Solstice Site/SunNet/Domain Manager
Application and Agent Development Guide

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No: 802-6109-10
Revision A, June 1996

Please
Recycle

Contents

Figures

Tables

Preface

Site/SunNet/Domain Manager™ 2.3 is comprised of software that contains services to help manage elements of a network. It provides a common platform for network management functions. It includes a Manager/Agent Services library to assist in monitoring various aspects of a network.

Who Should Read This Book

Part I of this manual describes the Manager Services library and the Application Programmer's Interface (API) that allows a programmer to use the services to request and receive reports from remote network agents. The primary audience is system designers and engineers who need to write new manager applications. Readers should be familiar with the overall Site/SunNet/Domain Manager architecture before writing a manager application.The introductory chapter of the *Administration Guide* as well as Part II of this manual should be understood to gain a view of the world from both a user's and an agent's perspective. It is also helpful to first build a simple agent for a "hands-on" understanding of the control flow between manager and agent.

Part II of this manual is intended for those C programmers and engineers who want to extend Site/SunNet/Domain Manager to manage resources not supported with the agents supplied in this product. Prospective agent writers should familiarize themselves with the Console

How This Book Is Organized

This manual is divided into three parts: Part I ”Writing Management Applications” includes the first nine chapters. Part II “Writing Agents” includes Chapters 10 through 14. Part III “Man Page Summaries” lists, with brief descriptions, the on-line man pages available for utilities, routines, and file formats for manager writers and for agent writers.

Part I—Writing Management Applications

Part I contains the following chapters:

Chapter 1, “Overview of Writing Management Applications,” provides a quick overview of the Manager/Agent Services library.

Chapter 2, “Registering for Data, Event, and Trap Reports,” describes the methods used to register your manager application to receive data and event reports.

Chapter 3, “Getting Data, Event, and Trap Reports,” suggests the algorithms to follow when getting data and event reports.

Chapter 4, “Requesting Data and Event Reports,” shows how an application asks for data and event reports. It also shows how to kill an existing request.

Chapter 5, “Setting Attribute Values,” discusses how to set attribute values.

Chapter 6, “Handling Error Reports,” details the procedures for handling error reports.

Chapter 7, “Unregistering the Application,” discusses how to unregister your application from the various data and event forwarding services.

Chapter 8, “Using the Database API Functions,” describes how to use the database API functions.

Chapter 9, “Miscellaneous Topics,” lists many issues related to writing manager applications not covered in earlier chapters.

Part II—Writing Agents

Part II contains the following chapters:

Chapter 10, “Overview of Writing Agents,” provides a high-level overview of the agent writing process.

Chapter 11, “Writing an Agent Schema,” presents the *schema,* the agent-specific portion of the Management Database.

Chapter 12, “Procedure for Writing an Agent,” details the set of steps involved in developing agent code.

Chapter 13, “Testing and Integration,” discusses how to test your agent and integrate it with this product.

Chapter 14, “Converting an Existing Application to an Agent,” offers an example of converting an existing application into an agent.

Part III—Man Page Summaries

Part III provides lists of available on-line man pages that describe each function in the Manager and Agent Services libraries. These lists, which include brief descriptions of each man page, are divided into the following appendices:

Appendix A, “Man Page Summary for Writers of Manager Applications,” lists man pages for utilities, routines, and file formats for reference when writing manager software.

Appendix B, “Man Page Summary for Writers of Agent Software,” lists man pages for utilities, routines, and file formats for reference when writing agent software.

Compatibility

See the *Site/SunNet/Domain Manager 2.3 Release Notes* that accompanies this product for definitive compatibility information.

Conventions Used in This Book

This section describes the conventions used in this book.

Compatibility-Related Conventions

All procedures and other information in this book applies to both the Solaris™ 2.x and 1.x operating environments, unless the text explicitly states otherwise.

Command Line Examples

All command line examples in this guide use the C-shell environment. If you use either the Bourne or Korn shells, refer to `sh`(1) and `ksh`(1) man pages for command equivalents to the C-shell.

What Typographic Changes and Symbols Mean

Table P-1, “Typographic Conventions,” describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password:`
<AaBbCc123>	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *<filename>*`.`
AaBbCc123	Book titles, new words or terms, or words to be emphasized	These are called *class* options. You *must* be root to do this.
Code samples are included in boxes and may display the following:		
%	UNIX C shell prompt	`system%`
$	UNIX Bourne and Korn shell prompt	UNIX Bourne and Korn shell prompt
#	Superuser prompt, all shells	Superuser prompt, all shells

Part 1— Writing Management Applications

Overview of Writing Management Applications 1

Site/SunNet/Domain Manager products are platforms upon which to develop management capabilities and explore issues surrounding the management of complex, heterogeneous environments. Site/SunNet/Domain Manager is designed for extensibility and includes a number of mechanisms to support customization. Part 1 focuses on the features for one of those areas—writing new managers.

1.1 Manager/Agent Model

The Site/SunNet/Domain Manager design is based on the manager/agent model in the Open Systems Interconnection (OSI) management framework. The manager is a process started by the user (for example, the Console). The agent is a process that collects data from the managed object and reports it to the manager. Figure 1-1 shows the manager and agent relationshipr.

The Manager/Agent Services libraries provide the management infrastructure and handle the communication services. The agent and manager need not be concerned with the underlying networking involved in their communication. The agent process need be concerned only with collecting data from the managed object. The manager and agent processes make use of the Services through Application Programming Interfaces (APIs).

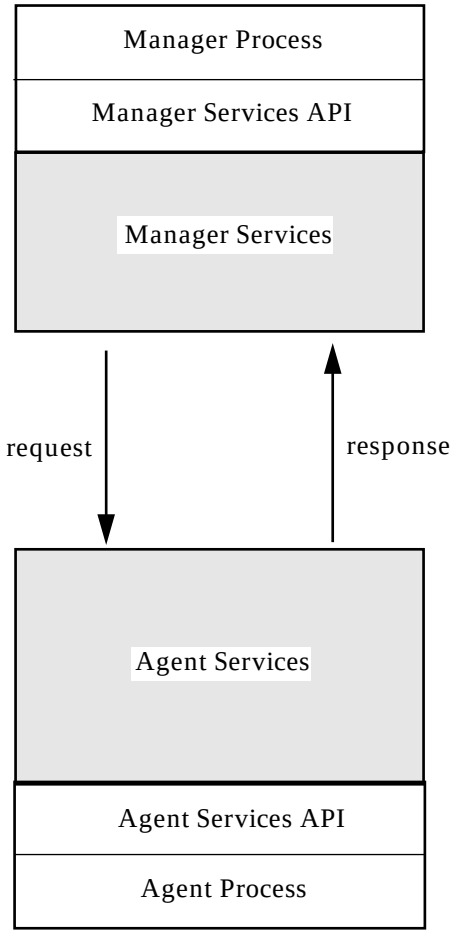

Figure 1-1 Manager / Agent Model

1.2 Manager Services

Managers can do many different kinds of things, depending on the goal of the application. These include keeping a database of network elements and their capabilities, sending and receiving requests for information, interacting with the user, and so on.

The Manager Services library provides services needed to support the manager-agent communication model. These services are provided to relieve the application writer of the burden of managing connections, encoding/decoding network data, etc. It also lays the groundwork for a standard mechanism where manager and agent applications can communicate without prior agreements.

As such, the Manager Services library simplifies the job an application programmer needs to do to send requests to agents and collect data and event reports. It does *not* help the manager support a database of network elements and requests and does not provide any support for user interaction. Manager application libraries will become available for these functions in the future.

Applications interact with the Manager Services library in several different functional areas. These areas are:

- Registering for data and event reports
- Getting data and event reports
- Requesting data and event reports, or setting attributes
- Stopping requests
- Handling error reports
- Unregistering the application
- Miscellaneous topics

Not all activities pertain to every manager application. For example, a trouble-ticketing application might only be interested in getting event reports, which would mean registering for them, getting them, and finally unregistering on application exit. In this case application will not need to start/stop requests, handle data reports, and so on.

The remaining chapters in Part 1 describe each area in detail.

1.3 Default File Locations

This section lists the default locations for files and directories that are mentioned in this guide, such as configuration and sample source code files.

1.3.1 API Examples Source Code

The default location for the example manager applications is indicated in Table 1-1.

Table 1-1 Locaion of API Example Source Code

SNM 2.3 version	Directory
for Solaris 1.x	`/usr/snm/src/API_examples`
for Solaris 2.x	`/opt/SUNWconn/snm/src/API_examples`

There is a `README` file in this directory that explains its contents.

Throughout the rest of this guide we will refer to this location as simply “the `API_examples` directory.”

1.3.2 Header Files

At various points in this guide we make reference to header files that you can include in your programs. Table 1-2 indicates the location of these files.

Table 1-2 Location of Header Files

SNM 2.3 version	Directory
for Solaris 1.x	`/usr/snm/include/netmgt`
for Solaris 2.x	`/opt/SUNWconn/snm/include/netmgt`

1.3.3 Default Locations for Various Files

In addition to the API examples and header files, a number of additional files referred to in this guide have different default directory locations in the Solaris 2.4 version than in the Solaris 1.1.1 version. Table 1-3 lists the default location for these files for both the Solaris 1.1.1 and Solaris 2.4 versions of Site/SunNet/Domain Manager 2.3.

Table 1-3 Default Locations for Various Files

Files	Default Location for Solaris 1.1.1 version	Default Location for Solaris 2.4 version	Description
`inetd.conf`	`/etc`	`/etc`	Internet servers database
`libnetmgt` library files	`/usr/snm/lib`	`/opt/SUNWconn/snm/lib`	Manager Services libraries
`na.logger`	`/usr/snm/agents`	`/opt/SUNWconn/snm/agents`	Agent logs data reports launched by `snm_cmd`
Sample agent	`/usr/snm/src/sample`	`/opt/SUNWconn/snm/src/sample`	Source code files
`rpc` file	`/etc`	`/etc`	Lists RPC numbers associated with network applications
`snm.conf`	`/etc`	`/etc/opt/SUNWconn/snm`	Configuration file used by agents and daemons
`snm.glue`	`/usr/snm/struct`	`/opt/SUNWconn/snm/struct`	Starting definitions for Console

Note – If you install any of the files mentioned in this section into non-default locations, then you will need to make appropriate modifications to the path names used in examples in this guide.

1

Registering for Data, Event, and Trap Reports

2

A management application can receive data, event, and trap reports from agents. It does not have to be able to send agent requests to receive reports from them. For data reports, the application sending the request to the agent has to specify your application as the recipient of the report. For event reports, the manager application always specifies an *event dispatcher* as the recipient of the report. Your application registers with the event dispatcher for specific kinds of event reports or traps. When the event dispatcher gets a report, it redistributes the event report or traps to all the applications it knows are interested.

Note – Agents do not know data reporting from event reporting; the facility is handled by the Agent Services library.

2.1 Background

To register for data, event, or trap reports, you must register your application's *RPC service* with the local *portmapper*, the porting utility described in the `portmap` (8) man page. This requires an RPC *program/version* number pair to identify the service. While it is possible for a manager application to have a static RPC program / version number pair, usually the application gets a temporary (or *transient*) RPC program / version number by calling `netmgt_register_callback` (3n). This function will obtain a transient service pair and register it with the portmapper.

Note – In the example program in Section 2.7, “Sample Code,” on page 2-6, you see that the protocol parameter for `netmgt_register_callback()` is set (correctly) to `IPPROTO_UDP|IPPROTO_TCP`. As a result of this, UDP transport is used for messages of 6144 bytes or less; TCP transport is used for messages longer than 6144 bytes. Because of this feature, if you registered, for example, only with UDP, you would not receive any large reports. If there is a possibility that you will receive reports larger than 6144 bytes, you should register with both TCP and UDP.

This “registration” is called the *RPC program number*, and is the basis for communication.

It is possible to register more than one function per process. For instance, you might have one function to handle data reports, another function to handle high priority event reports from the `ping` agent, and a third function to handle all other event reports. Because you register with a *particular* event dispatcher, it is possible to register with more than one at a time. You do this by calling `netmgt_register_callback` (3n) once for each callback function or for each event dispatcher.

Note – Error reports can be returned for any report request. You do not need to register for error reports. However, your manager application *must* be able to handle incoming error reports. See Chapter 3, “Getting Data, Event, and Trap Reports,” and Chapter 6, “Handling Error Reports,” for more information.

2.2 Data Reports

As described above, your application needs an RPC program number to get reports. When a manager application sends a request to an agent, it must specify your application as the *rendezvous* for you to get the resulting data report. Generally applications either direct their data reports to a cooperating rendezvous or to a general-purpose logger facility. Figure 2-1 illustrates this model.

Your application only need call `svc_run` (3n) (discussed below) to get data reports.

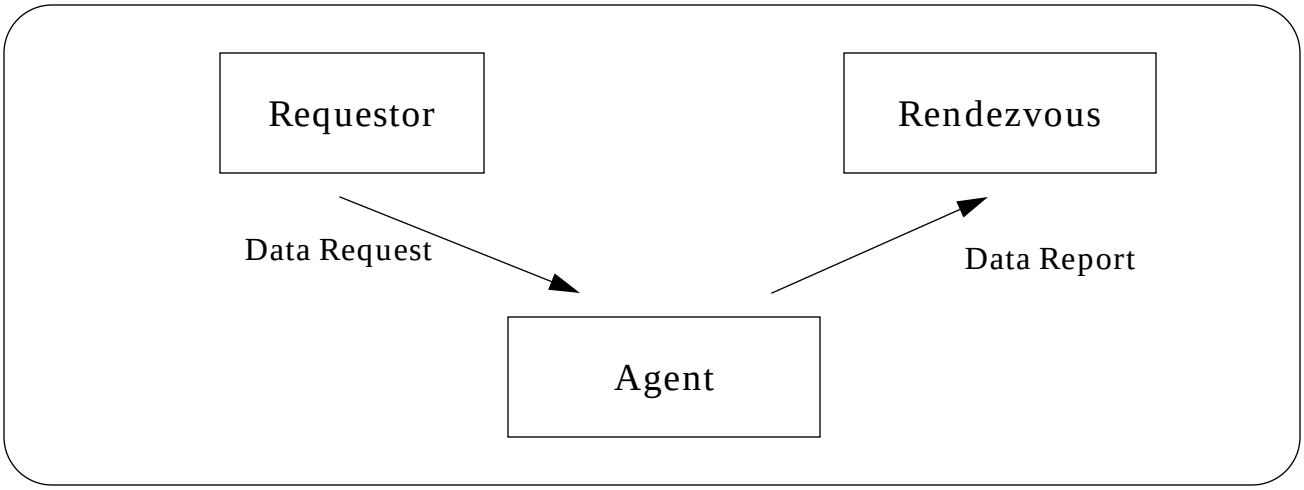

Figure 2-1 Data Reporting

2.3 *Event Reports*

Because manager applications are in general more interested in event reports, it is useful to have a central rendezvous for receiving them. The *event dispatcher* acts as a clearinghouse for event reports. When manager applications ask agents to send event reports, they should direct them to the local event dispatcher—an agent with a a known RPC program number—who will redistribute the reports to all interested processes.

To register with the event dispatcher, call `netmgt_register_rendez` (3n) with the appropriate parameters to describe the type of event reports you are interested in getting. Figure 2-2 illustrates several applications registered with the event dispatcher.

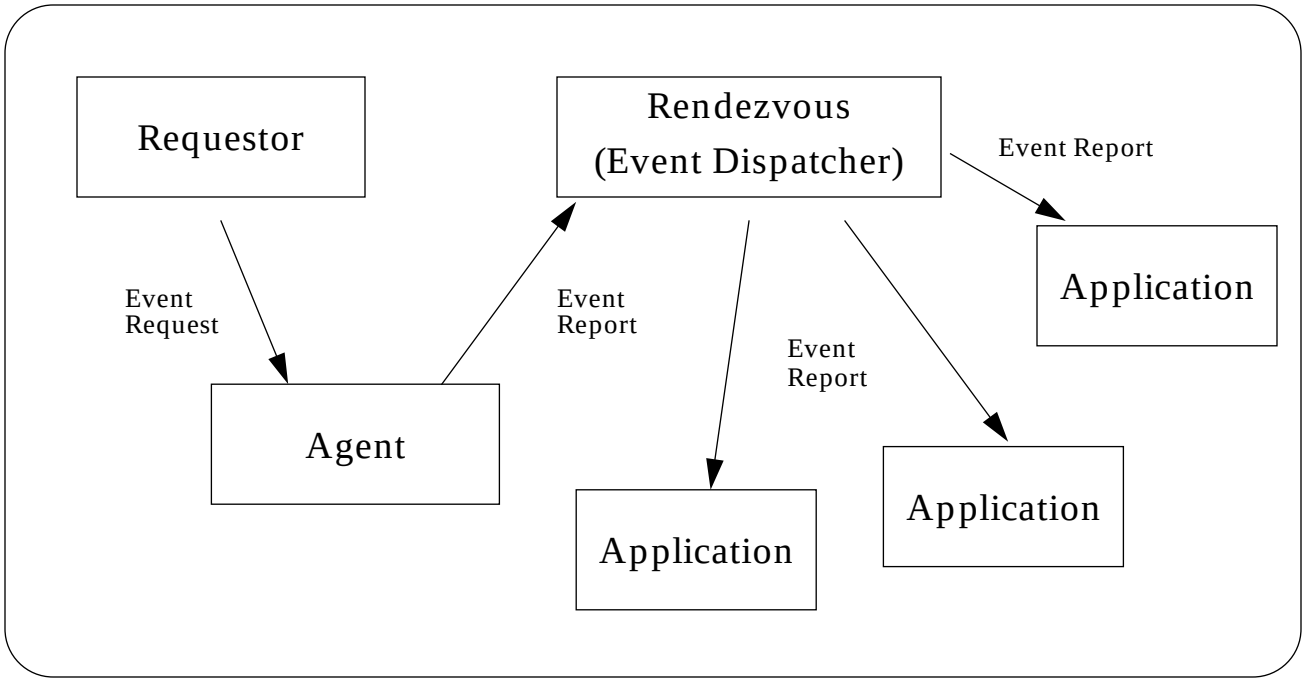

Figure 2-2 Event Reporting

2.4 *Trap Reports*

Traps are asynchronous or unrequested reports. Trap reports are sent to the event dispatcher, which then forwards traps and event reports to any registered manager application.

Trap reports can be generated by agents. For example, the SNMP trap daemon (`na.snmp-trap`) receives SNMP traps and sends them to the event dispatcher. Trap reports can also be generated by applications that use the database API functions to add, change, or delete elements in the database. In this case, low-priority trap reports are sent to the event dispatcher.

To register with the event dispatcher, call `netmgt_register_rendez` (3n) as you would to register your application to receive event reports. Figure 2-3 illustrates several applications registered with the event dispatcher for trap reports.

Note – If you only want to receive trap reports from changes in the database, specify NETMGT_DBMGR_PROG for the *agent_prog* argument in the `netmgt_register_rendez` (3n) call. No other event or trap reports will be sent from the event dispatcher.

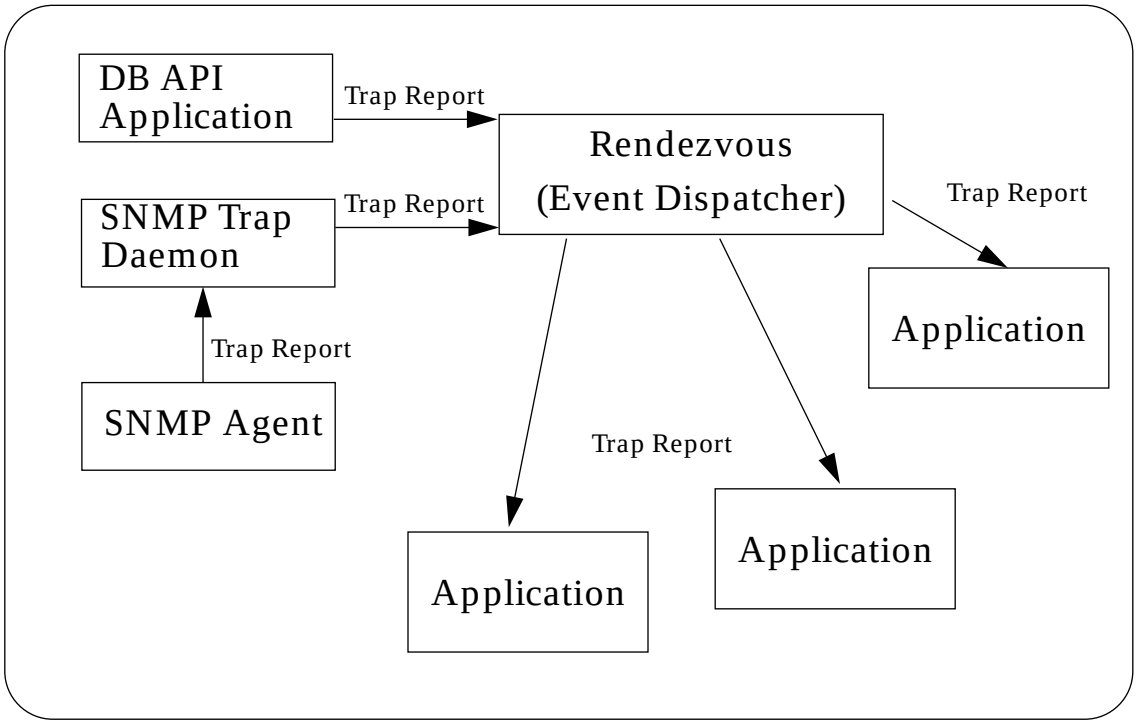

Figure 2-3 Trap Reporting

2.5 *Waiting for Reports*

Once you have registered as a rendezvous, you should call the RPC function `svc_run` (3n), which never returns. It causes your registered callback function to be called when a report arrives for your application.

2.6 Summary

1. Call `netmgt_register_callback` (3n) to get and register a transient callback RPC program / version number pair.
2. Call `netmgt_register_rendez` (3n) if you are interested in receiving event reports from the event dispatcher.
3. Call `svc_run` (3n) to wait for reports.

2.7 Sample Code

The following code fragment shows an example of registering a single function to handle both data and event reports. The complete program example from which this fragment is taken is included in the product structure in the following file `event_display.c` in the `API_examples` directory.

```
main(int argc, char *argv[])
{
        (void) signal(SIGINT, sigint);
        (void) signal(SIGTERM, sigint);

        /* get my hostname */
        gethostname(myname, MAXHOSTNAMELEN);

        /* Create a transient RPC program number using any UDP and TCP socket*/
        sock_udp = RPC_ANYSOCK;
        sock_tcp = RPC_ANYSOCK;

        /* Register the function “report_handler “ as the function to be called
         * for any report sent to the RPC program number returned in variable
         * “rendez”
         */
        if (!(rendez = netmgt_register_callback(report_handler, &sock_udp,
                &sock_tcp, NETMGT_VERS, IPPROTO_UDP | IPPROTO_TCP))) {

                (void) fprintf(stderr, “Can’t register callback: %s\n”,
                                netmgt_sperror());
                exit(CALLBACK_ERR);
        }

        /* Register the program identified by “rendez” with the local
         * event dispatcher in so the function “report_handler” will get
         * event reports too
         */
        if (!netmgt_register_rendez(myname, myname, rendez, NETMGT_VERS,
                NETMGT_ANY_AGENT, NETMGT_LOW_PRIORITY, timeout)) {

                (void) fprintf(stderr,
                        “Can’t register with the event dispatcher: %s\n”,
                        netmgt_sperror());

                netmgt_unregister_callback(rendez, NETMGT_VERS);
                exit(RENDEZ_ERR);
        }
        printf(“%s: starting\n”, argv[0]);

        /* wait for the reports to come in */
        svc_run();
```

Getting Data, Event, and Trap Reports 3

When you registered for data or event/trap reports, one parameter you specified was the function to handle incoming reports. Each time an event or data report comes in, your report-handling function is called with the following arguments:

`u_int type`
report type—either `NETMGT_DATA_REPORT`, `NETMGT_EVENT_REPORT`, `NETMGT_TRAP_REPORT`, or `NETMGT_ERROR_REPORT`.

`char *target`
name of the target system where the managed element resides.

`char *group`
name of the group whose attribute values are being reported.

`char *key`
unique row identifier for tables. An instance of the `group` on the managed system. This is used if a key was specified.

`u_int count`
the reporting count.

`timeval interval`
the reporting interval.

`u_int flags`
report flags.

The `count` and `interval` parameters are generally unused. They are provided if your application needs them.

The `group` identifies the collection of object *attributes* being reported by the agent.

3.1 Message Information

Once you get called, you'll want to find out who sent the report. Call `netmgt_fetch_msginfo`(3n). The information returned contains items like the RPC program number of the agent who sent the report and the timestamp identifying the request (more about request identification when we discuss sending requests).

An additional piece of information returned by `netmgt_fetch_msginfo`(3n) is a status code from the agent. It is usually `NETMGT_SUCCESS`, but could be `NETMGT_WARNING` or `NETMGT_FATAL` (or something else), in which case you need to find out more. Chapter 6, "Handling Error Reports," explains the procedures for handling error conditions.

Now you know who sent you the report. The `type` field tells you what type of report was sent—either `NETMGT_DATA_REPORT`, `NETMGT_EVENT_REPORT`, `NETMGT_TRAP_REPORT`, or `NETMGT_ERROR_REPORT`. The first three are discussed below. Error reports are discussed in Chapter 6, "Handling Error Reports."

3.2 Data Reports

If you received a data report, you need to cycle through the data statistics in the report by calling `netmgt_fetch_data`(3n) with a pointer to a local buffer of type `Netmgt_data`. Each time you call, your buffer will be filled in with the attribute's <*name*>, <*type*>, <*length*>, and <*value*>. The `type` is a standard C type, listed in `netmgt_arglist.h`, which describes the format of <*value*>.

Note – Agents that report tables can immediately follow `NETMGT_ENDOFROW` with `NETMGT_ENDOFARGS`. You need to check for this so you don't process a "blank" row.

The first time you call `netmgt_fetch_data`(3n) after getting a data report, you get the first data statistic in the report. Each successive call gets another data statistic. Two *sentinel* statistics are possible—if <*name*> is `NETMGT_ENDOFROW`, you've just hit the end of a row of a table. You might need this information to format your output report. The second sentinel you might get is a <*name*> of `NETMGT_ENDOFARGS`, which signals the end of the data report.

If the `flags` parameter contains the bit `NETMGT_LAST`, this is the last data report the agent is scheduled to send, and it is about to exit because it has finished its assigned task.

3.3 Event Reports

If you received an event report, cycle through the event statistics with `netmgt_fetch_event`(3n). The usage of this call is similar to `netmgt_fetch_data`(3n) except you provide it a pointer to a local buffer of type `Netmgt_event`. On return, the buffer is filled in with the usual <*name*>, <*type*>, <*length*>, and <*value*> fields, but three additional fields are present: <*relop*>, <*thresh_val*>, and <*priority*>.

Each of these three fields was set by the manager making the request. <*relop*> is the relational operator (defined in `netmgt_arglist.h`) that triggered the event on this attribute. If this attribute did not cause the event report, the field will be set to `NETMGT_NOP`. <*thresh_val*> is the value of the threshold. It has the same internal representation as <*value*>. Finally, <*priority*> is the "severity" of the report, one of `NETMGT_LOW_PRIORITY`, `NETMGT_MEDIUM_PRIORITY`, or `NETMGT_HIGH_PRIORITY`.

While cycling through the event statistics, it is possible to get the two sentinel values `NETMGT_ENDOFROW` and `NETMGT_ENDOFARGS`, just as with data reports.

If the `flags` parameter contains the bit `NETMGT_LAST`, this is the last event report the agent is scheduled to send, and it is about to exit because it has finished its assigned task.

It's possible for an agent to finish its processing but not have any events to send. You will be notified by a special event report with the `NETMGT_NO_EVENTS` bit set in the `flags` parameter.

3.4 Trap Reports

Even though you register with the event dispatcher to receive traps, traps are returned in data report format. This is because traps do not contain any threshold information. Therefore, to fetch attributes from a trap, call `netmgt_fetch_data`(3n).

The attributes returned in a trap are specific to each trap generator. SNMP is one entity that generates traps.

The attributes returned by the SNMP trap proxy consist of a fixed set of attributes, followed by an `ENDOFROW` indication, following by zero or more trap-specific attributes.

The fixed set of attributes are listed in Table 3-1.

Table 3-1 SNMP Trap Attributes

Attribute Name	Type	Description
sequence	unsigned long	sequence number of the trap
receive-time	unixtime	time trap was received by proxy
version	unsigned long	version returned by the device
community	string	community string in SNMP trap
enterprise	object id	enterprise in the SNMP trap
source time	timeticks	time since device was booted
trap type	string	name of the trap

3.5 Printing Data

Converting the attribute <*value*> to a printable entity is straightforward for most <*type*> definitions. One <*type*>, `NETMGT_OBJECT_IDENTIFIER`, is not as straightforward. This type is defined by Internet standard RFC 1065 for SNMP and by International Standards Organization for OSI, and in printed form looks like a string of numbers separated by dots.

The Manager Services library function `netmgt_oid2string`(3n) converts an object identifier—stored as an array of unsigned long integers—to a printable character string in “dot notation.” You should use this function when converting object identifiers to a printable format.

3.6 Summary

This isn't as complicated as it might seem at first. To summarize, your report handling routine needs to do the following:

1. call `netmgt_fetch_msginfo`(3n) to get additional message information.
2. call `netmgt_fetch_data`(3n) or `netmgt_fetch_event`(3n) to fetch individual attribute statistics.
3. optionally format the report.

3.7 Sample Code

The following code fragment is an example of a routine that handles event reports. Note that the print functions are not defined here: they can be found in the complete program example from which this fragment is taken, which is included in the product structure in the file `event_display.c` in the `API_examples` directory.

```
void report_handler(u_int type, char *target, char *group, char *key,
        u_int count, struct timeval interval, u_int flags)
{

Netmgt_msginfo  msginfo;                /* report message information */

        printf("report_handler: called\n");

        /* Get additional report message information. */
        if (!netmgt_fetch_msginfo(&msginfo)) {
                NETMGT_DBG("can't fetch message information: %s\n",
                    netmgt_sperror());
                return;
        }
        printf("IP address of the requesting manager : %s\n",
                inet_ntoa(msginfo.manager_addr));
        printf("IP address of the responding agent: %s\n",
                inet_ntoa(msginfo.agent_addr));

        if (flags & NETMGT_NO_EVENTS) {
                (void) printf("(last report from agent [got NETMGT_NO_EVENTS])\n");
                my_exit(NO_EVENT);
        }

        switch (type) {
        case NETMGT_ERROR_REPORT :
                print_agent_error();
                break;

        case NETMGT_DATA_REPORT:
                printf("report handler: data report\n");
                break;

        case NETMGT_TRAP_REPORT:
                printf("report handler: trap report\n");
                show_trap();
                break;

        case NETMGT_EVENT_REPORT:
                printf("report handler: event report\n");
                show_event();
                break;
```

```
        default:
                printf("report handler: unknown report\n");

        }

        if (flags & NETMGT_LAST) {
                (void) printf ("(last report from agent) \n");
                my_exit(LAST_REPORT);
        }

}

void
show_trap()
{
Netmgt_data     trap;                   /* data-report buffer */
char *s;

        strcpy(trap.name, "!a!");
        while (strcmp(trap.name, NETMGT_ENDOFARGS)) {
                if (!netmgt_fetch_data(&trap)) {
                        NETMGT_DBG1("can't fetch trap statistics\n");
                        print_agent_error();
                }

                if (!strcmp(trap.name, NETMGT_ENDOFROW)) {
                        (void) printf("(end of row) \n\n");
                        continue;
                }

                if (!strcmp(trap.name, NETMGT_ENDOFARGS)) {
                        (void) printf( "(end of report) \n\n");
                        continue;
                }
```

```
                /* print the received information */
                if (s = value2str(trap.type, trap.value, trap.length)) {
                        (void) printf (“%s = %s”, trap.name, s);
                } else {
                        (void) printf(“%s : has an UNKNOWN type \n”,
                            trap.name, trap.type);
                }

                (void) printf(“\n”);
        }

}
void
show_event()
{
Netmgt_event    event;                  /* event-report buffer */
char *s;


        strcpy(event.name, “!a!”);
        while (strcmp(event.name, NETMGT_ENDOFARGS)) {
                if (!netmgt_fetch_event(&event)) {
                        NETMGT_DBG1(“can’t fetch event statistics\n”);
                        print_agent_error();
                }

                if (!strcmp(event.name, NETMGT_ENDOFROW)) {
                        (void) printf(“end of row) \n\n”);
                        continue;
                }

                if (!strcmp(event.name, NETMGT_ENDOFARGS)) {
                        (void) printf( “(end of report) \n\n”);
                        continue;
                }

                /* print the received information */
                if (s = value2str(event.type, event.value, event.length)) {
                        (void) printf (“%s = %s”, event.name, s);
```

```
                        if (event.relop != NETMGT_NOP) {
                                s = value2str(event.type, event.thresh_val,
                                    event.length);
                                (void) printf("   (%s %s Priority %s)",
                                    relop2str(event.relop), s,
                                    prio2str(event.priority));
                        }
                } else {
                        (void) printf("%s : has an UNKNOWN type \n",
                            event.name, event.type);
                }

                (void) printf("\n");
        }

}
```

3

Requesting Data and Event Reports 4

All agents have their own permanent RPC program numbers compiled in. Managers, however, do not know (and should not assume) the agent's RPC numbers. They need to look up the RPC program numbers with a call to `getrpcbyname`(3n). When the manager sends the request to the agent, the agent is instructed where to send the resulting data or event reports. For data reports, usually the manager making the request will ask for the reports, so it uses the transient RPC program number it got when it first registered as a callback. A manager application usually asks for event reports to be sent to the local event dispatcher, so it needs to call `getrpcbyname`(3n) again to get the RPC program number of the event dispatcher before it can tell the agent where to send the reports.

4.1 Formulating a Request

The first call a manager makes in formulating an agent request is to `netmgt_set_instance`(3n). This allows the manager to set the name of the <*target*>, <*group*>, and optional <*key*> of the request. The <*target*> is the name of the object that contains the information the agent can obtain. It does not have to be a name the manager understands, since it is used only by the agent. For example, an agent that gets statistics from individual serial ports in a terminal controller might want a <*target*> name of "port17," which means nothing to the manager.

The <*group*> name is the logical collection of attributes the agent provides to the manager. When a manager application specifies the <*group*> name, all attributes in the group are returned. If a <*key*> is specified, it is also agent-

specific. Each agent's documentation will describe what attributes inside a table it uses for keys, and the method for selecting them. Again, keys mean nothing to the manager, so anything is fair game here.

Note – If your application will be using the `netmgt_kill_request2`(3n) function, you need to call `netmgt_set_manager_id`(3n) right after calling `netmgt_set_instance`(3n). See Section 4.7, "Stopping Requests," on page 4-8, for more information about `netmgt_kill_request2`(3n).

4.2 Optional Arguments

Sometimes the standard request mechanism isn't enough to get all needed information to an agent. The Manager Services library allows an "option" string to be specified. The definition of this field is up to the agent. For example, the `ping` agent uses the optional request arguments to set packet size, time to wait for echo replies, and so on, because it is not possible to specify these parameters through the "standard" request mechanism.

You can also use the option string for SNMP requests. You can specify certain options that you want the SNMP proxy agent to use when sending an SNMP request to a target device. The optional arguments are:

- `na.snmp-read-community`—community string for data and event requests
- `na.snmp-write-community`—community string for set requests
- `na.snmp-schema`—the name of the schema to be used for performing the request
- `na.snmp-proxy-device`—name of a vendor's proxy system with which the SNMP proxy agent will communicate. (If this option is used, the SNMP request is not sent to the target device, but will be sent instead to the specified proxy system. This argument should only be used only when a vendor has supplied an SNMP proxy agent to manage a particular device or set of devices. In this situation, the vendor's SNMP proxy agent communicates with theSite/ SunNe/Domain Manager SNMP proxy agent via SNMP, but communicates with the target device using either SNMP or a *different* protocol.)
- `na.snmp-get-list` (used only for data reports)—list of object identifiers in standard dot notation. Separate each object ID with a space. For each object ID in the list, the SNMP proxy agent will get the value of the instance specified by the object ID.

- `na.snmp-get-next-list` (used only for data reports)—list of object identifiers in standard dot notation. Separate each object ID with a space. For each object ID in the list, the SNMP proxy agent will get the value of the lexicographic next instance specified by the object ID. Note that a group or keyvalue should not be specified, rather, specify empty strings for those file IDs.
- `na.snmp-timeout` (number of seconds the SNMP proxy agent waits for a response from the target device)

If any optional arguments are to be specified, the manager should call `netmgt_set_argument`(3n).

The following code fragment shows how to specify an SNMP read community, timeout, and get-list as optional arguments in a request. The complete program example from which this fragment is taken is included in the product structure in the file `request.c` in the `API_examples` directory.

```
bool_t
set_snmp_args()
{
        /* argument for setting options */
        Netmgt_arg      option;

        /* read community string */
        char    *community= "community string";

        /* SNMP retry interval (in seconds) */
        int     retry = 10;

       /* OID list for instances (sysObjectID and sysUpTime) */
      char   *get_list = "1.3.6.1.2.1.1.2.0 1.3.6.1.2.1.1.3.0";


        NETMGT_DBG("setting snmp arguments \n");

        /* send SNMP read community */
        /*      not needed if read access is free
        strcpy(option.name, "na.snmp-read-community");
        option.type = NETMGT_STRING;
        option.length = strlen(community) + 1;
        option.value = community;
        if  (!netmgt_set_argument(&option)) {
                NETMGT_DBG ("netmgt_set_argument failed: %s \n",
                    netmgt_sperror());
                return (FALSE);
        }
        */


        /* send SNMP retry interval */
        strcpy(option.name, "na.snmp.retry");
        option.type = NETMGT_INT;
        option.length = sizeof (int);
        option.value = (char *) &retry;
        if  (!netmgt_set_argument(&option)) {
                NETMGT_DBG ("netmgt_set_argument failed: %s \n",
                    netmgt_sperror());
                return (FALSE);
        }
```

```
      /* send OID list to request data only for certain attributes
*/
        strcpy(option.name, "na.snmp-get-list");
        option.type = NETMGT_STRING;
        option.length = strlen(get_list) + 1;
        option.value = get_list;
        if  (!netmgt_set_argument(&option)) {
                NETMGT_DBG ("netmgt_set_argument failed: %s \n",
                    netmgt_sperror());
                return (FALSE);
        }

}
```

4.3 Count and Interval

Reports are sent according to a schedule the manager application sets. When the request is started, two parameters are specified to instruct the agent how often to report and how long to report. The two parameters, <*count*> and `interval`, will be interpreted by the agent according to this chart:

Table 4-1 Count / Interval Interpretations

Count	Interval	Interpretation
0	0	Send reports forever at an agent-specific interval
0	< *i*>	Send reports forever, every < *i*> seconds
1	0	Quick Dump — send one report
1	< *i*>	Quick Dump — send one report
<*c*>	0	Send <*c*> reports at an agent-specific interval
<*c*>	*i*	Send <*c*> reports every *i* seconds

For data reporting, agents report, sleep for <*interval*> and then report again, <*count*> times (or forever if <*count*> is zero). For event reporting, agents act the same way, but the Agent Services library prevents the report from getting back to the manager application if an event was not detected during the <*interval*>. Thus, it is possible for an agent to “send” <*count*> reports and exit without the

manager application ever seeing an event report, because no events have occurred. (However, the manager application will get a special event report with the NETMGT_NO_EVENTS flag set when the agent finishes processing the request.)

4.4 Request Flags

Managers can vary the behavior of agents via a special <*flags*> parameter at the time the request is sent. Two of these <*flags*>, NETMGT_RESTART and NETMGT_DO_DEFERRED, pertain to both data and event reports, while one of them, NETMGT_SEND_ONCE only pertains to event reports. By default, these <*flags*> are not set.

NETMGT_RESTART
: restarts the request if the agent abnormally terminates and is restarted. Otherwise this request is forgotten when the agent restarts.

  This flag is advisory; request restart is not guaranteed. The restart is not attempted until the agent parent is started (asked to start another request, or asked what requests it is working on) when all requests marked for restart will be restarted if possible.

NETMGT_DO_DEFERRED
: have the agent collect the report information but *don't send it just yet*.

  Often an agent collects useful information for debugging problems, but the information isn't useful under normal conditions. If the manager started the request only after the error condition started, it would have been started after the fact and valuable data would have been lost. On the other hand, if the request was started and the reports were continually streaming back to the manager before the error condition occurred, an unnecessary traffic and CPU load would be caused from many uninteresting reports coming back.

  With this flag, reports can be held on the agent's system until the manager is ready (if ever) to ask for them. When the manager asks via a call to netmgt_request_deferred(3n) for the collected reports, the "old" reports will be sent.

  This option (as well as netmgt_request_deferred(3n), described in its man page) is handled for the agent by the Agent Services library. The library keeps only the last 32 reports the agent "sent", memory permitting.

This option does *not* defer error reports.

`NETMGT_SEND_ONCE`
have the agent terminate after one event report has been sent. If this flag is not set, the agent continues to conditionally send event reports until otherwise directed.

To send more than one of these flags, OR them together.

4.5 Setting Thresholds

If you are about to send a request for an event report, you need to tell the agent what attributes you are interested in, and what constitutes an event. You do this by calling `netmgt_set_threshold`(3n) with a pointer to a `Netmgt_thresh` buffer, specifying the attribute name, a threshold value and the relational operator to be applied between the two. You can specify more than one relationship by making repeated calls to this function before you send the request.

Agents cannot compute inter-attribute relationships; they are only capable of reporting the attributes defined in their agent schema. While some of these attributes are derived values, the formulas for deriving them are hard-coded into the agent.

4.6 Sending the Request

You're now ready to send the request. If you are sending a request for a data report, call `netmgt_request_data`(3n). Otherwise, call `netmgt_request_events`(3n). The parameters are identical—they specify where to send the request (the host name, and RPC program and version number of the agent), where to send the reply (the host name, and RPC program and version number of the rendezvous, usually the manager making the request), the number of reports to send and how often to send them, the RPC <*timeout*> and any request <*flags*> (discussed above).

Note – Requests are identified by the agent host's IP address and the timestamp of the request.

When you send the request, you will get a reply from the Manager Services library indicating the status of the request. If the request was accepted by the agent, you will get a UNIX timestamp you can use to identify the request when you get reports or want to kill the request. If the request timestamp was `NULL`, either the agent didn't accept the request or there was an error sending the message to the agent. You can call `netmgt_fetch_error`(3n), discussed below, for more information on handling errors.

4.7 Stopping Requests

For manager applications created with the manager services libraries, there are two basic ways of killing requests:

- calling `netmgt_kill_request`(3n) tells an agent to terminate a single request. This function is supported by all versions of Site/SunNet/Domain Manager agents. To stop a single request, you call `netmgt_kill_request`(3n), specifying the host name, program and version number where the request is running, the request identification (the name of the <*manager*> who started the request and the request <*timestamp*>) and a <*timeout*> value.

  If you don't specify <*manager*>, requests started by any manager will be terminated. If you specify {0,0} for the <*timestamp*>, requests started at any time will be terminated.

- calling `netmgt_kill_request2`(3n) tells an agent to terminate all requests started by a specified manager from a given IP address. This function is *only* supported by Site/SunNet/domain Manager version 2.x agents or agents created with the Site/SunNet/Domain Manager 2.x agent services libraries.

Note – You cannot use `netmgt_kill_request2`(3n) to kill requests started by an agent created with Site/SunNet/Domain Manager version 1.x agent services libraries. Upon receiving a `netmgt_kill_request2`(3n) call, SunNet Manager 1.x agents will return `FALSE` and set the service error to `NETMGT_UNKNOWNREQUEST`. To work properly with 1.x agents, your application should call `netmgt_kill_requests`(3n), which is understood by all versions of Site/SunNet/Domain Manager agents.

To stop requests, you call `netmgt_kill_request2`(3n), specifying the host name, program and version number where the request is running, and a timeout value. Additionally, you must specify a manager identification that

uniquely identifies the manager application running on a particular host. (The manager identification is set by calling `netmgt_set_manager_id`(3n) right after calling `netmgt_set_instance`(3n), and before calling `netmgt_request_data`(3n) or `netmgt_request_events`(3n).)

4.8 Summary

Sending requests is not as complicated as it may seem. It can be summarized as follows:

1. Call `netmgt_set_instance`(3n).
2. Call `netmgt_set_manager_id`(3n) if you will be using `netmgt_kill_request2`(3n) to terminate requests.
3. Call `netmgt_set_argument`(3n) if you have optional arguments.
4. Call `netmgt_set_threshold`(3n) if you are requesting event reports.
5. Call `netmgt_request_data`(3n) or `netmgt_request_events`(3n) to start the request.
6. Call `netmgt_kill_request`(3n) or `netmgt_kill_request2`(3n) to terminate the request.

Most of the flags and options listed are not normally used; they are presented here rather than referring you to the `man` pages so you can get a more complete picture.

4.9 Sample Code

The following code fragment is an example of how to request an event report and a data report. The complete program example from which this fragment is taken is included in the product structure inthe file `request.c` in the `API_examples` directory.

```
/*
 *  request_event - send event report request to agent
 *      returns request timestamp if successful; otherwise NULL
 */
struct timeval *
request_event(count, interval, timeout, threshold, group, optstring, key, flags)
    u_int count;                /* report count */
    struct timeval interval;    /* reporting interval */
    struct timeval timeout;     /* rpc timeout */
    Netmgt_thresh *threshold;   /* event threshold */
    char *group;                /* group of statistics */
    char *optstring;            /* optstring to the  agent */
    char *key;                  /* key for table group */
    u_int flags;                /* restart */
{

        Netmgt_arg      arg;
        struct rpcent   *agent_rpc;     /* rpc entry for the agent */
        struct timeval  *timestamp;     /* request timestamp */

        u_long          agent_prog;     /* agent's RPC program number */
        u_long          rendez_prog;    /* rendez vous RPC program number */

        NETMGT_DBG("requesting events \n");


        /* declare managed object instance */
        if (!netmgt_set_instance(target, group, key)) {
                NETMGT_DBG("set_instance failed for %s: %s\n",
                    target, netmgt_sperror());
                return (struct timeval *) NULL;
        }

#ifndef SNMP_AGENT
        /* get the agent's RPC info */
        agent_rpc = getrpcbyname(agentname);
        if (!agent_rpc) {
                NETMGT_DBG("can't get RPC program number for %s\n", agentname);
                return (struct timeval *) NULL;
        }
        agent_prog = agent_rpc->r_number;
```

```
#else
        agent_prog = SNMP_RPC_NUM;
        /* set SNMP arguments */
        set_snmp_args();
#endif


        /* pass optional arguments */
        if (optstring) {
                (void) strcpy(arg.name, NETMGT_OPTSTRING);
                arg.type = NETMGT_STRING;
                arg.length = strlen(optstring) + 1;
                arg.value = optstring;
                if (!netmgt_set_argument(&arg)) {
                        NETMGT_DBG (“netmgt_set_argument failed: %s \n”,
                            netmgt_sperror());
                        return (struct timeval *) NULL;
                }
        }

        /* set threshold */
        if (!netmgt_set_threshold(threshold)) {
                NETMGT_DBG(“netmgt_set_threshold failed: %s\n”,
                    netmgt_sperror());
                return (struct timeval *) NULL;
        }

        /* send the event report to the event dispatcher */
        rendez_prog = NETMGT_EVENT_PROG;

        /* start the event report request */
        timestamp = netmgt_request_events(agent_host, agent_prog,
            NETMGT_VERS, event_host, rendez_prog, NETMGT_VERS,
            count, interval, timeout, flags);

        if (!timestamp)
                NETMGT_DBG(“Can’t request event report: %s\n”,netmgt_sperror());
        return timestamp;

}
```

```
/*
 *  request_data - send data report request to agent
 *      returns request timestamp if successful; otherwise NULL
 */
struct timeval *
request_data(count, interval, timeout, group, optstring, key, flags)
    u_int count;                /* report count */
    struct timeval interval;    /* reporting interval */
    struct timeval timeout;     /* rpc timeout */
    char *group;                /* group of statistics */
    char *optstring;            /* optstring to the  agent */
    char *key;                  /* key for table group */
    u_int flags;                /* restart */
{

        Netmgt_arg      arg;
        struct rpcent   *agent_rpc;     /* rpc entry for the agent */
        struct timeval  *timestamp;     /* request timestamp */

        u_long          agent_prog;     /* agent's RPC program number */
        u_long          rendez_prog;    /* rendez vous RPC program number */

        NETMGT_DBG("requesting data \n");


        /* declare managed object instance */
        if (!netmgt_set_instance(target, group, key)) {
                NETMGT_DBG("set_instance failed for %s: %s\n",
                    target, netmgt_sperror());
                return (struct timeval *) NULL;
        }

#ifndef SNMP_AGENT
        /* get the agent's RPC info */
        agent_rpc = getrpcbyname(agentname);
        if (!agent_rpc) {
                NETMGT_DBG("can't get RPC program number for %s\n", agentname);
                return (struct timeval *) NULL;
        }
        agent_prog = agent_rpc->r_number;
```

```
#else
        agent_prog = SNMP_RPC_NUM;
        /* set SNMP arguments */
        set_snmp_args();
#endif

        /* pass optional arguments */
        if (optstring) {
                (void) strcpy(arg.name, NETMGT_OPTSTRING);
                arg.type = NETMGT_STRING;
                arg.length = strlen(optstring) + 1;
                arg.value = optstring;
                if (!netmgt_set_argument(&arg)) {
                        NETMGT_DBG (“netmgt_set_argument failed: %s \n”,
                            netmgt_sperror());
                        return (struct timeval *) NULL;
                }
        }


        /* send the data report to the logger */
        rendez_prog = NETMGT_LOGGER_PROG;

        /* start the event report request */
        timestamp = netmgt_request_data(agent_host, agent_prog,
            NETMGT_VERS, event_host, rendez_prog, NETMGT_VERS,
            count, interval, timeout, flags);

        if (!timestamp)
                NETMGT_DBG(“Can’t request event report: %s\n”,netmgt_sperror());
        return timestamp;

}
```

Setting Attribute Values 5

In addition to sending data and event reports, some agents can change the values of managed objects (that is, set attributes values). Sending a request to set the values of one or more attributes is very similar to sending a data or event request. The basic steps are:

1. **Specify the system containing the attributes to set.**
2. **Set the values of any optional request arguments.**
3. **Specify each attribute and the value to set the attribute.**
4. **Send the set request to the agent. If the agent indicates it will be unable to set the requested attributes because the attribute is read-only or some other reason, fetch the error description and exit.**
5. **Run as an RPC callback server. When the callback function is dispatched, get the status message which indicates whether the agent succeeded in setting the requested attributes.**

5.1 Specify Target System

The first call the manager makes in formulating a set request is to `netmgt_set_instance`(3n). This allows the agent to indicate the name of the <*system*> containing the attributes to be set. Unlike data and event requests, *do not* use `netmgt_set_instance`(3n) to specify the <*group*> or <*key*> names. You set these using `netmgt_set_value`(3n).

5.2 Set Optional Arguments

Sometimes the standard request mechanism isn't enough to get all needed information to an agent. The Manager Services library allows an option string to be specified. The definition of this field is up to the agent. For example, the `ping` agent uses the optional request arguments to set packet size, time to wait for echo replies, etc, because it is not possible to specify these parameters through the "standard" request mechanism.

You can also use the option string for SNMP requests. You can specify certain options that you want the SNMP proxy agent to use when sending an SNMP request to a target device. The optional arguments are:

- `na.snmp-read-community`—community string for data and event requests
- `na.snmp-write-community`—community string for set requests
- `na.snmp-schema`—the name of the schema to be used for performing the request. Note that the specified filename must not contain a dot. For example, the filename, my.name.schema, crashes the daemon.
- `na.snmp-proxy-device`—name of a vendor's proxy system with which the SNMP proxy agent will communicate.
  If this option is used, the SNMP request is not sent to the target device, but will be sent instead to the specified proxy system. This argument should only be used only when a vendor has supplied an SNMP proxy agent to manage a particular device or set of devices. In this situation, the vendor's SNMP proxy agent communicates with the Site/SunNet/Domain Manager SNMP proxy agent via SNMP, but communicates with the target device using either SNMP or a *different* protocol.

- `na.snmp-get-list` (used only for data reports)—list of object identifiers in standard dot notation. Separate each object ID with a space. For each object ID in the list, the SNMP proxy agent will get the value of the instance specified by the object ID.
- `na.snmp-get-next-list` (used only for data reports)—list of object identifiers in standard dot notation. Separate each object ID with a space. For each object ID in the list, the SNMP proxy agent will get the value of the lexicographic next instance specified by the object ID.
- `na.snmp-timeout` (number of seconds the SNMP proxy agent waits for a response from the target device)

If any optional arguments are to be specified, the manager should call `netmgt_set_argument`(3n).

5.3 Specify Attribute Values

For each attribute value to be set, the manager should call `netmgt_set_value`(3n). This allows the manager to specify the <*group*> containing the attribute, a <*key*> if the group is a table, the <*name*> of the attribute, and a <*value*> to set the attribute. You can request attributes in more than one group to be set using one request. The attributes just need to be set-able by one agent.

5.4 Register to Receive Results

Similar to data and event requests, agents perform set operations in three steps:

1. Verify that they can perform the set request.
2. Set the requested attributes.
3. Send the set results to the manager.

Managers get a request timestamp as the return value from the call to `netmgt_request_set2`(3n). To receive the results of the set operations, managers run as callback RPC servers. When the agent sends the set results, the manager's callback service function is called. The manager gets the set results by calling `netmgt_fetch_error`(3n).

You should register your application as a temporary callback service by calling `netmgt_register_callback`(3n). This function registers a temporary (or transient) RPC service with the local portmapper which is described by `portmap`(8). Note, before exiting, the manager should unregister the RPC service by calling `netmgt_unregister_callback`(3n).

5.5 Send Set Request

You are now ready to send the set request by calling `netmgt_request_set2`(3n). When you send the request, the return value is the request timestamp if the request is successful. If the agent indicates it will not perform the request (`NULL` is returned), get the reason for the failure by calling `netmgt_fetch_error`(3n).

5.6 Get Set Results

Assuming the `netmgt_request_set2`(3n) return code indicated the agent will perform the request, run as a callback service to get the set results by calling `svc_run`(3n). When the agent sends the set results, your service dispatch function will be called with the following arguments:

`u_int type` - report type. This will be `NETMGT_SET_REPORT` indicating the report argument contains the set results.

`char *system` - name of the system where the attributes to be set reside.

`char *group` - group name (always `NULL`).

`char *key` - key name (always `NULL`).

`u_int count` - reporting count (always zero).

`struct timeval interval` - reporting interval (always zero).

`u_int flags` - report flags.

The dispatch function should call `netmgt_fetch_error`(3n) to get the set results. If the <*service_error*> is `NETMGT_SUCCESS`, the set request succeeded. Otherwise, the error buffer indicates the reason for the failure.

5.7 Sample Code

The following code fragment is an example of routines that send a set request to an SNMP agent and get the set results. The complete program example from which this fragment is taken is included in the product structure in the file `set.c` in the `API_examples` directory.

```
#include <sys/param.h>
#include <sys/types.h>
#include <netmgt/netmgt.h>

#include <rpc/rpcent.h>
#include <signal.h>

/* static functions */
static bool_t   register_results();
static void     display_results();
static void     unregister_results();

/* static data - global so exit handler can unregister RPC program */
u_long  callback_prog;          /* callback RPC program number */


char            agentname[NETMGT_NAMESIZ];      /* name of agent */

char            target[MAXHOSTNAMELEN];         /* name of target system */
char            agent_host[MAXHOSTNAMELEN];     /* agent hostname */
char            event_host[MAXHOSTNAMELEN];     /* host where the event dispatcher runs*/


/*----------------------------------------------------------------------
 * main routine
 *----------------------------------------------------------------------
 */

int
main (argc, argv)
int argc;
char ** argv;
{

        char    local_host[MAXHOSTNAMELEN];     /* local host name */
        struct timeval  timeout;                /* request RPC timeout */
        Netmgt_setval   setval;                 /* attribute buffer*/
        char    *optstring= (char *) NULL;      /* optional string */
        u_int   flags = 0;                      /* request option flags */

        char    *location = “1st floor, room 101”;
```

```
        /* sets the debug mode to print NETMGT_DBG messages */
        netmgt_debug = 1;

        timeout.tv_sec = NETMGT_TIMEOUT;
        timeout.tv_usec = 0;

        if (gethostname(local_host, sizeof(local_host)) == -1) {
                perror("gethostname");
                exit(1);
        }

        (void) strcpy(event_host, local_host);
        (void) strcpy(agent_host, local_host);
        (void) strcpy(target, local_host);

        (void) strcpy(agentname, "snmp");

        /* build set request argument */
        (void) strcpy(setval.group, "system");
        (void) strcpy(setval.key,"" );
        (void) strcpy(setval.name, "sysLocation");

        setval.type = NETMGT_STRING;
        setval.length = strlen(location) + 1;
        setval.value = location;


        /* send a set request - if it succeeds, run as a callback
         * RPC server waiting for the results
         */

        if (!do_set_request(target, &setval, &optstring, flags, timeout))
                exit(1);

        /* request was verified */
        svc_run();
        exit(1);
```

```
}
/*---------------------------------------------------------------------
 * do_set_request - send a set request and get the request verification
 * returns TRUE if successful; otherwise returns FALSE
 *---------------------------------------------------------------------
 */
bool_t
do_set_request(target, setvalp,  optstring, flags, timeout)

        char    *target;        /* target system name */
        Netmgt_setval   *setvalp;/* set request argument */
        char    *optstring;     /* optional argument string */
        u_int   flags;          /* request flags */
        struct timeval timeout; /* RPC timeout */
{
        u_long          agent_prog;     /* agent's RPC program number */
        struct rpcent *agent_rpc;       /* RPC database entry */
        Netmgt_arg      option;         /* option argument */

        /* get the agent's RPC info */
        agent_rpc = getrpcbyname(agentname);
        if (!agent_rpc) {
                NETMGT_DBG("can't get RPC program number for %s\n", agentname);
                return (FALSE);
        }
        agent_prog = agent_rpc->r_number;


        /* declare managed object instance */
        if (!netmgt_set_instance(target, (char *) NULL, (char *) NULL)) {
                NETMGT_DBG("set_instance failed for %s: %s\n",
                    target, netmgt_sperror());
                return (FALSE);
        }

#ifdef SNMP_AGENT
        /* set SNMP arguments */
        set_snmp_args();
#endif
```

```
        /* specifies the attribute to be set */
        if (!netmgt_set_value(setvalp)) {
                NETMGT_DBG("set_value failed : %s\n", netmgt_sperror());
                return (FALSE);
        }


        /* pass optional arguments */
        if (optstring) {
                (void) strcpy(option.name, NETMGT_OPTSTRING);
                option.type = NETMGT_STRING;
                option.length = strlen(optstring) + 1;
                option.value = optstring;
                if (!netmgt_set_argument(&option)) {
                        NETMGT_DBG ("netmgt_set_argument failed: %s \n",
                            netmgt_sperror());
                        return (FALSE);
                }
        }

        /* register a calback RPC program number to get results */
        if (!register_results(display_results))
                return(FALSE);


        /* and send the request ... */
        if (!netmgt_request_set2(target, agent_prog, NETMGT_VERS,
                event_host, callback_prog, NETMGT_VERS,
                timeout, flags)) {
                NETMGT_DBG("set_request failed : %s\n", netmgt_sperror());
                return (FALSE);
        }

        return(TRUE);
}

/*
 * chapter 4.2: Optional Arguments
 */
```

```
bool_t
set_snmp_args()
{
        /* argument for setting options */
        Netmgt_arg      option;

        /* write community string ( = default set during snmpd installation)*/
        char    *community= “private”;

        NETMGT_DBG(“setting snmp arguments \n”);

        /* send SNMP write community */
        strcpy(option.name, “na.snmp-write-community”);
        option.type = NETMGT_STRING;
        option.length = strlen(community) + 1;
        option.value = community;
        if  (!netmgt_set_argument(&option)) {
                NETMGT_DBG (“netmgt_set_argument failed: %s \n”,
                    netmgt_sperror());
                return (FALSE);
        }
}

/*---------------------------------------------------------------------
 * register_results - register callback function
 * returns TRUE if successful; otherwise returns FALSE
 *---------------------------------------------------------------------
 */

static bool_t
register_results(callback)
        void (*callback) ();            /* callback function pointer */
{
        int     udpSock;        /* UDP/IP socket descriptor */
        int     tcpSock;        /* TCP/IP socket descriptor */
        u_long  proto;          /* RPC transport protocol */

        NETMGT_DBG(“register_results\n”);

        udpSock = RPC_ANYSOCK;
        tcpSock = RPC_ANYSOCK;
        proto = (u_long) IPPROTO_UDP;
```

```
        if (! (callback_prog = netmgt_register_callback(callback, &udpSock,
                &tcpSock, NETMGT_VERS, proto)) ) {
                NETMGT_DBG("can't register RPC callback: %s\n",
                        netmgt_sperror());
                return (FALSE);
        }

        /* declare exit handler to clean up */
        (void) signal(SIGINT, unregister_results);
        (void) signal(SIGQUIT, unregister_results);
        (void) signal(SIGTERM, unregister_results);

        NETMGT_DBG( "registered callback: prog == %d\n", callback_prog );

        return (TRUE);
}

/*-----------------------------------------------------------------------
 * displays_results - displays results of set request and exit
 * no return
 *-----------------------------------------------------------------------
 */
static void
display_results(type, system, group, key, count, interval, flags)
        u_int   type;           /* request type */
        char    *system;        /* target system name */
        char    *group;         /* object group name */
        char    *key;           /* object group key */
        u_int   count;          /* report count */
        struct timeval  interval;       /* report interval */
        u_int   flags;          /* report flags */
{
        Netmgt_error    error;  /* error report argument */

        NETMGT_DBG("display_results\n");

        /* get error status */
        if (!netmgt_fetch_error(&error)) {
                NETMGT_DBG("can't get set report: %s\n",
                         netmgt_sperror());
                exit(1);
        }
```

```
        /* display confirmation if successful */
        if (error.service_error == NETMGT_SUCCESS) {
                NETMGT_DBG("request succeeded. \n");
                exit(0);
        }

        /* otherwise display error */
        NETMGT_DBG("request failed \n");
        NETMGT_DBG("service error code: %d\n", error.service_error);
        NETMGT_DBG("agent error code: %d ", error.agent_error);
        NETMGT_DBG("error message: %s \n", error.message);

        exit(0);
}

/*---------------------------------------------------------------------
 * unregister_results - unregister callback function before exiting
 * no return value
 *---------------------------------------------------------------------
 */

static void
unregister_results()
{
        NETMGT_DBG("unregister_mycallback\n");

        /* tell the local portmapper to unregister the callback RPC service */
        if (!netmgt_unregister_callback(callback_prog, NETMGT_VERS))
                NETMGT_DBG("can't unregister callback\n");
        exit(0);
}
```

Handling Error Reports 6

If you received an error report, you should pass a pointer to a `Netmgt_error` buffer to the function `netmgt_fetch_error`(3n), which will return your buffer with the error information. The first field of the buffer, <*service_error*> may be `NETMGT_SUCCESS`, which means no problems.

6.1 Agent-Specific Errors

If <*service_error*> is `NETMGT_WARNING` or `NETMGT_FATAL`, the agent has come across an agent-specific condition it has previously anticipated. The agent-specific error is given in <*agent_error*>. The value contained in <*agent_error*> corresponds to an error/text pair in the agent's schema file, in the `agentErrors` record. The third field in the error buffer is the <*error_message*> for specific information about where the error occurred, like the name of an interface.

You should look up the text string corresponding to the error code in the `agentErrors` agent schema record and present it to the user with the specific error given in <*error_message*>. (The agent sends back a number rather than a string so the corresponding string can be edited by the local user — for example, to change it to a different language.)

6.2 Generic Errors

A <*service_error*> larger than NETMGT_FATAL—a "generic" error code—is defined in `netmgt_errno.h`. An error like `NETMGT_RPCTIMEDOUT` indicates an error while contacting the agent. If you got this error while sending a request, you may wish to try and re-send the request. Otherwise, you can call `netmgt_sperror`(3n) to get the error message associated with <*service_error*>. Sometimes <*error_message*> contains additional information about the error.

6.3 Sample Code

The following code fragment is an example of how to handle errors. Note that the routine `retry_request` referenced in the example is a routine that you would write that retries the request.

```
Netmgt_error errbuf;
char agentMsg[NETMGT_NAMESIZ];                          /* agent message */

(void) netmgt_fetch_error(&errbuf);                     /* get error info */

if ((int)errbuf.service_error > (int)NETMGT_FATAL) {    /* generic error */
    if ((int)errbuf.service_error == (int)NETMGT_RPCTIMEDOUT)
        retry_request();                                /* retry request */
    else
        fprintf(stderr, "Generic error: %s, %s\n",
            netmgt_sperror(), errbufmessage);
} else {                                                /* agent error */
    /* get error message from agent schema */
    get_agentError(errbuf.agent_error, agentMsg);
    fprintf(stderr, "Agent %s: %s, %s\n",               /* print message */
          (errbuf.service_error == NETMGT_FATAL) ? "error" : "warning",
          agentMsg, errbuf.message);
}

return errbuf.service_error;
```

Unregistering the Application 7

Before exiting, your application must unregister from the portmappers and event dispatchers where it has previously registered. Failure to unregister with the portmapper results in old transient RPC program numbers being kept needlessly. Failure to unregister with the event dispatcher causes the dispatcher to continue to process event reports for your nonexistent process, consuming system resources.

7.1 Unregistering from the Event Dispatcher

If you have registered with the event dispatcher, always unregister the event rendezvous *before* unregistering any transient RPC program numbers. Use the call `netmgt_unregister_rendez`(3n) to tell the event dispatcher you are no longer interested in events. The event dispatcher keeps a list of its registered applications by the parameters sent to it with the `netmgt_register_rendez`(3n) call, so use the same parameters to unregister.

7.2 Unregistering the Transient RPC

If you have registered with the portmapper, call `netmgt_unregister_callback`(3n) to give back your transient RPC program number. Unlike `netmgt_unregister_rendez`(3n), the call to unregister your callback function only requires your transient RPC number and version (which is probably `NETMGT_VERS`).

If your application has a permanent registration with the portmapper you do not need to unregister your RPC program number.

7.3Sample Code

Here's an example code fragment to unregister the report handler registered earlier in the document:

```
/* Unregister the event manager rendezvous,
 * using the same parameters we registered with:
 */
if (!netmgt_unregister_rendez(myname,
        myname, rendez, NETMGT_VERS,
        NETMGT_ANY_AGENT, NETMGT_LOW_PRIORITY,
        timeout))
(void)fprintf(stderr, "Can't unregister from the event dispatcher: %s\n",
        netmgt_sperror());
(void) netmgt_unregister_callback(rendez, NETMGT_VERS);
```

Using the Database API Functions 8

This chapter describes the Site/SunNet/Domain Manager database API functions. The API functions allow you to do the following tasks:

- Open the database.
- Lock and unlock the database.
- Retrieve information about elements (components, buses, views, and connections) from the database.
- Add a new element instance to the database.
- Delete an element instance from the database.
- Modify element instances in the database.
- Loading an ASCII file into the Console.
- Saving element instances or saving the Console's runtime database into an ASCII file.

Example code segments are provided for each task. Refer to the man page for each function for a full description of syntax and returned values.

Before proceeding with this chapter, you should make sure that you have a thorough understanding of the Site/SunNet/Domain Manager database structures.

8.1 Building Your Program

The API functions are used in conjunction with a *header file* supplied with the product software. The header file `netmgt_db.h` defines the data structures for a *cluster record buffer* that is used with database operations. The cluster record

buffer is used to hold a copy of a cluster record retrieved from the database or to hold a new cluster record while it is being created. When you add a new cluster record or modify an existing cluster record, the information in the cluster record buffer is written into the database. The header file also defines errors that can be returned by API functions.

8.2 Static and Dynamic Linking

You can link your application with `libnetmgt_db` and `libnetmgt` either statically or dynamically. Dynamically is preferrable and recommended for the following reasons:

- Dynamically linked programs save disk space and main memory because they share library code at runtime.
- Shared library code can be enhanced without having to relink the applications that use it.
- Dynamically linked applications provide better compatibility. In order to ensure that your applications are compatible with future releases of this product, dynamic linking *must* be used.

8.2.1 Dynamic Linking

Dynamic linking, as shown in the samples below, is the preferrable and recommended linking method. When you dynamically link your program, you can include the following `libnetmgt` libraries:

- If you have installed the Solaris 1.1.1 version of this product:
  - `/usr/snm/lib/libnetmgt_db.so`
  - `/usr/snm/lib/libnetmgt.so`
- If you have installed the Solaris 2.4 version of this product:
  - `/opt/SUNWconn/snm/lib/libnetmgt_db.so`
  - `/opt/SUNWconn/snm/lib/libnetmgt.so`

Note – If you link with the `libnetmgt_db.so` library, the `libnetmgt.so` library must also be used, with `libnetmgt_db.so` being listed before `libnetmgt.so` in the command line. The `libnetmgt.so` library may be used without the `libnetmgt_db.so` library.

8.2.1.1 Dynamic Linking for the Solaris 2.4 Environment

To link with the libraries provided with the Solaris 2.4 version of this product, you must link with the network services library (`libnsl`) and the internationalization library (`libintl`). To *dynamically* link your application, follow this format:

```
host% cc myprog.c -o myprog -R /opt/SUNWconn/snm/lib -L /opt/SUNWconn/snm/lib -lnetmgt_db
-lnetmgt -lnsl -lintl
```

Since the Solaris 2.x version of this product does not make links in `/usr/lib`, all Site/SunNet/Domain Manager applications should link with the `-R` option, as shown above. This avoids forcing the user to set the `LD_LIBRARY_PATH` variable at runtime.

8.2.1.2 Dynamic Linking for the Solaris 1.1.1 Environment

To *dynamically* link your application with the libraries provided with the Solaris 1.1.1 version of this product, follow this format:

```
host% cc myprog.c -o myprog -L /usr/snm/lib -lnetmgt_db -lnetmgt -lnsl
```

8.2.2 Static Linking

When you statically link your program, you can include the following `libnetmgt` libraries:

- If you have installed the Solaris 1.1.1 version of this product:
  - `/usr/snm/lib/libnetmgt_db.a`
  - `/usr/snm/lib/libnetmgt.a`
- If you have installed the Solaris 2.4 version of this product:
  - `/opt/SUNWconn/snm/lib/libnetmgt_db.a`
  - `/opt/SUNWconn/snm/lib/libnetmgt.a`

Note – If you link with the `libnetmgt_db.a` library, the `libnetmgt.a` library must also be used, with `libnetmgt_db.a` being listed before `libnetmgt.a` in the command line. The `libnetmgt.a` library may be used without the `libnetmgt_db.a` library.

8.2.2.1 *Static Linking for the Solaris 2.4 Environment*

To link with the libraries provided with the Solaris 2.4 version of this product, you must link with the network services library (`libnsl`) and the internationalization library (`libintl`). To *statically* link your application, follow this format:

```
host% cc myprog.c -o myprog -L /opt/SUNWconn/snm/lib -Bstatic -lnetmgt_db -lnetmgt -lnsl
-lintl -Bdynamic -ldl
```

If you use the static method, you *must* specifically link in `libdl`.

8.2.2.2 *Static Linking in a Solaris 1.1.1 Environment*

To *statically* link your application with the libraries provided for the Solaris 1.1 version of this product, follow this format:

```
host% cc myprog.c -o myprog -L /usr/snm/lib -Bstatic -lnetmgt_db -lnetmgt -lnsl
-lintl -Bdynamic -ldl
```

If you use the static method, you *must* specifically link in `libdl`.

8.3 *Error Handling*

Most functions return `True` (1) or `False` (0) values upon completion. If a value of `False` is returned, an error has occurred. If any error occurs during the execution of a function, the external variable `snm_error` is set to the error number that indicates the cause of the error. Errors are defined in the header file `netmgt_db.h`. See the `snm_error`(3N) man page for error codes and explanations.

8.4 Opening the Database

The `snmdb_open` function is used to open the database. The database must be located in the directory specified by the `database` keyword in the `snm.conf` file. (The `snm.conf` file is located in `/etc` for the Solaris 1.1.1 version of this product, and in `/etc/opt/SUNWconn/snm` for the Solaris 2.4 version.) If there is no directory specified by the `database` keyword, then the database is located in the following directory:

- `/var/adm/snm` for the Solaris 1.1.1 version of this product
- `/var/opt/SUNWconn/snm` for the Solaris 2.4 version of this product

The database that is opened is the file `db.`<*name*>, where <*name*> is the name specified by the environment variable `SNM_NAME`, or the login user name is `SNM_NAME` is not specified. You should call `snmdb_open()` before you perform any database operations.

The following is an example of how to call `snmdb_open()`.

```
if (!snmdb_open())    exit(1);
```

8.5 Locking and Unlocking the Database

The database should be locked while you are *writing* information to the database. Locking the database is *not* necessary while you are reading information from the database or while you are performing operations on the cluster record buffer. Locking the database prevents other applications, such as the Console, from making changes to the database. If your program does not lock the database, you should make sure that there are no other active applications that can make changes to the database.

Note – While the database is locked, Console users cannot perform any operations (such as sending and viewing requests) from the Console. Therefore, you should minimize the time that the database is locked. `snmdb_lock` allows you the option of displaying a clock on the Console while the database is locked.

The `snmdb_lock()` and `snmdb_unlock()` functions lock and unlock the database. Call `snmdb_lock()` to lock the database before you make changes to the database. After the database updates are done, you should call `snmdb_unlock()` to release the lock.

Note – The lock and unlock functions cause low-priority trap reports to be generated when elements are added, changed, or deleted in the database. These trap reports are sent to the event dispatcher, which then forwards the reports to registered management applications. See Section 2.4, “Trap Reports,” on page 2-4, for more information.

The following is an example of how to call `snmdb_lock()` and `snmdb_unlock()`.

```
if (snmdb_lock(TRUE, TRUE)) {
   ......
   /* update the database record(s) */
   ......
   if (!snmdb_unlock())
     printf("Error: unable to unlock the database.\n");
}
else
   printf("Error: unable to lock the database. error = %d\n",
          snm_error);
```

8.6 Retrieving Element Information from the Database

The functions used to retrieve database information fall into two sets:

- Functions that retrieve information for one element.
- Functions that retrieve the names of one or more elements of a given type in one or more views.

Each of these sets of functions is described in the following sections.

Note – The examples shown in the following sections are program fragments only. A complete program example is included in the product structure in the file `view_db.c` in the `API_examples` directory.

8.6.1 Retrieving Information for a Single Element

To retrieve information about a single element from the database, you must first read the element cluster record into the cluster record buffer by calling `snmdb_read`. You can then retrieve any of the records contained in an element's cluster record. See the *Administration Guide* for information on cluster records. For agent and membership (view) records, you can either retrieve an enumerated list of the agent or view records or you can retrieve a specific agent or view record for the element.

Note – A call to `snmdb_read` initializes the cluster record buffer; therefore, you do not need to clear the buffer before reading subsequent cluster records into the internal buffer.

To retrieve a piece of element information for a single element, do the following:

1. Call `snmdb_read()`. Before you try to get any element information, you must call `snmdb_read()` to load the element record into the cluster record buffer.

2. Call one or more of the following functions:

```
snmdb_get_element_type()
snmdb_get_property()
snmdb_get_color()
snmdb_get_agent()
snmdb_get_view()
snmdb_enumerate_agents()
snmdb_enumerate_connections()
snmdb_enumerate_views()
```

3. For enumerated lists (agents, connections, or views), call `snmdb_free_list()` to free the storage allocated for the enumerated information.

The following examples assume that an element named “andrew” is defined in the database with the following information:

```
record component.ss1 (          # sparcstation 1
  string[32] Name
  string[40] IP_Address
  string[40] User
  string[40] Location
  string[80] Description
)

cluster(
  component.ss1 ( andrew 129.144.44.81 "Alice Wang" "Building
  #14" "SunNet Manager Engineer" )
  glyphColor ( 255 221 0 )
  agent ( diskinfo )
  agent ( etherif )
  agent ( hostif )
  agent ( hostmem )
  agent ( iproutes )
  proxy ( concord-mib"" )
  proxy ( hostperf"" )
  proxy ( ippath"" )
  membership ( SunNetMgr 170 120 0 )
  membership ( Home 150 120 0 )
  connect ( swap )
  connect ( golden )
)
```

In the header file `netmgt_db.h`, the data type of the cluster record buffer is defined as:

```
typedef struct {
        unsigned char data[SNMBUFFERSIZE];
} snmdb_buffer;
```

The following code segment shows how to retrieve information on the element “andrew:”

```
snmdb_buffer buf;       /*cluster record buffer of element */
int red, green, blue;   /* color values */
int isproxy;    /* indicator - whether agent is proxy */
int x, y;               /* x, y coordinates in a view */
char name[128];         /* name buffer */

/* first, we load the element record into the cluster record
   buffer for ‘andrew’ */
if (!snmdb_read("andrew", &buf)) { /* read error */
  printf("snmdb_read() failed, error = %d\n", snm_error);
  return;
}

/* get the type of the element, in this case, it should
   return the string "component.ss1" */
printf("The type of element ‘andrew’ is %s\n",
       (char *)snmdb_get_element_type(&buf));

/* get the value of property ‘User’, ‘Location’, and
   ‘Description’ */
printf("For element ‘andrew’: ‘User’ field = %s\n",
       snmdb_get_property(&buf, "User", 0).db_string);
printf("        ‘Location’ field = %s\n",
       snmdb_get_property(&buf, "Location", 0).db_string);
printf("        ‘Description’ field = %s\n",
       snmdb_get_property(&buf, "Description", 0).db_string);

/* get the color of it */
/* colors are represented in RGB numbers, snmdb_get_color()
   will return the RGB numbers of the element glyph */
/* in this case, red = 255, green = 221, blue = 0 */
if (snmdb_get_color(&buf, &red, &green, &blue))
  printf("Color values are: %d, %d, %d\n", red, green, blue);
else
  printf("There is no color defined for element.\n");

/* get agent information for agent ‘hostperf’ */
if (snmdb_get_agent(&buf, "hostperf", &isproxy, name)) {
  if (isproxy)
    printf("hostperf is proxy, proxy name = %s\n", name);
```

```
  else
    printf("hostperf is not proxy\n");
}
else
  printf("agent 'hostperf' is not running on element.\n");

/* get the location of element in view 'SunNetMgr' */
if (snmdb_get_view(&buf, "SunNetMgr", &x, &y, NULL, NULL)) {
  printf("Location of 'andrew' in view 'SunNetMgr' is :\n");
  printf(" x = %d, y = %d\n", x, y);
}
else
  printf("Element is not in view 'SunNetMgr', error = %d\n",
         snm_error);
```

The following code segment shows how to get all the agents that apply to the target element, how to get all the element names that the target element is connected to, and how to get all the views in which the target element exists:

```
snmdb_buffer buf;            /* buffer of element */
char **names;                /* pointer to the enumerated list */
char **str;                  /* temp. pointer */

/* first, we load the element record into the cluster record
   buffer for 'andrew' */
if (!snmdb_read("andrew", &buf)) { /* read error */
  printf("snmdb_read() failed, error = %d\n", snm_error);
  return;
}

/* get all the agent names that run on (or apply to)
   'andrew' */
if ((names = snmdb_enumerate_agents(&buf)) == NULL) {
  printf("Error: unable to enumerate agents, error = %d\n",
         snm_error);
  return;
}
/* got the enumerated list successfully, print the list */
for (str = names; *str; str++)
```

```
  printf("agent ‘%s’ found\n", *str);
/* remember to free the space used */
snmdb_free_list(names);

/* get all the element names that ‘andrew’ is connected to */
if ((names = snmdb_enumerate_connections(&buf)) == NULL) {
  printf("Error: unable to enumerate connections, error = %d\n",
        snm_error);
  return;
}
/* got the enumerated list successfully, print the list */
for (str = names; *str; str++)
  printf("connected to ‘%s’\n", *str);
/* remember to free the space used */
snmdb_free_list(names);

/* get all the view names that ‘andrew’ exists in */
if ((names = snmdb_enumerate_views(&buf)) == NULL) {
  printf("Error: unable to enumerate views, error = %d\n",
         snm_error);
  return;
}/* got the enumerated list successfully, print the list */
for (str = names; *str; str++)
  printf("element exists in view ‘%s’\n", *str);
/* remember to free the space used */
snmdb_free_list(names);
```

8.6.2 Retrieving Elements of a Given Type

To retrieve the names of elements of a given type in a particular view or in all views, do the following:

1. Call `snmdb_enumerate_elements()` to specify which element to enumerate in a particular view or in all views. This function also initializes storage for the list of elements.
2. Call `snmdb_get_next_element()` multiple times to retrieve the names of the elements in the list.
3. Call `snmdb_free_enumeration_handle()` to free the storage allocated for the enumerated element names.

The following code segment shows how to retrieve all element names in all views:

```
snmdb_handle *handle;     /* pointer to the enumerated list */
char *str;                /* temp. pointer */

if (!(handle = snmdb_enumerate_elements(NULL, NULL))) {
  printf("No element in any view\n");
  return;
}

printf("Elements of all types in all views are:\n");
while (str = snmdb_get_next_element(handle))
  printf("element = %s\n", str);
printf("end of list\n");

/* remember to free the space */
snmdb_free_enumeration_handle(handle);
```

The following code segment shows how to retrieve all element names of a given type in one particular view.

```
snmdb_handle *handle;    /* pointer to the enumerated list */
char *str;               /* temp. pointer */

if (!(handle =
      snmdb_enumerate_elements("SunNetMgr", "component.ss1"))) {
  printf("No element of type ‘component.ss1’ exists in view.\n");
  return;
}

printf("Elements of type ‘component.ss1’ in view
  ‘SunNetMgr’:\n");
while (str = snmdb_get_next_element(handle))
  printf("element = %s\n", str);
printf("end of view\n");

/* remember to free the space */
snmdb_free_enumeration_handle(handle);
```

8.7 Adding a New Element Instance into the Database

You can add new element instances to the database. To add a new element instance, you first initialize the cluster record buffer, then create one or more records that make up the element cluster record. Then, you write the cluster record from the buffer to the database. Note that in the cluster record you *must* define at least *one* view in which the element will reside. Otherwise, an error is returned and the element is not added to the database.

Note – The element <*type*> for the element you are adding must be defined in a schema file; there is no API for adding a new element <*type*>.

Note – See the *Administration Guide* for information on adding a new element types.

To add a new element into the database, do the following:

1. Call `snmdb_init_buffer()`. You need to call `snmdb_init_buffer()` to initialize the cluster record buffer for the new element.
2. Call one or more of the following functions:

```
snmdb_set_property()
snmdb_set_color()
snmdb_add_agent()
snmdb_add_connection()
snmdb_add_to_view()
```

3. Call `snmdb_add()` to add the new element to the database. You can optionally lock the database by calling `snmdb_lock()` *before* calling `snmdb_add()`. If the element is being added to a view that is currently displayed by the Console, call `snmdb_lock()` with its first parameter set to `TRUE`. This causes the Console to update its display when the element is added to the database. (Otherwise, the Console user must leave the view and then return to the view to see the new element.) Remember to unlock the database with `snmdb_unlock()`.

The following code segment shows how to add a new element “andrew”. A complete program example is included in the product structure in the file `add_db.c` in the `API_examples` directory.

```
snmdb_buffer buf;         /* buffer of element */

/* initialize the cluster record buffer of the element */
if (!snmdb_init_buffer("andrew", "component.ss1", &buf)) {
  printf("snmdb_init_buffer() failed, error = %d\n",
         snm_error);
  return;
}

/* set the property */
if (!snmdb_set_property(&buf, "User", "test machine") ||
    !snmdb_set_property(&buf, "Location", "Building 14")) {
  printf("snmdb_set_property() failed, error = %d\n",
         snm_error);
  return;
}

/* set the color of the element */
snmdb_set_color(&buf, 100, 220, 0);

/* add a non-proxy and a proxy agent */
if (!snmdb_add_agent(&buf, "hostif", NULL) ||
     !snmdb_add_agent(&buf, "hostperf", "andrew")) {
  printf("snmdb_add_agent() failed, error = %d\n",
         snm_error);
  return;
}

/* add a connection to another element */
/* Note: the element you want to connect to should be a
         valid element name that exists in the database */
if (!snmdb_add_connection(&buf, "swap")) { /* error */
  printf("snmdb_add_connection() failed, error = %d\n",
         snm_error);
  return;
}
```

```
/* add the element ‘andrew’ into the view ‘SunNetMgr’ */
if (!snmdb_add_to_view(&buf, "SunNetMgr", 10, 50, 0, 0, 0)) {
  printf("snmdb_add_to_view() failed, error = %d\n",
         snm_error);
  return;
}
/* now we’re ready to add the element to the database */
snmdb_lock (TRUE, FALSE);
if (!snmdb_add(&buf)) { /* error */
  printf("snmdb_add() failed, error = %d\n", snm_error);
  snmdb_unlock();
  return;
}
/* element added successfully */
snmdb_unlock();
```

8.8 Deleting an Element Instance from the Database

The function to delete an element instance operates directly on the database; the cluster record buffer is not used. You delete an element using `snmdb_delete()`. Note that you will not be able to delete an element if its subview is not empty.

If an element is being deleted from a view that is currently displayed by the Console, call `snmdb_lock()` with its first parameter set to `TRUE`. This causes the Console to update its display when the element is deleted. (Otherwise, the Console user must leave the view and then return to the view to see the element deleted.) Remember to unlock the database with `snmdb_unlock()`.

The following code segment shows how to delete the element “andrew” from the database. A complete program example is included in the product structure in the file `del_db.c` in the `API_examples` directory.

```
if (!snmdb_delete("andrew")) { /* unable to delete it */
  if (snm_error == SNMDB_SUBVIEW_IS_NOT_EMPTY)
    printf("Error: element’s subview is not empty.\n");
  else
    printf("Error: unable to delete element, error = %d\n",
           snm_error);
}
/* element is deleted successfully */
```

8.9 Modifying an Element in the Database

To modify an element instance, the element’s cluster record is first read into the cluster record buffer. After modifications are made to the records in the cluster record, the revised cluster record is written from the buffer back to the database.

To modify an element, do the following:

1. Call `snmdb_read()`. Before you try to change any element information, you must call `snmdb_read()` to load the element record into the cluster record buffer.

2. Call one or more of the following functions:

   `snmdb_set_property()`
   `snmdb_add_agent()` or `snmdb_delete_agent()`
   `snmdb_set_color()` or `snmdb_delete_color()`
   `snmdb_add_connection()` or `snmdb_delete_connection()`
   `snmdb_add_to_view()` or `snmdb_delete_from_view()`

3. Call `snmdb_update()` to write the updated element back to the database. You can optionally lock the database by calling `snmdb_lock()` *before* calling `snmdb_update()`. If the element is being modified in a view that is currently displayed by the Console, call `snmdb_lock()` with its first parameter set to `TRUE`. This causes the Console to update its display when the element is modified. (Otherwise, the Console user must leave the view and then return to the view to see the element modifications.) Remember to unlock the database with `snmdb_unlock()`.

Note that you can make multiple changes to an element at a time. The changed information is written to the cluster record buffer. `snmdb_update()` writes the contents of the buffer to the database.

The following code segment shows how to update the element “andrew”. A complete program example is included in the product structure in the file `mod_db.c` in the `API_examples` directory.

```
snmdb_buffer buf;/* cluster record buffer of element */

/* first, we load the element record into the cluster record
   buffer for element ‘andrew’ */
if (!snmdb_read("andrew", &buf)) { /* read error */
  printf("snmdb_read() failed, error = %d\n", snm_error);
  return;
}

/* modify the property value */
if (!snmdb_set_property(&buf, "User", "Admin System")) {
  printf("snmdb_set_property() failed, error = %d\n",
         snm_error);
  return;
}

/* add the element ‘andrew’ into the view ‘SunNetMgr’ */
if (!snmdb_add_to_view(&buf, "SunNetMgr", 10, 50, 0, 0, 0)) {
  printf("snmdb_add_to_view() failed, error = %d\n",
         snm_error);
  return;
}

/* delete the element ‘andrew’ from view ‘Engineering’ */
if (!snmdb_delete_from_view(&buf, "Engineering")) {
  printf("snmdb_delete_from_view() failed, error = %d\n",
         snm_error);
  return;
}

/* delete the connection between ‘andrew’ and ‘swap’ */
if (!snmdb_delete_connection(&buf, "swap")) {
  printf("snmdb_delete_connection() failed, error = %d\n",
         snm_error);
  return;
}
```

```
/* delete the agent record which describes 'hostif' runs
   on the element */
if (!snmdb_delete_agent(&buf, "hostif")) {
  printf("snmdb_delete_agent() failed, error = %d\n",
         snm_error);
  return;
}
/* delete the color record, use default color */
snmdb_delete_color(&buf);

/* now we write the updated element back to the database */
if (!snmdb_update(&buf)) {
  printf("snmdb_update() failed, error = %d\n", snm_error);
  return;
}
/* element is updated successfully */
```

8.10 Saving Database Records to an ASCII File

You can save an element cluster record that has been read into the cluster record buffer by calling `snmdb_save_element()`. Note that because this function only operates on a single element at a time and it does not save request instances, it is *not* equivalent to using the Save option in the Console's File menu. See Section 8.12, "Saving the Runtime Database to an ASCII File," if you want to save the entire database.

You can save all the element instances in a database to an ASCII file by enumerating through all the elements in the database, then reading and saving each element. This is shown in the example below.

```
/* Open the output file. */
if (!(fptr = fopen(outfile, “w”))) {
    printf(“Cannot open %s for writing.\n”, outfile);
    exit(1);
}

/* Enumerate through all the elements in the database. */
if (!(handle = snmdb_enumerate_elements(NULL, NULL))) {
  printf(“No element found in the database.\n”);
  exit(1);
}

/* Loop through the element records and save them into output
file. */
while (name = snmdb_get_next_element(handle)) {
  if (!snmdb_read(name, &buf))
   printf(“snmdb_read() failed, skipping element ‘%s’\n”, name);
  if (!snmdb_save_element(&buf, fptr))
    printf(“snmdb_save_element() failed, snm_error = %d\n”,
snm_error);
}

/* Free the enumerator handle. */
snmdb_free_enumeration_handle(handle);
```

8.11 Loading a Database File into the Console

You can load a database file into a Console session by calling `snmdb_console_load()`. Database load is accomplished through an RPC call to the Console. Note that the Console must be running. See the `snmdb_console_load`(3N) man page for information about this function.

8.12 Saving the Runtime Database to an ASCII File

You can have the Console save all element and request instances in its runtime database to a specified file in ASCII MDB form by calling `snmdb_console_save_components()`. Saving the runtime database is

accomplished through an RPC call to the Console. This call has a three parameters: the path name of the file in which the MDB is to be saved, a pointer to an integer for returned RPC error codes, and a pointer to an integer for returned Console errors. Note that the Console must be running. See the `snmdb_console_save_component`(3N) man page for information about this function.

Miscellaneous Topics 9

Since management applications can cover a wide range of uses, it is not always easy to present manager-writing topics in a cohesive fashion. This chapter attempts to gather various hints and recommendations related to writing manager applications not covered by the previous chapters. While many of these subjects touch on issues not directly related to the Manager Services library, they are things manager writers may need to know.

This chapter does not discuss high-level topics such as user-interface strategies, data reduction, analysis and presentation, and so on.

9.1 The Agent Schema

The agent schema is a file that describes the types and names of the object attributes an agent can get. Attributes are collected into *groups* for convenience, such as “input”, “output,” and so on. The agent schema is “built in” to most agents. The UNIX file manager applications use is the published interface specifications for the agent, written by the agent writer. The manager application uses this specification to know what information the agent can return from the managed objects. While it is not necessary for managers to use the agent schema to send requests and collect reports from agents, it is the supported method for agents to publish their available management information.

When a request is sent to an agent, the agent examines the request for the group and attribute names it understands so it can do the requested operation. If it does not understand the names in the request, it will reply with an error message.

The agent schema is contained in a regular UNIX file in ASCII format. The format is straightforward so manager applications—whether standard UNIX tools or custom applications—can spend as little time as possible parsing this file. The format of the agent schema is listed in `snm_schema`(5).

`snm_parser`(1) is an example application that parses schema files; the parser program is provided in source form. See the source program for more information.

9.2 Agent Identification

As objects change, so does the information they can provide. Since agents describe the information they can obtain via the *agent schema*, it is important for a manager to ensure the versions of the agent and schema file are equivalent, so the manager has an accurate picture of the agent's information base.

The Manager Services library function `netmgt_request_agent_ID`(3n) sends a request to the agent for any identifying information it can provide. The information returned includes the name of the agent, agent schema serial number, and the architecture of the machine where it is running. Not all data is always returned. For instance, the host architecture information may not be available.

The manager application can use this information to compare the agent schema serial number with the serial number the agent says represents its information base.

9.3 Security

From the manager application's standpoint, there's not much to do. Security is handled through the Manager / Agent Services library by:

- having agents run with Secure RPC. Each agent on each system can run at a variety of security levels. The actual security levels are determined by the administrator of the system where the agents run.

- having security *netgroups* that list users by security level.
- having the manager applications run by users in the appropriate security netgroups.

For more information on security, see the *Administration Guide*.

9.4 Dispatching Incoming RPC Calls

After you have registered a transient callback (with `netmgt_register_callback`(3n)) and optionally registered it with the event dispatcher (with `netmgt_register_rendez`(3n)), you need to pass control to the RPC libraries so incoming RPC calls are dispatched to your code. Normally, you do this by calling the procedure `svc_run`(3n). However, since `svc_run` never returns, it may be too restrictive for your management application. This section describes two alternatives to using `svc_run`.

If you are writing a manager that runs under XView, you can request that the XView notifier handle all the incoming RPC dispatching for you. To do this, call the XView procedure as follows:

```
notify_enable_rpc_svc(TRUE);
```

before you call `xv_main_loop`. Then, any incoming RPC calls will be dispatched to your code by the XView notifier. Note that the function `notify_enable_rpc_svc(TRUE)` is not documented in the *XView Programming Manual*.

If you are not writing an XView application and `svc_run` is too restrictive, you can use a `select`(2) loop to "wait for work." To use the select loop, you must build a mask consisting of the interesting file descriptors: both yours and any used by the incoming RPCs. To aid in building this mask, the RPC libraries export a global variable that has all the incoming RPC file descriptors set. The global variable is:

```
fd_set svc_fdset
```

When select returns, you should determine if any of the file descriptors indicate that an RPC operation requires service, and if so, call the routine `svc_getreqset()`. Refer to the man pages for more information on using `svc_fdset`(3n) and `svc_getreqset`(3n).

9.5 Blocking RPCs in XView

If your XView application makes requests, you may wish to fork a process to make the requests so that your application will not block if the request times out. Blocking occurs only when the request to an agent is made. There is no blocking while results are being received from an agent.

9.6 Agents and Managers that Generate Traps

If your agent or manager wishes to generate traps, you must fork a separate UNIX process to generate the traps. In addition, if you wish to send traps to multiple systems, each system requires a separate process.

9.7 Summary

You now have the knowledge to take advantage of the facilities in the Manager Services library. While the library does not provide every amenity possible for manager application writers, it does provide a straightforward means of using the services to communicate with agents and other Site/SunNet/Domain Manager applications.

Part 2— Writing Agents

Overview of Writing Agents 10

Site/SunNet/Domain Manager products are platforms upon which to develop management capabilities and explore issues surrounding the management of complex, heterogeneous environments. It also provides a set of tools for managing those networks.

Site/SunNet/Domain Manager includes a set of *agents* for managing Sun computers as well as standards-based mechanisms for managing products from other vendors. This set is not exhaustive. The Site/SunNet/Domain Manager agents are intended as examples of the kinds of management one can do as well as providing useful functionality in and of themselves.

Site/SunNet/Domain Manager is designed for extensibility and includes a number of mechanisms to support customization. This section focuses on the features for one of those areas—writing new agents. It is intended for Site/SunNet/Domain Manager users who have requirements beyond those served by the agents included in this release.

10.1 Manager-Agent Model

The Site/SunNet/Domain Manager design is based on the manager/agent model in the Open Systems Interconnection (OSI) management framework. The manager is a process started by the user (for example, the Console). The agent is a process that collects data from the managed object and reports it to the manager. Figure 10-1 shows the manager and agent relationship.

The Manager / Agent Services libraries provide the management infrastructure and handle the communication services. The agent and manager need not be concerned with the underlying networking involved in their communication. The agent process need be concerned only with collecting data from the managed object. The manager and agent processes make use of the Services through Application Programming Interfaces (APIs).

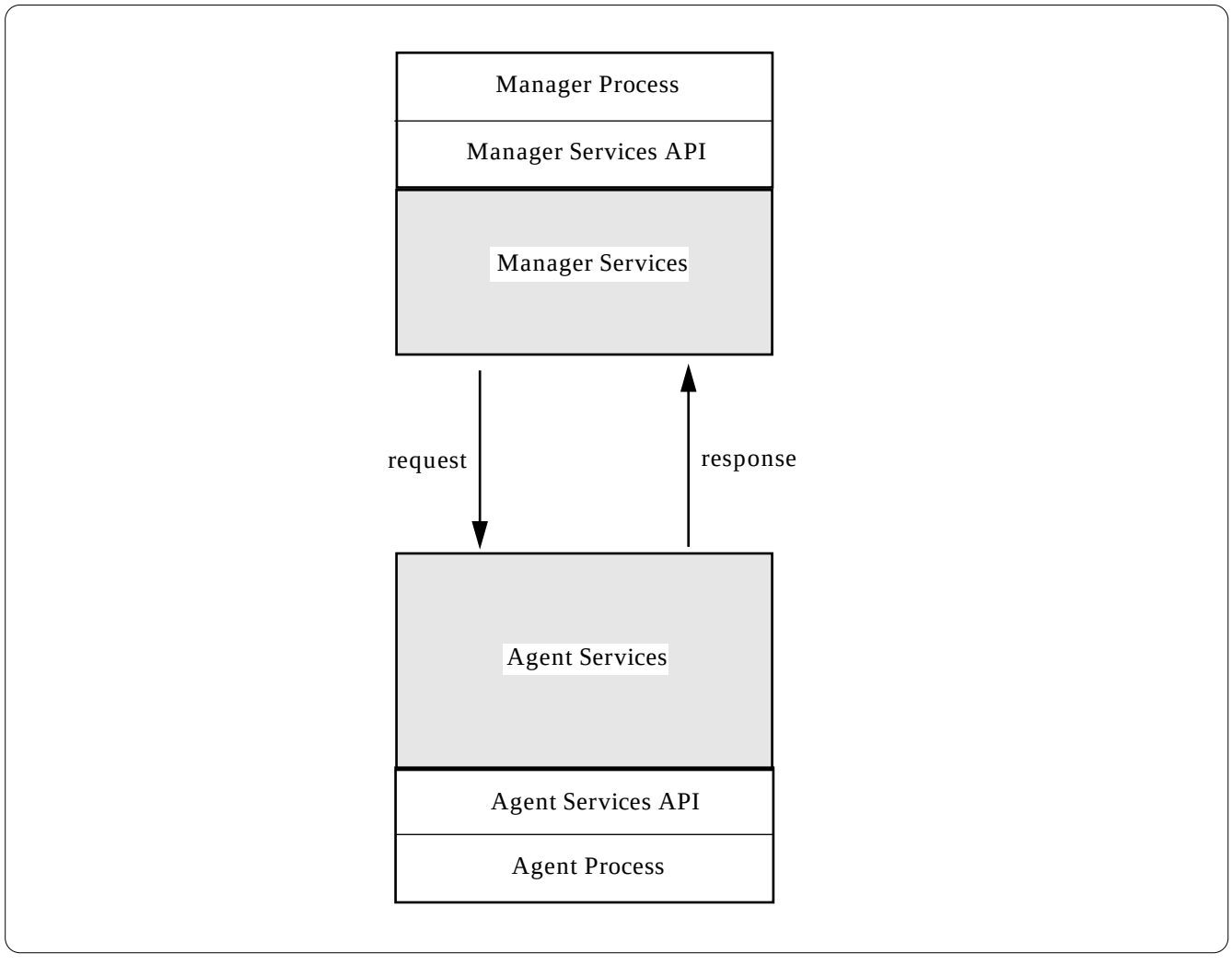

Figure 10-1 Manager Agent Model

Another aspect of the manager/agent model involves the definition of management data. Emerging open management standards (for example, OSI and the Simple Network Management Protocol (SNMP)) specify that agents abstract the properties (or attributes) of managed objects into data items (for example, "how busy a CPU is" becomes a value between 0 and 100). In Site/SunNet/Domain Manager, the attributes for a managed object are described in a portion of the Management Database (MDB)—the agent *schema*. The agent is able to respond to the manager's request, because both use the same data definitions for the managed object.

See the introductions to the *Administration Guide* for a discussion of the overall architecture and how the agent fits in.

10.2 Types of Agents

All Site/SunNet/Domain Manager agents communicate with the manager in the manner just described. Agent types differ in the relationship with their respective managed objects.

Agents can directly or indirectly access managed objects. Most of the agents provided with Site/SunNet/Domain Manager manage objects on the Sun workstations where they are installed (for example, the hostmem agent uses the same mechanism as `netstat -m` to get memory utilization data).

The second type of agent provides the ability to manage objects that are not directly accessible. Such agents are called *proxy agents*. Proxy agents run on Sun workstations, called *proxy systems*, and use protocol translation mechanisms to provide the necessary access to the managed objects. The proxy system may be the workstation on which the Console is running or another workstation on the network.

The ping proxy agent provides the ability to test the reachability of Internet Protocol (IP) devices, translating manager requests into Internet Control Message Protocol (ICMP) echo requests. Similarly, the hostperf proxy agent uses the *rstat* protocol to gather host statistics.

Proxy agents provide a mechanism allowing Site/SunNet/Domain Manager to extend into virtually any domain. The Simple Network Management Protocol (SNMP) proxy agent provides the ability to manage any device that supports SNMP, the widely accepted standard management protocol for the TCP/IP world.

10.3 Steps for Writing an Agent

From a high-level viewpoint, the steps involved in implementing an agent are as follows:

1. Access to the managed object must exist (that is, code must be written to get the required management data).

Note – The prefixes `NETMGT`, `Netmgt`, and `netmgt` are reserved for network management functions.

---

2. Assign a name to each discrete management data item—*attribute*. For example, if the input packet count is an attribute, `ipkts` would be a good name.
3. Determine the data type for each attribute. In the example, `ipkts` is an integer.
4. Use the attribute information to form the agent schema file, the portion of the Management Database specific to the particular agent.
5. Expand the original code written for accessing the managed object to incorporate the agent schema definitions.
6. Write the code that uses the Agent Services library. This includes code for agent initialization, request handling, and error reporting.
7. Incorporate any agent-specific error messages into the agent schema file.
8. Test and integrate the completed agent code and schema file.

The next three chapters provide detailed descriptions of the procedures for incorporating a new agent.

Step 1 is *not* addressed in this document. As exemplified by the set of Site/SunNet/Domain Manager agents, managed objects can be anything from a communications interface to a host CPU to an application program. It is impossible to generalize how to access a managed object. One hint: Investigate whether utilities already exist that provide access to useful information. Most of the agents in Site/SunNet/Domain Manager rely on such mechanisms.

Steps 2, 3, 4, and 7 are addressed in Chapter 11, “Writing an Agent Schema.”

Steps 5, 6, and 7 are discussed in Chapter 12, “Procedure for Writing an Agent,” and in Chapter 14, “Converting an Existing Application to an Agent.”

Step 8 is covered in Chapter 13, “Testing and Integration.”

10

Writing an Agent Schema 11

After you have chosen the information an agent will manage, the information must be described in an agent schema. This chapter discusses the steps in writing an agent schema. These steps are:

1. **Convert the information into a set of attributes.**
2. **Choose a name and data type for each attribute.**
3. **Arrange the attributes into logically related sets.**

   Section 11.2, “Agent Schema Attributes,” describes agent schema attributes.

4. **Write the agent schema.**
   This consists of defining the agent, defining any agent-specific enumerations, defining the groups/tables, defining the attributes, and defining any agent-specific errors.

   Section 11.3, “Agent Schema Syntax,” describes the agent schema syntax.

5. **Name the completed agent schema and place it in the appropriate directory so it can be used by the manager.**

   Section 11.4, “Schema File Conventions,” describes agent schema file name conventions.

11

11.1 What is an Agent Schema?

An agent schema is a data description that characterizes an agent's capabilities. It publishes agent interface information to management processes that submit requests and receive results. It may be used to interpret results from the agent.

While the terms "agent schema" and "schema file" are sometimes used interchangeably, they refer to slightly different things. The "agent schema" refers to the data description of a single agent. The term "schema file" is more generic—it refers to a file that may contain one or more descriptions.

11.2 Agent Schema Attributes

Managed objects are characterized by *attributes*. Attributes are variables an agent can get and set for a manager application. Attributes are characterized primarily by a data type (see Table 11-1 or consult the definition for the enumerated type `snmdb_type` in `netmgt_db.h`, located in `/usr/snm/include/netmgt` for SunOS 4.x machines or at `/opt/SUNWconn/snm/include/netmgt` for Solaris 2.x installations) and an agent-defined name. Attributes often have other associated information (for example, an extended textual description).

See the *Administration Guide* for a list of agents for the Solaris 1.x and 2.x versions of the products and any attribute changes made to them.

Because an agent can supply many attributes, a mechanism exists to collect them into logically-related sets called *groups*.

Certain kinds of groups known as *tables* have multiple instances of a given logically-related set. Consider a group of attributes that apply equally to a set of distinct interfaces or devices managed by an agent—such as the data for a group of RS-232-C ports. A particular instance of a table is identified by an associated *key*.

The first steps in writing an agent schema are defining the "attributes" (for example, variables, counters, state and error information) that you want your agent to collect, and naming and selecting data types for the attributes. Table 11-1, lists the defined data types.

Table 11-1 Data Types

Agent Schema	Service Library	C-language	Description
short	NETMGT_SHORT	short	16-bit integer
unsigned short	NETMGT_U_SHORT	unsigned short	unsigned 16-bit integer
int	NETMGT_INT	int	32-bit integer
unsigned int	NETMGT_U_INT	unsigned int	unsigned 32-bit integer
long	NETMGT_LONG	long	32-bit integer
unsigned long	NETMGT_U_LONG	unsigned long	unsigned 32-bit integer
float	NETMGT_FLOAT	float	32-bit floating point
double	NETMGT_DOUBLE	double	64-bit floating point
string[*n*]	NETMGT_STRING	char *	null-terminated ASCII string; <*n*> sets maximum size; length includes the null terminator
ipaddress	NETMGT_INADDR	u_char[4]	32-bit IP address (represented by an octet string of length 4, in *host* byte order)[1]
unixtime	NETMGT_UNIXTIME	long	signed long, number of seconds since Jan 1, 1970
enum *enumtype*	NETMGT_ENUM	enum	enumeration
octet[*n*]	NETMGT_OCTET	char[*n*]	opaque stream of <*n*> 8-bit bytes, not necessarily null-terminated
netaddress	NETMGT_IP_ADDRESS	u_char[4]	32-bit IP address (represented by an octet string of length 4, in *network* byte order)[1]
objectid	NETMGT_OBJECT_IDENTIFIER	u_long[*n*]	RFC 1065 object identifier
counter	NETMGT_COUNTER	unsigned int	unsigned integer, only increments, wraps after maximum value (2^{32}-1)
gauge	NETMGT_GAUGE	unsigned int	unsigned integer, may either increment or decrement, latches at maximum value (2^{32}-1)
timeticks	NETMGT_TIMETICKS	unsigned int	unsigned integer, elapsed time since some referenced event (in hundredths of seconds)

[1] While an IP address can be represented in host byte order, network byte order should be used to ensure correct operation with all management applications.

11.3 Agent Schema Syntax

After the attributes have been selected, the agent schema can be written. The examples shown in this section are arranged in the following top-down logical order:

- Naming and describing the agent.
- Naming and describing any agent specific enumerations.
- Naming and describing the agent's groups and tables.
- Naming and describing the attributes in each group or table.
- Naming and describing the agent-specific errors.

11.3.1 Syntax Rules

Use the following rules when defining an agent schema:

- Names may be any character above ASCII 0x20 (space). The following special characters must be enclosed in quotes if used: space, tab, left and right parenthesis, left and right brackets, and pound ("#").
- A pound sign (#) causes the remainder of the line to be treated as a comment.
- Strings are enclosed in double quotes. Repeat the double quote character (for example "") to include a double quote inside a string.
- Spaces, tabs and newlines (\n) are the token delimiters. All are equivalent when parsing a schema. Use them to improve the readability of your agent schema.
- If an argument string passed to `snm_cmd` contains spaces (ASCII 0x20), these will be treated as delimiters and the substrings on either side of the spaces will be interpreted as separate arguments.

11.3.2 Conventions

The agent schema descriptions shown in this section use the following conventions:

- **Bold** font indicates a keyword.
- *Italic* font indicates text you supply or fill in.

- An ellipsis (...) indicates zero or more occurrences of the preceding item.
- A vertical bar | inside braces { } indicates a choice of keywords.
- Brackets [ ] indicate an optional item.

11.3.3 Defining an Agent

An agent description consists of naming the agent, optionally giving it descriptive text, its serial number and RPC number, specifying its enumerations, groups, and tables, and optionally specifying its error return codes and messages. The syntax is:

```
{agent | proxy} <agentName>
[ description "descriptive text" ]
[ rpcid <program_number> ]
[ serial <serial_number> ]
(
    [ agent specific enumeration descriptions ]

    group/table description
    ...

    [ agent error description ]
)
```

where

{ **agent | proxy** }
is a keyword identifying this structure as an agent schema. A proxy agent is an "intermediate" agent that can carry out requests for another system (for example, a ping agent or an SNMP agent).

<agentName>
is the name of the agent schema as it appears in the agent schema list in the Element Properties window in the Console. *<agentName>* has a maximum length of 64 characters.

description
keyword that introduces descriptive text providing documentary information on this agent.

"<descriptive text>"

is the user-supplied text. This description appears in the agent schema list of the Element Properties window in the Console. The descriptive text has a maximum length of 1024 characters. (Note that the schema parser does not recognize any continuation conventions. All characters—including newlines (\n)—are included in the text up to the closing quote.)

`rpcid`
keyword that indicates that the RPC program number follows.

<*program_number*>
is the RPC program number of the agent. If specified, the Console and set tool use this number when making agent requests rather than consulting the NIS / NIS+ service or `the rpc` file (located in `/etc` on SunOS 4.x machines, or in `/etc/opt/SUNWconn/snm` in Solaris 2.x installations).

`serial`
keyword that indicates that the serial number follows.

<*serial_number*>
is the version number of the agent, beginning with 1 and increasing for each release of the agent. This is not the RPC version number, but a method for ensuring that the agent and its schema file are in sync. In the current release, the `snm` and `snm_cmd` managers do not look at this information.

<*agent specific enumeration descriptions*>
list of agent-specific enumeration definitions. See the discussion in Section 11.3.4, “Defining an Agent Enumeration.”

<*group/table description*>
list of group or table descriptions. See the discussion in Section 11.3.5, “Defining a Group and Table,” on page 11-8.

<*agent error description*>
list of errors returned by the agent. See the discussion in Section 11.3.7, “Defining an Agent Error,” on page 11-11.

The following example shows the agent portion of the sync agent schema. The complete sync agent schema can be found in Section 11.5, “An Example Agent Schema,” on page 11-12. ”

```
agent sync
  description "synchronous interface stats"
  serial 1
  rpcid 100104
(
    ...
```

11.3.4 *Defining an Agent Enumeration*

If any of the attributes of any agent are enumerated types, the enumeration should be defined in the agent schema. An agent schema may define zero or more agent-specific enumerations. The scope of an agent-specific enumeration is confined to the agent. The format is as follows:

```
enum <enumerationName> (
integer "<enumeration text>"
    ...
)
```

where

enum
keyword specifying an enumeration declaration.

<enumerationName>
name of the enumeration. An enumeration name has a maximum length of 64 characters.

integer “*<enumeration text>*”
specifies the mapping between an enumeration value and the *<enumeration text>* describing that member of the enumeration. An enumeration may contain up to 20 members. Each *<enumeration text>* has a maximum length of 64 characters.

For example, the following enumeration defines a boolean:

```
enum boolean (
    0   "False"
    1   "True"
)
```

11.3.5 Defining a Group and Table

A group or table description consists of the group or table name, optional descriptive text, and a list of attributes. Group and table descriptions have the following syntax:

```
{ group | table } <groupname>
[ description "<descriptive text>" ]
(
    <attribute description>
    ...
)
```

where

{ **group** | **table** }
keyword that specifies a group or table.

<groupname>
user-defined group or table name. This name is put into a namespace unique to this agent. Group and table names within a single agent description must be unique. A group or table name can have a maximum length of 64 characters.

description
keyword introducing descriptive text about a group or table.

"<descriptive text>"
describes the group or table and can have a maximum length of 1024 characters.

<attribute description>
list of attribute descriptions. See the discussion in Section 11.3.6, "Defining an Attribute."

The following example shows the agent and group/table sections from the example sync agent schema. The attribute definitions are indicated with an ellipsis (..).

```
agent sync
  description "synchronous interface stats"
  serial 1
  rpcid 100104
(
    table mode
    description "synchronous interface"
    (
        ...
    )
    table data
    description "I/O statistics"
    (
        ...
    )
```

11.3.6 Defining an Attribute

An attribute description consists of an attribute type specification (see Table 11-1 on page 11-3), the attribute name, and optional descriptive text. Attribute descriptions have the following syntax:

read/write <*type name*>
  [**description** "<*descriptive text*>"]
  [**units** <*unit type*>]

where

`read/write`
specifies the read / write status of the attribute. The four possible read / write parameter values are:

`readonly` or `ro`	the attribute can be read
`readwrite` or `rw`	the attribute can be read or set
`writeonly` or `wo`	the attribute can *only* be set
`notaccessible` or `na`	the attribute cannot be read or set

Either the full word or the abbreviation can be used. If no read / write parameter is present in an attribute definition, the attribute is assumed to be `readonly`.

<*type*>
is an agent schema type listed in Table 11-1 on page 11-3. String and octet types require a length specifier to declare the maximum size of the string.

An enumerated type is specified with the keyword **enum** followed by the enumeration type.

<*name*>
is the user-defined name of this attribute. Each group or table defines its own namespace in which names must be unique. However, the same name may occur in different groups or tables. A name has a maximum length of 64 characters.

description
keyword that introduces descriptive text providing documentary information about an attribute.

"<*descriptive text*>"
describes the attribute and can have a maximum length of 1024 characters.

units
keyword identifying a unit for a particular attribute.

<*unit type*>
optional clause specifying the units for a particular attribute. Currently this field is not used by the Console. It is described here for other manager applications. The maximum length is 1024 characters.

Below are some example attributes taken from the sync agent schema.

```
string[32]    ifname
    description "interface name"

enum tx       txclock
    description "transmit clock source"

int           iutil%
    description "input utilization"
         units "percent"
```

11.3.7 Defining an Agent Error

If an agent wishes to return agent-specific errors, they should be defined in the agentErrors section of the agent schema. This section defines a table of error numbers and associated error messages specific to a particular agent. The error numbers are represented by integers and the error messages are descriptive text enclosed in double quotes. Defining errors in the schema file localizes error messages to aid in making agents multinational.

The agentErrors section has the following syntax:

```
agentErrors (
    <integer>  "<error text>"
    ...
)
```

where

agentErrors
is the keyword that identifies agent-specific errors.

<integer> "<error text>"
specifies a mapping between an agent-specific error code <*integer*> and <*error text*>. The error text has a maximum length of 1024 characters.

The following example shows the agentErrors excerpt from the sync agent schema.

```
agentErrors (
    1    "No synchronous interface on this host"
    2    "Error in SIOCGIFCONG ioctl system call"
    3    "This key is not a synchronous interface"
)
```

11.4Schema File Conventions

By convention, the agent schema normally resides in the same directory as the agent. To be loaded by the Console, the agent schema file name must end with `.schema`.

At start-up time the Console may be instructed to reinitialize its runtime management database. A step in doing this is to load all agent schema. The *schemas* keyword in `/etc/the snm.conf` file contains a list of schema file directories. (The `snm.conf` file is located in the `/etc` directory on SunOS 4.x machines and in `/etc/opt/SUNWconn/snm` in Solaris 2.x environments.) Any file ending with `.schema` in those directories is assumed to be an agent schema and is loaded into the runtime database.

Agent schema may also be read into the Console using the File button Load option.

11.5An Example Agent Schema

The following section contains an abbreviated example of the agent schema for the sync agent. This agent returns statistics on synchronous serial ports.

```
agent sync                                          # Agent declaration
  description "synchronous interface stats"         #
  serial 1
  rpcid 100104
(

  enum tx (                                         # Agent specific
    0   "incoming transmit clock"                   # enumeration declarations
    1   "incoming receive clock"
    2   "baud rate generator"
    3   "phase-lock loop output"
  )



  enum rx (
    0   "incoming receive clock"
    1   "incoming transmit clock"
    2   "baud rate"
    3   "phase-lock loop"
  )



  enum bool (
    0   "false"
    1   "true"
  )

table mode                                          # Group/table declarations
  description "synchronous interface table (key on ifname)"
(
  string[32]   ifname      description "interface name"
  enum tx      txclock     description "transmit clock source"
  enum rx      rxclock     description "receive clock source"
  enum bool    loopback    description "do internal loopback"
  unsigned int baudrate    description "interface speed"
)
```

```
table data
  description "I/O statistics table (key on ifname)"
(
  string[32]   ifname      description "interface name"
  long         ipkts       description "input packets"
  long         opkts       description "output packets"
  int          ibytes      description "input bytes"
  int          obytes      description "output bytes"
  int          iutil%      description "% input utilization in the last timeperiod"
                           units "percent"
  int          outil%      description "% output utilization in the last time period"
                           units "percent"
  long         aborts      description "aborts"
  long         crcs        description "crc errors"
  long         over        description "receiver overruns"
  long         under       description "transmitter underruns"
)



agentErrors (                                    # Agent specific error
  1    "No synchronous interface on this host"   # declaration
  2    "Error in SIOCGIFCONG ioctl system call"
  3    "This key not a synchronous interface"
  )


)
```

11.6 Mapping Feature

The following discussion describes mapping, an advanced feature for agent writers.

When a user makes a request, the Console builds a request record that sets the parameters for the request. The Console has a set of defaulting rules that it uses to build the request record. Because these rules might not be correct for some agents, the Console provides a mapping mechanism by which an agent writer may force one or more fields of the request to be set from fields in the element's properties. This is done by adding a **map** line into the agent schema file. The **map** line has the following syntax:

```
map "<req_obj_field1>=<prop_field1>[,<req_obj_field2>=<prop_field2>]"
```

where

map
keyword indicates one or more mappings are desired.

<req_obj_field>
name of a field in the request record.

<prop_field>
name of a field in the element properties.

Each map string can be a maximum of 1024 characters.

The map entry immediately follows the "agent description"—see the following example:

```
agent XYZagent
    description "An agent that talks to XYZ elements"
    map "_targetsystem=Real_Name"
(
    ...
)
```

Note – The mapping occurs when the Property Sheet is brought up for a new request or when a “Quick Dump” request is launched. It does not occur when an existing request is modified.

For instance, consider an element of the following type:

```
record component.notASun (
    string[32]    Name        # Name of the system
    string[32]    Real_Name   # Its real name
)
```

If a request for the *XYZ<agent>* was launched from a element of type `component.notASun`, then the field *<_targetsystem>* in the request would be set from “Real_Name.”

To understand how to use mapping, an agent writer must understand how the request record fields are used to build a request sent to the agent. There are two types of request records, one for data requests and the other for event requests. These fields are in the “dataRequest” and “eventRequest” records as defined in the file `/snm.glue`. (The `snm.glue` file is located in the directory `/usr/snm/struct` for the SunOS 4.x version of this product and in `/opt/SUNWconn/snm/struct` for the Solaris 2.x version.)

The next table shows the fields in the request record, describes how they are used to build an agent request, and describes their default values when no mapping is specified.

Field	Usage or Default Rule
Name	Provides the name of the request. The name is displayed below the request glyph. This field is not seen by the agent. The default is a name of the form: <*agent-name*>.<*group-name*>.<*n*> where <*n*> is a number
_targetsystem	Provides the <system> value for a request. By default, this field is set to the Name field from the request's element object. If the field <*Proxy_System*> is NULL, this field also provides the name of the system to contact (for example, the system on which the agent runs).
Key	Provides the <*key*> value for a request. By default, this field is set to NULL.
Options	If not NULL, provides the <*options*> value for a request. By default, this field is set to NULL.
Proxy_System	If not NULL, provides the name of the system on which the agent resides. By default, this field is set to NULL for agents that are not defined as proxy agents. For proxy agents, this field is set to the value of the <*proxy*> field in the proxy subrecord of the request's system object.

Procedure for Writing an Agent 12

Once the agent schema has been defined, you can add the Site/SunNet/Domain Manager code to turn your program into an agent. This chapter discusses the procedure for developing a Site/SunNet/Domain Manager agent, including:

- Agent Initialization and Startup
- Agent Shutdown
- Request Verification and Dispatching
- Handling Requests for Data Reports, Event Reports, and Setting Attributes
- Error Reporting
- Generating and Sending Asynchronous Reports (Traps)

This chapter discusses an agent's flow of control, and how an agent interacts with the Agent Services library. For reference material, see the man pages in the Appendix B, "Man Page Summary for Writers of Agent Software." Use Chapter 14, "Converting an Existing Application to an Agent," as an outline for converting your application to an agent.

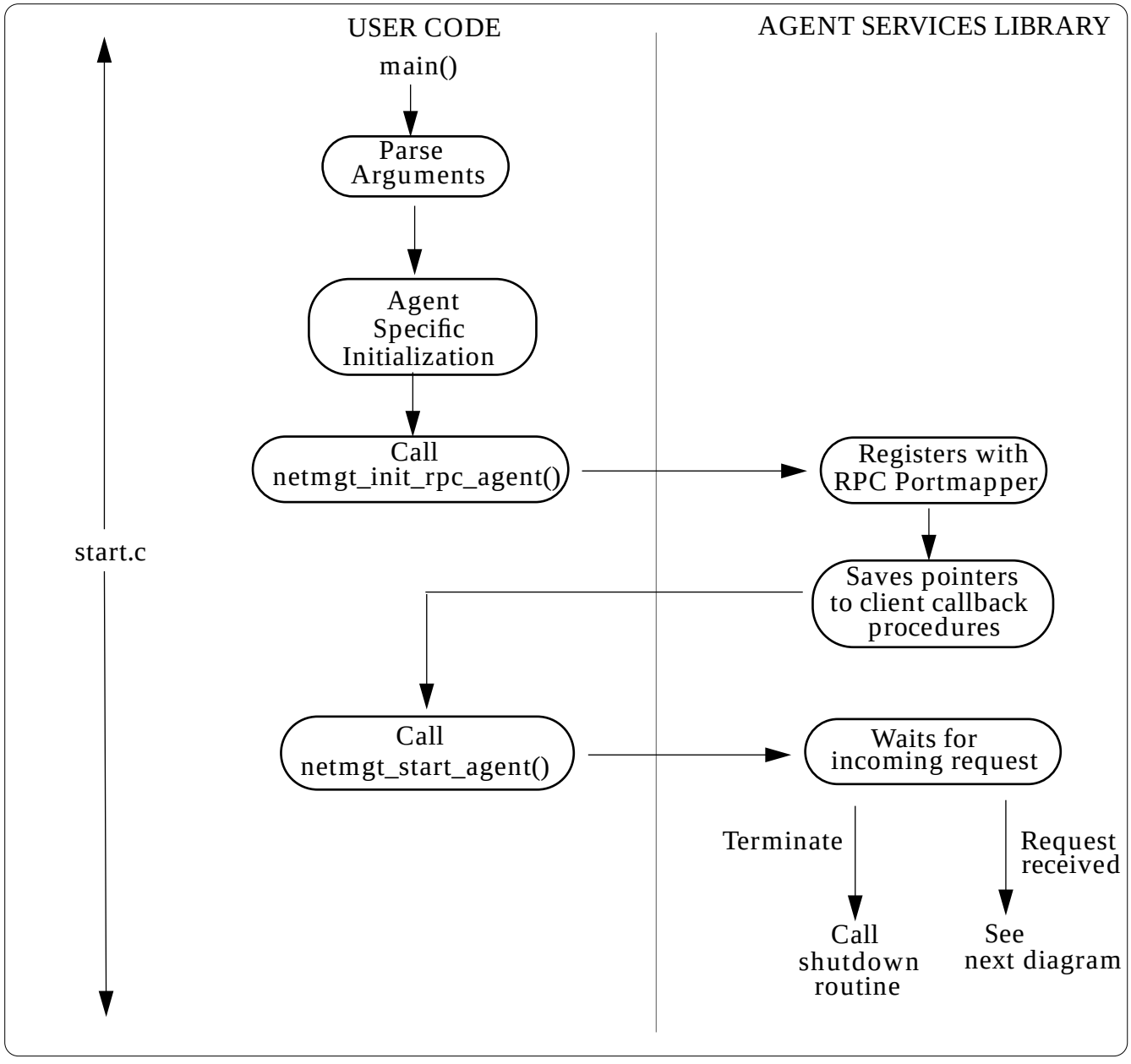

Figure 12-1 Agent Initialization

12.1 Agent Initialization and Startup

The first step for the agent is initialization. See Figure 12-1 for a flow diagram.

After parsing any command-line parameters and executing agent-specific initialization, the agent calls `netmgt_init_rpc_agent()` to register with the RPC system and initialize the Agent Services library data structures. The parameters of this call are protocol-specific. They do not directly affect the agent application. They fall into several categories:

Agent Identification
: identifies an agent by the RPC service name, agent-writer assigned serial number, RPC program number, and RPC version number parameters.

Transaction Characterization
: characterizes a transaction by the transport protocol to be used in manager/agent communication, the transaction time limit, and various flags such as whether agent requests run as subprocesses.

Callback Routines
: lists agent-supplied routines called by the Agent Services library to validate a request from a manager, dispatch the request, reap the agent child spawned by the request, and shutdown the agent's parent process.

After initialization, the agent starts itself by making a call to `netmgt_start_agent()`. This function has no parameters and never directly returns. It waits for incoming requests and returns control to the agent via the agent-defined callback routines.

12.2 Agent Shutdown

Normal agent shutdown occurs when the child process returns from its dispatch routine (discussed in Section 12.3, "Request Verification and Dispatching." If no agent-supplied `shutdown` function is specified during agent initialization, the Agent Services library will call `netmgt_shutdown_agent()` to unregister the agent from the RPC system, clean up some library data structures for the agent and have the agent exit with a return code of zero. If the agent supplies its own `shutdown` routine, that routine must call `netmgt_shutdown_agent()` as the last thing it does. An agent must *never* exit directly. It should call `netmgt_shutdown_agent()` to terminate.

If a report cannot be delivered to the report rendezvous, the Agent Services library automatically retries at appropriate intervals. If the rendezvous is a program with a permanent RPC number (like the logger or event dispatcher), the library tries to send the report forever. If the rendezvous is a program with a transient RPC number (like the Console or `snm_cmd`), the library tries to send the report a small number of times before considering it undeliverable. If a report is considered undeliverable the Agent Services library will shut down the agent child process.

The Agent Services library uses the following signals in the parent: `SIGINT`, `SIGQUIT`, `SIGTERM` and `SIGCHLD`. If the parent wishes to have control passed to its own functions when one of these signals occurs, it should specify the functions in the `shutdown` (for `SIGINT`, `SIGQUIT` and `SIGTERM`) and `reap` (for `SIGCHLD`) parameters in the call to `netmgt_init_rpc_agent()`. One signal is used in the child: `SIGUSR1`. Agents should avoid using this signal.

12.3 Request Verification and Dispatching

When a request is received from a manager via RPC, the agent parent process performs a verification routine. This routine should determine if the agent is able to do the requested operation. If the agent cannot execute the request, the verification routine should call `netmgt_send_error()`, specifying the reason for the failure, then return `FALSE`. Otherwise, it should return `TRUE` without calling `netmgt_send_error()`. Figure 12-2 illustrates request verification and dispatching.

The agent verification routine has a limited time to verify a request as dictated by the <*timeout*> argument specified by the manager making the request. This timeout is typically between 10 and 30 seconds, so the agent's verification routine should not do an extended amount of processing. For instance, verifying that the group name is valid would be appropriate, but retrieving information from a distant network would not be appropriate.

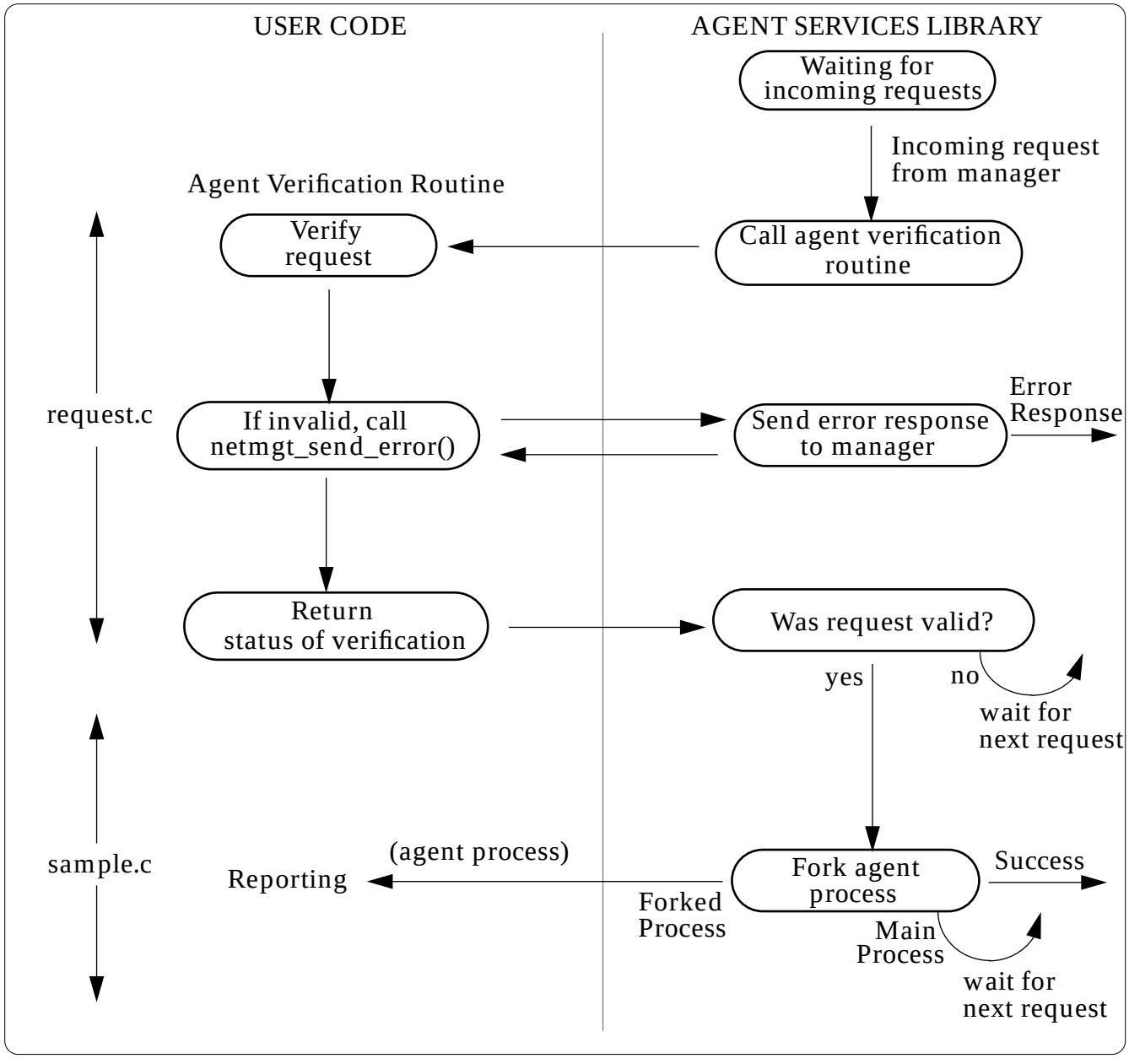

Figure 12-2 Request Dispatch

Note – Although Site/SunNet/Domain Manager forks a subprocess before calling your dispatch function, your verification function should *not* cache request arguments as global data. Caching request arguments may save a little time but it will probably make your agent incompatible with future Site/SunNet/Domain Manager releases.

Upon successful validation of the request, the Agent Services library creates a child process and calls the agent-supplied dispatch function in the context of the child process.

Note – If the verification routine opens files or allocates memory, it must be able to close the files or free the memory after control has passed to the child process. Otherwise, memory and file descriptor leakage may occur. For this reason, you should avoid agent setup tasks in the verification routine.

The dispatch function should execute the request, reporting data at the requested times. When all request servicing is completed, the dispatch function returns to the Agent Services library, who shuts down the child process.

Besides receiving requests from a manager, agents can also read requests from the `request.log` file on the agent system—this is the case when a request is restarted. For these requests, the verification routine is bypassed and the dispatch routine is called directly. Note that if any agent setup tasks are performed in the verification routine, the agent will not be set up correctly for restarted requests.

12.3.1 Verification and Dispatching Routine Parameters

The verify and dispatch routines share a common set of parameters. When an agent is used with the Console, most of these parameters come from the information in the request property window.

The following parameters are passed to the agent-supplied verification and dispatching procedures:

<*type*>
unsigned integer containing the request type code. The type code is either NETMGT_DATA_REQUEST for requesting data reports or NETMGT_EVENT_REQUEST for requesting event reports. Agents don't need to know the request type, but the code is provided for agents who want to do additional processing based on the request type.

<*target*>
pointer to a null-terminated ASCII string containing the name of the system where the managed element is located. If the agent is a proxy agent, this name may be different from the name of the system where the agent is running.

<*group*>
pointer to a null-terminated ASCII string containing the name of the group whose attribute values are to be collected. Group and attribute relationships are specified in the agent schema. (See Chapter 2, "Registering for Data, Event, and Trap Reports," for more information on the agent schema.)

<*key*>
optional pointer to a null-terminated ASCII string containing a key uniquely identifying an instance of the <*group*> on the managed <*target*>.

The interpretation of this string is left largely to the agent writer. Use white-space separated strings for key selectors, or a null string when all table entries are desired. The key is usually an attribute in the table. Managers have no knowledge of what agents will accept as keys; agent writers are encouraged to document their use of the key field.

<*count*>
unsigned integer containing the maximum reports to send. If <*count*> is zero, the agent should send reports until the request is asked to stop by a manager process.

<*interval*>
<*interval*> structure containing the reporting interval. If <*interval*> is zero, the agent should use an agent-default value.

<*interval*> and <*count*> taken together determine the type of reporting to do. Table 12-1 lists the possible combinations and their interpretations (*c* and *i* are positive integers):

Table 12-1 Count / Interval Interpretations

Count	Interval	Interpretation
0	0	Send reports forever at an agent-specific interval
0	*i*	Send reports forever, every *i* seconds
1	0	Quick Dump—send one report
1	*i*	Quick Dump—send one report
c	0	Send *c* reports at an agent-specific interval
c	*i*	Send *c* reports every *i* seconds

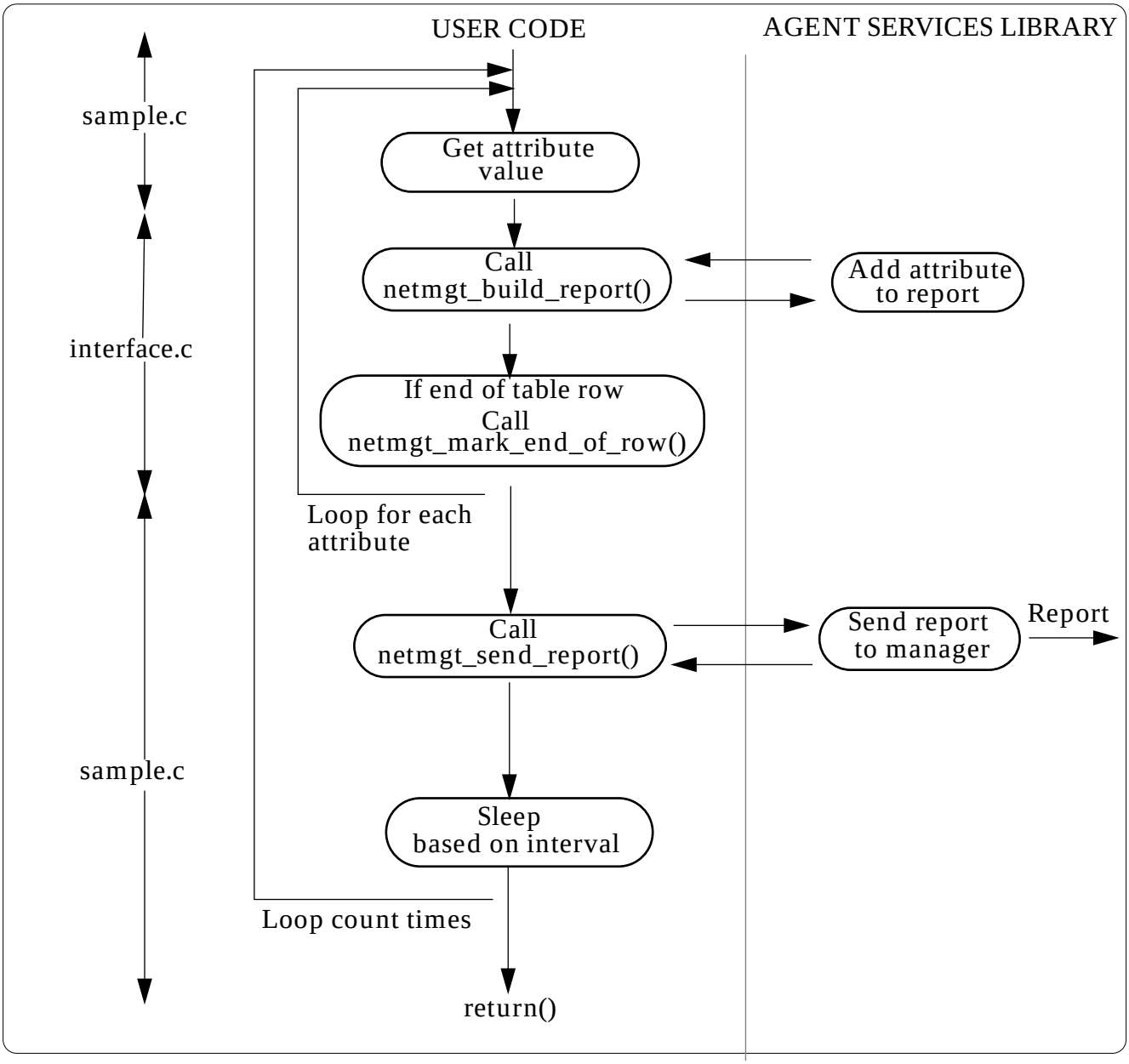

Figure 12-3 Sending Reports

Data gathering methods fall into two categories. Agents may be able to report data instantly or they may need to integrate it over time before reporting. For instant reporting, the agent reports the data then sleeps for the interval specified. For integrative reporting, the agent performs some set of actions over time before the data can be reported. For instance, the supplied `ping` agent sends many `ICMP ECHO` requests at one second intervals, then waits a small amount of time for the `ECHO REPLY` responses. It may take up to 20 seconds for a single report to be sent.

Agents who take a substantial amount of time to integrate and report should subtract their processing time from the manager-specified <*interval*>, so reports are sent as often as requested. Otherwise, an interval of 20 seconds and a processing time of 20 seconds would mean reports would be sent every 40 seconds—clearly not what the manager intended. Again, agent writers are encouraged to document any specific decisions they make along these lines, so users know what report intervals make sense for an agent.

<*flags*>
unsigned integer bit string specifying request options. Currently no request options are defined for use by agents.

12.4 Sending Reports

After a request is received and validated, a child process is created and the agent-supplied dispatch procedure is called. The dispatch procedure collects the information requested and sends reports at the specified intervals. Figure 12-3 provides a flow diagram for request handling.

When servicing a request, the child process should:

- Get any optional request argument by calling `netmgt_fetch_argument()` setting the parameter `argname` to `NETMGT_OPTSTRING` as defined in `netmgt_define.h`.
- Obtain the data.
- Build a reporting argument list with the current attribute data using `netmgt_build_report()`, once for each attribute. Check the return code for any errors. If you are interested in whether an event has occurred for the particular attribute (because you intend to do further processing), you can

check the value of the second parameter passed to `netmgt_build_report()`. Most agents do not check this "event" flag because they do no special processing when events occur.

- If the group represents a row of a table, once `netmgt_build_report()` has been called for all attributes in a group, the agent must call `netmgt_mark_end_of_row()` to signal the end of the row. `netmgt_mark_end_of_row()` should always be called to mark the end of a row, even if only one row is being returned.
- Agents that return tables should also return a key by returning an attribute called `netmgt_table_key`. Returning a key is useful when setting attribute values with the Console's Set tool (`snm_set`) or when table attributes are graphed from the Results Browser tool.
- Once the report has been completely built and is ready to be delivered, the agent should send the report to the rendezvous process with `netmgt_send_report()`. If this is the last report the agent will send, set the `NETMGT_LAST` flag.
- After the final report, return; otherwise, call `sleep(3V)` to suspend the child process for the time specified by the interval. A specified reporting interval of zero instructs the agent to choose an appropriate interval. A specified reporting count of zero instructs the agent process to continue sending reports every interval until the agent is asked to stop by a manager process. Otherwise, a total of <*count*> reports should be sent.

12.5 Handling Set Requests

In addition to sending data and event reports, some agents can change the values of managed objects (that is, set attributes values). Handling a request to set the values of one or more group attributes is very similar to handling a data or event request. The basic steps are:

- In your request verification function, determine whether the request is well formed. If it isn't, send an error report to the requester and return an error.
- Assuming the request is well formed, your request dispatch function should set the requested attribute values.
- Send an status report to the requester indicating whether you succeeded in setting the attributes and return.

12.5.1 Verifying the Request

When your agent receives a set request, your request verification function will be called just like a data or event request except for the following points:

- the request <*type*> is `NETMGT_SET_REQUEST`.
- the reporting interval is zero.
- the reporting count is zero.
- the group and key arguments are `NULL`.

Your request verification function should determine whether you believe you can set the requested attributes values. Do *not* actually set the attribute values here. Determine whether the set request is reasonable. For example, you should probably check whether the attributes have read-write or write-only access.

Check whether the values to set the attributes have the correct data types and are within acceptable ranges. (See Table 11-1 on page 11-3 for a list of valid data types.) If your agent can take optional arguments, fetch the arguments using `netmgt_fetch_argument`(3n).

Note – The Console `snm` and command-line manager `snm_cmd` set the optional argument name to `NETMGT_OPTSTRING` and the optional argument type to `NETMGT_STRING`.

Fetch the set request arguments using `netmgt_fetch_setval`. This function specifies the <*group*> name, optional <*key*> name if the group is a table, <*attribute*> name, and <*value*> to set the attribute. Unlike data and event requests, a manager can request your agent set attributes in more than one group. Continue calling `netmgt_fetch_setval` until the <*group*> name is `NETMGT_ENDOFARGS` which indicates the end of the request arguments. If you determine the request is well-formed, your request verification functions should return the boolean value `TRUE`. Otherwise, it should call `netmgt_send_error`(3n) to send an error report to the requester and return the boolean value `FALSE`.

12.5.2 Set Attribute Values

If you determined the request was well-formed, your request dispatch procedure will be called just like your verification function. Once again, you should fetch any request options and the set request arguments.

You should now set the requested attribute values. The way you do this is entirely up to your application.

12.5.3 Send a Status Report

Whether or not you were successful in setting the requested attribute values, you should send a status message to the requester. If you were successful, call `netmgt_send_error`(3n) setting the <*service_error*> code to `NETMGT_SUCCESS`. If you encountered an error setting the attributes, send an error report just like you would for a data or event request.

12.5.4 Sample Code

The following code fragment is an example of a routine which receives a set request and sends a status message to the requester:

```
/* agent error codes */
#define SET_FAILED 1

/* ------------------------------------------------------------
 * dispatch_request - dispatch agent request
 * no return value
 * ------------------------------------------------------------
 */
void
dispatch_request (type, system, group, key, count, interval, flags)
    u_int type,             /* request type */
    char *system,           /* target system */
    char *group,            /* object group */
    char *key,              /* table key */
    u_int count,            /* report count */
    struct timeval interval,/* report interval */
    u_int flags;            /* request flags */
{
    NETMGT_DBG("dispatch_request\n");
    /* dispatch request based on request type */
    switch (type) {
                                                        case NETMGT_SET_REQUEST:
            do_set_request(system);
            return;
    /* other cases */
    }
    return;
}

/* ------------------------------------------------------------
 * do_set_request - perform the set request
 * no return value
 * ------------------------------------------------------------
 */
void
do_set_request(system)

    char *system;           /* target system name */
{
    Netmgt_arg option;      /* optional argument */
    Netmgt_setval setval;   /* set request argument
```

```
    Netmgt_error status;/* status report argument */
    NETMGT_DBG("do_set_request\n");

    /* get any optional arguments */
    if (netmgt_fetch_argument (NETMGT_OPTSTRING, &option))

    /* handle request option */
    /* fetch request arguments and perform set operations */
    for (;;)
        /* get next set argument
        if (!netmgt_fetch_setval (&setval)) {
        (void) netmgt_fetch_error(&status);
        if (!netmgt_send_error(&status))
            NETMGT_DBG("netmgt_fetch_setval failed: %s\n",
                netmgt_sperror());
        return;
    }

            /* a sentinel string marks the end of arguments
            if (strcmp (setval.name, NETMGT_ENDOFARGS) == 0)
            return;

    /* assume here the set operation failed - send
     * an fatal error report to the requester
     */
    if (!perform_set(system, setval)) {
        status.service_error = NETMGT_FATAL;
        status.agent_error = SET_FAILED;
        status.message = "a description of why it failed";
        if (!netmgt_send_error(&status))
            NETMGT_DBG("netmgt_send_error failed: %s\n",
                netmgt_sperror());
        return;
    }
}

/* assume all the set operations succeeded - send a success status
 * message to the requester
 */
status.service_error = NETMGT_SUCCESS;
status.agent_error = (u_int)0;
status.message = (char *)NULL;
```

```
if (!netmgt_send_error(&status))
    NETMGT_DBG("netmgt_send_error failed: %s\n", netmgt_sperror());
    return;
}
```

12.6Error Reporting

The agent reports errors by calling `netmgt_send_error()`. There are two classes of errors, generic errors and agent-specific errors.

Generic errors are the fatal errors defined in the header file `netmgt_errno.h`. Generic errors may be generated by agents or the Agent Services library.

Agents may also report errors specific to the agent. Agent-specific errors are defined by a code/message pair in the `agentErrors` section of the agent's schema file. Agent-specific errors may also return supplementary text providing additional information about the error.

While all generic errors are fatal, an agent-specific error may be classified as either a warning or fatal. If the error is a warning, the manager assumes the agent will continue to return reports. If the error is fatal, the manager assumes the agent will stop servicing the request. The Console treats fatal errors as high-priority errors and warnings as low-priority errors.

`netmgt_send_error()` takes a single parameter, a pointer to a `Netmgt_error` structure with three fields:

`service_error`

an error code from the enumerated type `Netmgt_stat` as defined in `netmgt_errno.h`.

`NETMGT_WARNING`

signals an agent-specific non-fatal error. The `agent_error` field provides the agent error code.

`NETMGT_FATAL`

signals an agent-specific fatal error. The `agent_error` field provides the agent error code.

Any other error code signals a generic error, and `agent_error` should be set to zero.

`agent_error`

specifies the agent-specific error code from the list defined in the `agentErrors` section of the agent schema.

`message`

optional null-terminated string containing instance information about an agent-specific error that cannot be given in the `agentErrors` section of the agent schema—for example “ie0” or “port 7.”

12.7 Generating and Sending Asynchronous Reports (Traps)

Site/SunNet/Domain Manager provides support for asynchronous reports, also known as *traps*. A program generates a trap by first calling `netmgt_start_trap()`, then calling `netmgt_build_report()` for each attribute in the trap, and finally calling `netmgt_send_report()` to send the trap report. When `netmgt_send_report()` is called, traps are sent to the rendezvous as specified by `netmgt_start_trap()` arguments.

The following shows the syntax of the `netmgt_start_trap()` function:

```
bool_t netmgt_start_trap(system, agent_prog, agent_vers, group,
rendez_host, rendez_prog, rendez_vers, priority, timeout)
      char*   system;
      u_int   agent_prog;
      u_int   agent_vers;
      char*   group;
      char*   rendez_host;
      u_int   rendez_prog;
      u_int   rendez_vers;
      u_int   priority;
      struct timeval timeout;
```

where

`system`
is the name of the element associated with the trap.

`agent_prog`
specifies the RPC program number of the agent that is sending the trap. Specify `0` if unknown.

`agent_vers`
specifies the RPC program version number of the agent that is sending the trap. Specify 0 if `agent_prog` is set to 0.

`group`
specifies the group within the agent schema to which the trap attributes belong. Specify "trap" if the attributes are not defined in any agent schema.

`rendez_host`
specifies the name of the system where the trap is to be sent.

`rendez_prog`
rendezvous for the trap. This should be set to `NETMGT_EVENT_PROG`, the event dispatcher.

`rendez_vers`
specifies the version number of the rendezvous for the trap. This should be `NETMGT_EVENT_VERS`, the event dispatcher's version number.

`priority`
specifies the priority of the trap. Choices are `NETMGT_LOW_PRIORITY`, `NETMGT_MEDIUM_PRIORITY`, and `NETMGT_HIGH_PRIORITY`.

timeout
specifies the maximum time (in seconds) that the `netmgt_start_trap()` call is to wait for a reply from the rendezvous when sending a trap report.

`netmgt_start_trap()` sets up the parameters for the trap. After calling `netmgt_start_trap()`, the trap is constructed by one or more calls to the procedure `netmgt_build_report()`. As with Data and Event Reports, `netmgt_build_report()` is called for each attribute in the trap.

A trap is sent to the rendezvous using `netmgt_send_report()`. Note that `netmgt_send_report()` may block for the timeout period if the trap cannot be sent to the rendezvous. If the trap cannot be sent, it will be retried the next time a trap or other report is generated.

A point to note is that `netmgt_start_trap()` does not need to be called again in order to send another trap. You only need to call `netmgt_start_trap()` if you want to change groups or the value of any argument you specified in the previous call to `netmgt_start_trap()`.

Note – Reporting data and issuing traps should occur in separate processes. If an agent collects and sends data in addition to issuing traps, then the agent should issue a trap by calling the `fork()` system routine to spawn a child process and the building and sending of the trap should be carried out by the child process. This procedure is necessary to ensure that the parent process is not interrupted from collecting and reporting data while the trap is being built and sent. This will prevent loss or delay of reports, especially if the reporting interval is very short.

12.7.1 Sample Code

The following code fragment is an example of a routine that calls `netmgt_start_trap()`, then calls `netmgt_build_report()` for each attribute in the trap, and then calls `netmgt_send_report()` to send the trap report.

```
char myname[MAXHOSTNAMELEN+1];/* system name */
struct timeval timeout;/* timeout buffer */

timeout.tv_sec = 10;/* set timeout value */
timeout.tv_usec = 0;

sysinfo (SI_HOSTNAME, MYNAME, 128);

netmgt_start_trap(myname, /* system name */
                  myprog_number,/* agent program number */
                  myver_number,/* agent version number */
                  "trap", /* group */
                  "localhost",/* event dispatcher host */
                  NETMGT_EVENT_PROG,
                         NETMGT_EVENT_VERS,
                  NETMGT_HIGH_PRIORITY,
                         timeout))
...
/* Later on, build the report.  Do this as many times as you like, once
   for each attribute in the report. */
Netmgt_data data;/* data buffer */
bool_t event;  /* whether an event occurred */
Netmgt_error error;/* error buffer */

(void)strcpy(data.name, attr_name);/* set the attribute name */
data.type = NETMGT_STRING;/* set attribute data type */
data.length = strlen(attr_value);/* set attribute length */
data.value = attr_value;/* set attribute value */
```

```
/* build the report */
if (!netmgt_build_report(&data, &event))
  fprintf(stderr, "Cannot build report: %s\n", netmgt_sperror());

/* Finally send the trap report to the rendezvous specified by
   netmgt_start_trap. */

struct timeval delta_time;

delta_time.tv_sec = delta_time.tv_usec = 0;
if (!netmgt_send_report(delta_time, 0, 0))
  fprintf(stderr, "Cannot send report: %s\n", netmgt_sperror());
```

12.8 Summary

Now that you know how to initialize and shut down your agent, handle requests and report errors, you're ready to integrate your agent into the Site/SunNet/Domain Manager environment.

Testing and Integration 13

This chapter describes how to:

- Set up your agent development environment
- Test your agent to verify it is executing correctly
- Integrate your agent into the Site/SunNet/Domain Manager environment

13.1 Building Your Program

When you compile your agent, you must include a Site/SunNet/Domain Manager header file in each of your source code files and link your object modules with a Site/SunNet/Domain Manager library.

13.1.1 Header Files

The header file you need to include in each source code file is `netmgt.h`. This file includes the other header files, located in the `/usr/snm/include/netmgt` directory for Solaris 1.x installations or `/opt/SUNWconn/snm/include/netmgt` for the Solaris 2.x version. These files declare Site/SunNet/Domain Manager functions and data structures. To simplify portability, include this file in your source code by adding the line:

```
#include <"netmgt/netmgt.h">
```

at the top of the file, and specifying the search path to this header file using the `cpp`(1) `-I` switch in your agent Makefile.

13.1.2 Static and Dynamic Linking

You can link your application with `libnetmgt_db` and `libnetmgt` either statically or dynamically. Dynamically is preferrable and recommended for the following reasons:

- Dynamically linked programs save disk space and main memory because they share library code at runtime.
- Shared library code can be enhanced without having to relink the applications that use it.
- Dynamically linked applications provide better compatibility. In order to ensure that your applications are compatible with future releases of the Site/SunNet/Domain Manager product, dynamic linking *must* be used.

13.1.2.1 Dynamic Linking

Dynamic linking, as shown in the sample below, is the preferrable and recommended linking method. When you dynamically link your program, you can include the following `libnetmgt` libraries:

If you have installed the Solaris 1.1.1 version of this product:

- `/usr/snm/lib/libnetmgt_db.so`
- `/usr/snm/lib/libnetmgt.so`

If you have installed the Solaris 2.4 version of this product:

- `/opt/SUNWconn/snm/lib/libnetmgt_db.so`
- `/opt/SUNWconn/snm/lib/libnetmgt.so`

Note – If you link with the `libnetmgt_db.so` library, the `libnetmgt.so` library must also be used, with `libnetmgt_db.so` being listed before `libnetmgt.so` in the command line. The `libnetmgt.so` library may be used without the `libnetmgt_db.so` library.

13.1.2.2 Dynamic Linking for the Solaris 2.4 Environment

To link with the libraries provided with the Solaris 2.4 version of this product, you must link with the network services library (`libnsl`) and the internationalization library (`libintl`). To *dynamically* link your application, follow this format:

```
host% cc myprog.c -o myprog -R /opt/SUNWconn/snm/lib -L /opt/SUNWconn/snm/lib -lnetmgt_db
-lnetmgt -lnsl -lintl
```

Since the Solaris 2.x version of this product does not make links in `/usr/lib`, all Site/SunNet/Domain Manager applications should link with the `-R` option, as shown above. This avoids forcing the user to set the `LD_LIBRARY_PATH` variable at runtime.

13.1.2.3 Dynamic Linking for the Solaris 1.1 Environment

To *dynamically* link your application with the libraries provided with the Solaris 1.1.1 version of this product, follow this format:

```
host% cc myprog.c -o myprog -L /usr/snm/lib -lnetmgt_db -lnetmgt -lnsl
```

13.1.3 Static Linking

When you statically link your program, you can include the following `libnetmgt` libraries:

- If you have installed the Solaris 1.1.1 version of this product:
  - `/usr/snm/lib/libnetmgt_db.a`
  - `/usr/snm/lib/libnetmgt.a`
- If you have installed the Solaris 2.4 version of this product:
  - `/opt/SUNWconn/snm/lib/libnetmgt_db.a`
  - `/opt/SUNWconn/snm/lib/libnetmgt.a`

Note – If you link with the `libnetmgt_db.a` library, the `libnetmgt.a` library must also be used, with `libnetmgt_db.a` being listed before `libnetmgt.a` in the command line. The `libnetmgt.a` library may be used without the `libnetmgt_db.a` library.

13.1.3.1 Static Linking for the Solaris 2.4 Environment

To link with the libraries provided with the Solaris 2.4 version of this product, you must link with the network services library (`libnsl`) and the internationalization library (`libintl`). To *statically* link your application, follow this format:

```
host% cc myprog.c -o myprog -L /opt/SUNWconn/snm/lib -Bstatic -lnetmgt_db -lnetmgt -lnsl
-lintl -Bdynamic -ldl
```

If you use the static method, you *must* specifically link in `libdl`.

13.1.3.2 Static Linking for the Solaris 1.1.1 Environment

To *statically* link your application with the libraries provided for the Solaris 1.1.1 version of this product, follow this format:

```
host% cc myprog.c -o myprog -L /usr/snm/lib -Bstatic -lnetmgt_db -lnetmgt -lnsl
-lintl -Bdynamic -ldl
```

If you use the static method, you *must* specifically link in `libdl`.

13.1.4 API Differences from the 2.0 Release

Two API functions in the SunNet Manager 2.1 and later releases differ from the SunNet Manager 2.0 release. Arguments to the `reap` and `shutdown` functions, specified as arguments to the `netmgt_init_rpc_agent()` function, have been simplified, as shown in the definitions below.

SunNet Manager 2.0 `reap` function:

Code Example 5

```
reaper(sig, code, scp, addr, child_pid, status, rusage)
int sig;/* signal number caught */
 int code;/* additional signal detail */
 struct sigcontext *scp;/* saved process context */
 char *addr;/* additional address info */
 int child_pid;/* child's pid */
 union wait *status;/* child's return status */
struct rusage *rusage;/* child resource usage */
```

SunNet Manager 2.1, 2.2, 2.2.3, and 2.3 `reap` function:

Code Example 6

```
reaper(sig, child_pid, status, rusage)
int sig;              /* signal number caught */
int child_pid;        /* child's pid */
int *status;          /* child's return status */
struct rusage *rusage;/* child resource usage */
```

SunNet Manager 2.0 `shutdown` function:

Code Example 7

```
shutdown(sig, code, scp, addr)
int sig;              /* signal caught */
int code;             /* additional signal info */
struct sigcontext;    /* saved process context */
char *addr;           /* additional address info */
```

SunNet Manager 2.1, 2.2, 2.2.3, and 2.3 `shutdown` function:

Code Example 8

```
shutdown_agent (sig)
int sig;              /* signal caught */
```

13.1.9 Assign an Agent Name

To communicate with an agent, Site/SunNet/Domain Manager services map the agent name to a unique RPC program number. Given the agent RPC program number and the name of the host where the agent resides, a Site/SunNet/Domain Manager application can send requests directly to the agent.

Each agent name must be unique within an NIS/NIS+ (or local host) namespace. Uniqueness is required to ensure each agent name is mapped to a unique RPC program number. This means that if the NIS/NIS+ is running, the agent name must be unique within the NIS domain's `rpc.bynumber` map. If NIS/NIS+ is not running, the agent name must be unique within the local `/etc/rpc` file.

When you pick a name for your agent you should verify the name is unique. If NIS is running, type the following from the command line:

```
host% ypcat rpc.bynumber | grep <my-agent-name>
```

If NIS is not running, type the following:

```
host% grep <my-agent-name> /etc/rpc
```

If you don't see a match, your name is unique.

13.1.10 Register the Agent RPC Program Number

Once you have a name for your agent, assign your agent an RPC program number. While you are developing your agent, you can use a temporary RPC program number, which must be unique within your `rpc.bynumber` map (or the `rpc` file). You should assign a number from the range intended for debugging new programs, which extends from `0x20000000` to `0x3fffffff`.

Before picking your temporary RPC number, verify its uniqueness:

```
host% ypcat rpc.bynumber | grep <my-RPC-number>
```

or if you are not running NIS, enter the following:

```
host% grep <my-RPC-number> /etc/rpc
```

Again, if you don't see a match, your number is unique.

The RPC version number is also required to uniquely identify an agent service. For this release, all agents should use the RPC version number `NETMGT_VERS`, as defined in the header file `netmgt_rpc.h`.

Once you have your agent RPC program name, number and version, ask your system administrator to add the information to the NIS `rpc.bynumber` map (or `rpc` file).

After you have completed your agent and want to share it with others, ask Sun Microsystems to assign you a permanent RPC number. Refer to Chapter 3 of the *SunOS Remote Procedure Call Programming Guide* for instructions.

13.2Test the Agent

You need to start the logger `na.logger` from the command line before you start the generic manager `snm_cmd`. On a SunOS4.x machine, enter the following:

```
host% /usr/snm/agents/na.logger
```

If you've installed this product on a Solaris 2.x machine, the command would be the following:

```
host% /opt/SUNWconn/snm/agents/na.logger
```

The logger writes all reports to the log file specified by the "monitor-log" keyword in the `snm.conf` file. You can change the path name of the log file by editing the line associated with the keyword.

The logger registers with the local event dispatcher to receive event reports. When your agent sends an event report, it is sent to the event dispatcher. The event dispatcher logs a copy in the file specified by the “event-log” keyword in `snm.conf`. The event dispatcher then sends the report to the logger. The logger writes it in its own file.

The `snm.conf` file is located in the `/etc` directory if you’ve installed this product on a Solaris 1.1 machine; it is located in `/etc/opt/SUNWconn/snm` for Solaris 2.x installations.

13.2.1 `snm_cmd`

`snm_cmd` is a generic manager for Site/SunNet/Domain Manager agents. You can test agents without having to write a dedicated manager or running the Console.

When you test your agent using `snm_cmd`, your agent sends reports to the logger. The logger stores them in the log file on your local host. See `snm.logfile(5)` for the format of the file.

When testing and debugging your agent, start your agent from the command line and tell it to set the global debugging variable `netmgt_debug` to a value between one and three so you can see agent debugging messages on your screen. All the supplied Site/SunNet/Domain Manager agents allow `netmgt_debug` to be set via the `-d` command line switch. Your agent should do the same.

```
host% myagent -d 3
```

When you start your agent without asking it to start debugging (when `netmgt_debug` is set to zero) from the command line or from `inetd`, you will *not* see debugging messages on your screen.

13.2.2 Verifying the Agent Schema

Once you have written your agent schema, use the `–v` option of `snm_cmd` to verify the group and attribute names and values defined in the agent schema match those in your agent:

```
host% snm_cmd –v –f myagent.schema
```

13.2.3 Test the Agent

The verification check done by `snm_cmd -v` can give you a high degree of confidence that your agent and schema match, and that your agent is at least reporting information. However, there are several tests you should do yourself. Use `snm_cmd` to do your testing. These are recommended tests, and may be done in any order.

- Look at the values in the logger file, and compare them against the values that should be returned. In other words, make sure the agent is reporting *correct* values.
- Test the behavior of the agent with various count and interval values (use the `-c` and `-i` options). Recommended values:

Table 13-1 Recommended Count/Interval Tests

Count	Interval	Interpretation
0	0	Forever. Agent picks the interval.
1	0	Once. Interval doesn't matter.
5	0	Five times. Agent picks the interval.
0	1	Forever, one second apart.
0	10	Forever, ten seconds apart.
5	10	Five times, ten seconds apart.

- Ask for group names (use the `-g` option) the agent doesn't know about.
- Ask for an event report (use the `-n`, `-r`, and `-e` options) for attributes the agent doesn't know about.

- Test the agent with both bad and good keys (use the `-k` option) and with no key. Make sure the agent can return the first row and the last row of the table (in different reports).
- If the agent reports tables, see what is returned when only one row of the table has an event occur in it. What is returned when more than one row has an event occur? Test event reports for the first and last rows.
- Test the agent with various options (use the `-o` option).
- Try various target names (use the `-t` option) that are the same and different as the name of the host where the agent is running.
- Try to get the agent to return each possible error code defined in the `agentErrors` record in the agent schema.
- If the agent does extra processing when an event occurs, you should test it.

13.3 Console Integration

Once you are satisfied your agent is performing correctly, install it on the systems where you want it to run, then integrate it with the Console's Management Database (MDB).

13.3.1 Install the Agent

For *each system* where you want your agent to run,

- Copy your agent to the directory where the other Site/SunNet/Domain Manager agents are installed. If no Site/SunNet/Domain Manager agents have been installed on the system, first run the utility `getagents.`
- Add the following entry to `snm.conf`:

```
na.<my-agent-name>0
```

This will set your agent security level on this host to zero (no security checking). Optionally, you can set your agent to any value between 1 and 5. See the *Administration Guide* for information on security.

- Add an entry for your agent to `/etc/inetd.conf` to allow `inetd` to automatically start your agent when a manager sends a request to your agent. Here's a summary of the `inetd.conf`(5) file format.

```
na.<agent-name>/10 tli rpc/udp wait root <agent program absolute pathname> <agent-name> <arguments>
```

For example, the entry for the `iproutes` agent (on a Solaris 2.x machine) is:

```
na.hostmem/10 tli rpc/udp wait root /opt/SUNWconn/snm/agents/na.iproutes na.iproutes
```

- Force `inetd` to reread its configuration file by sending it a `SIGHUP` signal:

```
host% kill -HUP <inetd's process ID>
```

13.3.2 Update the MDB

On the management station (where the Console runs),

- Copy your agent schema to where the agents and schema files have been installed. This directory is defined by the *schema* keyword in `snm.conf`. By default, agent schema files are in the `agents` directory.
- Add the agent or proxy sub-record to the MDB element cluster record. See the *Administration Guide* for more information.
- Start the Console with the `-i` option described in `snm(1)`. Your agent name will appear in the glyph menus for each element where your agent is defined. It will also appear in the Properties sheet for every element, where you can optionally specify what systems can be managed by your agent.

For complete details on the operation of the Console and the MDB, see the *Administration Guide.*

13.4 Summary

You have completed the testing and integration of your agent. Make sure you document your agent so others know how to use it, or you know how to use it six months from now.

If you are going to distribute the agent outside your site, make sure you first change your agent's RPC number to a permanent one assigned by Sun Microsystems. Refer to Chapter 3 of the *SunOS Remote Procedure Call Programming Guide* to request a permanent RPC number.

Converting an Existing Application to an Agent

14

This chapter shows an example of quick agent development using an existing application. The example introduces a simple intermediate programming layer between a generic agent services interface and the application. Thus, converting the application to an agent becomes an almost mechanical process.

The examples assume that Site/SunNet/Domain Manager has been installed on a system being used to develop agents. All files cited are presumed to be in the sample agent directory:

- `/opt/SUNWconn/snm/src/sample`, if you've installed the software on a Solaris 2.x machine
- `/usr/snm/src/sample` for a SunOS 4.x machine

(Chapter 13, "Testing and Integration," has details on how to compile your agent and install it into the network.)

The example uses the following staged approach:

1. **Write and test a standalone application that gets the information desired.**
2. **Organize the information into groups of attributes and write the agent schema file.**
3. **Rewrite and test the application with a reporting interface not including agent services.**
4. **Build an agent by including agent services and testing with `snm_cmd`.**
5. **Test the agent with the Site/SunNet/Domain Manager Console, `snm`.**

14

In this chapter, you will write an agent by modifying an existing sample agent. The output shown herein is based on the sample agent. Yours will be different, specific to your application. Also, note this chapter does not discuss agent initialization, startup, or shutdown.

14.1 Write and Test the Standalone Program

Start with a standalone application. The example application reports system information, for example, machine type, and the date.

The following listing shows the source code for the kernel memory buffer application.

```
/* application.c */

#include <sys/systeminfo.h>
#include <time.h>

extern void send_error();

static char *day_of_week[7] = {
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
};

static char *month_of_year[12] = {
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December" };

#ifndefAGENT
/*
```

```
------------------------------------------------------------------------------------------
*  main (standalone application)
------------------------------------------------------------------------------------------
*/
main(argc, argv)
     int argc;
     char **argv; {

#else
#include <netmgt/netmgt.h>

/*
------------------------------------------------------------------------------------------
*  sample_data - sample performance data and build report
*   returns TRUE if successful; otherwise returns FALSE
------------------------------------------------------------------------------------------
*/
bool_t
sample_data(system, group, key)
    char *system;/* system name */
    char *group;/* group name */
    char *key;/* table key name */
{
#endif

#define SYSNAME_LEN 25
#define RELEASE_LEN 7
#define MACHINE_LEN 7
#define SERIALNUMBER_LEN 50

    char message[64];/* error message buffer */

    char sysname[SYSNAME_LEN];/* OS name */
    char release[RELEASE_LEN];/* OS release */
    char machine[MACHINE_LEN];/* machine type (eg. sun4c) */
    char serialnumber[SERIALNUMBER_LEN];/* machine's built in serial # */

    char *weekday;
    char *month;
    shortday;
    shortyear;
```

```
    struct tm *time_info;
    time_ttime_val;

    char *group;
    if (argc != 2) {
       printf("usage:  na.sample group\n");
       exit(0);
    }
    group = argv[1];

   if (strcmp(group, "sysinfo") == 0) {
       sysinfo(SI_SYSNAME, sysname, SYSNAME_LEN);
       sysinfo(SI_RELEASE, release, RELEASE_LEN);
       sysinfo(SI_MACHINE, machine, MACHINE_LEN);
       sysinfo(SI_HW_SERIAL, serialnumber, SERIALNUMBER_LEN);

    /* print sysinfo results */
    (void) printf("sysname = %s\n", sysname);
    (void) printf("release = %s\n", release);
     (void) printf("machine = %s\n", machine);
     (void) printf("serialnumber = %s\n", serialnumber);

}
    if (strcmp(group, "date") == 0) {
       time_val = time((time_t *)NULL);
       time_info = localtime(&time_val);

       /* convert values */
       weekday = day_of_week[time_info->tm_wday];
       month = month_of_-year[time_info->tm_mon];
       day = time_info->tm_mday;
       year = time_info->tm_year + 1900;

       /* print out date report */
       (void) printf("weekday = %s\n", weekday );
       (void) printf("month = %s\n", month);
       (void) printf("day = %d\n", day);
       (void) printf("year = %d\n", year);

    }
```

```
#ifndef AGENT
    exit(0);
#else
     /* indicate success */
     return TRUE;
#endif
}
```

Use the Makefile in the sample agent directory to build the application. Run the application to get a feel for the information it collects. (Note the values reported in these examples will probably differ from those on your system.) Enter the command:

```
host% cd <sample-agent-path>
```

where <*sample-agent-path*> is `/opt/SUNWconn/snm/src/sample` for a Solaris 2.x installation or `/usr/snm/src/sample` for the Solaris 1.x version.

Now enter:

```
host% make application
```

This command creates an executable application in the file `Application/na.sample`. Run the application:

```
host% Application/na.sample sysinfo
sysname = SunOS
release = 5.1
machine = sun4m
serialnumber = 1918901280

host% Application/na.sample date
weekday = Friday
month = March
day = 12
year = 1993
host%
```

14.2 Organize the Information and Write a Schema File

The information provided falls into two groups: a summary of memory buffer statistics and a detailed report of statistics. From this are defined the two groups *sysinfo* and *date*.

Within these groups attributes are defined, one for each piece of information reported. Note data types of the data and assign types from Table 11-1 on page 11-3. Name each data field (using no more than 64 characters) and write a short description of each field (using no more than 1024 characters). Note any error messages in the standalone application code.

The following is the agent schema file, `sample.schema`, written for this example.

```
#
# @(#)sample.schema 2.7 93/02/01 SMI
#
# sample.schema - SunNet Manager sample statistics agent schema definition
#

agent sample
    description "sample system info agent"
    rpcid 100113
    serial 1
(

    group sysinfo
    description "system info"
    (
    string[20]sysnamedescription "operating system name"
    string[5] releasedescription "operating system release"
    string[5] machinedescription "machine name (eg. sun4, sun4c, sun4m)"
    string[50]serialnumberdescription "ascii representation of serial number"
    )
    group date
    description "date according to system"
    (
    string[10] weekdaydescription "day of the week"
    string[10] monthdescription "current month "
    short        daydescription "current day of month [number 1 - 31]"
    short        yeardescription "year"

    )
    agenterrors
    (
    1  "Can't report summary group"
    2  "Can't report detail group"
    )
)
```

14.3 `isGroup()` Function

After defining the schema, you should write a function to validate group names. The validation routine will be used by the request verification routine, `verify_request()`, in the file, `request.c`. In the example, the group names are placed into the `groupnames` array in the file `schema.c` as shown below. (The `#ifdef`'ed code in the file is used during the testing described in Section 14.4, "Rewrite with the Reporting Interface," on page 14-11.)

```
/* schema.c */

#include <netmgt/netmgt.h>

/* ------------------------------------------------------------------------------------
 *  groupnames - group names defined in the agent schema
 * ------------------------------------------------------------------------------------
 */
static char *groupnames[] = {
    "sysinfo",
    "date",
    0
};
/* ------------------------------------------------------------------------------------
 *  isGroup - determines if 'groupname' is a group
 *  returns TRUE if a group; otherwise returns FALSE
 * ------------------------------------------------------------------------------------
 */
bool_t
isGroup(groupname)
    char *groupname;
{
    register char **p;/* utility pointer */

    for (p = groupnames; *p; p++)
    if (strcmp(groupname, *p) == 0)
        return TRUE;

    return FALSE;
}

#ifdef NOAGENTSERVICES
/* ------------------------------------------------------------------------------------
 *  main - main routine to debug the report building routines.  This section of code is
 *  used to test the application with the agentinterface but without using the agent services
 * ------------------------------------------------------------------------------------
 */
```

```
main()
{

    register char **p;/* utility pointer */
    char *system = "<local host>";/* system name */
    char *key = "<no key>";/* table key */

    /* report all agent groups */
    for (p = groupnames; *p; p++) {
    (void) printf("system = \"%s\", group = \"%s\", key = \"%s\"\n",system, *p, key);
    sample_data(system, *p, key);
    }
    exit(0);
}

#endif
```

14.4 Rewrite with the Reporting Interface

The reporting interface is a simplified programming interface to agent services. The idea is to isolate the application and agent services so the application can be transformed into an agent independent of the way those services are used. In this way, you are “bringing agent services to the application” rather than vice versa.

The reporting interface consists of the files: `start.c`, `request.c`, and `interface.c`. These files are in the sample agent directory. Each file is responsible for a specific function as follows:

`start.c`
: contains the agent main routine and executes agent initialization and startup.

`request.c`
: contains routines that execute request verification and dispatching. The dispatch function contains the reporting algorithm used by the agent. In the example, the agent attributes are assumed to be instantaneously sampled and reported periodically, on the interval specified in the request.

`interface.c`
contains a wrapper to the agent service routines that build message lists for event reports, data reports, and error reports. The wrapper reduces these calls to routines that sample attributes by data type. Calls to the wrapper routines can be embedded in the application at the point where attributes are sampled. This file also contains `#ifdef`'ed code for testing with the sample interface but without agent services.

These files contain the interfaces to agent services. The last file is the intermediate layer that ties the agent service interface to the application. In this section, you will add the intermediate layer to the application. In the next section, you will add the agent services interface.

The intermediate layer consists of the following wrapper routines in `interface.c`:

`send_error()`
sends back errors detected by the standalone application code. The arguments consist of a system error code, an optional agent error code for agent defined errors and an optional message string which describes the specific error.

`get_option_string()`
retrieves the options string passed in the agent request. The options string is set in the Options field of the Console's data report and event report windows.

`fe()`
is a set of routines for building data buffers by data type. These routines call `build_report()` which builds the data or event report.

```
/* interface.c */

#include <netmgt/netmgt.h>

#ifndef NOAGENTSERVICES
/* ------------------------------------------------------------------------
 * send_error - send an error report
 *   no return value
 * ------------------------------------------------------------------------
 */
void
send_error(service_error, agent_error, message)
     Netmgt_stat service_error;/* service error code */
     u_int agent_error;/* agent error code defined in agent schema */
     char *message;/* error message string */
{
     Netmgt_error error;/* error report buffer */

     error.service_error = service_error;
     error.agent_error = agent_error;
     error.message = message;
     if (!netmgt_send_error(&error))
          NETMGT_DBG("na.sample: can't send error report: %s\n",
              netmgt_sperror());
     return;
}

#else
void
send_error(system_error, agent_error, message)
     Netmgt_stat system_error;/* system error code */
     u_int agent_error;/* agent error code */
     char *message;/* error message */
{
     printf("system_error = %d, agent_error = %d, message = \"%s\"\n",
          system_error, agent_error, message);
     return;
}
#endif
```

```
#ifndef NOAGENTSERVICES
/* ------------------------------------------------------------------
 * get_option_string - retrieve the options string from the agent
 *   request. The options string is set in the Options field of
 *   the Console's data and event request windows. This
 *   function returns the option string if there is one; otherwise
 *   it returns (char *)NULL.
 * ------------------------------------------------------------------
 */
char *
get_option_string()
{
     Netmgt_arg arg;/* request argument buffer */

     if (!netmgt_fetch_argument(NETMGT_OPTSTRING, &arg))
          return (char *) NULL;
     return (char *) arg.value;
}
#else
char *
get_option_string()
{
     return ("<option string>");
}
#endif


#ifndef NOAGENTSERVICES
/* ------------------------------------------------------------------
 * build_report - build a data or event report from data buffer
 *   returns TRUE if successful; otherwise returns FALSE
 * ------------------------------------------------------------------
 */
bool_t
build_report(data)
     Netmgt_data *data;/* data buffer */
{
     bool_t event;/* whether an event occurred */
     Netmgt_error error;/* error buffer */
```

```
     /* build the report */
     if (!netmgt_build_report(data, &event)) {
          /* fetch error buffer */
          if (!netmgt_fetch_error(&error)) {
               NETMGT_DBG("na.event: can't fetch error: %s\n",
                    netmgt_sperror());
               return FALSE;
          }
          /* send the error report to the rendezvous */
          if (!netmgt_send_error(&error)) {
               NETMGT_DBG("na.event: can't send error report: %s\n",
                    netmgt_sperror());
               return FALSE;
          }
          return FALSE;
     }
     /* indicate that an event occurred if tracing */
     if (event)
          NETMGT_DBG("*** na.sample: event occurred ***\n");
     return TRUE;
}

#else
bool_t
build_report(data)
     Netmgt_data *data;     /* data buffer */
{
     printf("<build report>\n");
}
#endif


#ifndef NOAGENTSERVICES
/* ------------------------------------------------------------------------
 * build_<data type>
 *   The following are a set of routines for building data and
 *   event reports by data type. These routines return TRUE if
 *   successful; otherwise they return FALSE.
 * ------------------------------------------------------------------------
 */
```

```
bool_t
build_short(name, value)
     char *name;
     short value;
{
     Netmgt_data data;      /* data buffer */

     (void) strcpy(data.name, name);
     data.type = NETMGT_SHORT;
     data.length = sizeof(short);
     data.value = (caddr_t) & value;

     return build_report(&data);
}


...
#else
/* ------------------------------------------------------------------
 * This section of code is used for testing the application with
 * the reporting interface but without using agent services.
 * ------------------------------------------------------------------
 */
bool_t
build_short(name, value)
     char *name;
     short value;
{
     printf("%s: (short) %d\n", name, value);
     return TRUE;
}
...
#endif
```

14.4.1 Modify the Application

To rewrite the application, first include <netmgt/netmgt.h>. Next, rewrite the main routine to be callable from the dispatch_request() routine with the parameters system, group, and key. The three parameters are character strings indicating the system for which information is desired, the group name of the attribute group desired, and the key selector to be used if the group name specifies a table. For tables, the attribute netmgt_table_key should also be returned. This is useful for setting attribute values with the Console's Set tool or for graphing attributes from the Results Browser tool.

Error messages should be converted to calls to send_error() with the agent error code for the appropriate message as defined by the agent schema.

Reporting is carried out by group name. Calls to the build_<data_type>() routines should be bracketed by an if statement that checks for the appropriate group name.

When a report is built, the returned (boolean) value of the call should be checked. The rewritten application should return the boolean value FALSE if any of the build_<data_type>() calls failed. Otherwise it should return TRUE.

Reporting is done by using the appropriate build_<data_type>() routine for the type of information to be collected. This involves replacing printf() calls with build_<data_type>() calls. The reporting routines transcribe the information into an event or data report according to the algorithm in the file request.c.

The rewritten application is shown below. Note the modified code sections are indicated using #ifdef AGENT statements. The application can be re-compiled with the -DAGENT compiler switch set to build an agent. It can be compiled without it to obtain the original application.

```
/* application.c */

#include <sys/systeminfo.h>
#include <time.h>

extern void send_error();

static char *day_of_week[7] = {
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
};

static char *month_of_year[12] = {
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
};

#ifndefAGENT
/* ------------------------------------------------------------------------
 *  main (standalone application)
 * ------------------------------------------------------------------------
 */
```

```
*/
main(argc, argv)
    int argc;
    char **argv;
{

#else
#include <netmgt/netmgt.h>

/* ---------------------------------------------------------------------
 *  sample_data - sample performance data and build report
 *  returns TRUE if successful; otherwise returns FALSE
 * ---------------------------------------------------------------------
 */
bool_t
sample_data(system, group, key)
    char *system;/* system name */
    char *group;/* group name */
    char *key;/* table key name */
{
#endif

#define SYSNAME_LEN 25
#define RELEASE_LEN 7
#define MACHINE_LEN 7
#define SERIALNUMBER_LEN 50

    char message[64];/* error message buffer */

    char sysname[SYSNAME_LEN];/* OS name */
    char release[RELEASE_LEN];/* OS release */
    char machine[MACHINE_LEN];/* machine type (eg. sun4c) */
    char serialnumber[SERIALNUMBER_LEN];/* machine's built in serial # */

    char *weekday;
    char *month;
    shortday;
    shortyear;

    struct tm*time_info;
    time_ttime_val;
```

```
#ifndef AGENT
    char *group;
    if (argc != 2) {
       printf("usage:  na.sample2 group\n");
       exit(0);
    }
    group = argv[1];
#endif
   }
   if (strcmp(group, "sysinfo") == 0) {
       sysinfo(SI_SYSNAME, sysname, SYSNAME_LEN);
       sysinfo(SI_RELEASE, release, RELEASE_LEN);
       sysinfo(SI_MACHINE, machine, MACHINE_LEN);
       sysinfo(SI_HW_SERIAL, serialnumber, SERIALNUMBER_LEN);

#ifndef AGENT
    /* print sysinfo results */
    (void) printf("sysname = %s\n", sysname);
    (void) printf("release = %s\n", release);
    (void) printf("machine = %s\n", machine);
    (void) printf("serialnumber = %s\n", serialnumber);
#else

   /* build sysinfo report */
      if (!build_string("sysname", sysname) ||
          !build_string("release", release) ||
          !build_string("machine", machine) ||
     !build_string("serialnumber", serialnumber)) {
     /*
    * fatal error - send an error report to the rendezvous and return
         */
    send_error(NETMGT_FATAL, (u_int) 1, (char *) NULL);
        return FALSE;
       }
#endif

    }
    if (strcmp(group, "date") == 0) {
          time_val = time((time_t *)NULL);
       time_info = localtime(&time_val);
```

```
        /* convert values */
        weekday = day_of_week[time_info->tm_wday];
        month = month_of_year[time_info->tm_mon];
        day = time_info->tm_mday;
        year = time_info->tm_year + 1900;

#ifndef AGENT
        /* print out date report */
        (void) printf("weekday = %s\n", weekday );
        (void) printf("month = %s\n", month);
        (void) printf("day = %d\n", day);
        (void) printf("year = %d\n", year);
#else

  /* build date report */
        if (!build_string("weekday", weekday) ||
            !build_string("month", month) ||
            !build_short("day", day) ||
    !build_short("year", year)) {
    /*
    * fatal error - send an error report to the rendezvous and return
              */
        send_error(NETMGT_FATAL, (u_int) 1, (char *) NULL);
        return FALSE;
        }
#endif
    }

#ifndef AGENT
    exit(0);
#else
    /* indicate success */
    return TRUE;
#endif
```

14.4.2 Build with Report Interface

Having made these changes to the application, build a test version that omits agent services using the following commands:

```
host% cd <sample-agent-path>
```

where <*sample-agent-path*> is `/opt/SUNWconn/snm/src/sample` if this is a Solaris 2.x installation or `/usr/snm/src/sample` if you have installed the Solaris 1.1 version of this product.

Then enter the following command:

```
host% make test
```

This command creates a test application executable in the file `Test/na.sample`.

14.4.3 Test with Report Interface

Run the test application and verify the group names and that all appropriate attributes are sampled. It is an error to return an attribute not specified for a group as defined by the agent schema.

The problem is that only root can access kernel memory structures. Become root and try again:

```
host% Test/na.sample
system = "<local host>", group = "sysinfo", key = "<no key>"
sysname: "SunOS"'
release: "5.1"'
machine: "sun4m"'
serialnumber: "1918901280"'
system = "<local host>", group = "date", key = "<no key>"
weekday: "Friday"'
month: "March"'
day: (short) 12
year: (short) 1993
host%
```

14.5 Build the Agent and Test with `snm_cmd`

Build the agent with the agent services library. In the sample code directory enter the command:

```
host% make agent
```

This command creates an executable agent in the file `Agent/na.sample`. Testing the agent with `snm_cmd`. `snm_cmd` is described in greater detail in Section 13.2, “Test the Agent,” on page 13-7. Ensure the logger is running. If it isn’t running, start it with the following command,

```
host% <agents-path>/na.logger
```

where *<agents-path>* is `/opt/SUNWconn/snm/agents` if this is a Solaris 2.x installation or `/usr/snm/agents` if this is the SunOS 4.x version.

Ensure the sample agent service is registered. (See Section 13.1.10, “Register the Agent RPC Program Number,” on page 13-6.) The mechanism for starting the agent must be determined. The agent can be started either automatically by `inetd` or manually from the command line. Manually starting an agent has the advantage that messages from the agent will be reported and any command

line debug switches can be used. These are unavailable when using `inetd`. When using `inetd` to start the agent, verify that an entry for the sample agent exists in `/etc/inetd.conf`, and it refers to the proper file to execute.

Assuming that a Solaris 2.4 version of Site/SunNet/Domain Manager is installed in `/opt/SUNWconn/snm`, here is what the sample agent entry should look like.

```
sample/10 TLI rpc/udp wait root /opt/SUNWconn/snm/src/sample/Agent/na.sample na.sample
```

Assuming the Solaris 1.1.1 version of this product installed in `/usr/snm`, here is what the sample agent entry should look like.

```
sample/10 TLI rpc/udp wait root /usr/snm/src/sample/Agent/na.sample na.sample
```

If the `inetd.conf` file must be modified, remember to send a `SIGHUP` signal to the `inetd` process so that the change will be noted.

Once the mechanism for starting the agent has been established, it can be tested with `snm_cmd`. Test the agent by making data requests for all groups. The following is an example with the Solaris 2.4 version of this product:

```
host% /opt/SUNWconn/snm/bin/snm_cmd -d -a sample -g sysinfo -s
success: timestamp = 629774235.520002
sysname: SunOS
release: 5.1
machine: sun4m
serialnumber: 1918901280

host% /opt/SUNWconn/snm/bin/snm_cmd -d -a sample -g date -s
success: timestamp = 629774257.220002
weekday: Friday
month: March
day: (short) 12
year: (short) 1993
host%
```

The next example illustrates this use of `snm_cmd` in a Solaris 1.1 environment:

```
host% /usr/snm/bin/snm_cmd -d -a sample -g sysinfo -s
success: timestamp = 629774235.520002
sysname: SunOS
release: 4.1.3
machine: sun4m
serialnumber: 1918901280

host% /usr/snm/bin/snm_cmd -d -a sample -g date -s
success: timestamp = 629774257.220002
weekday: Friday
month: December
day: (short) 17
year: (short) 1993
host%
```

14.6 Test the Agent with the Console

Test the agent with `snm`. Ensure the new agent schema file is specified either by being moved into a directory listed by the *schemas* keyword in `snm.conf` or by being included on the `snm` command line.

The system where the test will be made should have its management database (MDB) record cluster modified to include a reference to the sample agent. For more information on the MDB and how to specify element representations, see the *Administration Guide*. The agent will be tested on the same system on which it was built. Following is an example record cluster for the test system used in this example:

```
cluster(
    component.sun3 ( prospero 128.10.2.26 "Earvin Johnson" )
    membership ( Home )
    glyphColor ( 208 0 255 )
    agent ( sample )
)
```

1. Run `snm` and bring up the Data Log window.

2. **Bring up the glyph menu of the test system and do a Quick Dump of all the groups under the `sample` agent.**

3. **Try to send some event reports.**
   This agent application collects static data so it is easy to set an event on any particular value. Agents with rapidly changing data can often be induced to send a report by setting an attribute threshold to a value the attribute does not normally have.

By using and modifying the existing `na.sample` agent code, you streamline the agent writing process. At this point, `na.sample` is your new agent. To complete the process, rename the agent as appropriate and follow the process described in Section 13.1.9, “Assign an Agent Name,” on page 13-6.

Part 3— Man Page Summaries

Man Page Summary for Writers of Manager Applications

A

This appendix lists the on-line manual pages for utilities, routines, and file formats that are useful to manager writers. User manual pages are listed in the *Administration Guide*. Manual pages for agent writers are listed in Appendix B, "Man Page Summary for Writers of Agent Software."

A.1 Setting the MANPATH Variable

To use the `man`ual pages on-line, add the Site/SunNet/Domain Manager man pages directory to the `MANPATH` environment variable statement in your `.cshrc` or `.profile` file.

A.1.1 MANPATH Setting for Solaris 2.4

If you installed the Solaris 2.4 version of Site/SunNet/Domain Manager software in `/opt/SUNWconn/snm`, enter one of the lines below on a command line or in your shell startup file.

For a C-shell, in your `$HOME/.cshrc` file enter:

```
setenv MANPATH ${MANPATH}:/opt/SUNWconn/man
```

For a Bourne or Korn shell, in your `$HOME/.profile` file enter:

```
set MANPATH=${MANPATH}:/opt/SUNWconn/man
export MANPATH
```

A.1.2 MANPATH Setting for Solaris 1.1.1

If you installed the Solaris 1.1.1 version of Site/SunNet/Domain Manager software in `/usr/snm`, enter one of the lines below on a command line or in your shell startup file.

For a C-shell, in your `$HOME/.cshrc` file enter:

```
setenv MANPATH ${MANPATH}:/usr/man
```

For a Bourne or Korn shell, in your `$HOME/.profile` file enter:

```
set MANPATH=${MANPATH}:/usr/man
export MANPATH
```

A.2 Utilities

Manual pages are available for the following utilities:

`snm_cmd`
: The command-line manager for agents. It provides a way to send requests to agents without having to run the Console or write dedicated test programs.

`snm_kill`
: Stop one or more agent requests. This command allows you to stop requests from other than where they were started.

`snm_parser`
: An example application for parsing schema files. This program is provided in source form to allow you to easily build applications that can read schema files.

A.3 Manager Services Library Routines

Manual pages are available for the following Manager Services library routines:

`netmgt_alloc_manager_id`
Allocates a unique ID to the calling manager application. This ID is the inode number of a temporary file in the product directory.

`netmgt_dbg`
Three macros—for each of three debug levels—used for print tracing information. The arguments are identical to those of `printf`.

`netmgt_fetch_data`
Gets data statistics from a data report and places them into a local buffer that you specify. When a rendezvous receives a data report, the first call to this function gets the first data statistic in the report. Subsequent calls get successive statistics.

`netmgt_fetch_error`
Copies the relevant portions of the global `netmgt_error` structure into a user-specified buffer.

`netmgt_fetch_event`
Gets event statistics out of an event report and places them into a local buffer. When a rendezvous receives a data report, the first call to this function gets the first data statistic in the report. Subsequent calls get successive statistics.

`netmgt_fetch_msginfo`
Obtains additional message information from a data or event report and places it into a local buffer.

`netmgt_free_manager_id`
Deallocates a manager ID previously allocated by a call to `netmgt_alloc_manager_id`().

`netmgt_get_manager_id`
Returns the value of the manager ID field in data, event, set, trap, and error reports. This function may only be called by applications that receive such reports.

`netmgt_kill_request`
Terminate a specific request on a specific host.

`netmgt_kill_request2`
Terminate all requests on a specific host started by a specific manager application.

`netmgt_oid2string`
Converts an SNMP object identifier to a text string.

`netmgt_register_callback`
Establishes a rendezvous for receiving data or event reports by registering a transient RPC callback function.

`netmgt_register_rendez`
Instructs the event dispatcher on a specific host to start sending event reports from a specified agent to a specified rendezvous.

`netmgt_request_agent_ID`
Requests identification from an agent.

`netmgt_request_data`
Sends a request to an agent for a specific group's attribute values. The caller must have previously called `netmgt_set_instance`(3n) to define the system, group, and optional key of the request. If any agent-specific options are to be set, the caller must have previously called `netmgt_set_argument`(3n).

`netmgt_request_deferred`
Get any available deferred data reports (that is, data reports for which a manager application has previously requested an agent to defer sending).

`netmgt_request_events`
Sends a request to an agent to watch for interesting events related to specific attributes.

`netmgt_request_set`
Sends a request to an agent to set the values of one or more group or table attributes. The return value is a boolean value (true or false).

`netmgt_request_set2`
Identical to `netmgt_request_set` except that the return value for this function is the timestamp of the request, if successful, Otherwise the return is null.

`netmgt_request_set`

`netmgt_restore_argument`
Sets an argument from the contents of the buffer pointed to by ‘argument’.

`netmgt_restore_request`
Sets the current request state from the contents of the buffer pointed to by 'request_info'.

`netmgt_restore_threshold`
Sets a threshold from the contents of the buffer pointed to by 'threshold'.

`netmgt_save_argument`
Man page also includes `netmgt_restore_argument`. Copies information about an optional argument into the buffer pointed to by the sole argument to the function. The “restore” function sets an argument from the contents of the buffer pointed to by the sole argument to `netmgt_restore_argument`.

`netmgt_save_attribute`
This routine is usedprimarily by the agent programs. When a data request is sent to an agent, the agent program should invoke this routine to save all the attributes requested and accordingly fetch the data from the managed device. The number of attributes requested is available in the request info structure; this routine should be called accordingly. If no attributes are specified then all the attributes in a given group are retrieved.

`netmgt_save_request`
Man page also includes `netmgt_restore_request`. Copies information about the current request state into the buffer pointed to by the sole argument to the function. The “restore” function sets the current request state from the contents of the buffer pointed to by the sole argument.

`netmgt_save_threshold`
Man page also includes `netmgt_restore_threshold`. Copies information about an event threshold into the buffer pointed to by the sole argument to the function. The “restore” function sets a threshold from the contents of the buffer pointed to by the sole argument to `netmgt_restore_threshold`.

`netmgt_set_argument`
Specifies optional request arguments to be sent to an agent. These optional arguments are agent-specific, for directing the agent to perform actions not accounted for by the standard request mechanisms.

`netmgt_set_attribute`
Allows the management applications, such as the console, to specify the set of attributes to be retrieved in a data report request.

While requesting data requests this routine must be called after *netmgt_set_instance()* and *netmgt_set_argument()*, but before the *netmgt_request_data()* call. This routine should be called separately for each and every attribute to be retrieved in a data request. If this routine is not invoked, then the behaviour is same as in earlier product releases, which is to retrieve all the attributes in any given group.

`netmgt_set_debug`
Sets one of four debugging levels for application execution tracing.

`netmgt_set_instance`
Specifies the parameters—including the system name, attribute group, and optional key into a table of attributes—an agent should use when making a request.

`netmgt_set_manager_id`
Covers four functions, to allocate, free, set, and get a manager ID. A manager ID is associated with an individual manager application. It is used to distinguish among requests started by multiple manager applications running on the same machine.

`netmgt_set_threshold`
Sets the event report threshold for an attribute value.

`netmgt_set_value`
Specifies a set request argument to be sent to an agent. This function is called once for each attribute to be set.

`netmgt_sperror`
Returns the error description string associated with the last error contained in the global variable `netmgt_error`.

`netmgt_unregister_callback`
Unregisters a specified transient RPC callback function.

`netmgt_unregister_rendez`
Asks the event dispatcher on a specified host to stop sending event reports of a specified priority from agents of a specified RPC program number to the rendezvous process with a specified RPC program and version number on a specified host.

A.4 Database Library Routines

Manual pages are available for the following database API library routines:

`snm_error`
An external variable containing the error reason for the last error. It should not be used if the return code of the `snmdb` routine is true.

`snmdb_add`
Adds a new element into the database.

`snmdb_add_agent`
Adds an agent record for the element. You add an agent record in the element buffer by specifying the name of the agent and, if the agent is a proxy, the name of the system providing the proxy function.

`snmdb_add_alias`
Adds an alias record for the element. The name of element used at creation time is its primary name. There may be one or more secondary names added later for the element by adding alias records. Both primary and secondary names should be unique in the elements name space.

`snmdb_add_connection`
Connections are described in the connect record in the element buffer. This function adds a connect record into the buffer, specifying the target that the element is connected to.

`snmdb_add_to_view`
The view in which the element exists is described in the element's membership record. This function adds a membership record with the specified view name into the element buffer.

`snmdb_console_load`
Causes the Console to load a specified ASCII management database file.

`snmdb_console_reload`
Causes the Console to reinitialize the runtime database and load the specific server.

`snmdb_console_save_components`
Causes the SNM Console to save all instances in its runtime database to a specified ASCII management database file.

`snmdb_delete`
Deletes an existing element from the database.

`snmdb_delete_agent`
Deletes an agent record from the element buffer.

`snmdb_delete_alias`
Deletes an alias record for the element.

`snmdb_delete_color`
The color of an element is described in the glyphColor record in the element buffer. This function deletes the glyphColor record from the buffer.

`snmdb_delete_connection`
Connections are described in the connect record in the element buffer. This function looks through all of the connect records in the buffer and deletes the connection between the element and the target that the element is connected to.

`snmdb_delete_from_view`
The view in which the element exists is described in the element's membership record. This function deletes a membership record with the specified view name from the element buffer.

`snmdb_enumerate_agents`
Agents and proxy agents are described in the agent record in the element buffer. This function enumerates the agent records in the buffer and returns a pointer to the list of all agent names.

`snmdb_enumerate_aliases`
An element may be identified by its secondary names specified in the alias records in the element buffer. This function enumerates the alias records in the buffer and returns a pointer pointing to the list of all alias names.

`snmdb_enumerate_connections`
Connections are described in the connect record in the element buffer. This function enumerates the connect records in the buffer and returns a pointer to the list of all the names to which the element is connected.

`snmdb_enumerate_elements`
Enumerates all the elements in a specified view or in all views if no view is specified.

`snmdb_enumerate_views`
The view in which the element exists is described in the element's membership record. This function enumerates the membership records in the element buffer and returns a pointer to the list of all view names in which the element exists.

`snmdb_free_enumeration_handle`
Frees the storage allocated previously for enumerating element names.

`snmdb_free_list`
Frees the storage allocated previously for enumerating agents, views, or connections.

`snmdb_get_agent`
Retrieves an agent record for a specified agent, indicating whether the agent is a proxy and, if it is, the name of the system that provides the proxy function.

`snmdb_get_color`
Determines the color value in RGB numbers of the element loaded in the buffer. The colors are contained in the glyphColor record in the buffer.

`snmdb_get_element_glyph`
Returns the glyph/icon file name for a particular type of element.

`snmdb_get_element_type`
Returns the type of an element loaded in the internal buffer.

`snmdb_get_next_element`
Gets the next element in an enumerated list that is generated by `snmdb_enumerate_elements`.

`snmdb_get_property`
Gets the value and data type of a particular property of an element that has been read into the internal buffer.

`snmdb_get_view`
The membership records in the element buffer describe the x and y coordinate values of the element in the views and whether the glyph of the element is stacked on top of another glyph (`z` value). This function retrieves the x and y coordinates in the membership record that describe the position of the element in a specified view. It also retrieves the `z` value, used when glyphs are stacked on top of one another.

`snmdb_init_buffer`
Before adding a new element to the database, you must call this function to initialize the buffer for the element and to set the name and type for the element. See this man page for other function calls needed to add an element into the database.

`snmdb_lock`
Locks the database so that entries can be changed. Returns a boolean.

`snmdb_open`
Called before any database operations. Opens the database and performs necessary setup work. Returns a boolean.

`snmdb_read`
Reads an existing element from the database into the specified internal buffer. This is a pre-requirement for retrieving or updating any properties of a particular element.

`snmdb_save_element`
Writes the contents in the element buffer into a specified ASCII file, which duplicates the Console's save-file.

`snmdb_set_color`
Changes the color value in RGB numbers of the element in the buffer. The colors are contained in the glyphColor record in the buffer. If the glyphColor record is not already defined in the buffer, a new one is created.

`snmdb_set_property`
Sets the value of a particular property of an element. Called in a sequence with other functions, described in this man page, to initialize, read, and update an element.

`snmdb_unlock`
Unlocks the database and notifies the Console to update its views.

`snmdb_update`
Writes an updated element back into the database. Called as the last function in the sequence described in the `snmdb_set_property` man page.

Man Page Summary for Writers of Agent Software

B

This appendix lists the on-line manual pages for utilities, routines, and file formats useful to agent writers. The *Administration Guide* lists the manual pages for the user commands. Appendix A, "Man Page Summary for Writers of Manager Applications," lists the manual pages for manager writers.

B.1 Setting the MANPATH Variable

To use the `man`ual pages on-line, add the Site/SunNet/Domain Manager manual pages directory to the `MANPATH` environment variable statement in your `.cshrc` or `.profile` file.

B.1.1 MANPATH Setting for Solaris 2.4

If you installed the Solaris 2.4 version of Site/SunNet/Domain Manager software in `/opt/SUNWconn/snm`, enter one of the lines below on a command line or in your shell startup file.

For a C-shell, in your `$HOME/.cshrc` file enter:

```
setenv MANPATH ${MANPATH}:/opt/SUNWconn/man
```

For a Bourne or Korn shell, in your `$HOME/.profile` file enter:

```
set MANPATH=${MANPATH}:/opt/SUNWconn/man
export MANPATH
```

B.1.2 MANPATH Setting for Solaris 1.1.1

If you installed the Solaris 1.1.1 version of Site/SunNet/Domain Manager software in `/usr/snm`, enter one of the lines below on a command line or in your shell startup file.

For a C-shell, in your `$HOME/.cshrc` file enter:

```
setenv MANPATH ${MANPATH}:/usr/man
```

For a Bourne or Korn shell, in your `$HOME/.profile` file enter:

```
set MANPATH=${MANPATH}:/usr/man
export MANPATH
```

B.2 Utilities

This appendix contains manual pages for the following utilities:

`snm_cmd`
: Command-line manager for Site/SunNet/Domain Manager agents. It provides a way to send requests to agents without having to run the SNM Console.

`snm_kill`
: Stop one or more agent requests according to specified options.

B.3 Agent Services Routines

The appendix also contains manual pages for the following Agent Services API routines:

`netmgt_build_report`
Gets the current attribute value from a specified data statistic and adds it to a report. If and event has occurred for this attribute, indication of the event will be set upon return.

`netmgt_dbg`
This man page describes the three macros (corresponding to the three debugging levels) that are used for printing debugging output.

`netmgt_fetch_argument`
Fetches an optional request argument.

`netmgt_fetch_error`
Copies the relevant portions of the global structure `netmgt_error` into a user-specified error buffer.

`netmgt_fetch_setval`
Gets the next set request argument from a set request and places it into a local buffer. When an agent receives a set request, the first call to this function gets the first set request argument in the request. Successive calls get successive arguments.

`netmgt_init_rpc_agent`
Initializes the Agent Services library and registers the agent with the RPC system to receive requests via RPC from manager processes.

`netmgt_mark_end_of_row`
Adds an end-of-row string that is defined in a header file to a report. This string allows the rendezvous to determine the end of each row of a table.

`netmgt_send_error`
Sends an error report to the rendezvous, indicating that an error occurred while handling a request.

`netmgt_send_report`
Sends a data or event report to a rendezvous process. You should have previously built a report argument list with calls to `netmgt_build_report`.

`netmgt_set_debug`
Sets the debugging level for application execution tracing. Used with the macros described in the man page for `netmgt_dbg`.

`netmgt_shutdown_agent`
Unregisters the calling agent from the RPC system and then shuts it down.

`netmgt_sperror`
Returns the error description string associated with the last error contained in the global variable `netmgt_error`.

`netmgt_start_agent`
Allows the agent to receive requests. It suspends the agent to wait for incoming requests, which are passed to the agent's verify routine. If the verification succeeds, the request is then passed to the agent's dispatch routine to dispatch and execute the request.

`netmgt_start_trap`
Sets the context for sending one or more trap reports. Agents must call this function once before calling the functions to build and send a trap. Agents should not call this function again to send another trap unless they need to change trap parameters.

B.4 File Format

One file format manual page is included:

`snm_schema`
Provides information in ASCII form used to build the management database.

Index

O

P

R