

1396

IMPLEMENTING SNMP ON DEC-ALPHA OSF/1 SYSTEM

Dissertation Submitted to
JAWAHARLAL NEHRU UNIVERSITY
*in partial fulfilment of requirements
for the award of the degree of*
Master of Technology
in
Computer Science & Technology

by

SUNIL VIJAYA KUMAR GADDAM



Jawaharlal Nehru University

**SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067**

January 1997


JAWAHARLAL NEHRU UNIVERSITY
School of Computer & Systems Sciences
New Delhi - 110067

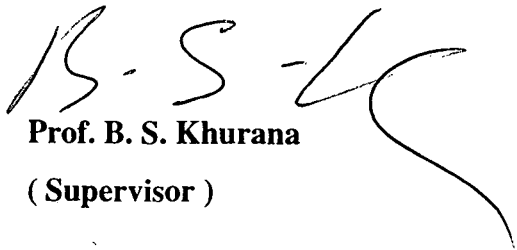
CERTIFICATE

This is to certify that the dissertation entitled
Implementing SNMP On DEC-ALPHA OSF/1
System

which is being submitted by **Mr. Sunil Vijaya Kumar Gaddam** to the **School of Computer and Systems Sciences , Jawaharlal Nehru University, New Delhi** for the award of **Master of Technology in Computer Sciences & Technology**, is a record of bonafide work by him under the supervision and guidance of **Prof.B.S. Khurana**.

This work is original and has not been submitted in part or full any University or Institution for the award of any degree.


Prof. G.V. Singh
(Dean SC & SS)


Prof. B. S. Khurana
(Supervisor)

Acknowledgements

The present work is not a result of my *effort alone*. The following persons have lent their helping hands in giving it the final shape. I sincerely acknowledge their contribution.

I want to express my heartfelt gratitude to **Prof.B.S. Khurana**, SC&SS, JNU for his *scholarly guidance* and *encouragement* during my Project Work.

I wish to thank **Prof.G.V. Singh**, Dean, SC&SS, JNU for extending lab facilities and **Prof. P.C. Saxena** for his valuable comments.

I thank my friends **Sanjay, Sanjeev, Paulraj, Joy, Bala, Vivek, Narendra** and **Subba Rao** for their help.

I acknowledge the financial assistance provided by UGC.

I'm grateful to all of them.



(**Sunil Vijay Kumar G.**)

ABSTRACT

The unprecedented growth of networks world wide and the diversification in the underlying hardware technologies have made the maintenance, management and monitoring of networks a major issue. Further, *Network Management* has assumed far greater importance with the steady growth of *networks* and *data communications* in recent times. With the ever increasing *complexity* and *heterogeneity* in the area of *networking*, there arises a need to keep the network running consistently at all times.

The *Simple Network Management Protocol* (SNMP) consists of simply composed set of network communication specifications that cover all the basics of *network management* in a method that poses little stress on an existing network. Network elements supporting SNMP provide management information to the management station on receipt of periodical query messages. The Finite State Machine (FSM) residing over the network elements (agent) interrupts these messages and provides means for access and control of the various management objects. Each SNMP *agent* maintains a conceptual set of variables specific to its functionality and depending on the request from the SNMP manager it either sets or gets the value of that particular variable.

This project report describes the Implementation of an SNMP *Manager* developed for an agent supporting BSD Unix system (OSF/1). It studies the various issues involved in a *Central Network Management Station*(CNMS) and it implements *tools* required by the SNMP Protocol(core of SNMP) such as *MIB tree* loading, *Encoder* and *Decoder* that use SNMP specific Basic Encoding Rules (BER).

Contents

1. Introduction	1
2. Network Management	4
2.1 Motivation	4
2.2 OSI Network Management Model	5
2.3 Level of Management Protocols	7
2.4 Centralised Management Vs. Decentralised Management	7
2.5 Polling Vs. Alerts	8
2.5.1 Reliable detection of failures	9
2.5.2 Complexity and performance of the ME	9
2.5.3 Response Time for problem detection	10
2.5.4 Volume of network management traffic	11
2.5.5 Ease of configuration	11
2.5.6 Potential to over-inform the NMS	11
2.6 Management Paradigms	12
2.7 Asynchronous Communication	12
2.8 The SNMP Network Management	13
2.8.1 Components of SNMP Management System	13
2.9 Summary	14
3. SNMP - The Management Protocol for TCP/IP Networks	15
3.1 Motivation	15
3.2 SNMP Architecture	15
3.2.1 Managed Nodes	16
3.2.2 Network Management Stations	16
3.2.3 Network Management Protocol	17
3.3 Elements of SNMP Architecture	19
3.3.1 Scope of Management Information	19
3.3.2 Representation of Management Information	20
3.3.3 Operation Supported on Management Information	20
3.3.4 Form and Meaning of Protocol Exchanges	21
3.3.5 Definition of Administrative Relationships	22
3.4 Proxy Management	23
3.5 Use of Transport Service	23
3.6 Instance Identification	24
3.7 Protocol	25
3.8 summary	30
4. Structure and Identification of Management Information	31
4.1 Definition of Management Information	31
4.1.1 SMI	31
4.1.2 MIB	32

4.2 Structure of Management Information	35
4.2.1 Names to identify Managed Objects	35
4.2.2 Syntax to define Object Types	36
4.2.3 Format of Managed Objects	38
4.3 Summary	39
5. Design and Implementation of SNMP on OSF/1 system	40
5.1 Major criterion in designing of an SNMP Manager	40
5.2 Design considerations	41
5.3 Modules specific to SNMP	41
5.3.1 MIB tree-loading	41
5.3.2 Encoder and Decoder that use SNMP specific Basic Encoding Rules(BER)	42
5.4 Representation	43
5.5 Implementation	44
5.5.1 Configuration	44
5.5.2 Mechanism	44
5.5.3 Space versus Time	45
5.5.4 Description of Important Routines	45
5.6 Summary	47
6. Conclusions and Future Extensions	49
6.1 Conclusions	49
6.2 Comment & Extensions	49

REFERENCES

APPENDICES

Introduction

The field of computer networks has been undergoing rapid growth. While interconnecting a collection of autonomous computers was the main theme of 70's, in 80's various economic factors and technological advantages made internetworking feasible. Internetworking is a scheme that provides universal communication services by interconnecting a collection of autonomous computer networks, irrespective of the underlying networking hardware. The rationale for such an internetworking are: no single computer network can serve all users, users desire universal communication. Thus in an Internet although each *network* might consist of an entirely different underlying technology, all hosts attached to those networks have a common view of the network. This is the power of Internet abstraction. But such an abstraction makes network management quite difficult because of the following reasons:

- * because of the Internet abstraction in computer networks there are quite a lot of different hardware products available in the market, and to develop a management technology that works on all these products is difficult
- * different administrations; all networks are not under the same administration
- * the complexity involved in an Internet is more complicated than in a simple computer network

The Simple Network Management Protocol (SNMP) was designed in mid-1980's as an answer to the communication problems between different types of networks. SNMP was developed because Internet users needed a simple, reliable, inexpensive way to manage network devices. Its initial aim was to be a *band-aid* solution to internetwork management difficulties until a better designed and more complete network manager became available. However, no better choice became available and SNMP became the network management protocol of choice. The SNMP, put forth and promoted by Internet Activities Board (IAB), provides communication between the management station and the managed node in terms of abstract objects.

SNMP, as the name suggests, shows a simplified approach to the Internet management. The way it works is very simple: It exchanges network information through messages, technically known as Protocol Data Units (PDUs). From a high level perspective, the message (PDU) can be looked at as an object that contains variables that have both titles and values.

There are five types of PDUs that SNMP employs to monitor a network: two deal with reading terminal data, two deal with setting setting terminal data, and one, the trap, is used for monitoring network events such as terminal *start-ups* or *shut-downs*.

Therefore, if a user wants to see if a terminal is attached to the network, he would use SNMP to send out a read PDU to that terminal. If the terminal was attached to the network, the user would receive back the PDU, its value being "yes, *the terminal is attached*". If the terminal was shut off, the user would receive a packet sent out by the terminal being shut off informing them of the shut down. In this instance a *trap* PDU would have been dispatched.

Because SNMP emphasizes simplicity it uses a connection less protocol to limit the amount of network overhead. SNMP built on the Simple Gateway Management Protocol (SGMP), a protocol unit was used primarily to manage internet routers. SNMP provides tools for tracking workstations, compiling statistics and resetting network links remotely. Network managers can also use SNMP to check paths for data packets.

The Advantages of SNMP

The largest advantage to using SNMP is that its design is simple, hence it is easy to implement on a large network, for it neither takes a long time to set up nor poses a lot of stress on the network. Also, its simple design makes it easy for a user to program variables they would like to have monitored, for in a more low-level perspective each variable consists of the following information:

- * the *variable* title
- * the data type of the variable (eg. integer, string)

- * whether the variable is *read-only* or *read-write*
- * the value of the variable

The net result of this simplicity is a network manager that is easy to implement and not too stressful on an existing network.

Another advantage of SNMP is that it is in very wide use today. This popularity came about when no other network managers appeared to replace the "band-aid" implementation of SNMP. The result of this is that almost all major vendors of internetworking hardware, such as bridges and routers, design their products to support SNMP, making it very easy to implement.

Expandability is another benefit of SNMP. Because of its simple design, it is easy for the protocol to be updated so that it can expand to the needs of users in the future. The ramification of this will be seen later on.

This report describes the implementation of a Network Manager (SNMP Manager) on the Dec Alpha OSF/1 system. The salient features of the tools are:

- * MIB tree loading for network elements with an SNMP agent simultaneously using asynchronous communication;
- * Encoder and Decoder that use SNMP specific BER

In chapter 2, the various issues in a central network management station are discussed. In chapter 3, the Internet management framework (the major issues in the network management) and its associated management protocol 'SNMP', are dealt with. The structure and identification of management information used for TCP/IP based Internets are covered in chapter 4. Chapter 5 describes design and implementation of tools developed for SNMP Manager. Chapter 6 talks about conclusions and future extensions.

2.1 Motivation

The success of the protocol suite has given birth to Internets -- large communications infrastructures involving computers, media specific devices and a host of other network devices. Because of the open nature of the protocol suite, these internets are heterogeneous in nature. The need to keep such networks running, and the need for traffic and utilisation data which is useful in designing, and planning new networks and extensions, makes network management essential. However the vastness, heterogeneity involved, complexity and the interrelatedness of the components in the internets make the management of these networks quite difficult. Typical problems faced by technical staff are:

- * equipment additions and changes that often lead to configuration errors.
- * increased scale makes format adhoc tools impartial
- * increased heterogeneity makes proprietary tools unusable
- * wider range of staff expertise requires more sophisticated tools which are easier to use
- * different administrations; makes effective management difficult because of the complicated interactions that it requires.

What is network management?

Network management can be broadly defined as that functionality which allows one to debug problems in a network, control routing, find computers that violate protocol standards. The following section describes the OSI network management can be taken as ideal model for network management.

2.2 The OSI Network Management Model

The functional approach to OSI network management is to view the problem as five sub-problems:

- * ***Fault Management*** - detecting, diagnosing, and recovering from network faults.
- * ***Configuration Management*** - defining, changing, monitoring, and controlling network resources and data.
- * ***Accounting Management*** - recording usage of network resources
- * and generating billing information
- * ***Performance Management*** - controlling and analysing the throughput and error rate of the network(including historical information)
- * ***Security Management*** - ensuring only secured and authorised access to network management system and network resources.

To support these different views, a *Common Management Information Service*(CMIS) and associated protocol *Common Management Information Protocol*(CMIP) are introduced. Orthogonal to the functional decomposition, the management service provides three types of usage which can be used to accomplish the five management functions listed above. They are

- * ***monitoring***, in which management information is retrieved;
- * ***control***. in which devices are manipulated; and
- * ***reporting***, in which devices report abnormal events.

In OSI parlance, *System Management* consists of managing the OSI porting of a system. Each layer consists of a *Layer Management Entity*(LME), which knows about the protocol operating at that layer. The LMEs communicate with a *System Management Application Entity*(SMAE) using a local mechanism, which in turn

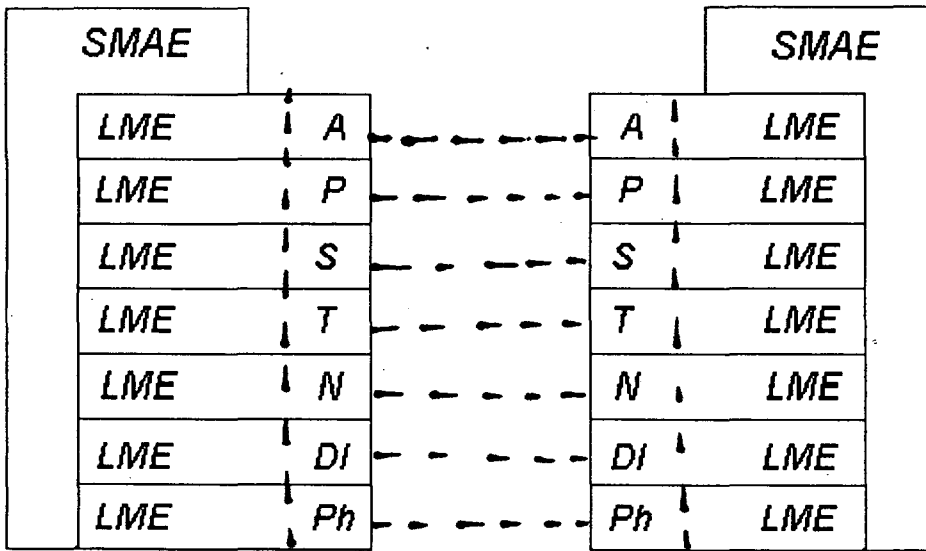


Fig. 2.1 OSI Network Management

uses CMIP to communicate with other SMAEs. By this process of abstraction, OSI layer management is accomplished.

As depicted in the figure 2.1 the SMAE has access to the LME at each layer. Further, the SMAE is a user of the OSI application layer in order to access CMIP. This architecture is general enough in that the same protocols and mechanisms used for normal data transfer at the application layer can be used for network management.

The OSI Common Management Information(CMI) is based on a connection oriented model before management activity occurs an application association is formed between the two management entities. Following this there are several services:

- * *get*, which is used to retrieve specific management information;
- * *set*, which is used to manipulate management information.
- * *action*, which is used to perform some imperative command;
- * *create*, which is used to create a new instance of a management object;
- * *delete*, which is used to delete an instance of a management object;
- * *event-report*, which is used to report extraordinary events;

All of these services may be performed in a confirmed fashion, while set, action and event-report can be used in non-confirmed form also. In order to specify the management objects of interest, the service employs two concepts : *Scoping* and *Filtering*. Management information is organised in a hierarchical structure. *Scoping* occurs by identifying a particular node in the tree with a depth. This marks a subtree rooted at that node and of the specified depth. Next an arbitrary boolean expression is applied to the attributes of the management information held within that subtree. This is called *Filtering*. Once the management information is Scoped and Filtered the desired management activity occurs. Beyond this basic model, the key issues of how management information is structured and defined, is not clearly mentioned. In the following sections a qualitative analysis of network management models as seen from different perspectives is given.

2.3 Level Of Management Protocols

Earlier most of the hardware vendors used to provide their own proprietary network management solutions that work for their products. Needless to say, such solutions will not work for another hardware platform. Also these protocols assume homogeneous environment i.e., same underlying network technology and generally operate at common like level. So such solutions work only within a single network. However an Internet management protocol is different from usual network management in that it has to work in heterogeneous networks. This makes mandatory that any Internet management protocol work and application level. Such a protocol offers several advantages:

- * they are independent of underlying hardware.
- * they are uniform
- * they can control gateways across an entire TCP/IP Internet without having direct attachment to every physical network or gateway.

Having the management protocol work at application level has one serious drawback: as the protocol depends on the underlying transport services, if the transport service goes wrong the management ceases to function.

2.4 Centralised Management Versus Decentralised Management

Network management can be classified into two types:

Centralised and **Decentralised** management. In a centralised monitoring station data collection and analysis are centralised by using a single network monitoring station. All the elements in the network will be monitored only by this management station. In a decentralised management model the whole domain of network elements will be divided into convenient sub-domains with one NMS for each subdomain. Monitoring could then be done in a hierarchical fashion, with NMS of each sub-domain reporting to a central NMS.

The advantage of a decentralised management model is that less traffic is generated because the MEs in the same sub-domain would report to their own NMS only a hop or two away, rather than a central NMS several hops away. Other advantages over a centralised system include greater reliability, and a smaller probability of the NMS being a bottleneck in the data collection process.

The problem with decentralised is the division into sub-domains which is often very difficult, mainly because of administrative reasons. Also it requires an NMS for each sub domain which may prove to be costly for a NMS may generally require a dedicated work stations. Though such a problem can be overcome say by installing the NMS over the backbone routers/gateways, such an effort is not advisable for the simple reason that management function should not burden the primary functionality of a functions.

Failure of the NMS in a centralised mode will seriously affect the management functionality, but this can be taken care of by duplicating the NMS software on two or three other hosts. With such an arrangement at the most a monitoring data over a short time will be lost, which does not pose any serious degradation to the monitoring function.

2.5 Polling Versus Alerts/Traps

There are two methods of transferring network management information from the MEs to the NMS: request-response driven polling and unsolicited sending of alert by the MEs. In a *Polled* system, the NMS periodically sends each ME a request for network management information pertaining to that entity. On receipt of the request the ME responds to the desired data. On the other hand in an alert based system *Alerts* or *Traps* are delivered by the MEs on their own to the NMS, because of the occurrence of some event at the ME. The following factors are to be considered when choosing these methods:

2.5.1 Reliable detection of failures

The NMS should be informed of any failures that occur. The NMS that polls the ME for management information can more readily determine failures in the ME or the network than an alert based system. A lack of a response to a query indicates failures.

In the case of an alert-based system the NMS assumes that the ME is good when it does not receive an alert, even when a failure has occurred and the alert message cannot be generated by the failed ME (e.g. a power failure) or the alert message is generated but cannot reach the NMS (e.g. network partition).

Alerts will not be delivered when the ME fails (e.g. a power failure) or the network experiences some problems (e.g. it is congested or partitioned). However, these are precisely the type of information that the NMS must receive. The lack of an alert will cause the NMS to assume that the network or the ME is in a healthy state.

2.5.2 Complexity and performance of the ME

A management protocol that imposes less complexity on an ME will free more of its resources for its primary function. A polling system allows the NMS to initiate a poll when it is in position to receive management traffic from an ME. There is no need to introduce any extra intelligence in to the ME to handle the the initiation of communication with the NMS or any retransmissions.

On the other hand an alert based system requires that the ME alerts the NMS whenever a monitored event occurs. This introduces an added complexity in the ME because the ME has to initiate alert messages to the NMS.

Performance impact on ME: In a polled system, there is no need for the ME to generate an alert every time a monitored event occurs. It merely takes note of the monitored event whenever it occurs, and responds to a poll with the entire set of events that have occurred during the last sample interval. The performance impact of responding to a poll periodically is minimal, as long as the polling interval is not too frequent (eg. hourly compared to one minute intervals).

In an alert-based system, the ME must generate an alert every time a monitored event occurs. This involves forming the alert packet and sending it to the NMS. This places an overhead on the ME depriving a possibly loaded processor of CPU cycles that could otherwise be used for its primary function.

2.5.3 Response Time for Problem Detection

The time interval between the occurrence of a problem and its detection should be minimized. In a poll based system the NMS may have to poll a large number of MEs, each for many parameters. It may be a while before the NMS can complete a cycle and poll a particular ME again. If an ME should fail immediately after it responds to a poll, there could be a significant delay before it is polled again. During this interval, the failed ME is assumed to be in good health.

A NMS could either use sequential or parallel polling. In sequential polling, the NMS waits for any outstanding requests to be responded before polling the next ME. In parallel polling the NMS can have a certain number of outstanding requests pending. Thus for the same number of MEs, the polling cycle for a particular ME can be smaller since the NMS spends less time in a busy waiting state.

In parallel polling increasing the number of MEs results in more management traffic as well as reduced performance. In large networks, the NMS is limited by the number of polls that it can process in unit time. The larger network also requires the time-outs to be larger to accommodate responses that are still in transit. Also a response could already have been received, but was queued while waiting to be processed. As a result, the time to detect a failure is increased.

However, alerts allow the NMS to be notified immediately of any problem in the ME once it occurs. This eliminates the delay imposed by polling a large numbers of MEs, for a large number of parameters.

2.5.4 Volume of Net Management Traffic

The amount of network management information that is transferred between the ME and the NMS, Polling many parameters on many machines results in a large amount of network management traffic flowing across the network and into the NMS to be stored. One way around this is to use hierarchical polling, where the NMS polls a set of intermediate stations for the general status of the machines that they poll. This localizes the management traffic, but results in increased time to failure detection.

In an alert based system as alerts are sent out only when errors are detected. By eliminating the need to transfer large amounts of healthy status information, network and system resources will be less heavily loaded.

2.5.5 Ease of Configuration

It should be relatively easy for the NMS to configure the MEs. In a poll based system the ME only responds to the request it receives, so long as it is an authorized party. Thus, the number of NMSs or the identity of the NMS which polls a particular ME can be changed without any need for reconfiguration of numerous MEs.

In an alert based system the remote ME need to be configured to report management information to specific destinations. So the ease of configuring the NMS to manage an ME is quite difficult.

2.5.6 Potential To Over-Inform The NMS

The use of polling allows the NMS to directly control the amount of management traffic across the network. There are no retransmissions unless the NMS asks for them. Thus, there is less chance that the NMS will be flooded with management packets at a time when it cannot handle them.

In an *alert-based* system the NMS could be flooded by an excessive number of alerts. For example in situations when an MN experiences a long burst or errors, causing it to send an equally long burst of alerts to the NMS. This flood of information places an extra burden on the network, as well as the NMS. It could even prevent the NMS

from disabling the ME that is generating the alerts, because all available network bandwidth into the NMS or the NMS itself, is saturated with incoming alerts.

In a polled system, the NMS need only be informed in a single poll response that the uptime for the vacillating link is less than the poll interval or that a burst of errors was received.

2.6 The Management Paradigms

Depending on the paradigm used for network management there are several forms a network management protocol can take. Basically there are two important network management paradigms:

The *remote debugging* paradigm and the remote execution paradigm. In the *remote execution* paradigm the management protocol is used to exchange *program fragments* which are executed on the managed node. For example protocols that use a remote execution paradigm incorporate commands like *reboot* etc., into the Protocol. In the remote debugging paradigm each managed node is viewed as having several variables. By reading the values of these *variables* the MEs are monitored. By changing the value of these variables the managed node is controlled. The advantage of this paradigm are:

- * **Stability:** The approach is stable for if in future if new management information or new commands are to be incorporated in can be simply done adding a new list of variables without changing the basic structure.
- * **Simplicity:** The approach is fairly simple and thus doesn't burden the MNs.
- * **Flexibility :** The approach is flexible since simply by changing the semantic interpretation of the values that a variable can take management functionality can be changed.

2.7 Asynchronous Communication

Any communication between the Manager and the MN will be asynchronous in nature, from the Manger's point of view. A Manager needn't wait for a response after sending a message to a managed node; because of the non deterministic nature of

response time over a network, it is not wise to wait for a response, without doing anything else. So ideally it can send other messages or do other activities after sending a message. Also because the manager has to communicate with multiple MNs it can be afford to block itself waiting for a response to come back from a specific MN.

2.8 The SNMP Network Management Model

Because SNMP emphasizes simplicity, it uses a connectionless protocol to limit the amount of network overhead. From a philosophical point of view, it is preferable to have the management protocol associated with SNMP decide issue, such as the number of retries and timeouts, rather than have a connection-oriented transport protocol decide these issues. Similarly, SNMP developers made a conscious decision to place management functions in the network management station that manages network functions rather than in the devices that are managed. This decision contrasts with OSI network management protocols discussed in section 2.1 where the network management station and devices share a more *peer-to-peer* relationship.

2.8.1 Components of SNMP management system

The components of an SNMP management system include the *Network Management System*(NMS) , *network agents*, *protocol data units*, the *Management Information Base* (MIB) and the Structure of Management Information (SMI).

The *Network Management Station* (NMS) monitors and controls the SNMP agents found within devices. An NMS can use a SNMPSTAT command to interrogate the variables associated with a particular device as found in the SNMP database known as the *Management Information Base*(MIB).

SNMP Manager software software generally consists of an application that generates specific SNMP commands to a network device (the agent) and receiving responses from the agent including information on critical network events.

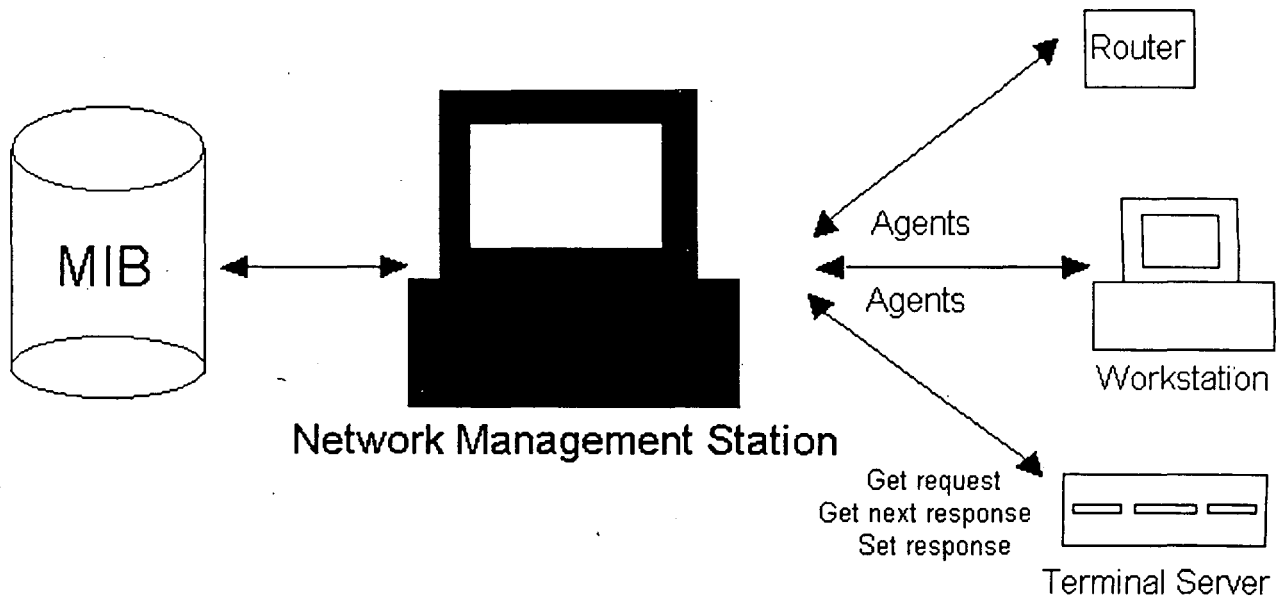


Fig. 2.2 SNMP Network Management

Network agents are devices such as routers or bridges that are to be managed on an SNMP network. These agents monitor network information such as the number of connections, packets transmitted, or error conditions at their specific locations. They provide this information to network management stations when they are polled and information is requested using the *User Datagram Protocol*(UDP).

Agents and Managers communicate with protocol data units (PDUs). Generally, the managers send commands (GetRequest, SetRequest, etc.) and receive responses from the agents (a GetResponse to a GetRequest, a Trap) to tell a manager that a major event has occurred.

Fig. 2.2 illustrates how SNMP network management works. About MIB and SMI we will discuss in later chapters. SNMP components are extensively discussed in[7].

2.9 Summary

Depending on the mechanisms used to achieve the goal of network management a model can be classified in many ways. A brief and qualitative analysis of such classes is given in this chapter. One important point to be stressed here is that when all else fails the manager should keep on functioning smoothly; otherwise the very purpose of the management is in jeopardy. The elements SNMP architecture and Internet management framework and the corresponding management protocol are discussed in the next chapter.

SNMP - The Management Protocol For TCP/IP Networks

3.1 Motivation

The Internet management framework focuses on the critical need for operational network management capabilities in the Internet. These immediate needs mandate the development of a system based on proven and well-understood practices rather than catering to every theoretical requirement. The framework is based on the philosophy of minimal requirements, which can be designated as the fundamental axioms of the framework, elaborated below:

The impact of adding network management to managed nodes must be minimal representing a lowest common denominator.

In this chapter, the major issues in the *network management* with reference to SNMP are dealt. In sections 2 and 3 the SNMP *architecture* and *elements of its architecture* are discussed, where as in sections 4 to 7 deal with the mechanism adopted by SNMP. Section 8 gives the summary. At the end of this chapter the reader will appreciate the underlying well thought out engineering decisions.

3.2 SNMP Architecture

A network management system consists of the three basic components:

- * several managed nodes, each containing an agent;
- * at least one Network Management Station(NMS) ; and
- * a network management protocol, which is used by the management station and the agents to exchange management information.

3.2.1 Managed Nodes

A managed node refers to a device of some kind falling into one of the following categories:

- * a *host system*, such as a workstation, terminal server, or printer;
- * a *gateway system*, or
- * a *media device*, such as a bridge, hub, or multiplexor;

All these devices have some sort of network capability. As can be seen, the potential diversity of managed nodes is quite high, spanning the spectrum from mainframes to modems. The first two categories implement the Internet suite of protocols, whilst the primary function of devices in the third category is media_dependent.

Another fundamental point to be taken note of is that the impact of adding network management to managed nodes must be minimal. Any managed node can be conceptualised as containing three components:

- * useful protocols, which perform the function desired by the user
- * a management protocol, which permits the monitoring and control of the managed node; and,
- * management instrumentation, which interacts with the implementation of the managed node in order to achieve the monitoring and control.

The instrumentation acts as GLUE between the useful protocols and the management protocol. This is usually achieved by an internal communications mechanism in which the data structures for the useful protocols may be accessed and manipulated at the request of the management protocol.

3.2.2 Network Management Stations

A network management station refers to a host system which is running

- * the *network management protocol* and
- * *network management applications*

The network management protocol provides the mechanism for management and the network management applications determine the policy that is used for management. An analogy helps to understand the concept in better way: the protocol is very much like an environment provided for debugging and just as how to debug and when to debug is the programmers botheration; how to manage and when to manage is the management application's botheration.

One should see to it that the managed nodes have the minimal impact of network management. As a consequence, the burden is shifted to the management station. Thus, it is expected that the host systems supporting a management station be relatively powerful in comparison to the managed nodes. Such an approach is logical and pragmatic; for

- * In an Internet there will be many managed nodes than managed stations and it is better to require significant functionality from a small percentage of devices rather than the vast majority.
- * As the requirement on MN side is very small, adding network management to the vast majority of NMs need not necessarily be adding another memory board or a faster processor.

The management station run by the operations staff of a large network is termed a Network Operations Centre(NOC). There is no fixed relationship between managed nodes and NOCs. There might be many network elements assigned to a single NOC station, but there might also be one or many NOC stations for each network element.

3.2.3 Network Management Protocol

Any network management protocol should be powerful enough to allow the following management operations

- * a *monitoring operation*, which allows a management station to examine what is happening at a managed-node;
- * a *control operation*, which allows a management station to control the managed node;
- * a *traversal operation*, which allows a management station to determine which variables a managed node supports; and
- * a *trap operation*, which allows a managed node to report an extraordinary event to a management station;

How a management protocol accomplishes all these depends on the network management paradigm that particular protocol uses. A thorough treatment of how SNMP achieves this is given in section 3.7. For the sake of completion and flow, it is briefly discussed here. The network management protocol in the Internet standard network management framework uses a remote debugging paradigm. Each managed node is viewed as having several *variables*.

Monitoring and Control

By reading the value of variables the managed node is monitored and changing the values of the variables means controlling the node. Such an approach is used because it is straight forward to build a simple, general and if required, extensible protocol.

Traversal

As already noted the diversity of managed nodes is quite high and as a consequence different managed nodes contain different management variables. Given this background there must be an efficient means for a management station to determine which variables are supported. Hence is the need for traversal operation. There is even more basic need for traversal operation. The NMS should be provided with an efficient means to traverse tables. More specifically the traversal mechanism should be able to

- * Retrieve a specific column or a row in a table
- * Browse through a table or the whole set of management variables

Traps

The advantages and disadvantages of polling and traps are extensively dealt in section 2.5. In the Internet standard network management framework the model used is trap directed polling. When an extraordinary event occurs the managed node sends a single and simple trap to the NOC stating the gist of the Event. The NOC is then responsible for initiating further interactions with the managed node in order to determine the nature and extent of the problem. This has proven to be effective and flexible; the impact on the managed nodes remain small; the impact on the network bandwidth is minimised; and problems can be dealt with in a timely fashion if the NMS happen to be *intelligent* it may weigh the information content of a trap using *heuristic rules* and past experience and may either decide to proceed with further questioning or simply ignore it.

3.3 Elements of the SNMP Architecture

The SNMP architecture articulates a solution to the network management problem in terms of the following aspects:

- 1) the scope of the management information communicated by the protocol.
- 2) the representation of the management information communicated by the protocol,
- 3) operations on the management information supported by the protocol,
- 4) the form and meaning of exchanges among management entities and
- 5) the definition of the administrative relationships among management entities.

3.3.1 Scope of Management Information

The scope of the management information communicated by operation of the SNMP is exactly that represented by instances of all non-aggregate object types defined in Internet standard MIB.

3.3.2 Representation of Management Information

Management information communicated by operation of the SNMP is represented according to the subset of a ASN.1 language that is specified for the definition of non_aggregate types in the RFC on Structure of Management Information (SMI). The SNMP uses an extended subset of ASN.1 for describing managed objects and for describing the protocol data units used for managing those objects. It should be noted that different implementations of the protocol will use different *internal representations* and the actual layout of each data structure depends on the programming language, language compiler and the machine architecture of each platform. Finally when management information is sent over a network SNMP used only a subset of the basic encoding rules of ASN.1 namely all encoding use the definite-length form. A more elaborate discussion can be found in [1].

3.3.3 Operations Supported on Management Information

Only four operations are available in the protocol:

- * *get*, to retrieve specific management information
- * *get-next*, the basic and only traversal tool, to retrieve management information via traversal;
- * *set*, to manipulate management information;
- * *trap*, to report extraordinary events.

SNMP models all management agent functions as alterations or inspections of variables. Thus a protocol on a logically remote host interacts with the management agent resident on the network elements in order to retrieve (get) or alter(set) variables. This type of support has two distinct advantages:

- 1) It has the effect of limiting the number of essential management functions realised by the management agent to two: one operation to assign a value to a specified configuration or other parameter and another to retrieve such a value

- 2) A second advantage of this decision is to avoid introducing into the protocol definition support for imperative management commands; the number of such commands is in practice ever-increasing and the semantics of such commands are in general arbitrarily complex.

The strategy implicit in the SNMP is that the monitoring of the network state at any significant level of detail is accomplished primarily by polling for appropriate information on the part of the monitoring centre(s). A limited number of unsolicited messages(traps) guide the timing and focus of the polling. The exclusion of imperative commands from the set of explicitly supported management functions is unlikely to preclude any desirable management agent operation. Currently most commands are requests either to set the values of some parameter or to retrieve such a value and the function of the few imperative commands currently supported is easily accommodated by this management model. In this scheme, the imperative command might be realised as the setting of a parameter value that subsequently triggers the desired action. For example, rather than implementing a *reboot command* this action might be invoked by simply setting a parameter indicating the number of seconds until system reboot.

3.3.4 Form and Meaning of Protocol Exchanges

The communication of management information among management entities is realised in the SNMP through the exchange of protocol messages. The complete description of these messages is given in Rose[1]. Consistent with the goal of minimising complexity of the management agent, the exchange of SNMP messages requires only an unreliable datagram service, and every message is entirely and independently represented by a single transport datagram. While this report specifies the exchange of messages via the UDP protocol, the mechanisms of the SNMP are generally suitable for use with a wide variety of transport services.

TH-6668

3.3.5 Definition of Administrative Relationships

The SNMP architecture admits a variety of administrative relationships among entities that participate in the protocol. The pairing of an SNMP agent with some arbitrary set of SNMP Managers is called an SNMP community. Each SNMP community is named by a string of octets, that is called the community name for said community.

Authentication

SNMP offers only trivial authentication that is the community name is placed clearly in an SNMP message. IF the community name corresponds to a community name known to the receiving SNMP entity the sending SNMP entity is considered to be authenticated as member of that community.

Authorisation

Authorisation determines the level of access to an authenticated member of the community. Authorisation is done using what is called as community profile. For any network element, a subset of objects in the MIB that pertain to that element is called a SNMP MIB view. The names of the object types represented in a SNMP MIB view need not belong to single sub-tree of the object type name space. An element of the set { READ-ONLY, READ-WRITE } is called an SNMP access mode.

A pairing of a SNMP access mode with a SNMP MIB view is called an SNMP community profile. A SNMP community profile represents specified access privileges to variables in a specified MIB view. For every variable in the MIB view in a given SNMP community profile access to that variable is represented by the profile according to the following conventions:

- 1) if said variable is defined in the MIB with *Access:* of *none* it is unavailable as an operand for any operator.

- 2) if said variable is defined in the MIB with *Access:* of *read-write* or *write-only* and the access mode of the given profile is READ-WRITE, that variable is available as an operand for the *get*, *set* and *trap* operations.
- 3) otherwise, the variable is available as an operand for the *get* and *trap* operations.
- 4) in those cases where a *write-only* variable is a operand used for the *get* or *trap* operations, the value given for the variable is implementation specific.

3.4 Proxy Management

So far the discussion assumes that all managed nodes support the Internet management protocol. However, in the cases of media devices which appear on the network such as repeaters and bridges this need not be the case. In addition if host or gateway systems, which implement other protocol suites, but not the Internet suite of protocols are on the network, then they too cannot be managed. Such devices are termed FOREIGN devices. The management framework provides a scheme to manage such *foreign* devices. A special agent termed a PROXY AGENT acts on behalf of the foreign device. When the foreign device is to be managed the management station contacts the proxy agent, and indicates the identity of the foreign device. The PROXY AGENT then translates the protocol interactions it receives from the management station into whatever interactions are supported by the foreign device. So if the foreign device supports a different management protocol, the proxy agent acts as an *application gateway*.

There is room for efficiency in the use of PROXY AGENT : caching of management information. If a managed node is being asked the same management questions frequently and if the answers don't change as frequently a proxy agent might be placed between the managed node and several NMSs so as to minimise the processing burden on the managed node.

3.5 Use of Transport Service

The transport requirements of SNMP are modest. This is consistent with the fundamental axiom of the management framework. Network management usually

occur in a trouble shooting or *fire-fighting* mode. The management application entity is in the best position to decide what the reliability constraints are on for management traffic. The lowest common denominator is a connectionless-mode transport service, so this is what is preferred for use in SNMP. This choice allows the management station to determine the appropriate level of retransmission in order to accommodate lossy or congested networks. However, the mechanisms of the SNMP are generally suitable for use with a wide variety of transport services.

3.6 Instance Identification and Lexicographical Ordering

Instance identification is important for traversal operation, especially while traversing a table. SNMP identifies an instance of an object by concatenating a *suffix* to the Object Identifier. The form of suffix is calculated according to

1. Only instances of leaf objects may be identified. Thus table and row objects are not manipulated, as aggregates, in SNMP.
2. If the object is not a column in a table, the *suffix* is simply 0(zero)
3. Otherwise, the object is a column in a table. The textual description of that table in the correspondent MIB defines how the suffix is formed, by selecting those columns necessary to make the suffix unique for that column.

Using object identifiers to name instances has a powerful advantage a lexicographical ordering is enforced on all object instances. This means that for instance names a and b one of three conditions consistently holds : either $a < b$, $a = b$, $a > b$. Having such a lexicographical ordering of instances allows one to browse through the entire object tree just by following a link. Infact the `get_next` operator uses this lexicographical ordering and is the most powerful operation provided by SNMP. Using the `get_next` operator and some simple logic one can manipulate the tabular objects and can retrieve information as though an *aggregate* type is provided. Thus the clever naming of the instances obviated the use of aggregate types, implementation of which would have been a burden on the managed nodes. Not supporting the aggregate type is also reasonable, for rarely do managers want to retrieve a table as a whole.

3.7 Protocol

SNMP is an asynchronous request/response protocol. An SNMP entity needn't wait for a response after sending a message. It can send other messages or do other activities. Further the request /response might be lost by the underlying transport service, it is up to the sending SNMP entity to implement the desired level of reliability. There are four primitive protocol interactions.

- 1) The manager retrieves management information from agent.
- 2) The manager traverses a portion of the agent's view.
- 3) The manager stores management information with the agent
- 4) The agent reports an extra ordinary event.

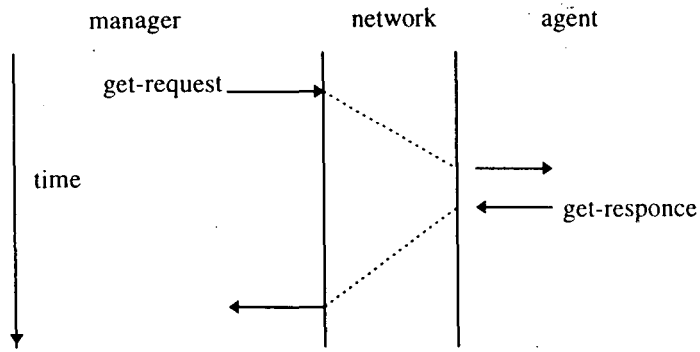
All these operations are shown in fig 3.1. These operations are subject to the community profile(authentication and authorisation) used by the sending SNMP entity.

Messages

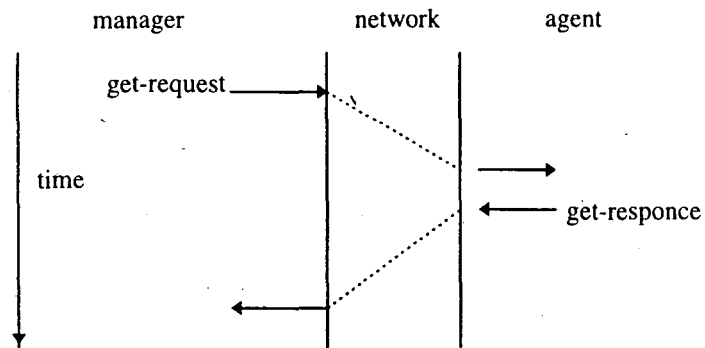
Communication among protocol entities is accomplished by the exchange of messages, each of which is entirely and independently represented within a single UDP datagram using the basic encoding rules of ASN.1[Rose]. A message consists of a version

identifier, an SNMP community name and a protocol data unit(PDU). A protocol entity receives messages at UDP *port 161* on the host with which it is associated for all messages except for those which report traps (i.e., all messages except those which contain the Trap (PDU). Messages which report traps would be received on UDP *port 162* for further processing. An implementation of this protocol need not accept messages whose length exceeds 484 octets. However, it is recommended that implementations support larger datagrams whenever feasible. [1] gives the asn.1 definition of the SNMP. We will now briefly discuss about the use of authentication and the data field viz., PDU.

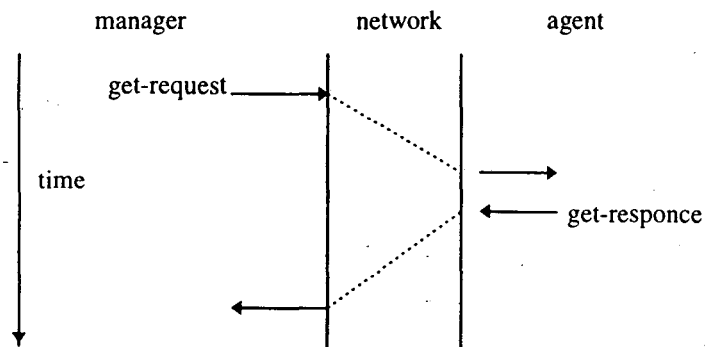
The *Manager* retrieves management information from the *Agent* :



The *Manager* traverses a portion of the *Agent's* view :



The *Manager* stores management information with the *Agent* :



The *Agent* reports an extraordinary event:

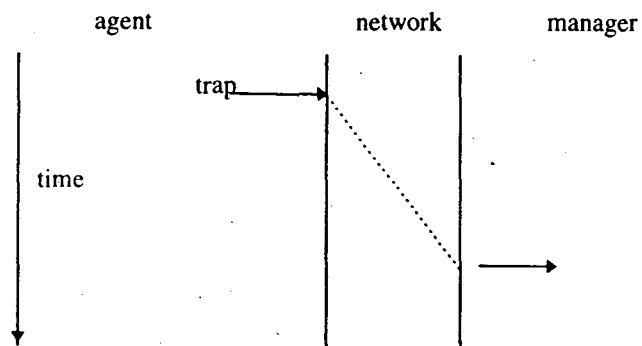


Fig. 3.1 Communication between Manager & Agent

***PDU*s**

The PDUs data type is actually one of two other ASN.1 types; a PDU which is used for the majority of operations and the trap-PDU which is used for traps. The fields of PDU data type are now described.

requested-id: An integer-value used by a manager to distinguish among outstanding requests. This allows a management application, if it so desires, to rapidly send several SNMP messages. The incoming replies can then be correlated to the correspondent operations. further, this provides a simple but effective, means for identifying messages duplicated by the network(or operations duplicated by retransmissions).

error-status: If non-zero, this indicates an exception occurred when processing the request. The values are :

- * **tooBig**, the agent could not fit the results of an operation into a single SNMP message;
- * **noSuchName**, the requested operation identified an unknown variable name(according to the community profile);
- * **badValue**, the requested operation specified an incorrect syntax or value when trying to modify a variable;
- * **readOnly**, the requested operation tried to modify a variable that according to the community profile, may not be written; and,
- * **genErr**, otherwise.

error-index: if non-zero, this indicates which variable in the request was in error. this field is non-zero only for the error no such name, badvalue, and readonly. In this case it is the positive offset into the variable-bindings field (the first variable is said to be at offset 1).

variable-bindings: A list of variables each containing a name and value. The value portion of a variable is not meaningful for GetRequest-PDU and GetNextRequest-PDU data types; by convention the value is always an instance of the ASN.1 DATA TYPE

null. However, the receiving SNMP entity should simply ignore whatever value is supplied by the sending SNMP entity.

Get Request PDU

Upon receiving a GET request PDU for each variable in the request the named instance is retrieved in the context of the community profile. If the instance does not exist, a get-response is returned with error noSuchName, otherwise, a get-response is returned identical to the request but with the value portions of the variables filled in accordingly.

Get Next Request PDU

Upon receiving a GET-NEXT request PDU for each variables in the request, the instance lexicographically following the named instance, in the context of the community profile is retrieved. If the end of the lexicographically space is reached, a get-response is returned with error nosuchname. Otherwise, a get response is returned identically to the request but with the name and value portions of the variables filled in accordingly.

Both the get and the get-next operators work sequentially. For example, if an error occurs while processing the first variable in a request, the remainder of the operands are not processed.

An Illustration of GET and GET_NEXT

Suppose a manager is interested in the value of two variables

sysDescr.o

sysName.o .

The former is defined in the Internet standard MIB and the latter is defined in MIB-II.

Therefore it is possible that the call

get(sysDescr.o,sysName.o).

might fail because the agent does not support the object type associated with the second operand. The solution is to issue

```
get(sysDescr,sysName)
```

instead. If the agent supports MIB-II, then the names and values of

```
sysDescr.o
```

```
sysName.o
```

will be returned. Otherwise, the names and values of

```
sysDescr.o
```

```
ifNumber.o
```

will be returned. Since the name of the sysName and ifNumber are different, the manager can easily determine that the sysName object is not available.

Set Request PDU

For each variable in the SET requested PDU received, the named instance is identified, in the context of the community profile. If the instance does not exist, a get-response is returned with error noSuchName. If the instance does exist but does not permit writing, a get-response is returned with the readOnly. If the instance exists and permits writing, but the value supplied in the request is poorly-formed(wrong syntax) or poorly-valued (range error), a get-response is returned with error badValue. Otherwise, all of the variable are updated simultaneously, and a get-response is returned identical to the request.

Get Response

The manager checks its list of previously sent requests to locate the one which matches this response. If no record is found the response is discarded. Otherwise, the manager handles the response in an appropriate fashion.

Traps

The fields of a trap-PDU are now described:

enterprise: the value of the agent's sysObjectId

agent-addr: the value of the agent's Network Address

generic-trap: one of a few extraordinary events

- * **coldStart**, the agent is (re-) initialising itself, and objects in its view may be altered (e.g., the protocol entities on the managed node are starting);
- * **warmstart**, the agent is reinitialising itself, but the objects in its view will not be altered;
- * **linkDown**, an attached interface has changed from the up to the down state (the first variable identifies the interface);
- * **linkUp**, an attached interface has changed to the up state (the first variable identifies the interface);
- * **authenticationFailure**, an SNMP message has been received from an SNMP entity which falsely claimed to be in a particular community;
- * **egpNeighborLoss**, an EGP peer has transitioned to state down (the first variable identifies the IP address of the EGP peer); and
- * **enterpriseSpecific**, some other extraordinary event has occurred, identified in the specific-trap field (using an enterprise-specific, e.g., private, value).

specific-trap: identifies the enterprise Specific trap which occurred, otherwise this value is zero.

time-stamp: the value of the agent's sysUpTime MIB object when the event occurred.

variable-bindings: a list of variables containing information about the trap.

Sending Trap-PDU

When an exception event occurs, the agent identifies those managers which it sends traps to, if any. For each manager, it selects an appropriate community and sends a Trap-PDU to that manager. Upon receiving a Trap-PDU, the manager handles the message in an appropriate fashion.

3.8 Summary

The SNMP architecture is modelled in to three groups of managed nodes, management stations and the supporting management protocol. In perspective, the fundamental axiom of the Internet management framework is based on the notion of universal deployment:

If network management is viewed as an essential aspect of an Internet, then it must be universally deployed on the largest possible connection of devices in the network.

By taking a minimalist approach, the management framework enjoys significant leverage in terms of economy of scale. As there are many more agents than management stations, minimising the impact of management on the agents is more attractive solution to the problem.

A second important tenet in network management is:

When all else fails, network management must continue to function, if at all possible.

This tenet mandates that many of the functions traditionally found in the transport layer be directly addressed by the applications in the management station since it is only applications which know the reliability requirements of each operation. So the transport service must not be helpful.

Structure and Identification of Management Information

In the previous chapter, the network management concepts related to the SNMP were discussed. The management information is represented by a separate syntax which is independent of the machine architecture and language. In this chapter, the representation scheme of management information is described.

4.1 Definition Of The Management Information

SNMP mandates that management information at a managed node be stored according to the conventions set forth in Internet-standard SMI. If one views the whole management information at a managed node as a database, the SMI defines the *Schema* for that database. The database is called Internet-standard Management Information Base (**MIB**). A brief description of SMI is given here

4.1.1 SMI

Management information is stored in terms of variables termed *Managed Object*. A managed object is described using an ASN.1 macro defined in the SMI. This macro basically consists of two definitions: TYPE NOTATION and VALUE NOTATION. The type notation defines the syntactical type (such as an ASN.1 INTEGER), the access to the object (whether read-only or read-write or write-only or not accessible), and the status of the object (whether mandatory or optional or obsolete /deprecated). The value notation specifies the actual name given to the object. All objects are given an authoritative name using an unique object identifier.

The syntax of an object is defined using the data type called OBJECT SYNTAX. This can be any of the following data types: simple ASN.1 types such as INTEGER, NULL etc., application-wide data types such as IpAdress, counter etc. and simply constructed types such as List and Table etc. [RFC 1155] gives elaborate details about SMI.

Objects as defined in SMI are just templates. More than just the name of an object is needed for management. It is the instances which are to be manipulated. So in addition to an object name an instance identifier is to be defined. Though the SMI doesn't specify any instance for non-tabular objects, a '.0' append to the object name represents the instance. For tabular objects rows are distinguished by using a set of columns which are sufficient to uniquely represent a row.

4.1.2 MIB

The Internet standard MIB describes those objects which are expected to be implemented by managed nodes running the TCP/IP. The criteria in fixing these objects are

- * the object must be essential for either fault or configuration analysis.
- * due to lack of secure authentication framework, any control objects must have weak properties.
- * the object must have evidenced *utility*.
- * the object must not be easily derivable from other objects.
- * the object must be sufficiently general in nature as to be found on many different platforms.

Using these criterion in MIB-II a revised version of an earlier MIB called MIB-I, a total of 10 groups are defined, each group corresponding to a protocol in the Internet suite. A brief summary of each group is given below. The MIB-II is elaborately described in [RFC 1156].

(I) System Group

The system group is mandatory for all MNs and contains information regarding the system (MN) such as system description, system uptime, system location etc. In all, the group contains seven variables.

(ii) Interfaces Group

This group contains generic information on the entities at the interface layer. At the top level the group has objects: one denoting the number of interface attachments on the node the second a table with detailed information such as no. of packets arriving at an interface, no. of errors etc., about each of the interface attachment. The group is mandatory for all MNs.

(iii) Address Translation Group

The group contains address resolution information and corresponds to Address Resolution Protocol(ARP). In fact the group contains a single table used for mapping ipaddress into media specific addresses. The group is marked depreciated in MiB-II.

(iv) IP Group

This group is mandatory for all MNs. The group corresponds to the IP layer and contains several scalars and four tables. Typical *scalars* are:

ipForWarding — acting as a gateway or host

ipInHdrErrors — datagrams discarded due to format errors etc.,

The *tables* to be maintained are:

ipAddrTable — to keep track of the ip addresses associated with the MN.

ipRountinTable — to keep track of the ip routes associated with the MN.

ipNetToMediaTable — to keep track of the mapping between IP and media - specific addresses.

(v) ICMP Group

This group is mandatory for all MNs. The group corresponds to the ICMP protocol used at IP layer and consists of 26 counters. The group can be summarised as:

⇒ for each ICMP message type, two counters exist, one counting the number of times this message type was generated by the local IP entity, the other counting the number of times the message type was received by the local IP entity.

⇒ there are four additional counters which keep track of the total number of ICMP messages received, sent, received in error or not sent due to error.

(vi) TCP Group

The TCP group is mandatory for all MNs. The group corresponds to the TCP layer and contains several scalars and table. The scalars maintain information relevant to TCP layer such as number of open connections, no of segments retransmitted etc., The table is used to keep track of applications entities which are using the TCP.

(vii) UDP Group

This group is mandatory for all MNs which implement UDP. The group corresponds to the UDP and has four counters and a table. As in TCP group the table keeps track of application entities using UDP.

(viii) EGP Group

This group is mandatory for all MNs which implement the Exterior Gateway Protocol(EGP), a reachability protocol used between autonomous systems.

(ix) Transmission Group

This group is introduced in MIB-II as a place-holder for media specific MIBs. These MIBs start out in the experimental space and may ultimately be placed in the Internet standard MIB.

(x) SNMP Group

This group is introduced in MIB-II and is intended to keep track of the various counters and any configuration parameters that are relevant to SNMP. This group facilitates monitoring the information about the management activity at each MN.

4.2 Structure of Management Information

Managed objects are accessed via a virtual information store, termed the Management Information Base or MIB. Objects in the MIB are defined using Abstract Syntax Notation one (ASN.1). Each type of object (termed an object type) has a name, a syntax, and an encoding. The name is represented uniquely as an OBJECT IDENTIFIER. An OBJECT IDENTIFIER is an administratively assigned name. The syntax for an object type defines the abstract data structure corresponding to that object type. For example, the structure of a given object type might be an INTEGER or OCTET STRING. The encoding of an object type is simply how instances of that object type are represented using the object's type syntax. Implicitly tied to the notion of an object's syntax and encoding is how the object is represented when being transmitted on the network.

4.2.1 Names to Identify Managed Objects

Names are used to identify managed objects which are hierarchical in nature. The OBJECT IDENTIFIER concept is used to model this notion. An OBJECT IDENTIFIER can be used for purposes other than naming managed object type: for example, each international standard has an OBJECT IDENTIFIER assigned to it for the purposes of identification. In short, OBJECT IDENTIFIER are a means for identifying some object, regardless of the semantics associated with the object.

An OBJECT IDENTIFIER is a sequence of integers which traverse a global tree. The tree consists of a root connected to a number of labelled nodes via edges. Each node may, in turn, have children of its own which are labelled. In this case, we may term the node a subtree. The root node itself is unlabeled, but has at least three children directly under it: one node is administered by the International Organisation for Standardisation (ISO), with label iso(1); another is administered by the International Telegraph and Telephone (ITTCC)

Consultative Committee, with label ccitt(0); and the third is jointly administered by the ISO and the CCITT, joint-iso-ccitt(2). The Internet subtree of OBJECT IDENTIFIER starts with the prefix: 1.3.6.1

The four nodes specified under the Internet subtree of OBJECT IDENTIFIERS are :

directory OBJECT IDENTIFIER ::= { Internet 1 }

mgmt OBJECT IDENTIFIER ::= { Internet 2 }

experimental OBJECT IDENTIFIER ::= { Internet 3 }

private OBJECT IDENTIFIER ::= { Internet 4 }

1) Directory

The directory(1) subtree is reserved for use in future that discusses how the OSI Directory may be used in the Internet.

2) Mgmt

The mgmt(2) subtree is used to identify objects which are defined in the standard management information base. So an OBJECT IDENTIFIER under the mgmt subtree would be accessed using

{mgmt 1} or 1.3.6.1.2

3) Experimental

The experimental(3) subtree is used to identify objects used in Internet experiments.

4) Private

The private(4) subtree is used to identify objects defined unilaterally. Initially, this subtree has at least one child:

enterprises OBJECT IDENTIFIER ::= { private 1 }

The enterprises(1) subtree is used, among other things, to permit parties providing networking subsystems to register models of their products.

4.2.2 Syntax To Define Object Types

Syntax is used to define the structure corresponding to object types. ASN.1 constructs are used to define this structure. The ASN.1 type ObjectSyntax defines the different syntaxes which may be used in defining an object type.

(i) Primitive Types

Only the ASN.1 primitive types INTEGER, OCTET STRING, OBJECT IDENTIFIER, and NULL are permitted. These are sometimes referred to as non-aggregate types.

(ii) Constructor Types

The ASN.1 constructor type SEQUENCE is permitted, providing that it is used to generate either lists or tables.

For lists, the syntax takes the form:

SEQUENCE { <type1> ,...,<typeN> }

Where each <type> resolves to one of the ASN.1 primitive types listed above. Further, these ASN.1 types are always present (the DEFAULT AND OPTIONAL clauses do not appear in the SEQUENCE definition)

For tables, the syntax takes the form:

SEQUENCE OF <entry>

Where <entry> resolves to a list constructor.

(iii) Defined Types

New application-wide types may be defined, so long as they resolve into an IMPLICITLY defined ASN.1 primitive type list, table, or some other application-wide type. A few of the application-wide types used are:

* *NetworkAddress*

This CHOICE represents an address from one of possibly several protocol families.

* ***IpAddress***

This application-wide type represents a 32 bit Internet address. It is represented as an OCTET STRING of length 4, in the network byte-order. When this ASN.1 type is encoded using the ASN.1 basic encoding rules, only the primitive encoding form shall be used.

* ***Counter***

This application-wide type represents a non-negative integer which monotonically increases until it reaches a maximum value, when it wraps around and starts increasing again from zero.

* ***Gauge***

This application-wide type represents a non-negative integer, which may increase or decrease, but which latches at a maximum value.

* ***TimeTicks***

This application-wide type represents a non-negative integer which counts the time in hundredths of a second since some epoch. When object types are defined in the MIB which use the ASN.1 type, the description of the object type identifies the reference epoch.

* ***Opaque***

This application-wide type supports the capability to pass arbitrary ASN.1 syntax. A value is encoded using the ASN.1 basic rules into a string of octets. This is then encoded as a OCTET STRING in effect *double wrapping* the original ASN.1 value.

4.2.3 Format Of Managed Objects

An object type definition consists of five fields:

OBJECT :

A textual name, termed the OBJECT DESCRIPTOR, for the object type, along with its corresponding OBJECT IDENTIFIER

SYNTAX:

The abstract syntax for the object type. This must resolve to an instance of the ASN.1 type ObjectSyntax.

DEFINITION:

A textual description of the semantics of the object type.

ACCESS:

One of read-only read-write, write-only or not accessible.

STATUS:

One of mandatory, optional or obsolete

In order to facilitate the use of tools for processing the definition of the MIB, the object type macro may be used. This macro permits the key aspects of an object type to be represented in a format way.

4.3 Summary

The structure of the management information is represented in a well defined manner. Grouping the management information in such a fashion facilitates the easy access of the objects through some identification mechanism. All the management information is represented by machine independent and language independent syntax called ASN.1

Design and Implementation

The previous chapters discuss the fundamental concepts of network management and the mechanism to identify the management information. In this chapter I delve into the intricacies involved in the implementation of an SNMP Manager.

5.1 Major criterion in the design of an SNMP Manager

As discussed in the earlier chapters, at least *one* **SNMP Agent** should reside on each of the managed nodes and at least *one* **SNMP Manager** should be on **Network Management Station**. The nodes of a network management station may manage by any of the following : a *workstation, gateway, router, bridge* or a *multiplexor*. Whenever any agent is residing on a particular network element, it is expected to support certain objects whose status or values may be queried by the **SNMP Manager** at any time. What variables a particular managed node supports is largely dependent on the type of network element. By default the **SNMP Manager** contains certain standard objects with definitions and syntaxes of these objects well defined in a store called the Management Information Base(MIB).

This store of objects is a very generalised database and may query sufficient information from specialised network elements like a router. Network elements like routers support a host of other variables whose information plays a crucial role. In order to the information regarding these objects be queried by the network management station, the list of the objects supported by that particular network element is added to the MIB at the NMS. However, adding list of objects to MIB will result in other managed nodes also being queried, even though they don't support such objects. This would result in unnecessary network congestion and the very purpose of network management is lost.

So each such list of objects supported by network element called Private MIB is given an enterprise specific number and whenever the management station polls that

particular node, it sends request for objects in standard MIB as well as the objects under the enterprise specific class corresponding to the number allocated to it. Thus it is seen that elements specialised in particular tasks can be queried about information related to those objects.

5.2 Design Considerations

The key aspects that influenced the design are:

- * Asynchronous communication with the agents- Any communication with an SNMP entity should be asynchronous as one can not predict the response time over a network.
- * Protocol specific constraints.
- * Collection and storage of data.
- * Processing of data and fault detection.

This Project implements *tools* required by the protocol(SNMP) such as *MIB tree* loading, *Encoder* and *Decoder* that use SNMP specific Basic Encoding Rules(*BER*).

5.3 Modules specific to SNMP

5.3.1 MIB tree-loading

The foremost step in communicating with SNMP agents while implementing an SNMP manager is to formulate the MIB tree in memory. An MIB object is basically represented by an object with data fields: the name of the object, the ASN.1 Object identifier, the syntax of the object, the child and sibling links and the important “next-link” for the net_next operation.

Forming the MIB tree involves four major steps:

- 1) reading the entire set of objects into memory(probably from a text file).

- 2) forming the child and sibling links that are required to maintain and traverse a tree structure.
- 3) connecting the entire set of MIB objects by the next link, using lexicographical ordering.
- 4) storing Enumerated information about a set of given MIB tree Objects.

Once the MIB tree is formed next logical step is providing some tools that allow one to search for different objects in all Possible ways. These details are given in the implementation part.

The syntax is represented as an Object having two data fields, one for storing information about *Enumerated Syntax* and the other for simple syntax. The various functions associated with this object are:

a *Parsing routine* - to read the value into the object from a string.

an *Encode routine* - to encode a value of the associated syntax into a SNMP value structure

a *Decode routine* - to complement the above routine.

a *Print routine* - to print the value in appropriate format, according to the syntax on the console.

5.3.2 Encoder and Decoder that use SNMP specific Basic Encoding Rules(BER)

These tools are written in a very generic way so that given a structure of any complexity, valid according to SNMP specific ASN.1 and BER, one can use the same decoding or encoding tool without the need for recompiling. An Information Structure is maintained for all the fields in every ASN.1 structure that is to be encoded or decoded. The Information Structure essentially reflects the ASN.1 tag, class, primitive/constructive information and other information related to the internal representation(C language Rep). At present, these information structures are generated manually, but the process can be automated. In such tools the user gives an ASN.1

structure, and the tool returns an equivalent internal representation, and a set of functions one for encoding and one for decoding. These functions actually use the Prototype encoding and decoding routines. Other details are discussed in implementation part.

5.4 Representation

In deciding the representation for agents, and requests which are forwarded to the agents, the major concern was to achieve

- * Management Protocol Independence
- * Extensibility

Each agent is represented as a set of features that are characteristic of a network element. Typical features are name, network address, network address type, MIB tree etc. To avoid protocol dependency, first some generic functions, that are required for Management activity to take place, are identified. Typical functions identified are:

- * *authentication function* - to authenticate properly the sending Management queries;
- * *formMessage* - to form a protocol specific message from an abstract request;
- * *encodeMessage* - to convert protocol specific message into the protocol specific serialised packet;
- * *sendPacket* - to send the serialised packet over the network;
- * *decodeMessage* - to decode the packet received over the network into a protocol specific message;
- * *optimizePacketSending* - to take care of appropriate buffering so as to make the process of polling efficient;

Apart from these, some other functions which do any kind of pre-processing or interfacing with user are attached to the agent. These approach makes the design Object Oriented. That is, a Management Protocol supported by a particular agent, will have it's

own set of functions. For example, if two agents support different management protocols, they will have different set of functions attached to them. Similarly if an agent supports two management protocols, it will have two sets of functions - one for each management protocol. Such a representation allows, with minor changes, easy adoption to the advent of any new management protocols.

Since SNMP is used here as the Management Protocol, all these functions are written assuming SNMP.

Secondly the services offered by the management protocol are made transparent to the user. The user is given some highly generalised services. Some of these services are not supported directly by the underlying Management Protocol. Towards this end an abstract Request List is defined. This finally comes down to a generic reuse, which is represented by typical features such as names of variables, frequency with which the variables have to be polled, number of retries and time-out, file where the information received is to be logged etc.

5.5 Implementation

5.5.1 Configuration

The request/response *Manager* is implemented on *DEC-ALPHA* work station with *OSF/1* OS version 2.0. The environment consists of 2 DEC workstations and 4 X-terminals connected over *Ethernet*. Only DEC-ALPHA1 station run an SNMP agent in the background. The project is implemented using *C* language, and the *Berkeley Socket* interface for transport services. Though C++ could have been a better language, especially because of the inherent Object Orientedness involved in the project, C is used for the lack of resources and time. However, all efforts have been put to see that the implementation is *object oriented* and data driven.

5.5.2 The mechanism

The complete *Manager* program is implemented as a single Unix Process. *Sending*, *receiving* and *processing* are all done in a single process sequentially, giving *top*

priority to sending. This is working well for the configuration described above. Care is taken not to cause any problem even, if there are too many agent. For this distributed computing is simulated using the Unix processes and Inter Process Communication(*IPC*).

5.5.3 Space versus Time

It is decided that any NMS requires a dedicated and some times more than one Work Station. So stress is given to speed rather than conserving memory. Whenever it is felt that buffering will increase speed, it has been employed without hesitation. For example using two processes one for sending and another for receiving resulted in duplicating of data structures, though this could have been avoided using a more powerful *IPC* which would be inefficient speed wise. Shared memory could not be used because of the inherent limitation of the OS on the size. Similarly since an agent is polled for the same queries periodically, all queries as they are finally dispatched over the network i.e., in byte stream form are buffered. This resulted in saving lot of CPU time.

In Th. following section a detailed description of important functions, starting from main is given.

5.5.4 Description of Important Routines

The whole *source code* is given in Appendices. So, for more details reader can refer Appendices. However, the important routines are explained here briefly.

init_snmp()

Gets initial request ID for all transactions.

snmp_open(session)

Sets up the session with the *snmp_session* information provided by the user. Then opens and binds the necessary UDP port. A handle to the created session is returned (this is different than the pointer passed to *snmp_open()*). On any error, NULL is returned and *snmp_errno* is set to the appropriate error code.

free_request_list(rp)

Free each element in the input request list.

snmp_close(session)

Close the input session. Frees all data allocated for the session, dequeues any pending requests, and closes any sockets allocated for the session. Returns 0 on error, 1 otherwise.

snmp_build(session, pdu, packet, out_length)

Takes a session and a pdu and serialises the ASN PDU into the area pointed to by packet. out_length is the size of the data area available. Returns the length of the completed packet in out_length. If any errors occur, -1 is returned (for error index). If all goes well, 0 is returned.

snmp_parse(session, pdu, data, length)

Parses the packet received on the input session, and places the data into the input pdu. length is the length of the input packet. If any errors are encountered, -1 is returned. Otherwise, a 0 is returned. It authenticates message and returns length if valid.

snmp_send(session, pdu)

Sends the input pdu on the session after calling snmp_build to create a serialised packet. If necessary, set some of the pdu data from the session defaults. Add a request corresponding to this pdu to the list of outstanding requests on this session, then send the pdu. Returns the request id of the generated packet if applicable, otherwise 1. On any error, 0 is returned. The pdu is freed by snmp_send() unless a failure occurred.

snmp_free_pdu(pdu)

Frees the pdu and any malloc'd data associated with it.

snmp_read(fdset)

Checks to see if any of the fd's set in the fdset belong to SNMP. Each socket with it's fd set has a packet read from it and snmp_parse is called on the packet received. The resulting pdu is passed to the callback routine for that session. If the callback routine

returns successfully, the pdu and it's request are deleted. If it finds error it prints "Mangled packet". But there shouldn't be any more request with the same reqid.

snmp_select_info(numfds, fdset, time-out, block)

Returns info about what SNMP requires from a select statement. numfds is the number of fds in the list that are significant. All file descriptors opened for SNMP are OR'd into the fdset. If activity occurs on any of these file descriptors, snmp_read should be called with that file descriptor set. The time-out is the latest time that SNMP can wait for a time-out. The select should be done with the minimum time between time-out and any other time-outs necessary. This should be checked upon each invocation of select. If a time-out is received, snmp_timeout should be called to check if the time-out was for SNMP. (snmp_timeout is idempotent) Block is 1 if the select is requested to block indefinitely, rather than time out. If block is input as 1, the time-out value will be treated as undefined, but it must be available for setting in snmp_select_info. On return, if block is true, the value of time-out will be undefined. snmp_select_info returns the number of open sockets. (i.e. The number of sessions open).

snmp_timeout()

snmp_timeout should be called whenever the time-out from snmp_select_info expires, but it is idempotent, so snmp_timeout can be polled (probably a cpu expensive proposition). snmp_timeout checks to see if any of the sessions have an outstanding request that has timed out. If it finds one (or more), and that pdu has more retries available, a new packet is formed from the pdu and is resent. If there are no more retries available, the callback for the session is used to alert the user of the time-out.

5.6 Summary

In summary, the design is object oriented in nature and easily extensible. Program flow is data driven and all functionalities are implemented in the most generic way possible. Speed is given prime importance than space; this is justifiable because of the enormous

magnitude of the management functionality. The next chapter gives conclusions and future extensions, which could not be implemented because of time constraints.

Conclusions and Future Extensions

6.1 Conclusions

The *SNMP Manager* implemented over the BSD Unix(OSF/1) environment with a minimal set of variables in the MIB and Manager program is run on a DEC-ALPHA2 work station which is connected to 4 X-terminals over Ethernet. DEC-ALPHA1 station run an *SNMP Agent* in the background. The Manager program was able to successfully communicate with the SNMP agent running on DEC-ALPHA1.

SNMP Manager sends get or getnext request to retrieve Management information from any connected SNMP agent. For example

get ipForWarding.0 and *get ipInHdrErrors.0* responses will be like this:

ipForWarding.0 = 1(host)

ipInHdrErrors.0 = 0

SNMP basic functionality is provided. In future, there is an enormous potential for extensions in the development of Manager.

6.2 Comments & Extensions

Magnitude of the Network Management Traffic:

Network Management skews the data being retrieved. So as possible the traffic generated by the management functionality should be minimum. Care has been taken to minimise the management traffic, by classifying the requests to be dispatched and grouping them within a single SNMP message. Right now the maximum limit on the no. of variables sent within a single packet is set at 5. If the user wants change it, he has to change the corresponding macro definition and recompile it.

Such a scheme causes some problems. Imagine that 5 SNMP variables are being requested using a single packet. Even if one of them is not supported by the agent on the other side he sends a error packet. So eventhough seven variables are there we couldn't get that information. Intelligence have to be incorporated to deal with such situations. Heuristics will help a lot here.

Controlling

The program allows the user to specific variables in the MIB tree, but it is upto the receiving SNMP agent to authenticate it and take the decision whether to honour it or not.

Private MIBs

The program works well with the private MIBs. The program takes any input regarding SNMP variables from files. By manipulating these files and rerunning the program, one can use this as a generic tool. However, if one feels that such a process is laborious, he can develop a user interface that interacts with the user and send SNMP messages very easily.

Artificial Intelligence tools

The field of AI has a wide range of arsenal. Using various AI techniques one can port this to real time situations, which would be very useful in proactive network management. Important fields within network management where AI methods can be used are Planning and designing to help in future expansions, fault diagnosis. Using Graphic User Interface(GUI), one can make up with intuitive visual representation.

Right now the program is developed as a group of programs which are tightly bound together, i.e., the functionalities are not clearly dissected. One very good extension is to develop a black board like architecture using a scheduler. Information comes from over the network and many experts such as diagnose expert, a fault isolation expert, a designer etc., are given access to the information by the scheduler. Each expert will work

independently and comes into the picture whenever his expertise is the need of the hour. However, time and infrastructure would be the major criterion for such extensions.

Distributed management

Finally as the number of the network elements to be monitored become large, Distributed management models have to be used. Basically there are many managers who supervise agents who come under his purview. If a manager wants to know something about an agent who is under a different Manager, he has to communicate with him. So some sort of protocol has to be designed for Manager-Manager communication. One can implement a prototype Network Monitor using SNMP as the underlying protocol. One can always extend this by adding network management applications such as Network Monitoring, Network Control, Network Operation Centre On-Line(NOCOL), etc. By reading values of variables the managed node is monitored and changing the values of the variables means controlling the node.

References

- [1] Marshall T. Rose, "The Simple Book: An Introduction to Management of TCP/IP based Internets", Prentice-Hall, EngleWood Cliffs, New Jersey.
- [2] Request For Comments(RFC) 1155, "Structure and Identification of Management Information for TCP/IP based internets".
- [3] Request For Comments(RFC) 1156, "Management Information Base for Network Management of TCP/IP based internets".
- [4] Request For Comments(RFC) 1157, "A Simple Network Management Protocol(SNMP)"
- [5] Request For Comments(RFC) 1089, "SNMP over Ethernet"
- [6] Request For Comments(RFC) 1161, "SNMP over OSI"
- [7] Stan Schatt, "Understanding Network Management: Strategies & Solutions", Applied Networking Series Windcrest/McGraw-Hill.
- [8] Douglas E. Comer, "Internetworking with TCP/IP: Principles, Protocols, and Architecture" vol.1, Prentice-Hall, EngleWood Cliffs, New Jersey.
- [9] Andrew S. Tanenbaum, "Computer Networks", Prentice-Hall, EngleWood Cliffs, New Jersey.
- [10] William Stallings, "Data and Computer Communications", Prentice-Hall, EngleWood Cliffs, New Jersey.
- [11] Darren L.Spohn, "Data Network Design", McGraw-Hill Series on Computer Communications.
- [12] Kornel Terplan, "Communication Network Management", Prentice-Hall Computer Communication Series.
- [13] Allan Leinwand & Karen Fang, "Network Management: A Practical Perspective", Addison-Wesley, Don Mills.
- [14] Ulysess Black, "Computer Networks: Protocol Standards and Interfaces", Prentice-Hall EngleWood Cliffs, New Jersey.
- [15] Ulysess Black, "Network Management Standards: The OSI, SNMP and CMOL Protocols", Ulysess Black Series On Computer Communications.
- [16] Richard Stevens, "Unix Networking Programming", Prentice-Hall, EngleWood Cliffs, New Jersey.
- [17] Maurice Bach, "The Design of Unix Operating System", Prentice-Hall, India.
- [18] Chris Brown, "Unix Distributed Programming", Prentice-Hall, EngleWood Cliffs, New Jersey.

```

/*****
 * jnu_snmp.c - send snmp requests to a network entity.
 *****/
#include <sys/types.h>
#include <netinet/in.h>
#include <stdio.h>
#include <ctype.h>
#include <errno.h>

#include "../jnusnmp/snmp.h"
#include "../jnusnmp/snmp_impl.h"
#include "../jnusnmp/asn1.h"
#include "../jnusnmp/snmp_api.h"
#include "../jnusnmp/snmp_client.h"

extern int  errno;
int  command = GET_REQ_MSG;
int  snmp_dump_packet = 0;

main(argc, argv)
    int  argc;
    char *argv[];
{
    struct snmp_session session, *ss;
    struct snmp_pdu *pdu, *response;
    struct variable_list *vars, *vp;
    int  arg, ret;
    char *gateway = NULL;
    char *community = NULL;
    int  status, count;

    init_mib();
    /* usage: snmptest gateway-name community-name */
    for(arg = 1; arg < argc; arg++){
        if (argv[arg][0] == '-'){
            switch(argv[arg][1]){
                case 'd':
                    snmp_dump_packet++;
                    break;
                default:
                    fprintf(stderr, "invalid option: -%c\n", argv[arg][1]);
                    break;
            }
            continue;
        }
        if (gateway == NULL){
            gateway = argv[arg];
        } else if (community == NULL){
            community = argv[arg];
        } else {
            fprintf(stderr, "usage: snmptest gateway-name community-name\n");
            exit(1);
        }
    }
}

```

```

    }
}
if (community == NULL)
    community = "public"; /* default to public */

if (!(gateway && community)){
    fprintf(stderr, "usage: snmpctest gateway-name community-name\n");
    exit(1);
}

bzero((char *)&session, sizeof(struct snmp_session));
session.peername = gateway;
session.community = (u_char *)community;
session.community_len = strlen((char *)community);
session.retries = SNMP_DEFAULT_RETRIES;
session.timeout = SNMP_DEFAULT_TIMEOUT;
session.authenticator = NULL;
snmp_synch_setup(&session);
ss = snmp_open(&session);
if (ss == NULL){
    fprintf(stderr, "Couldn't open snmp\n");
    exit(-1);
}

while(1){
    vars = NULL;
    for(ret = 1; ret != 0;){
        vp = (struct variable_list *)malloc(sizeof(struct variable_list));
        vp->next_variable = NULL;
        vp->name = NULL;
        vp->val.string = NULL;

        while((ret = input_variable(vp)) == -1)
            ;
        if (ret == 1){
            /* add it to the list */
            if (vars == NULL){
                /* if first variable */
                pdu = snmp_pdu_create(command);
                pdu->variables = vp;
            } else {
                vars->next_variable = vp;
            }
            vars = vp;
        } else {
            /* free the last (unused) variable */
            if (vp->name)
                free((char *)vp->name);
            if (vp->val.string)
                free((char *)vp->val.string);
            free((char *)vp);
        }
    }
    status = snmp_synch_response(ss, pdu, &response);
    if (status == STAT_SUCCESS){

```

```

switch(response->command){
    case GET_REQ_MSG:
        printf("Received GET REQUEST ");
        break;
    case GETNEXT_REQ_MSG:
        printf("Received GETNEXT REQUEST ");
        break;
    case GET_RSP_MSG:
        printf("Received GET RESPONSE ");
        break;
    case SET_REQ_MSG:
        printf("Received SET REQUEST ");
        break;
    case TRP_REQ_MSG:
        printf("Received TRAP REQUEST ");
        break;
}
printf("from %s\n", inet_ntoa(response->address.sin_addr));
printf("requestid 0x%x errstat 0x%x errindex 0x%x\n",
    response->reqid, response->errstat, response->errindex);
if (response->errstat == SNMP_ERR_NOERROR){
    for(vars = response->variables; vars; vars = vars->next_variable)
        print_variable(vars->name, vars->name_length, vars);
} else {
    fprintf(stderr, "Error in packet.\nReason: %s\n", snmp_errstring(response->errstat));
    if (response->errstat == SNMP_ERR_NOSUCHNAME){
        for(count = 1, vars = response->variables; vars && count != response->errindex;
            vars = vars->next_variable, count++)
            ;
        if (vars){
            printf("This name doesn't exist: ");
            print_objid(vars->name, vars->name_length);
        }
        printf("\n");
    }
}

} else if (status == STAT_TIMEOUT){
    fprintf(stderr, "No Response from %s\n", gateway);
} else { /* status == STAT_ERROR */
    fprintf(stderr, "An error occurred, Quitting\n");
}

if (response)
    snmp_free_pdu(response);
}

int
ascii_to_binary(cp, bufp)
    u_char *cp;
    u_char *bufp;
{
    int subidentifier;
    u_char *bp = bufp;

```



```

for(; *cp != '\0'; cp++){
    if (isspace(*cp))
        continue;
    if (!isdigit(*cp)){
        fprintf(stderr, "Input error\n");
        return -1;
    }
    subidentifier = atoi(cp);
    if (subidentifier > 255){
        fprintf(stderr, "subidentifier %d is too large (> 255)\n", subidentifier);
        return -1;
    }
    *bp++ = (u_char)subidentifier;
    while(isdigit(*cp))
        cp++;
    cp--;
}
return bp - bufp;
}
int
hex_to_binary(cp, bufp)
    u_char *cp;
    u_char *bufp;
{
    int subidentifier;
    u_char *bp = bufp;

    for(; *cp != '\0'; cp++){
        if (isspace(*cp))
            continue;
        if (!isxdigit(*cp)){
            fprintf(stderr, "Input error\n");
            return -1;
        }
        sscanf(cp, "%x", &subidentifier);
        if (subidentifier > 255){
            fprintf(stderr, "subidentifier %d is too large (> 255)\n", subidentifier);
            return -1;
        }
        *bp++ = (u_char)subidentifier;
        while(isxdigit(*cp))
            cp++;
        cp--;
    }
    return bp - bufp;
}
input_variable(vp)
    struct variable_list *vp;
{
    u_char buf[256], value[256], ch;

    printf("Please enter the variable name: ");
    fflush(stdout);
    gets(buf);
}

```

```

if (*buf == 0){
    vp->name_length = 0;
    return 0;
}
if (*buf == '$'){
    switch(buf[1]){
        case 'G':
            command = GET_REQ_MSG;
            printf("Request type is GET REQUEST\n");
            break;
        case 'N':
            command = GETNEXT_REQ_MSG;
            printf("Request type is GETNEXT REQUEST\n");
            break;
        case 'S':
            command = SET_REQ_MSG;
            printf("Request type is SET REQUEST\n");
            break;
        case 'D':
            if (snmp_dump_packet){
                snmp_dump_packet = 0;
                printf("Turned packet dump off\n");
            } else {
                snmp_dump_packet = 1;
                printf("Turned packet dump on\n");
            }
            break;
        case 'Q':
            printf("Quitting, Goodbye\n");
            exit(0);
            break;
        default:
            fprintf(stderr, "Bad command\n");
    }
    return -1;
}
vp->name_length = MAX_NAME_LEN;
if (!read_objid(buf, value, &vp->name_length))
    return -1;
vp->name = (oid *)malloc(vp->name_length * sizeof(oid));
bcopy((char *)value, (char *)vp->name, vp->name_length * sizeof(oid));

if (command == SET_REQ_MSG){
    printf("Please enter variable type [i|s|x|d|n|o|t|a]: ");
    fflush(stdout);
    gets(buf);
    ch = *buf;
    switch(ch){
        case 'i':
            vp->type = INTEGER;
            break;
        case 's':
            vp->type = STRING;
            break;
    }
}

```

```

case 'x':
    vp->type = STRING;
    break;
case 'd':
    vp->type = STRING;
    break;
case 'n':
    vp->type = NULLOBJ;
    break;
case 'o':
    vp->type = OBJID;
    break;
case 't':
    vp->type = TIMETICKS;
    break;
case 'a':
    vp->type = IPADDRESS;
    break;
default:
    fprintf(stderr, "bad type \"%c\", use \"i\", \"s\", \"x\", \"d\", \"n\", \"o\", \"t\", or \"a\".\n", *buf);
    return -1;
}
printf("Please enter new value: "); fflush(stdout);
gets(buf);
switch(vp->type){
case INTEGER:
    vp->val.integer = (long *)malloc(sizeof(long));
    *(vp->val.integer) = atoi(buf);
    vp->val_len = sizeof(long);
    break;
case STRING:
    if (ch == 'd'){
        vp->val_len = ascii_to_binary(buf, value);
    } else if (ch == 's'){
        strcpy(value, buf);
        vp->val_len = strlen(buf);
    } else if (ch == 'x'){
        vp->val_len = hex_to_binary(buf, value);
    }
    vp->val.string = (u_char *)malloc(vp->val_len);
    bcopy((char *)value, (char *)vp->val.string, vp->val_len);
    break;
case NULLOBJ:
    vp->val_len = 0;
    vp->val.string = NULL;
    break;
case OBJID:
    vp->val_len = MAX_NAME_LEN;;
    read_objid(buf, value, &vp->val_len);
    vp->val_len *= sizeof(oid);
    vp->val.objid = (oid *)malloc(vp->val_len);
    bcopy((char *)value, (char *)vp->val.objid, vp->val_len);
    break;
case TIMETICKS:
    vp->val.integer = (long *)malloc(sizeof(long));

```

```

        *(vp->val.integer) = atoi(buf);
        vp->val_len = sizeof(long);
        break;
    case IPADDRESS:
        vp->val.integer = (long *)malloc(sizeof(long));
        *(vp->val.integer) = inet_addr(buf);
        vp->val_len = sizeof(long);
        break;
    default:
        fprintf(stderr, "Internal error\n");
        break;
    }
} else { /* some form of get message */
    vp->type = NULLOBJ;
    vp->val_len = 0;
}
return 1;
}

/*****
 * jnu_api.c - API for access to snmp.
 *****/

#include <stdio.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>

#include "../jnusmp/asn1.h"
#include "../jnusmp/snmp.h"
#include "../jnusmp/snmp_impl.h"
#include "../jnusmp/snmp_api.h"

#define PACKET_LENGTH      4500

#ifndef BSD4_3
#define BSD4_2
#endif

#if defined(FD_SET) || defined(BSD4_3)
#define HAVE_FD_MACROS
#endif

#ifndef HAVE_FD_MACROS

typedef long    fd_mask;
#define NFDBITS      (sizeof(fd_mask) * NBBY)      /* bits per mask */

#define FD_SET(n, p)  ((p)->fds_bits[(n)/NFDBITS] |= (1 << ((n) % NFDBITS)))
#define FD_CLR(n, p)  ((p)->fds_bits[(n)/NFDBITS] &= ~(1 << ((n) % NFDBITS)))
#define FD_ISSET(n, p) ((p)->fds_bits[(n)/NFDBITS] & (1 << ((n) % NFDBITS)))
#define FD_ZERO(p)    bzero((char *) (p), sizeof(*(p)))

```

```

#endif /* HAVE_FD_MACROS */

oid default_enterprise[] = { 1, 3, 6, 1, 4, 1, 3, 1, 1 }; /* enterprises.jnu.systems.jnuSNMP */

#define DEFAULT_COMMUNITY "public"
#define DEFAULT_RETRIES 4
#define DEFAULT_TIMEOUT 1000000L
#define DEFAULT_REMPORT SNMP_PORT
#define DEFAULT_ENTERPRISE default_enterprise
#define DEFAULT_TIME 0

/* Internal information about the state of the snmp session. */
struct snmp_internal_session {
    int sd; /* socket descriptor for this connection */
    ipaddr addr; /* address of connected peer */
    struct request_list *requests; /* Info about outstanding requests */
};

/* A list of all the outstanding requests for a particular session. */
struct request_list {
    struct request_list *next_request;
    u_long request_id; /* request id */
    int retries; /* Number of retries */
    u_long timeout; /* length to wait for timeout */
    struct timeval time; /* Time this request was made */
    struct timeval expire; /* time this request is due to expire */
    struct snmp_pdu *pdu; /* The pdu for this request (saved so it can be retransmitted) */
};

/* The list of active/open sessions. */
struct session_list {
    struct session_list *next;
    struct snmp_session *session;
    struct snmp_internal_session *internal;
};

struct session_list *Sessions = NULL;

u_long Reqid = 0;
int snmp_errno = 0;

char *api_errors[4] = {
    "Unknown session",
    "Unknown host",
    "Invalid local port",
    "Unknown Error"
};

static char *
api_errstring(snmp_errnumber)
    int snmp_errnumber;
{
    if (snmp_errnumber <= SNMPERR_BAD_SESSION && snmp_errnumber >= SNMPERR_GENERR){
        return api_errors[snmp_errnumber + 4];
    } else {

```

```

        return "Unknown Error";
    }
}

/* Gets initial request ID for all transactions. */
static
init_snmp(){
    struct timeval tv;
    gettimeofday(&tv, (struct timezone *)0);
    srand(tv.tv_sec ^ tv.tv_usec);
    Reqid = random();
}

/* Sets up the session with the snmp_session information provided by the user. Then opens and binds the necessary
UDP port. A handle to the created session is returned (this is different than the pointer passed to snmp_open()). On
any error, NULL is returned and snmp_errno is set to the appropriate error code. */
struct snmp_session *
snmp_open(session)
    struct snmp_session *session;
{
    struct session_list *slp;
    struct snmp_internal_session *isp;
    u_char *cp;
    int sd;
    u_long addr;
    struct sockaddr_in me;
    struct hostent *hp;
    struct servent *servp;
    if (Reqid == 0)
        init_snmp();
    /* Copy session structure and link into list */
    slp = (struct session_list *)malloc(sizeof(struct session_list));
    slp->internal = isp = (struct snmp_internal_session *)malloc(sizeof(struct snmp_internal_session));
    bzero((char *)isp, sizeof(struct snmp_internal_session));
    slp->internal->sd = -1; /* mark it not set */
    slp->session = (struct snmp_session *)malloc(sizeof(struct snmp_session));
    bcopy((char *)session, (char *)slp->session, sizeof(struct snmp_session));
    session = slp->session;
    /* now link it in. */
    slp->next = Sessions;
    Sessions = slp;
    /* session now points to the new structure that still contains pointers to data allocated elsewhere. Some of this
data is copied to space malloc'd here, and the pointer replaced with the new one. */

    if (session->peername != NULL){
        cp = (u_char *)malloc((unsigned)strlen(session->peername) + 1);
        strcpy((char *)cp, session->peername);
        session->peername = (char *)cp;
    }

    /* Fill in defaults if necessary */
    if (session->community_len != SNMP_DEFAULT_COMMUNITY_LEN){
        cp = (u_char *)malloc((unsigned)session->community_len);
        bcopy((char *)session->community, (char *)cp, session->community_len);
    } else {

```

```

    session->community_len = strlen(DEFAULT_COMMUNITY);
    cp = (u_char *)malloc((unsigned)session->community_len);
    bcopy((char *)DEFAULT_COMMUNITY, (char *)cp, session->community_len);
}
session->community = cp;    /* replace pointer with pointer to new data */

if (session->retries == SNMP_DEFAULT_RETRIES)
    session->retries = DEFAULT_RETRIES;
if (session->timeout == SNMP_DEFAULT_TIMEOUT)
    session->timeout = DEFAULT_TIMEOUT;
isp->requests = NULL;

/* Set up connections */
sd = socket(AF_INET, SOCK_DGRAM, 0);
if (sd < 0){
    perror("socket");
    snmp_errno = SNMPERR_GENERR;
    if (!snmp_close(session)){
        fprintf(stderr, "Couldn't abort session: %s. Exiting\n", api_errstring(snmp_errno));
        exit(1);
    }
    return 0;
}
isp->sd = sd;
if (session->peername != SNMP_DEFAULT_PEERNAME){
    if ((addr = inet_addr(session->peername)) != -1){
        bcopy((char *)&addr, (char *)&isp->addr.sin_addr, sizeof(isp->addr.sin_addr));
    } else {
        hp = gethostbyname(session->peername);
        if (hp == NULL){
            fprintf(stderr, "unknown host: %s\n", session->peername);
            snmp_errno = SNMPERR_BAD_ADDRESS;
            if (!snmp_close(session)){
                fprintf(stderr, "Couldn't abort session: %s. Exiting\n", api_errstring(snmp_errno));
                exit(2);
            }
            return 0;
        } else {
            bcopy((char *)hp->h_addr, (char *)&isp->addr.sin_addr, hp->h_length);
        }
    }
    isp->addr.sin_family = AF_INET;
    if (session->remote_port == SNMP_DEFAULT_REMPORT){
        servp = getservbyname("snmp", "udp");
        if (servp != NULL){
            isp->addr.sin_port = servp->s_port;
        } else {
            isp->addr.sin_port = htons(SNMP_PORT);
        }
    } else {
        isp->addr.sin_port = htons(session->remote_port);
    }
} else {
    isp->addr.sin_addr.s_addr = SNMP_DEFAULT_ADDRESS;
}
}

```

```

me.sin_family = AF_INET;
me.sin_addr.s_addr = INADDR_ANY;
me.sin_port = htons(session->local_port);
if (bind(sd, (struct sockaddr *)&me, sizeof(me)) != 0){
    perror("bind");
    snmp_errno = SNMPERR_BAD_LOCPORT;
    if (!snmp_close(session)){
        fprintf(stderr, "Couldn't abort session: %s. Exiting\n", api_strerror(snmp_errno));
        exit(3);
    }
    return 0;
}
return session;
}
/* Free each element in the input request list. */
static
free_request_list(rp)
    struct request_list *rp;
{
    struct request_list *orp;

    while(rp){
        orp = rp;
        rp = rp->next_request;
        if (orp->pdu != NULL)
            snmp_free_pdu(orp->pdu);
        free((char *)orp);
    }
}

/* Close the input session. Frees all data allocated for the session, dequeues any pending requests, and closes any
sockets allocated for the session. Returns 0 on error, 1 otherwise. */
int
snmp_close(session)
    struct snmp_session *session;
{
    struct session_list *slp = NULL, *oslp = NULL;

    if (Sessions->session == session){ /* If first entry */
        slp = Sessions;
        Sessions = slp->next;
    } else {
        for(slp = Sessions; slp; slp = slp->next){
            if (slp->session == session){
                if (oslp) /* if we found entry that points here */
                    oslp->next = slp->next; /* link around this entry */
                break;
            }
            oslp = slp;
        }
    }
    /* If we found the session, free all data associated with it */
    if (slp){
        if (slp->session->community != NULL)

```



```

        free((char *)slp->session->community);
    if(slp->session->peername != NULL)
        free((char *)slp->session->peername);
    free((char *)slp->session);
    if (slp->internal->sd != -1)
        close(slp->internal->sd);
    free_request_list(slp->internal->requests);
    free((char *)slp->internal);
    free((char *)slp);
} else {
    snmp_errno = SNMPERR_BAD_SESSION;
    return 0;
}
return 1;
}

```

/* Takes a session and a pdu and serializes the ASN PDU into the area pointed to by packet. out_length is the size of the data area available. Returns the length of the completed packet in out_length. If any errors occur, -1 is returned. If all goes well, 0 is returned. */

```

static int
snmp_build(session, pdu, packet, out_length)
    struct snmp_session *session;
    struct snmp_pdu *pdu;
    register u_char *packet;
    int *out_length;
{
    u_char buf[PACKET_LENGTH];
    register u_char *cp;
    struct variable_list *vp;
    int length;
    long zero = 0;
    int totallength;

    length = *out_length;
    cp = packet;
    for(vp = pdu->variables; vp; vp = vp->next_variable){
        cp = snmp_build_var_op(cp, vp->name, &vp->name_length, vp->type, vp->val_len, (u_char *)vp-
>val.string, &length);
        if (cp == NULL)
            return -1;
    }
    totallength = cp - packet;

    length = PACKET_LENGTH;
    cp = asn_build_header(buf, &length, (u_char)(ASN_SEQUENCE | ASN_CONSTRUCTOR), totallength);
    if (cp == NULL)
        return -1;
    bcopy((char *)packet, (char *)cp, totallength);
    totallength += cp - buf;

    length = *out_length;
    if (pdu->command != TRP_REQ_MSG){
        /* request id */
        cp = asn_build_int(packet, &length,
            (u_char)(ASN_UNIVERSAL | ASN_PRIMITIVE | ASN_INTEGER),

```

```

    (long *)&pdu->reqid, sizeof(pdu->reqid));
if (cp == NULL)
    return -1;
/* error status */
cp = asn_build_int(cp, &length,
    (u_char)(ASN_UNIVERSAL | ASN_PRIMITIVE | ASN_INTEGER),
    (long *)&pdu->errstat, sizeof(pdu->errstat));
if (cp == NULL)
    return -1;
/* error index */
cp = asn_build_int(cp, &length,
    (u_char)(ASN_UNIVERSAL | ASN_PRIMITIVE | ASN_INTEGER),
    (long *)&pdu->errindex, sizeof(pdu->errindex));
if (cp == NULL)
    return -1;
} else { /* this is a trap message */
/* enterprise */
cp = asn_build_objid(packet, &length,
    (u_char)(ASN_UNIVERSAL | ASN_PRIMITIVE | ASN_OBJECT_ID),
    (oid *)&pdu->enterprise, pdu->enterprise_length);
if (cp == NULL)
    return -1;
/* agent-addr */
cp = asn_build_string(cp, &length,
    (u_char)(ASN_UNIVERSAL | ASN_PRIMITIVE | ASN_OCTET_STR),
    (u_char *)&pdu->agent_addr.sin_addr.s_addr, sizeof(pdu->agent_addr.sin_addr.s_addr));
if (cp == NULL)
    return -1;
/* generic trap */
cp = asn_build_int(cp, &length,
    (u_char)(ASN_UNIVERSAL | ASN_PRIMITIVE | ASN_INTEGER),
    (long *)&pdu->trap_type, sizeof(pdu->trap_type));
if (cp == NULL)
    return -1;
/* specific trap */
cp = asn_build_int(cp, &length,
    (u_char)(ASN_UNIVERSAL | ASN_PRIMITIVE | ASN_INTEGER),
    (long *)&pdu->specific_type, sizeof(pdu->specific_type));
if (cp == NULL)
    return -1;
/* timestamp */
cp = asn_build_int(cp, &length,
    (u_char)(ASN_UNIVERSAL | ASN_PRIMITIVE | ASN_INTEGER),
    (long *)&pdu->time, sizeof(pdu->time));
if (cp == NULL)
    return -1;
}
if (length < totallength)
    return -1;
bcopy((char *)buf, (char *)cp, totallength);
totallength += cp - packet;

length = PACKET_LENGTH;
cp = asn_build_header(buf, &length, (u_char)pdu->command, totallength);
if (cp == NULL)

```

```

        return -1;
    if (length < totallength)
        return -1;
    bcopy((char *)packet, (char *)cp, totallength);
    totallength += cp - buf;

    length = *out_length;
    cp = snmp_auth_build(packet, &length, session->community, &session->community_len, &zero, totallength);
    if (cp == NULL)
        return -1;
    if ((*out_length - (cp - packet)) < totallength)
        return -1;
    bcopy((char *)buf, (char *)cp, totallength);
    totallength += cp - packet;
    *out_length = totallength;
    return 0;
}

```

/* Parses the packet recieved on the input session, and places the data into the input pdu. length is the length of the input packet. If any errors are encountered, -1 is returned. Otherwise, a 0 is returned. */

```

static int
snmp_parse(session, pdu, data, length)
    struct snmp_session *session;
    struct snmp_pdu *pdu;
    u_char *data;
    int length;
{
    u_char msg_type;
    u_char type;
    u_char *var_val;
    long version;
    int len, four;
    u_char community[128];
    int community_length = 128;
    struct variable_list *vp;
    oid objid[MAX_NAME_LEN], *op;

    /* authenticates message and returns length if valid */
    data = snmp_auth_parse(data, &length, community, &community_length, &version);
    if (data == NULL)
        return -1;
    if (version != SNMP_VERSION_1){
        fprintf(stderr, "Wrong version: %d\n", version);
        fprintf(stderr, "Continuing anyway\n");
    }
    if (session->authenticator){
        data = session->authenticator(data, &length, community, community_length);
        if (data == NULL)
            return 0;
    }
    data = asn_parse_header(data, &length, &msg_type);
    if (data == NULL)
        return -1;
    pdu->command = msg_type;
    if (pdu->command != TRP_REQ_MSG){

```

```

data = asn_parse_int(data, &length, &type, (long *)&pdu->reqid, sizeof(pdu->reqid));
if (data == NULL)
    return -1;
data = asn_parse_int(data, &length, &type, (long *)&pdu->errstat, sizeof(pdu->errstat));
if (data == NULL)
    return -1;
data = asn_parse_int(data, &length, &type, (long *)&pdu->errindex, sizeof(pdu->errindex));
if (data == NULL)
    return -1;
} else {
pdu->enterprise_length = MAX_NAME_LEN;
data = asn_parse_objid(data, &length, &type, objid, &pdu->enterprise_length);
if (data == NULL)
    return -1;
pdu->enterprise = (oid *)malloc(pdu->enterprise_length * sizeof(oid));
bcopy((char *)objid, (char *)pdu->enterprise, pdu->enterprise_length * sizeof(oid));

four = 4;
data = asn_parse_string(data, &length, &type, (u_char *)&pdu->agent_addr.sin_addr.s_addr, &four);
if (data == NULL)
    return -1;
data = asn_parse_int(data, &length, &type, (long *)&pdu->trap_type, sizeof(pdu->trap_type));
if (data == NULL)
    return -1;
data = asn_parse_int(data, &length, &type, (long *)&pdu->specific_type, sizeof(pdu->specific_type));
if (data == NULL)
    return -1;
data = asn_parse_int(data, &length, &type, (long *)&pdu->time, sizeof(pdu->time));
if (data == NULL)
    return -1;
}
data = asn_parse_header(data, &length, &type);
if (data == NULL)
    return -1;
if (type != (u_char)(ASN_SEQUENCE | ASN_CONSTRUCTOR))
    return -1;
while((int)length > 0){
    if (pdu->variables == NULL){
        pdu->variables = vp = (struct variable_list *)malloc(sizeof(struct variable_list));
    } else {
        vp->next_variable = (struct variable_list *)malloc(sizeof(struct variable_list));
        vp = vp->next_variable;
    }
    vp->next_variable = NULL;
    vp->val.string = NULL;
    vp->name = NULL;
    vp->name_length = MAX_NAME_LEN;
    data = snmp_parse_var_op(data, objid, &vp->name_length, &vp->type, &vp->val_len, &var_val, (int
*)&length);
    if (data == NULL)
        return -1;
    op = (oid *)malloc(((unsigned)vp->name_length * sizeof(oid));
    bcopy((char *)objid, (char *)op, vp->name_length * sizeof(oid));
    vp->name = op;

```

```

len = PACKET_LENGTH;
switch((short)vp->type){
  case ASN_INTEGER:
  case COUNTER:
  case GAUGE:
  case TIMETICKS:
    vp->val.integer = (long *)malloc(sizeof(long));
    vp->val_len = sizeof(long);
    asn_parse_int(var_val, &len, &vp->type, (long *)vp->val.integer, sizeof(vp->val.integer));
    break;
  case ASN_OCTET_STR:
  case IPADDRESS:
  case OPAQUE:
    vp->val.string = (u_char *)malloc((unsigned)vp->val_len);
    asn_parse_string(var_val, &len, &vp->type, vp->val.string, &vp->val_len);
    break;
  case ASN_OBJECT_ID:
    vp->val_len = MAX_NAME_LEN;
    asn_parse_objid(var_val, &len, &vp->type, objid, &vp->val_len);
    vp->val_len *= sizeof(oid);
    vp->val.objid = (oid *)malloc((unsigned)vp->val_len);
    bcopy((char *)objid, (char *)vp->val.objid, vp->val_len);
    break;
  case ASN_NULL:
    break;
  default:
    fprintf(stderr, "bad type returned (%x)\n", vp->type);
    break;
}
}
return 0;
}

```

/* Sends the input pdu on the session after calling snmp_build to create a serialized packet. If necessary, set some of the pdu data from the session defaults. Add a request corresponding to this pdu to the list of outstanding requests on this session, then send the pdu. Returns the request id of the generated packet if applicable, otherwise 1. On any error, 0 is returned. The pdu is freed by snmp_send() unless a failure occurred. */

```

int
snmp_send(session, pdu)
  struct snmp_session *session;
  struct snmp_pdu *pdu;
{
  struct session_list *slp;
  struct snmp_internal_session *isp = NULL;
  u_char packet[PACKET_LENGTH];
  int length = PACKET_LENGTH;
  struct request_list *rp;
  struct timeval tv;

  for(slp = Sessions; slp; slp = slp->next){
    if (slp->session == session){
      isp = slp->internal;
      break;
    }
  }
}

```

```

    if (isp == NULL){
        snmp_errno = SNMPERR_BAD_SESSION;
        return 0;
    }
    if (pdu->command == GET_REQ_MSG || pdu->command == GETNEXT_REQ_MSG
        || pdu->command == GET_RSP_MSG || pdu->command == SET_REQ_MSG){
        if (pdu->reqid == SNMP_DEFAULT_REQID)
            pdu->reqid = ++Reqid;
        if (pdu->errstat == SNMP_DEFAULT_ERRSTAT)
            pdu->errstat = 0;
        if (pdu->errindex == SNMP_DEFAULT_ERRINDEX)
            pdu->errindex = 0;
    } else {
        /* fill in trap defaults */
        pdu->reqid = 1; /* give a bogus non-error reqid for traps */
        if (pdu->enterprise_length == SNMP_DEFAULT_ENTERPRISE_LENGTH){
            pdu->enterprise = (oid *)malloc(sizeof(DEFAULT_ENTERPRISE));
            bcopy((char *)DEFAULT_ENTERPRISE, (char *)pdu->enterprise, sizeof(DEFAULT_ENTERPRISE));
            pdu->enterprise_length = sizeof(DEFAULT_ENTERPRISE)/sizeof(oid);
        }
        if (pdu->time == SNMP_DEFAULT_TIME)
            pdu->time = DEFAULT_TIME;
    }
    if (pdu->address.sin_addr.s_addr == SNMP_DEFAULT_ADDRESS){
        if (isp->addr.sin_addr.s_addr != SNMP_DEFAULT_ADDRESS){
            bcopy((char *)&isp->addr, (char *)&pdu->address, sizeof(pdu->address));
        } else {
            fprintf(stderr, "No remote IP address specified\n");
            snmp_errno = SNMPERR_BAD_ADDRESS;
            return 0;
        }
    }
}
if (snmp_build(session, pdu, packet, &length) < 0){
    fprintf(stderr, "Error building packet\n");
    snmp_errno = SNMPERR_GENERR;
    return 0;
}
if (snmp_dump_packet){
    int count;

    for(count = 0; count < length; count++){
        printf("%02X ", packet[count]);
        if ((count % 16) == 15)
            printf("\n");
    }
    printf("\n\n");
}

gettimeofday(&tv, (struct timezone *)0);
if (sendto(isp->sd, (char *)packet, length, 0, (struct sockaddr *)&pdu->address, sizeof(pdu->address)) < 0){
    perror("sendto");
    snmp_errno = SNMPERR_GENERR;
    return 0;
}

```

```

if (pdu->command == GET_REQ_MSG || pdu->command == GETNEXT_REQ_MSG || pdu->command ==
SET_REQ_MSG){
    /* set up to expect a response */
    rp = (struct request_list *)malloc(sizeof(struct request_list));
    rp->next_request = isp->requests;
    isp->requests = rp;
    rp->pdu = pdu;
    rp->request_id = pdu->reqid;

    rp->retries = 1;
    rp->timeout = session->timeout;
    rp->time = tv;
    tv.tv_usec += rp->timeout;
    tv.tv_sec += tv.tv_usec / 1000000L;
    tv.tv_usec %= 1000000L;
    rp->expire = tv;
}
return pdu->reqid;
}

```

/* Frees the pdu and any malloc'd data associated with it. */

```

void
snmp_free_pdu(pdu)
    struct snmp_pdu *pdu;
{
    struct variable_list *vp, *ovp;

    vp = pdu->variables;
    while(vp){
        if (vp->name)
            free((char *)vp->name);
        if (vp->val.string)
            free((char *)vp->val.string);
        ovp = vp;
        vp = vp->next_variable;
        free((char *)ovp);
    }
    if (pdu->enterprise)
        free((char *)pdu->enterprise);
    free((char *)pdu);
}

```

/* Checks to see if any of the fd's set in the fdset belong to snmp. Each socket with it's fd set has a packet read from it and snmp_parse is called on the packet received. The resulting pdu is passed to the callback routine for that session. If the callback routine returns successfully, the pdu and it's request are deleted. */

```

void
snmp_read(fdset)
    fd_set *fdset;
{
    struct session_list *slp;
    struct snmp_session *sp;
    struct snmp_internal_session *isp;
    u_char packet[PACKET_LENGTH];
    struct sockaddr_in from;
    int length, fromlength;

```

```

struct snmp_pdu *pdu;
struct request_list *rp, *orp;

for(slp = Sessions; slp; slp = slp->next){
    if (FD_ISSET(slp->internal->sd, fdset)){
        sp = slp->session;
        isp = slp->internal;
        fromlength = sizeof from;
        length = recvfrom(isp->sd, (char *)packet, PACKET_LENGTH, 0, (struct sockaddr *)&from,
&fromlength);
        if (length == -1)
            perror("recvfrom");
        if (snmp_dump_packet){
            int count;

            printf("recieved %d bytes from %s:\n", length, inet_ntoa(from.sin_addr));
            for(count = 0; count < length; count++){
                printf("%02X ", packet[count]);
                if ((count % 16) == 15)
                    printf("\n");
            }
            printf("\n\n");
        }

        pdu = (struct snmp_pdu *)malloc(sizeof(struct snmp_pdu));
        pdu->address = from;
        pdu->reqid = 0;
        pdu->variables = NULL;
        pdu->enterprise = NULL;
        pdu->enterprise_length = 0;
        if (snmp_parse(sp, pdu, packet, length) != SNMP_ERR_NOERROR){
            fprintf(stderr, "Mangled packet\n");
            snmp_free_pdu(pdu);
            return;
        }

        if (pdu->command == GET_RSP_MSG){
            for(rp = isp->requests; rp; rp = rp->next_request){
                if (rp->request_id == pdu->reqid){
                    if (sp->callback(RECEIVED_MESSAGE, sp, pdu->reqid, pdu, sp->callback_magic) ==
1){

                        /* successful, so delete request */
                        orp = rp;
                        if (isp->requests == orp){
                            /* first in list */
                            isp->requests = orp->next_request;
                        } else {
                            for(rp = isp->requests; rp; rp = rp->next_request){
                                if (rp->next_request == orp){
                                    rp->next_request = orp->next_request;      /* link around it */
                                    break;
                                }
                            }
                        }
                    }
                }
            }
            snmp_free_pdu(orp->pdu);
        }
    }
}

```



```

        free((char *)orp);
        break; /* there shouldn't be any more request with the same reqid */
    }
}
} else if (pdu->command == GET_REQ_MSG || pdu->command == GETNEXT_REQ_MSG
    || pdu->command == TRP_REQ_MSG || pdu->command == SET_REQ_MSG){
    sp->callback(RECEIVED_MESSAGE, sp, pdu->reqid, pdu, sp->callback_magic);
}
snmp_free_pdu(pdu);
}
}
}

```

/* Returns info about what snmp requires from a select statement. numfds is the number of fds in the list that are significant. All file descriptors opened for SNMP are OR'd into the fdset. If activity occurs on any of these file descriptors, snmp_read should be called with that file descriptor set

* The timeout is the latest time that SNMP can wait for a timeout. The select should be done with the minimum time between timeout and any other timeouts necessary. This should be checked upon each invocation of select. If a timeout is received, snmp_timeout should be called to check if the timeout was for SNMP. (snmp_timeout is idempotent)

* Block is 1 if the select is requested to block indefinitely, rather than time out. If block is input as 1, the timeout value will be treated as undefined, but it must be available for setting in snmp_select_info. On return, if block is true, the value of timeout will be undefined. * snmp_select_info returns the number of open sockets. (i.e. The number of sessions open) */

int

snmp_select_info(numfds, fdset, timeout, block)

```

    int *numfds;
    fd_set *fdset;
    struct timeval *timeout;
    int *block; /* should the select block until input arrives (i.e. no input) */

```

```

{
    struct session_list *slp;
    struct snmp_internal_session *isp;
    struct request_list *rp;
    struct timeval now, earliest;
    int active = 0, requests = 0;

```

timerclear(&earliest);

/* For each request outstanding, add it's socket to the fdset, and if it is the earliest timeout to expire, mark it as lowest. */

```

    for(slp = Sessions; slp; slp = slp->next){
        active++;
        isp = slp->internal;
        if ((isp->sd + 1) > *numfds)
            *numfds = (isp->sd + 1);
        FD_SET(isp->sd, fdset);
        if (isp->requests){
            /* found another session with outstanding requests */
            requests++;
            for(rp = isp->requests; rp; rp = rp->next_request){
                if (!timerisset(&earliest) || timercmp(&rp->expire, &earliest, <))
                    earliest = rp->expire;
            }
        }
    }

```

if (requests == 0) /* if none are active, skip arithmetic */

```

return active;

/* Now find out how much time until the earliest timeout. This transforms earliest from an absolute time into a
delta time, the time left until the select should timeout. */
gettimeofday(&now, (struct timezone *)0);
earliest.tv_sec--; /* adjust time to make arithmetic easier */
earliest.tv_usec += 1000000L;
earliest.tv_sec -= now.tv_sec;
earliest.tv_usec -= now.tv_usec;
while (earliest.tv_usec >= 1000000L){
    earliest.tv_usec -= 1000000L;
    earliest.tv_sec += 1;
}
if (earliest.tv_sec < 0){
    earliest.tv_sec = 0;
    earliest.tv_usec = 0;
}
/* if it was blocking before or our delta time is less, reset timeout */
if (*block == 1 || timercmp(&earliest, timeout, <)){
    *timeout = earliest;
    *block = 0;
}
return active;
}

/* snmp_timeout should be called whenever the timeout from snmp_select_info expires,
* but it is idempotent, so snmp_timeout can be polled (probably a cpu expensive proposition). snmp_timeout
checks to see if any of the sessions have an outstanding request that has timed out. If it finds one (or more), and that
pdu has more retries available, a new packet is formed from the pdu and is resent. If there are no more retries
available, the callback for the session is used to alert the user of the timeout. */
void
snmp_timeout(){
    struct session_list *slp;
    struct snmp_session *sp;
    struct snmp_internal_session *isp;
    struct request_list *rp, *orp, *freeme = NULL;
    struct timeval now;

    gettimeofday(&now, (struct timezone *)0);
    /* For each request outstanding, check to see if it has expired. */
    for(slp = Sessions; slp; slp = slp->next){
        sp = slp->session;
        isp = slp->internal;
        orp = NULL;
        for(rp = isp->requests; rp; rp = rp->next_request){
            if (freeme != NULL){ /* frees rp's after the for loop goes on to the next_request */
                free((char *)freeme);
                freeme = NULL;
            }
            if (timercmp(&rp->expire, &now, <)){
                /* this timer has expired */
                if (rp->retries >= sp->retries){
                    /* No more chances, delete this entry */
                    sp->callback(TIMED_OUT, sp, rp->pdu->reqid, rp->pdu, sp->callback_magic);
                    if (orp == NULL){

```

```

        isp->requests = rp->next_request;
    } else {
        orp->next_request = rp->next_request;
    }
    snmp_free_pdu(rp->pdu);
    freeme = rp;
    continue;    /* don't update orp below */
} else {
    u_char packet[PACKET_LENGTH];
    int length = PACKET_LENGTH;
    struct timeval tv;

    /* retransmit this pdu */
    rp->retries++;
    rp->timeout <<= 1;
    if (snmp_build(sp, rp->pdu, packet, &length) < 0){
        fprintf(stderr, "Error building packet\n");
    }
    if (snmp_dump_packet){
        int count;

        for(count = 0; count < length; count++){
            printf("%02X ", packet[count]);
            if ((count % 16) == 15)
                printf("\n");
        }
        printf("\n\n");
    }
    gettimeofday(&tv, (struct timezone *)0);
    if (sendto(isp->sd, (char *)packet, length, 0, (struct sockaddr *)&rp->pdu->address, sizeof(rp->pdu->address)) < 0){
        perror("sendto");
    }
    rp->time = tv;
    tv.tv_usec += rp->timeout;
    tv.tv_sec += tv.tv_usec / 1000000L;
    tv.tv_usec %= 1000000L;
    rp->expire = tv;
}
}
orp = rp;
}
if (freeme != NULL){
    free((char *)freeme);
    freeme = NULL;
}
}
}
}

```

```

/*****
* jnu_snmp_client.c - a toolkit of common functions for an SNMP client.
*****/

```

```

#include <sys/types.h>
#include <sys/param.h>
#include <stdio.h>

```

```

#include <netinet/in.h>
#include <sys/time.h>
#include <errno.h>

#include "../jnusnmp/asn1.h"
#include "../jnusnmp/snmp.h"
#include "../jnusnmp/snmp_impl.h"
#include "../jnusnmp/snmp_api.h"
#include "../jnusnmp/snmp_client.h"

#ifdef BSD4_3
#define BSD4_2
#endif

#if defined(FD_SET) || defined(BSD4_3)
#define HAVE_FD_MACROS
#endif

#ifdef HAVE_FD_MACROS

typedef long    fd_mask;
#define NFDBITS    (sizeof(fd_mask) * NBBY)    /* bits per mask */

#define FD_SET(n, p)    ((p)->fds_bits[(n)/NFDBITS] |= (1 << ((n) % NFDBITS)))
#define FD_CLR(n, p)    ((p)->fds_bits[(n)/NFDBITS] &= ~(1 << ((n) % NFDBITS)))
#define FD_ISSET(n, p)  ((p)->fds_bits[(n)/NFDBITS] & (1 << ((n) % NFDBITS)))
#define FD_ZERO(p)      bzero((char *) (p), sizeof(*(p)))
#endif /* HAVE_FD_MACROS */
extern int errno;
struct synch_state snmp_synch_state;

struct snmp_pdu *
snmp_pdu_create(command)
    int command;
{
    struct snmp_pdu *pdu;

    pdu = (struct snmp_pdu *) malloc(sizeof(struct snmp_pdu));
    bzero((char *) pdu, sizeof(struct snmp_pdu));
    pdu->command = command;
    pdu->errstat = SNMP_DEFAULT_ERRSTAT;
    pdu->errindex = SNMP_DEFAULT_ERRINDEX;
    pdu->address.sin_addr.s_addr = SNMP_DEFAULT_ADDRESS;
    pdu->enterprise = NULL;
    pdu->enterprise_length = 0;
    pdu->variables = NULL;
    return pdu;
}

/* Add a null variable with the requested name to the end of the list of variables for this pdu. */
snmp_add_null_var(pdu, name, name_length)
    struct snmp_pdu *pdu;
    oid *name;
    int name_length;
{

```

```

struct variable_list *vars;

if (pdu->variables == NULL){
    pdu->variables = vars = (struct variable_list *)malloc(sizeof(struct variable_list));
} else {
    for(vars = pdu->variables; vars->next_variable; vars = vars->next_variable)
        ;
    vars->next_variable = (struct variable_list *)malloc(sizeof(struct variable_list));
    vars = vars->next_variable;
}

vars->next_variable = NULL;
vars->name = (oid *)malloc(name_length * sizeof(oid));
bcopy((char *)name, (char *)vars->name, name_length * sizeof(oid));
vars->name_length = name_length;
vars->type = ASN_NULL;
vars->val.string = NULL;
vars->val_len = 0;
}

```

```

snmp_synch_input(op, session, reqid, pdu, magic)

```

```

int op;
struct snmp_session *session;
int reqid;
struct snmp_pdu *pdu;
void *magic;
{
    struct variable_list *var, *newvar;
    struct synch_state *state = (struct synch_state *)magic;
    struct snmp_pdu *newpdu;

    if (reqid != state->reqid)
        return 0;
    state->waiting = 0;
    if (op == RECEIVED_MESSAGE && pdu->command == GET_RSP_MSG){
        /* clone the pdu */
        state->pdu = newpdu = (struct snmp_pdu *)malloc(sizeof(struct snmp_pdu));
        bcopy((char *)pdu, (char *)newpdu, sizeof(struct snmp_pdu));
        newpdu->variables = 0;
        var = pdu->variables;
        if (var != NULL){
            newpdu->variables = newvar = (struct variable_list *)malloc(sizeof(struct variable_list));
            bcopy((char *)var, (char *)newvar, sizeof(struct variable_list));
            if (var->name != NULL){
                newvar->name = (oid *)malloc(var->name_length * sizeof(oid));
                bcopy((char *)var->name, (char *)newvar->name, var->name_length * sizeof(oid));
            }
            if (var->val.string != NULL){
                newvar->val.string = (u_char *)malloc(var->val_len);
                bcopy((char *)var->val.string, (char *)newvar->val.string, var->val_len);
            }
            newvar->next_variable = 0;
            while(var->next_variable){
                newvar->next_variable = (struct variable_list *)malloc(sizeof(struct variable_list));
                var = var->next_variable;
            }
        }
    }
}

```

```

        newvar = newvar->next_variable;
        • bcopy((char *)var, (char *)newvar, sizeof(struct variable_list));
          if (var->name != NULL){
              newvar->name = (oid *)malloc(var->name_length * sizeof(oid));
              bcopy((char *)var->name, (char *)newvar->name, var->name_length * sizeof(oid));
          }
          if (var->val.string != NULL){
              newvar->val.string = (u_char *)malloc(var->val_len);
              bcopy((char *)var->val.string, (char *)newvar->val.string, var->val_len);
          }
          newvar->next_variable = 0;
      }
  }
  state->status = STAT_SUCCESS;
} else if (op == TIMED_OUT){
    state->status = STAT_TIMEOUT;
}
return 1;
}
/* If there was an error in the input pdu, creates a clone of the pdu that includes all the variables except the one
marked by the errindex. The command is set to the input command and the reqid, errstat, and errindex are set to
default values. If the error status didn't indicate an error, the error index didn't indicate a variable, the pdu wasn't a
get response message, or there would be no remaining variables, this function will return NULL. If everything was
successful, a pointer to the fixed cloned pdu will be returned.*/
struct snmp_pdu *
snmp_fix_pdu(pdu, command)
    struct snmp_pdu *pdu;
    int command;
{
    struct variable_list *var, *newvar;
    struct snmp_pdu *newpdu;
    int index, copied = 0;

    if (pdu->command != GET_RSP_MSG || pdu->errstat == SNMP_ERR_NOERROR || pdu->errindex <= 0)
        return NULL;
    /* clone the pdu */
    newpdu = (struct snmp_pdu *)malloc(sizeof(struct snmp_pdu));
    bcopy((char *)pdu, (char *)newpdu, sizeof(struct snmp_pdu));
    newpdu->variables = 0;
    newpdu->command = command;
    newpdu->reqid = SNMP_DEFAULT_REQID;
    newpdu->errstat = SNMP_DEFAULT_ERRSTAT;
    newpdu->errindex = SNMP_DEFAULT_ERRINDEX;
    var = pdu->variables;
    index = 1;
    if (pdu->errindex == index){ /* skip first variable */
        var = var->next_variable;
        index++;
    }
    if (var != NULL){
        newpdu->variables = newvar = (struct variable_list *)malloc(sizeof(struct variable_list));
        bcopy((char *)var, (char *)newvar, sizeof(struct variable_list));
        if (var->name != NULL){
            newvar->name = (oid *)malloc(var->name_length * sizeof(oid));
            bcopy((char *)var->name, (char *)newvar->name, var->name_length * sizeof(oid));
        }
    }
}

```

```

    }
    if (var->val.string != NULL){
        newvar->val.string = (u_char *)malloc(var->val_len);
        bcopy((char *)var->val.string, (char *)newvar->val.string, var->val_len);
    }
    newvar->next_variable = 0;
    copied++;

    while(var->next_variable){
        var = var->next_variable;
        if (++index == pdu->errindex)
            continue;
        newvar->next_variable = (struct variable_list *)malloc(sizeof(struct variable_list));
        newvar = newvar->next_variable;
        bcopy((char *)var, (char *)newvar, sizeof(struct variable_list));
        if (var->name != NULL){
            newvar->name = (oid *)malloc(var->name_length * sizeof(oid));
            bcopy((char *)var->name, (char *)newvar->name, var->name_length * sizeof(oid));
        }
        if (var->val.string != NULL){
            newvar->val.string = (u_char *)malloc(var->val_len);
            bcopy((char *)var->val.string, (char *)newvar->val.string, var->val_len);
        }
        newvar->next_variable = 0;
        copied++;
    }
}
if (index < pdu->errindex || copied == 0){
    snmp_free_pdu(newpdu);
    return NULL;
}
return newpdu;
}
int
snmp_synch_response(ss, pdu, response)
    struct snmp_session *ss;
    struct snmp_pdu *pdu;
    struct snmp_pdu **response;
{
    struct synch_state *state = &snmp_synch_state;
    int numfds, count;
    fd_set fdset;
    struct timeval timeout, *tvp;
    int block;
    if ((state->reqid = snmp_send(ss, pdu)) == 0){
        *response = NULL;
        snmp_free_pdu(pdu);
        return STAT_ERROR;
    }
    state->waiting = 1;

    while(state->waiting){
        numfds = 0;
        FD_ZERO(&fdset);
        block = 1;

```

```

tvp = &timeout;
timerclear(tvp);
snmp_select_info(&numfds, &fdset, tvp, &block);
if (block == 1)
    tvp = NULL; /* block without timeout */
count = select(numfds, &fdset, 0, 0, tvp);
if (count > 0){
    snmp_read(&fdset);
} else switch(count){
    case 0:
        snmp_timeout();
        break;
    case -1:
        if (errno == EINTR){
            continue;
        } else {
            perror("select");
        }
        /* FALLTHRU */
    default:
        return STAT_ERROR;
}
}
*response = state->pdu;
return state->status;
}

snmp_synch_setup(session)
struct snmp_session *session;
{
    session->callback = snmp_synch_input;
    session->callback_magic = (void *)&snmp_synch_state;
}

char *error_string[6] = {
    "No Error",
    "Response message would have been too large.",
    "There is no such variable name in this MIB.",
    "The value given has the wrong type or length",
    "This variable is read only",
    "A general failure occurred"
};

char *
snmp_errstring(errstat)
int errstat;
{
    if (errstat <= SNMP_ERR_GENERR && errstat >= SNMP_ERR_NOERROR){
        return error_string[errstat];
    } else {
        return "Unknown Error";
    }
}
}
/*****
* Abstract Syntax Notation One, ASN.1 * asn1.c *

```


* As defined in ISO/IS 8824 and ISO/IS 8825 * This implements a subset of the above International Standards that
 * is sufficient to implement SNMP. * Encodes abstract data types into a machine independent stream of bytes.
 *****/

```
#ifndef KINETICS
```

```
#include "gw.h"
```

```
#endif
```

```
#if (defined(unix) && !defined(KINETICS))
```

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

```
#include <netinet/in.h>
```

```
#endif
```

```
#include "../jnusnmp/asn1.h"
```

```
#ifndef NULL
```

```
#define NULL 0
```

```
#endif
```

```
#define ERROR(x)
```

```
/* asn_parse_int - pulls a long out of an ASN int type. On entry, datalength is input as the number of valid bytes  

  following "data". On exit, it is returned as the number of valid bytes following the end of this object. Returns a  

  pointer to the first byte past the end
```

```
 * of this object (i.e. the start of the next object). Returns NULL on any error. */
```

```
u_char *
```

```
asn_parse_int(data, datalength, type, intp, intsize)
```

```
    register u_char *data; /* IN - pointer to start of object */
```

```
    register int *datalength; /* IN/OUT - number of valid bytes left in buffer */
```

```
    u_char *type; /* OUT - asn type of object */
```

```
    long *intp; /* IN/OUT - pointer to start of output buffer */
```

```
    int intsize; /* IN - size of output buffer */
```

```
{
```

```
/* ASN.1 integer ::= 0x02 asnlength byte {byte}* */
```

```
    register u_char *bufp = data;
```

```
    u_long asn_length;
```

```
    register long value = 0;
```

```
    if (intsize != sizeof (long)){
```

```
        ERROR("not long");
```

```
        return NULL;
```

```
    }
```

```
    *type = *bufp++;
```

```
    bufp = asn_parse_length(bufp, &asn_length);
```

```
    if (bufp == NULL){
```

```
        ERROR("bad length");
```

```
        return NULL;
```

```
    }
```

```
    if (asn_length + (bufp - data) > *datalength){
```

```
        ERROR("overflow of message");
```

```
        return NULL;
```

```
    }
```

```
    if (asn_length == intsize + 1 && *bufp == 0) {
```

```
/* this will cause a positive 32 bit integer to be returned as negative, which is only OK if our result is to be treated as  

  unsigned - in practice its likely to be OK... */
```

```
        asn_length--;
```

```
        bufp++;
```

```

}
if (asn_length > intsize){
    ERROR("I don't support such large integers");
    return NULL;
}
*datalength -= (int)asn_length + (bufp - data);
if (*bufp & 0x80)
    value = -1; /* integer is negative */
while(asn_length--)
    value = (value << 8) | *bufp++;
*intp = value;
return bufp;
}

```

/* asn_build_int - builds an ASN object containing an integer. On entry, datalength is input as the number of valid bytes following "data". On exit, it is returned as the number of valid bytes following the end of this object. * Returns a pointer to the first byte past the end of this object (i.e. the start of the next object). * Returns NULL on any error. */

```

u_char *
asn_build_int(data, datalength, type, intp, intsize)
    register u_char *data; /* IN - pointer to start of output buffer */
    register int *datalength; /* IN/OUT - number of valid bytes left in buffer */
    u_char type; /* IN - asn type of object */
    register long *intp; /* IN - pointer to start of long integer */
    register int intsize; /* IN - size of *intp */
{
    /* ASN.1 integer ::= 0x02 asnlength byte {byte} */
    register long integer;
    register u_long mask;

    if (intsize != sizeof (long))
        return NULL;
    integer = *intp;

    /* Truncate "unnecessary" bytes off of the most significant end of this 2's complement integer. There should be no
    sequence of 9 consecutive 1's or 0's at the most significant end of the integer. */
    mask = 0x1FF << ((8 * (sizeof(long) - 1)) - 1);
    /* mask is 0xFF800000 on a big-endian machine */
    while((((integer & mask) == 0) || ((integer & mask) == mask)) && intsize > 1){
        intsize--;
        integer <<= 8;
    }
    data = asn_build_header(data, datalength, type, intsize);
    if (data == NULL)
        return NULL;
    if (*datalength < intsize)
        return NULL;
    *datalength -= intsize;
    mask = 0xFF << (8 * (sizeof(long) - 1));
    /* mask is 0xFF000000 on a big-endian machine */
    while(intsize--){
        *data++ = (u_char)((integer & mask) >> (8 * (sizeof(long) - 1)));
        integer <<= 8;
    }
    return data;
}

```

```

}
/* asn_parse_string - pulls an octet string out of an ASN octet string type. On entry, datalength is input as the
number of valid bytes following "data". On exit, it is returned as the number of valid bytes following the beginning
of the next object. "string" is filled with the octet string. Returns a pointer to the first byte past the end of this object
(i.e. the start of the next object). Returns NULL on any error. */

```

```

u_char *

```

```

asn_parse_string(data, datalength, type, string, strlen)
    u_char      *data; /* IN - pointer to start of object */
    register int *datalength; /* IN/OUT - number of valid bytes left in buffer */
    u_char      *type; /* OUT - asn type of object */
    u_char      *string; /* IN/OUT - pointer to start of output buffer */
    register int *strlen; /* IN/OUT - size of output buffer */

```

```

{
/* ASN.1 octet string ::= primstring | cmpdstring primstring ::= 0x04 asnlength byte {byte}* cmpdstring ::= 0x24
asnlength string {string}* This doesn't yet support the compound string. */

```

```

    register u_char *bufp = data;
    u_long          asn_length;

    *type = *bufp++;
    bufp = asn_parse_length(bufp, &asn_length);
    if (bufp == NULL)
        return NULL;
    if (asn_length + (bufp - data) > *datalength){
        ERROR("overflow of message");
        return NULL;
    }
    if (asn_length > *strlen){
        ERROR("I don't support such long strings");
        return NULL;
    }
    bcopy((char *)bufp, (char *)string, (int)asn_length);
    *strlen = (int)asn_length;
    *datalength -= (int)asn_length + (bufp - data);
    return bufp + asn_length;
}

```

```

/* asn_build_string - Builds an ASN octet string object containing the input string.
* On entry, datalength is input as the number of valid bytes following "data". On exit, it is returned as the number
of valid bytes following the beginning of the next object. Returns a pointer to the first byte past the end of this
object (i.e. the start of the next object). Returns NULL on any error. */

```

```

u_char *

```

```

asn_build_string(data, datalength, type, string, strlen)
    u_char      *data; /* IN - pointer to start of object */
    register int *datalength; /* IN/OUT - number of valid bytes left in buffer */
    u_char      type; /* IN - ASN type of string */
    u_char      *string; /* IN - pointer to start of input buffer */
    register int *strlen; /* IN - size of input buffer */

```

```

{
/* ASN.1 octet string ::= primstring | cmpdstring
* primstring ::= 0x04 asnlength byte {byte}*
* cmpdstring ::= 0x24 asnlength string {string}*
* This code will never send a compound string. */
data = asn_build_header(data, datalength, type, strlen);
if (data == NULL)
    return NULL;

```

```

if (*datalength < strlen)
    return NULL;
bcopy((char *)string, (char *)data, strlen);
*datalength -= strlen;
return data + strlen;
}
/* asn_parse_header - interprets the ID and length of the current object. On entry, datalength is input as the number
of valid bytes following "data". On exit, it is returned as the number of valid bytes in this object following the id
and length. Returns a pointer to the first byte of the contents of this object. Returns NULL on any error. */
u_char *
asn_parse_header(data, datalength, type)
    u_char      *data; /* IN - pointer to start of object */
    int         *datalength; /* IN/OUT - number of valid bytes left in buffer */
    u_char      *type; /* OUT - ASN type of object */
{
    register u_char *bufp = data;
    register      header_len;
    u_long        asn_length;

    /* this only works on data types < 30, i.e. no extension octets */
    if (IS_EXTENSION_ID(*bufp)){
        ERROR("can't process ID >= 30");
        return NULL;
    }
    *type = *bufp;
    bufp = asn_parse_length(bufp + 1, &asn_length);
    if (bufp == NULL)
        return NULL;
    header_len = bufp - data;
    if (header_len + asn_length > *datalength){
        ERROR("asn length too long");
        return NULL;
    }
    *datalength = (int)asn_length;
    return bufp;
}
/* asn_build_header - builds an ASN header for an object with the ID and length specified. On entry, datalength is
input as the number of valid bytes following
* "data". On exit, it is returned as the number of valid bytes in this object following the id and length. This only
works on data types < 30, i.e. no extension octets. The maximum length is 0xFFFF; Returns a pointer to the first
byte of the contents of this object. Returns NULL on any error. */
u_char *
asn_build_header(data, datalength, type, length)
    register u_char *data; /* IN - pointer to start of object */
    int         *datalength; /* IN/OUT - number of valid bytes left in buffer */
    u_char      type; /* IN - ASN type of object */
    int         length; /* IN - length of object */
{
    if (*datalength < 1)
        return NULL;
    *data++ = type;
    (*datalength)--;
    return asn_build_length(data, datalength, length);
}

```

/* asn_parse_length - interprets the length of the current object. On exit, length contains the value of this length field. Returns a pointer to the first byte after this length field (aka: the start of the data field). Returns NULL on any error. */

```
u_char *
asn_parse_length(data, length)
    u_char *data; /* IN - pointer to start of length field */
    u_long *length; /* OUT - value of length field */
{
    register u_char lengthbyte = *data;

    if (lengthbyte & ASN_LONG_LEN){
        lengthbyte &= ~ASN_LONG_LEN; /* turn MSb off */
        if (lengthbyte == 0){
            ERROR("We don't support indefinite lengths");
            return NULL;
        }
        if (lengthbyte > sizeof(long)){
            ERROR("we can't support data lengths that long");
            return NULL;
        }
        bcopy((char *)data + 1, (char *)length, (int)lengthbyte);
        *length = ntohl(*length);
        *length >>= (8 * ((sizeof *length) - lengthbyte));
        return data + lengthbyte + 1;
    } else { /* short asnlength */
        *length = (long)lengthbyte;
        return data + 1;
    }
}

u_char *
asn_build_length(data, datalength, length)
    register u_char *data; /* IN - pointer to start of object */
    int *datalength; /* IN/OUT - number of valid bytes left in buffer */
    register int length; /* IN - length of object */
{
    u_char *start_data = data;

    /* no indefinite lengths sent */
    if (length < 0x80){
        *data++ = (u_char)length;
    } else if (length <= 0xFF){
        *data++ = (u_char)(0x01 | ASN_LONG_LEN);
        *data++ = (u_char)length;
    } else { /* 0xFF < length <= 0xFFFF */
        *data++ = (u_char)(0x02 | ASN_LONG_LEN);
        *data++ = (u_char)((length >> 8) & 0xFF);
        *data++ = (u_char)(length & 0xFF);
    }

    if (*datalength < (data - start_data)){
        ERROR("build_length");
        return NULL;
    }

    *datalength -= (data - start_data);
    return data;
}
```

```
}
```

/* asn_parse_objid - pulls an object identifier out of an ASN object identifier type. On entry, datalength is input as the number of valid bytes following "data". On exit, it is returned as the number of valid bytes following the beginning of the next object. "objid" is filled with the object identifier. Returns a pointer to the first byte past the end of this object (i.e. the start of the next object). Returns NULL on any error. */

```
u_char *
```

```
asn_parse_objid(data, datalength, type, objid, objidlength)
```

```
    u_char      *data; /* IN - pointer to start of object */
```

```
    int         *datalength; /* IN/OUT - number of valid bytes left in buffer */
```

```
    u_char      *type; /* OUT - ASN type of object */
```

```
    oid         *objid; /* IN/OUT - pointer to start of output buffer */
```

```
    int         *objidlength; /* IN/OUT - number of sub-id's in objid */
```

```
{
```

```
/* ASN.1 objid ::= 0x06 asnlength subidentifier {subidentifier}*
```

```
* subidentifier ::= {leadingbyte}* lastbyte
```

```
* leadingbyte ::= 1 7bitvalue
```

```
* lastbyte ::= 0 7bitvalue */
```

```
register u_char *bufp = data;
```

```
    register oid *oidp = objid + 1;
```

```
    register u_long subidentifier;
```

```
    register long length;
```

```
    u_long      asn_length;
```

```
    *type = *bufp++;
```

```
    bufp = asn_parse_length(bufp, &asn_length);
```

```
    if (bufp == NULL)
```

```
        return NULL;
```

```
    if (asn_length + (bufp - data) > *datalength){
```

```
        ERROR("overflow of message");
```

```
        return NULL;
```

```
    }
```

```
    *datalength -= (int)asn_length + (bufp - data);
```

```
    length = asn_length;
```

```
    (*objidlength)--; /* account for expansion of first byte */
```

```
    while (length > 0 && (*objidlength)-- > 0){
```

```
        subidentifier = 0;
```

```
        do { /* shift and add in low order 7 bits */
```

```
            subidentifier = (subidentifier << 7) + (*(u_char *)bufp & ~ASN_BIT8);
```

```
            length--;
```

```
        } while (*(u_char *)bufp++ & ASN_BIT8); /* last byte has high bit clear */
```

```
        if (subidentifier > (u_long)MAX_SUBID){
```

```
            ERROR("subidentifier too long");
```

```
            return NULL;
```

```
        }
```

```
        *oidp++ = (oid)subidentifier;
```

```
    }
```

/* The first two subidentifiers are encoded into the first component with the value $(X * 40) + Y$, where: X is the value of the first subidentifier. Y is the value of the second subidentifier. */

```
subidentifier = (u_long)objid[1];
```

```
objid[1] = (u_char)(subidentifier % 40);
```

```
objid[0] = (u_char)((subidentifier - objid[1]) / 40);
```

```
*objidlength = (int)(oidp - objid);
```

```

    return bufp;
}
/* asn_build_objid - Builds an ASN object identifier object containing the input string.
 * On entry, datalength is input as the number of valid bytes following "data". On exit, it is returned as the number
of valid bytes following the beginning of the next object.
 * Returns a pointer to the first byte past the end of this object (i.e. the start of the next object). Returns NULL on
any error. */
u_char *
asn_build_objid(data, datalength, type, objid, objidlength)
    register u_char *data; /* IN - pointer to start of object */
    int *datalength; /* IN/OUT - number of valid bytes left in buffer */
    u_char type; /* IN - ASN type of object */
    oid - *objid; /* IN - pointer to start of input buffer */
    int objidlength; /* IN - number of sub-id's in objid */
{
/* ASN.1 objid ::= 0x06 asnlength subidentifier {subidentifier}*
 * subidentifier ::= {leadingbyte}* lastbyte
 * leadingbyte ::= 1 7bitvalue
 * lastbyte ::= 0 7bitvalue */
u_char buf[MAX_OID_LEN];
    u_char *bp = buf;
    oid objbuf[MAX_OID_LEN];
    oid *op = objbuf;
    register int asnlength;
    register u_long subid, mask, testmask;
    register int bits, testbits;
bcopy((char *)objid, (char *)objbuf, objidlength * sizeof(oid));
    /* transform size in bytes to size in subid's */
    /* encode the first two components into the first subidentifier */
    op[1] = op[1] + (op[0] * 40);
    op++;
    objidlength--;
while(objidlength-- > 0){
    subid = *op++;
    mask = 0x7F; /* handle subid == 0 case */
    bits = 0;
    /* testmask *MUST* !!!! be of an unsigned type */
    for(testmask = 0x7F, testbits = 0; testmask != 0; testmask <=<= 7, testbits += 7){
        if (subid & testmask){ /* if any bits set */
            mask = testmask;
            bits = testbits;
        }
    }
    /* mask can't be zero here */
    for(;mask != 0x7F; mask >>= 7, bits -= 7){
        if (mask == 0x1E00000) /* fix a mask that got truncated above */
            mask = 0xFE00000;
        *bp++ = (u_char)(((subid & mask) >> bits) | ASN_BIT8);
    }
    *bp++ = (u_char)(subid & mask);
}
asnlength = bp - buf;
data = asn_build_header(data, datalength, type, asnlength);
if (data == NULL)
    return NULL;

```

```

    if (*datalength < asnlength)
        return NULL;
    bcopy((char *)buf, (char *)data, asnlength);
    *datalength -= asnlength;
    return data + asnlength;
}
/* asn_parse_null - Interprets an ASN null type. On entry, datalength is input as the number of valid bytes following
"data". On exit, it is returned as the number of valid bytes following the beginning of the next object. Returns a
pointer to the first byte past the end of this object (i.e. the start of the next object). Returns NULL on any error. */
u_char *
asn_parse_null(data, datalength, type)
    u_char      *data; /* IN - pointer to start of object */
    int         *datalength; /* IN/OUT - number of valid bytes left in buffer */
    u_char      *type; /* OUT - ASN type of object */
{
/* ASN.1 null ::= 0x05 0x00 */
    register u_char *bufp = data;
    u_long      asn_length;

    *type = *bufp++;
    bufp = asn_parse_length(bufp, &asn_length);
    if (bufp == NULL)
        return NULL;
    if (asn_length != 0){
        ERROR("Malformed NULL");
        return NULL;
    }
    *datalength -= (bufp - data);
    return bufp + asn_length;
}
/* asn_build_null - Builds an ASN null object. On entry, datalength is input as the number of valid bytes following
"data". On exit, it is returned as the number of valid bytes following the beginning of the next object. Returns a
pointer to the first byte past the end of this object (i.e. the start of the next object). Returns NULL on any error. */
u_char *
asn_build_null(data, datalength, type)
    u_char      *data; /* IN - pointer to start of object */
    int         *datalength; /* IN/OUT - number of valid bytes left in buffer */
    u_char      type; /* IN - ASN type of object */
{
/* ASN.1 null ::= 0x05 0x00 */
    return asn_build_header(data, datalength, type, 0);
}

```



```

/*****
 * Definitions for Abstract Syntax Notation One, ASN.1
 * As defined in ISO/IS 8824 and ISO/IS 8825   * asn1.h *
 *****/

#ifndef EIGHTBIT_SUBIDS
typedef u_long  oid;
#define MAX_SUBID 0xFFFFFFFF
#else
typedef u_char  oid;
#define MAX_SUBID 0xFF
#endif

#define MAX_OID_LEN 64 /* max subid's in an oid */

#define ASN_BOOLEAN (0x01)
#define ASN_INTEGER (0x02)
#define ASN_BIT_STR (0x03)
#define ASN_OCTET_STR (0x04)
#define ASN_NULL (0x05)
#define ASN_OBJECT_ID (0x06)
#define ASN_SEQUENCE (0x10)
#define ASN_SET (0x11)

#define ASN_UNIVERSAL (0x00)
#define ASN_APPLICATION (0x40)
#define ASN_CONTEXT (0x80)
#define ASN_PRIVATE (0xC0)

#define ASN_PRIMITIVE (0x00)
#define ASN_CONSTRUCTOR (0x20)

#define ASN_LONG_LEN (0x80)
#define ASN_EXTENSION_ID (0x1F)
#define ASN_BIT8 (0x80)

#define IS_CONSTRUCTOR(byte)((byte) & ASN_CONSTRUCTOR)
#define IS_EXTENSION_ID(byte) (((byte) & ASN_EXTENSION_ID) == ASN_EXTENSION_ID)

u_char *asn_parse_int();
u_char *asn_build_int();
u_char *asn_parse_string();
u_char *asn_build_string();
u_char *asn_parse_header();
u_char *asn_build_header();
u_char *asn_parse_length();
u_char *asn_build_length();
u_char *asn_parse_objid();
u_char *asn_build_objid();
u_char *asn_parse_null();
u_char *asn_build_null();
/*****

```

* Definitions for the Simple Network Management Protocol (RFC 1067). *snmp.h*

```
#define SNMP_PORT      161
#define SNMP_TRAP_PORT 162

#define SNMP_MAX_LEN   484

#define SNMP_VERSION_1 0

#define GET_REQ_MSG    (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x0)
#define GETNEXT_REQ_MSG (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x1)
#define GET_RSP_MSG    (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x2)
#define SET_REQ_MSG    (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x3)
#define TRP_REQ_MSG    (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x4)
```

```
#define SNMP_ERR_NOERROR (0x0)
#define SNMP_ERR_TOOBIG  (0x1)
#define SNMP_ERR_NOSUCHNAME (0x2)
#define SNMP_ERR_BADVALUE (0x3)
#define SNMP_ERR_READONLY (0x4)
#define SNMP_ERR_GENERR   (0x5)
```

```
#define SNMP_TRAP_COLDSTART      (0x0)
#define SNMP_TRAP_WARMSTART      (0x1)
#define SNMP_TRAP_LINKDOWN      (0x2)
#define SNMP_TRAP_LINKUP        (0x3)
#define SNMP_TRAP_AUTHFAIL      (0x4)
#define SNMP_TRAP_EGPNEIGHBORLOSS (0x5)
#define SNMP_TRAP_ENTERPRISESPECIFIC (0x6)
```

* snmp_api.h - API for access to snmp.

```
typedef struct sockaddr_in ipaddr;
```

```
struct snmp_session {
    u_char *community; /* community for outgoing requests. */
    int community_len; /* Length of community name. */
    int retries; /* Number of retries before timeout. */
    long timeout; /* Number of uS until first timeout, then exponential backoff */
    char *peername; /* Domain name or dotted IP address of default peer */
    u_short remote_port; /* UDP port number of peer. */
    u_short local_port; /* My UDP port number, 0 for default, picked randomly */
    /* Authentication function or NULL if null authentication is used */
    u_char *(*authenticator)();
    int (*callback)(); /* Function to interpret incoming data */
    /* Pointer to data that the callback function may consider important */
    void *callback_magic;
};
```

/* Set fields in session and pdu to the following to get a default or unconfigured value. */

```
#define SNMP_DEFAULT_COMMUNITY_LEN 0 /* to get a default community name */
```

```
#define SNMP_DEFAULT_RETRIES -1
```

```

#define SNMP_DEFAULT_TIMEOUT      -1
#define SNMP_DEFAULT_REMPORT      0
#define SNMP_DEFAULT_REQID        0
#define SNMP_DEFAULT_ERRSTAT      -1
#define SNMP_DEFAULT_ERRINDEX     -1
#define SNMP_DEFAULT_ADDRESS      0
#define SNMP_DEFAULT_PEERNAME     NULL
#define SNMP_DEFAULT_ENTERPRISE_LENGTH  0
#define SNMP_DEFAULT_TIME         0

extern int snmp_errno;
/* Error return values */
#define SNMPERR_GENERR            -1
#define SNMPERR_BAD_LOCPORT      -2 /* local port was already in use */
#define SNMPERR_BAD_ADDRESS      -3
#define SNMPERR_BAD_SESSION      -4
#define SNMPERR_TOO_LONG        -5

struct snmp_pdu {
    ipaddr address;      /* Address of peer */

    int    command;     /* Type of this PDU */

    u_long reqid; /* Request id */
    u_long errstat; /* Error status */
    u_long errindex; /* Error index */

    /* Trap information */
    oid    *enterprise; /* System OID */
    int    enterprise_length;
    ipaddr agent_addr; /* address of object generating trap */
    int    trap_type; /* trap type */
    int    specific_type; /* specific type */
    u_long time; /* Uptime */

    struct variable_list *variables;
};

struct variable_list {
    struct variable_list *next_variable; /* NULL for last variable */
    oid    *name; /* Object identifier of variable */
    int    name_length; /* number of subid's in name */
    u_char type; /* ASN type of variable */
    union { /* value of variable */
        long    *integer;
        u_char  *string;
        oid     *objid;
    } val;
    int    val_len;
};

/* struct snmp_session *snmp_open(session)
 *      struct snmp_session *session;

```

```

* Sets up the session with the snmp_session information provided by the user. Then opens and binds the
necessary UDP port. A handle to the created session is returned (this is different than the pointer passed to
snmp_open()). On any error, NULL is returned and snmp_errno is set to the appropriate error code. */
struct snmp_session *snmp_open();

/* int snmp_close(session)
* struct snmp_session *session;
* Close the input session. Frees all data allocated for the session, dequeues any pending requests, and
closes any sockets allocated for the session. Returns 0 on error, 1 otherwise. */
int snmp_close();

/* int snmp_send(session, pdu)
* struct snmp_session *session;
* struct snmp_pdu *pdu;
* Sends the input pdu on the session after calling snmp_build to create a serialized packet. If necessary, set
some of the pdu data from the session defaults. Add a request corresponding to this pdu to the list of
outstanding requests on this session, then send the pdu. Returns the request id of the generated packet if
applicable, otherwise 1. On any error, 0 is returned. The pdu is freed by snmp_send() unless a failure
occured. */
int snmp_send();
/* void snmp_read(fdset)
* fd_set *fdset;
* Checks to see if any of the fd's set in the fdset belong to snmp. Each socket with it's fd set has a packet
read from it and snmp_parse is called on the packet received. The resulting pdu is passed to the callback
routine for that session. If the callback routine returns successfully, the pdu and it's request are deleted. */
void snmp_read();
/* void
* snmp_free_pdu(pdu)
* struct snmp_pdu *pdu;
* Frees the pdu and any malloc'd data associated with it. */
void snmp_free_pdu();

/* int snmp_select_info(numfds, fdset, timeout, block)
* int *numfds;
* fd_set *fdset;
* struct timeval *timeout;
* int *block;
* Returns info about what snmp requires from a select statement. numfds is the number of fds in the list that
are significant. All file descriptors opened for SNMP are OR'd into the fdset. If activity occurs on any of
these file descriptors, snmp_read should be called with that file descriptor set.
* The timeout is the latest time that SNMP can wait for a timeout. The select should be done with the
minimum time between timeout and any other timeouts necessary. This should be checked upon each
invocation of select. If a timeout is received, snmp_timeout should be called to check if the timeout was
for SNMP. (snmp_timeout is idempotent)
* Block is 1 if the select is requested to block indefinitely, rather than time out. If block is input as 1, the
timeout value will be treated as undefined, but it must be available for setting in snmp_select_info. On
return, if block is true, the value of timeout will be undefined.
* snmp_select_info returns the number of open sockets. (i.e. The number of sessions open) */
int snmp_select_info();

/* void snmp_timeout();
* snmp_timeout should be called whenever the timeout from snmp_select_info expires, but it is
idempotent, so snmp_timeout can be polled (probably a cpu expensive proposition). snmp_timeout checks
to see if any of the sessions have an outstanding request that has timed out. If it finds one (or more), and

```

that pdu has more retries available, a new packet is formed from the pdu and is resent. If there are no more retries available, the callback for the session is used to alert the user of the timeout.

```
*/
void snmp_timeout();

/* This routine must be supplied by the application:
* u_char *authenticator(pdu, length, community, community_len)
* u_char *pdu;          The rest of the PDU to be authenticated
* int *length;          The length of the PDU (updated by the authenticator)
* u_char *community;   The community name to authenticate under.
* int community_len    The length of the community name.
* Returns the authenticated pdu, or NULL if authentication failed. If null authentication is used, the
authenticator in snmp_session can be set to NULL(0). */
/* This routine must be supplied by the application:
* int callback(operation, session, reqid, pdu, magic)
* int operation;
* struct snmp_session *session;  The session authenticated under.
* int reqid;                    The request id of this pdu (0 for TRAP)
* struct snmp_pdu *pdu;        The pdu information.
* void *magic                   A link to the data for this routine.
* Returns 1 if request was successful, 0 if it should be kept pending. Any data in the pdu must be copied
because it will be freed elsewhere. Operations are defined below: */
#define RECEIVED_MESSAGE 1
#define TIMED_OUT        2
extern int snmp_dump_packet;
/*****
* snmp_client.h
*****/
struct synch_state {
    int waiting;
    int status;
/* status codes */
#define STAT_SUCCESS 0
#define STAT_ERROR 1
#define STAT_TIMEOUT 2
    int reqid;
    struct snmp_pdu *pdu;
};

extern struct synch_state snmp_synch_state;

struct snmp_pdu *snmp_pdu_create();
struct snmp_pdu *snmp_fix_pdu();
char *snmp_errstring();
/*****
* Definitions for SNMP (RFC 1067) implementation. *snmp_impl.h
*****/

#if (defined vax) || (defined mips))
/*
* This is a fairly bogus thing to do, but there seems to be no better way for
* compilers that don't understand void pointers.
*/
#define void char
```

```

#endif
/* Error codes: */
/* These must not clash with SNMP error codes (all positive). */
#define PARSE_ERROR -1
#define BUILD_ERROR -2

#define SID_MAX_LEN 64
#define MAX_NAME_LEN 64 /* number of subid's in a objid */

#ifndef NULL
#define NULL 0
#endif

#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

#define READ 1
#define WRITE 0

#define RONLY 0xA000 /* read access for everyone */
#define RWRITE 0xA001 /* add write access for community private */
#define NOACCESS 0x0000 /* no access for anybody */

#define INTEGER ASN_INTEGER
#define STRING ASN_OCTET_STR
#define OBJID ASN_OBJECT_ID
#define NULLOBJ ASN_NULL

/* defined types (from the SMI, RFC 1065) */
#define IPADDRESS (ASN_APPLICATION | 0)
#define COUNTER (ASN_APPLICATION | 1)
#define GAUGE (ASN_APPLICATION | 2)
#define TIMETICKS (ASN_APPLICATION | 3)
#define OPAQUE (ASN_APPLICATION | 4)

#ifndef DEBUG
#define ERROR(string) printf("%s(%d): %s", __FILE__, __LINE__, string);
#else
#define ERROR(string)
#endif

/* from snmp.c */
extern u_char sid[]; /* size SID_MAX_LEN */

u_char *snmp_parse_var_op();
u_char *snmp_build_var_op();

u_char *snmp_auth_parse();
u_char *snmp_auth_build();

```