position in the array. The array you get back from snmpgetnext() differs from the array returned by snmpget() in that the value of each object is preceded by the object's ID, in the form:

OID:*value*

This routine returns both the OID and the value because with the get-next operation you don't necessarily know what the next object in the MIB tree is.

E.2.3 snmpwalk( )

The snmpwalk() routine could easily be implemented with repeated calls to snmpgetnext(); it traverses the entire object tree, starting with the object passed to it. Its syntax is:

snmpwalk(*community@host:port:timeout:retries:backoff, OID*)

If snmpwalk() fails, it returns undef.

Unlike many of the routines in this module, snmpwalk() allows only one OID as an argument. Like the other routines, it returns an array of values; each element of the array consists of an object's ID followed by its value, separated by a colon. For example, after executing the following code:

```
@system = snmpwalk("public\@cisco.ora.com","system");
```

The contents of the array @system would be something like:

```
1.0:cisco.ora.com Cisco
2.0:1.3.6.1.4.1.0
3.0:23 days, 11:01:57
```

```
4.0:Ora Network Admin Staff
5.0:cisco.ora.com
6.0:Atlanta, GA
7.0:4
```

Note that the array doesn't include the entire object ID. We've told snmpwalk() to walk the tree starting at the system object, which has the OID .1.3.6.1.2.1.1. The first child object, and the first item in the array, is sysName, which is .1.3.6.1.2.1.1.1.0. snmpwalk() returns 1.0:cisco.ora.com because it omits the generic part of the OID (in this case, system) and prints only the instance-specific part (1.0). Similarly, the next item in the array is system.2.0, or system.sysObjectID.0 ; its value is Cisco's enterprise ID.

E.2.4 snmpset( )

The snmpset() routine allows you to set the value of an object on an SNMP-managed device. In addition to the standard arguments (hostname, community, etc.), this routine expects three arguments for each object you want it to set: the object's ID, datatype, and value. The syntax for this routine is:

```
snmpset(community@host:port:timeout:retries:backoff,
        OID, type, value, [OID, type, value...])
```

The type argument must be one of the following strings:

string

Represents the string type

int

Represents the 32-bit integer type

ipaddr

Represents the IP address type

oid

Represents the object identifier (OID) type

If snmpset() fails, it returns undef.

Performing a set from a script is straightforward. The following code sets the value of sysContact to "Joe@Ora". If the operation succeeds, snmpset() returns the new value for sysContact. If the operation fails, the fs variable is not set and snmpset() prints an error message:

```
$setResponse =
    snmpset("private\@cisco.ora.com", sysContact,"string","Joe\@Ora");
if ($setResponse) {
    print "SET: sysContact: $setResponse\n";
} else {
    print "No response from cisco.ora.com\n";
}
```

The most common reasons for an **snmpset()** to fail are that the host isn't up, the host isn't running an SNMP agent, or the community string is wrong.

E.2.5 snmptrap( )

The **snmptrap()** routine generates an SNMPv1 trap. Most of the arguments are familiar:

```
snmptrap(community@host:port:timeout:retries:backoff,
        enterpriseOID, agent, generalID, specificID,
        OID, type, value, [OID, type, value...])
```

The enterpriseOID, agent, generalID, and specificID arguments are discussed in Chapter 10. Each OID/type/value triplet defines a data binding to be included in the trap. OID is the object ID of the variable you want to send, value is the value you want to send for this object, and type is the object's datatype. type must be one of the following three strings:

string

Represents the string type

int

Represents the 32-bit integer type

oid

Represents the object identifier (OID) type

If **snmptrap()** fails, it returns **undef**. See Chapter 10 for a more detailed discussion of SNMP traps.

Appendix F. SNMPv3

Security has been the biggest weakness of SNMP since the beginning. Authentication in SNMP Versions 1 and 2 amounts to nothing more than a password (community string) sent in clear text between a manager and agent. Any security-conscious network or system administrator knows that clear-text passwords provide no real security at all. It is trivial for someone to intercept the community string, and once he has it, he can use it to retrieve information from devices on your network, modify their configuration, and even shut them down.

The Simple Network Management Protocol Version 3 (SNMPv3) addresses the security problems that have plagued both SNMPv1 and SNMPv2. For all practical purposes, security is the only issue SNMPv3 addresses; there are no other changes to the protocol. There are no new operations; SNMPv3 supports all the operations defined by Versions 1 and 2. There are several new textual conventions, but these are really just more precise ways of interpreting the datatypes that were defined in earlier versions.

This appendix provides an introduction to SNMPv3 and covers SNMPv3 configuration for a Cisco router and the Net-SNMP agent. Although SNMPv3 is not yet a full standard, a few vendors sell products with SNMPv3 support. We chose to cover two popular SNMPv3 implementations for our configuration examples.

F.1 Changes in SNMPv3

Although SNMPv3 makes no changes to the protocol aside from the addition of cryptographic security, its developers have managed to make things look much different by introducing new textual conventions, concepts, and terminology. The changes to the terminology are so radical that it's hard to believe the new terms essentially describe the same software as the old ones, but they do. However, they do differ in terms of how they relate to each other and in that they specify much more precisely the pieces that an SNMP implementation needs.

The most important change is that Version 3 abandons the notion of managers and agents. Both managers and agents are now called SNMP *entities*. Each entity consists of an SNMP engine and one or more SNMP applications, which are discussed in the following sections. These new concepts are important because they define an architecture, rather than simply defining a set of messages; the architecture helps to separate different pieces of the SNMP system in a way that makes a secure implementation possible. Let's look at what these concepts mean, starting with the RFCs that define them (Table F-1).

Table F-1. RFCs for SNMPv3

Name	Number	Status	Last Activity Date
Architecture for SNMP Frameworks	RFC 2571	Draft	April 1999
Message Processing and Dispatching	RFC 2572	Draft	April 1999
SNMP Applications	RFC 2573	Draft	April 1999
User-based Security Model	RFC 2574	Draft	April 1999
View-based Access Control Model	RFC 2575	Draft	April 1999
Protocol Operations for SNMPv2	RFC 1905	Draft	January 1996

Transport Mappings for SNMPv2	RFC 1906	Draft	January 1996
MIB for SNMPv2	RFC 1907	Draft	January 1996
Coexistence Between SNMP Versions	RFC 2576	Proposed	March 2000
Introduction to SNMPv3	RFC 2570	Informational	April 1999
Diffie-Hellman USM Key Management	RFC 2786	Experimental	March 2000

F.1.1 The SNMPv3 Engine

The engine is composed of four pieces: the Dispatcher, the Message Processing Subsystem, the Security Subsystem, and the Access Control Subsystem. The Dispatcher's job is to send and receive messages. It tries to determine the version of each received message (i.e., v1, v2, or v3) and, if the version is supported, hands the message off to the Message Processing Subsystem. The Dispatcher also sends SNMP messages to other entities.

The Message Processing Subsystem prepares messages to be sent and extracts data from received messages. A message processing system can contain multiple message processing modules. For example, a subsystem can have modules for processing SNMPv1, SNMPv2, and SNMPv3 requests. It may also contain a module for other processing models that are yet to be defined.

The Security Subsystem provides authentication and privacy services. Authentication uses either community strings (SNMP Versions 1 and 2) or SNMPv3 user-based authentication. User-based authentication uses the MD5 or SHA algorithms to authenticate users without sending a password in the clear. The privacy service uses the DES algorithm to encrypt and decrypt SNMP messages. Currently, DES is the only algorithm used, though others may be added in the future.

The Access Control Subsystem is responsible for controlling access to MIB objects. You can control what objects a user can access as well what operations she is allowed to perform on those objects. For example, you might want to limit a user's read-write access to certain parts of the mib-2 tree, while allowing read-only access to the entire tree.

F.1.2 SNMPv3 Applications

Version 3 divides most of what we have come to think of as SNMP into a number of applications:

Command generator

Generates get, get-next, get-bulk, and set requests and processes the responses. This application is implemented by a Network Management Station (NMS), so it can issue queries and set requests against entities on routers, switches, Unix hosts, etc.

Command responder

Responds to get, get-next, get-bulk, and set requests. This application is implemented by an entity on a Cisco router or Unix host. (For Versions 1 and 2, the command responder is implemented by the SNMP agent.)

Notification originator

Generates SNMP traps and notifications. This application is implemented by an entity on a router or Unix host. (For Versions 1 and 2, the notification originator is part of an SNMP agent. Freestanding utilities for generating traps are also available.)

Notification receiver

Receives traps and inform messages. This application is implemented by an NMS.

Proxy forwarder

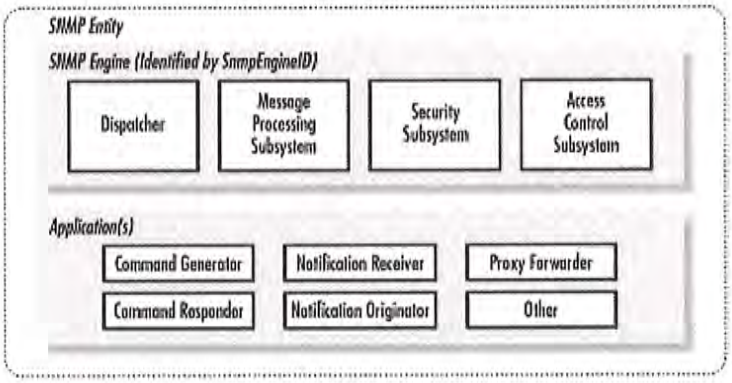
Facilitates message-passing between entities.

RFC 2571 allows additional applications to be defined over time. This ability to extend the SNMPv3 framework is a significant advantage over the older SNMP versions.

F.1.3 What Does an Entity Look Like?

Thus far we've talked about the SNMPv3 entity in terms of abstract definitions. Figure F-1 (taken from RFC 2571) shows how the components that make up an entity fit together.

Figure F-1. SNMPv3 entity



F.1.4 SNMPv3 Textual Conventions

SNMPv3 defines a number of additional textual conventions, outlined in Table F-2.

Table F-2. SNMPv3 Textual Conventions

Textual Convention	Description
SnmpEngineID	An administratively unique identifier for an SNMP engine. Objects of this type are for identification, not for addressing, even though an address can be used in the generation of a specific value. RFC 2571 provides a detailed discussion of how SnmpEngineIDs are created.
SnmpSecurityModel	An SNMP securityModel (SNMPv1, SNMPv2, or USM). USM stands for User-based Security Model, which is the security method used in SNMPv3.
SnmpMessageProcessingModel	A Message Processing Model used by the Message Processing Subsystem.
SnmpSecurityLevel	The level of security at which SNMP messages can be sent, or the level of security at which operations are being processed. Possible values are noAuthNoPriv (without authentication and without privacy), authNoPriv (with authentication but without privacy), and authPriv (with authentication and with privacy). These three values are ordered such that noAuthNoPriv is less than authNoPriv and authNoPriv is less than authPriv.
SnmpAdminString	An octet string containing administrative information, preferably in human-readable form. The string can be up to 255 bytes in length.
SnmpTagValue	An octet string containing a tag value. Tag values are preferably in human-readable form. According to RFC 2573, valid tags include acme, router, and host.
SnmpTagList	An octet string containing a list of tag values. Tag values are preferably in human-readable form. According to RFC 2573, valid examples of a tag list are the empty

	string, acme router, and host managerStation.
KeyChange	An object used to change authentication and privacy keys.

F.2 Configuring SNMPv3

Now we get to put the SNMPv3 concepts to use. We'll look at two examples: configuring a Cisco router and setting up the Net-SNMP tools on a system running Unix. The concepts are the same for both entities; the only difference is how you configure SNMPv3.

Most of the work in administering SNMPv3 has to do with managing users and their passwords. It shouldn't be surprising that the table of users, passwords, and other authentication information is just another SNMP table, called usmUser. The table's full object ID is .iso.org.dod.internet.snmpV2.snmpModules.snmpUsmMIB.usmMIBObjects usmUser ; the numeric form is .1.3.6.1.6.3.15.1.2.

F.2.1 Configuring SNMPv3 for a Cisco Router

Chapter 7 describes how to configure SNMP on a Cisco router. This section assumes that you're already familiar with IOS and that we don't have to tell you the basics, such as how to log into the router and get to privileged mode. It also assumes that you've read Chapter 7 and have configured basic SNMP on your router.

The first task in configuring SNMPv3 is to define a view. To simplify things, we'll create a view that allows access to the entire internet subtree:

```
router(config)#snmp-server view readview internet included
```

This command creates a view called readview. If you want to limit the view to the system tree, for example, replace internet with system. The included keyword states that the specified tree should be included in the view; use excluded if you wanted to exclude a certain subtree.

Next, create a group that uses the new view. The following command creates a group called readonly ; v3 means that SNMPv3 should be used. The auth keyword specifies that the entity should authenticate packets without encrypting them; read readview says that the view named readview should be used whenever members of the readonly group access the router.

```
router(config)#snmp-server group readonly v3 auth read readview
```

Now let's create a user. The following command creates a user called kschmidt, who belongs to the readonly group. auth md5 specifies that the router should use MD5 to authenticate the user (the other possibility is sha). The final item on the

command line is the user's password or passphrase, which may not exceed 64 characters.

```
router(config)#snmp-server user kschmidt readonly v3 auth md5 mysecretpass
```

This configuration uses encryption only to prevent passwords from being transferred in the clear. The SNMP packets themselves, which may contain information that you don't want available to the public, are sent without encryption and can therefore be read by anyone who has a packet sniffer and access to your network. If you want to go a step further and encrypt the packets themselves, use a command like this:

```
router(config)#snmp-server user kschmidt readonly v3 auth md5 mysecretpass \
priv des56 passphrase
```

The additional keywords on this command specify privacy (i.e., encryption for all SNMP packets), use of DES 56-bit encryption, and a passphrase to use when encrypting packets.

The encrypted passwords and passphrases depend on the engine ID, so if the engine ID changes you'll need to delete any users you have defined (with the familiar IOS *no* command), and recreate them (with *snmp-server user* commands). Why would the engine ID change? It's possible to set the engine ID on the IOS command line. You shouldn't ever need to set the engine ID explicitly, but if you do, you'll have to delete and recreate your users.

This has been the briefest of introductions to configuring SNMPv3 on a Cisco router. For more information see Cisco's documentation, which is available at http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120t/120t3/snmp3.htm.

F.2.2 Configuring SNMPv3 for Net-SNMP

Chapter 7 describes basic configuration for Net-SNMP. In this section, we discuss how to configure Net-SNMP's Version 3 features. First, we will discuss how to configure SNMPv3 by editing the *snmpd.conf* [F] files. Note that you must install OpenSSL before editing the files if you want to use either DES or SHA. OpenSSL is available from http://www.openssl.org.

> [F] There are two *snmpd.conf* files in play here: the normal */usr/share/snmp/snmpd.conf* file and the persistent */var/ucd-snmp/snmpd.conf* file. The persistent file will be discussed momentarily.

To create a user named kschmidt who has read-write access to the system subtree, add the following line to your *snmpd.conf* file:

```
rwuser  kschmidt auth system
```

To create a user with read-only access, use the command `rouser` instead of `rwuser`. The `auth` keyword requests secure

authentication, but not privacy: the SNMP packets themselves aren't encrypted. The other possibilities are **noauth** (no authentication and no privacy) and **priv** (authentication and privacy). Now add the following line to */var/ucd-snmp/snmpd.conf*:

```
createUser kschmidt MD5 mysecretpass
```

This creates an MD5 password for the user kschmidt. The password assigned to kschmidt is **mysecretpass**. To create a user with a DES passphrase in addition to an MD5 password, add the following line to */var/ucd-snmp/snmpd.conf:*

```
createUser kschmidt MD5 mysecretpass DES mypassphrase
```

If you omit **mypassphrase**, Net-SNMP sets the DES passphrase to be the same as the MD5 password. The RFCs for SNMPv3 recommend that passwords and passphrases be at least eight characters long; Net-SNMP enforces this recommendation and won't accept shorter passwords.

After making these changes, stop and restart the agent. When the agent is started, it reads the configuration file, computes secret keys for the users you have added, and deletes the **createUser** commands from the file. It then places the secret key in the configuration file. This behavior has a number of consequences. The secret key is based on the engine ID, which for Net-SNMP is based on the IP address. Therefore, you can't copy configuration files from one machine to another. Furthermore, if you change a machine's IP address, you will have to reconfigure Net-SNMP: stop the agent, edit */var/ucd-snmp/snmpd.conf*, delete any entries Net-SNMP has added for your users, add **createUser** commands to recreate your users, and start the agent again.

Now we can perform an *snmpwalk* using Version 3 authentication. The following command specifies Version 3, with the username **kschmidt**, requesting authentication without privacy using the MD5 algorithm. The password is **mysecretpass**:

```
$ snmpwalk -v 3 -u kschmidt -l authNoPriv -a MD5 -A mysecretpass \
server.ora.com
system.sysDescr.0 = Linux server 2.2.14-VA.2.1 #1 Mon Jul 31 21:58:22 PDT 2000 i686
system.sysObjectID.0 = OID: enterprises.ucdavis.ucdSnmpAgent.linux
system.sysUpTime.0 = Timeticks: (1360) 0:00:13.60
system.sysContact.0 = "Ora Network Admin"
system.sysName.0 = server
system.sysLocation.0 = "Atlanta, Ga"
system.sysServices.0 = 0
system.sysORLastChange.0 = Timeticks: (0) 0:00:00.00
system.sysORTable.sysOREntry.sysORID.1 = OID: ifMIB
...
system.sysORTable.sysOREntry.sysORUpTime.9 = No more variables left in this MIB View
```

Note that we see only objects from the system subtree, even though the command tries to walk the entire tree. This limitation occurs because we have given kschmidt access only to

the system subtree. If kschmidt tries to query a subtree he is not allowed to access, he gets the following result:

```
$ snmpwalk -v 3 -u kschmidt -l authNoPriv -a MD5 -A mysecretpass \
server.ora.com interfaces
interfaces = No more variables left in this MIB View
```

If you want privacy in addition to authentication, use a command like this:

```
$ snmpwalk -v 3 -u kschmidt -l authPriv -a MD5 -A mysecretpass -x DES -X \
mypassphrase server.ora.com
```

Remember that to use DES privacy, you must install the OpenSSL library.

F.2.2.1 Using snmpusm to manage users

The Net-SNMP utility snmpusm is used to maintain SNMPv3 users. The following command creates the user kjs by cloning the kschmidt user:

```
$ snmpusm -v 3 -u kschmidt -l authNoPriv -a MD5 -A mysecretpass localhost create \ kjs
kschmidt
```

Since kjs was cloned from kschmidt, the two users now have the same authorization, password, and passphrase. It's obviously essential to change kjs 's password. To do so, use snmpusm with the *-Ca* option. Similarly, to change the privacy passphrase, use *-Cx*. The following two commands change the password and passphrase for the new user kjs:

```
$ snmpusm -v3 -l authNoPriv -u kjs -a MD5 -A mysecretpass localhost passwd \
-Co -Ca mysecretpass mynewpass
$ snmpusm -v3 -l authPriv -u kjs -a MD5 -A mysecretpass localhost passwd \
-Co -Cx mypassphrase mynewphrase
```

There are many things to note about this seemingly simple operation:

- You must know both the password and passphrase for kschmidt to set up a new password and passphrase for kjs.
- According to the documentation, Net-SNMP allows you to clone any given user only once. It's not clear whether this means that you can create only one clone of a user or that once you have created a clone, you can't create a clone of that clone. In any case, this restriction doesn't appear to be enforced.
- snmpusm can only clone users; it can't create them from scratch. Therefore, you must create the initial user by hand, using the process described above. (This isn't quite true. snmpusm can create a user, but once you've done so you have to assign it a password by changing its previous password. So you're in a catch-22: the new user doesn't have a password, so you can't change its password.)

For the user to be written to the persistent *snmpd.conf* file, you must either stop and restart the agent or send an HUP signal to the snmpd process. This forces the agent to write the

current state of the user table to disk, so the agent can reread it upon startup. Note that *kill -9* does not produce the desired result.

The *snmpusm* command exists primarily to allow end users to manage their own passwords and passphrases. As the administrator, you may want to change your users' passwords and passphrases periodically. This is possible only if you keep a master list of users and their passwords and passphrases.

If the engine ID changes, you will have to regenerate all the usernames, passwords, and passphrases. (Remember that the engine ID depends in part on the host's IP address and therefore changes if you have to change the address.) To do this, stop the agent and edit the */var/ucd-snmp/snmpd.conf* file. Remove all the persistent usmUser entries and add new createUser commands (as described previously) for your users. A usmUser entry looks something like this:

```
usmUser 1 3 0x800007e580e134af77b9d8023b 0x6b6a7300 0x6b6a7300 NULL
.1.3.6.1.6.3.10.1.1.2 0xb84cc525635a155b6eb5fbe0e3597873
.1.3.6.1.6.3.10.1.2.2 0x1cfd8d3cadd95abce8efff7962002e24 ""
```

F.2.2.2 Simplifying commands by setting defaults

At this point you may be wondering why anyone would use SNMPv3, because the commands are so painfully long and complex that it's practically impossible to type them correctly. Fortunately, there's a way around this problem. Net-SNMP allows you to set configuration variables that the commands pick up when they execute. Create a directory in your home directory called *.snmp*, then edit the *snmp.conf* file. Add entries that look like this:

```
defSecurityName   kschmidt
defAuthType       MD5
defSecurityLevel  authPriv
defAuthPassphrase mysecretpass
defPrivType       DES
defPrivPassphrase mypassphrase
defVersion        3
```

The fields in this file are:

defSecurityName

The SNMPv3 username.

defAuthType

The authentication method (either MD5 or SHA).

defSecurityLevel

The security level for the user. Valid levels are noAuthNoPriv, authNoPriv, and authPriv.

defAuthPassphrase

Your password; must be at least eight characters long.

defPrivType

The privacy protocol to use. Only DES is supported at this time.

defPrivPassphrase

Your privacy passphrase; not needed if the security level is noAuthNoPriv or authNoPriv. Must be at least eight characters long.

defVersion

The SNMP version to use (in this case, SNMPv3).

You can also use the *snmpconf* command to set up this file. *snmpconf* prompts you for the various passwords and keywords that need to be in the file. In our opinion, it's easier to write the file yourself.

Once you've created *snmp.conf*, you can use defaults to simplify your commands. For example, the following command:

```
$ snmpwalk -v3 -u kschmidt -l authPriv -a MD5 -A mysecretpass -x DES -X \ mypassphrase
localhost
```

becomes:

```
$ snmpwalk localhost
```

These defaults apply to all Net-SNMP commands, including *snmpusm*.

F.2.2.3 Sending SNMPv3 traps with Net-SNMP

Sending an SNMPv3 trap with Net-SNMP is easy.[F] Simply run *snmptrap* with the normal SNMPv2 trap options combined with SNMPv3 options. For example:

[F] SNMPv3 traps are simply SNMPv2 traps with added authentication and privacy capabilities.

```
$ snmptrap -v3 -l authPriv -u kjs -a MD5 -A mysecretpass -x DES -X mypassphrase \
localhost '' .1.3.6.1.6.3.1.1.5.3 ifIndex i 2 ifAdminStatus i 1 ifOperStatus i 1
```

Setting the appropriate configuration options in *~/.snmp/snmp.conf* greatly reduces the complexity of the command:

```
$ snmptrap localhost '' .1.3.6.1.6.3.1.1.5.3 ifIndex i 2 ifAdminStatus i 1 \ ifOperStatus
i 1
```

F.3 Final Words on SNMPv3

While vendors have begun to support SNMPv3 in their products, keep in mind that it is still a draft standard, not a full standard. If you would like to keep track of SNMPv3 happenings, you can visit the Internet Engineering Task Force's (IETF) SNMPv3 working group site at http://www.ietf.org/html.charters/snmpv3-charter.html.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animals on the cover of *Essential SNMP* are red deer (Cervus elaphus). Male red deer, also known as stags or harts, can grow to over 400 lbs. and stand 42-54 inches tall at the shoulder. Females, or hinds, are more slightly built and usually reach a weight of only about 200 lbs. The color of the red deer's coat ranges from a warm reddish-brown in the summer to a darker grayish-brown in winter. Calves are spotted at birth, but the spots fade after about two months.

The typical family group consists of a hind, a new calf, a yearling calf, and perhaps a 2-3 year old stag. Mature stags and hinds live in separate groups for most of the year, with the hinds tending to monopolize the better, more grassy habitats. At the start of the mating season (the rut) in the early fall, the stags split up and join the females. Each eligible stag establishes a harem of up to 20 or more hinds, which he defends vigorously during the rut. During this period, which typically lasts 6-8 weeks, the stags often forego eating and can lose as much as 15% of their body mass.

Red deer are one of the most widely distributed deer species: though they are native to Europe, today they can be found everywhere from New Zealand to North America. They are herbivores, feeding mainly on rough grasses, young tree shoots, and shrubs. Forest-dwellers by nature, they can adapt easily to different climates and terrain. In many of the areas in which they were introduced red deer are commercially farmed for venison and antler velvet, which has been used in traditional Chinese medicine for over 2,000 years to treat a broad range of ailments including anemia, arthritic pain and rheumatism, kidney disorders, and stress.

Rachel Wheeler was the production editor and copyeditor for *Essential SNMP*. Colleen Gorman was the proofreader and Catherine Morris provided quality control. Sada Preisch provided production assistance. Jan Wright wrote the index.

Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with Quark™XPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout based on a series design by Nancy Priest. Neil Walls converted the files from Microsoft Word to FrameMaker 5.5.6 using tools created by Mike Sierra. The text and heading fonts are ITC Garamond Light and Garamond Book; the code font is Constant Willison. The

illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. This colophon was written by Rachel Wheeler.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.