CHAPTER III

PROGRAM ALGORITHM & FLOW CHARTS
3.1 A. SNMP and UDP:

SNMP uses the *User Datagram Protocol* (UDP) as the transport protocol for passing data between managers and agent's. UDP was chosen over the *Transmission Control Protocol* (TCP) because it is connectionless; that is, no end-to-end connection is made between the agent and the NMS when *datagrams* (packets) are sent back and forth. This aspect of UDP makes it unreliable, since there is no acknowledgment of lost datagrams at the protocol level. It's up to the SNMP application to determine if datagrams are lost and retransmit them if it so desires. This is typically accomplished with a simple timeout. The NMS sends a UDP request to an agent and waits for a response. The length of time the NMS waits depends on how it's configured. If the timeout is reached and the NMS has not heard back from the agent, it assumes the packet was lost and retransmits the request. The number of times the NMS retransmits packets is also configurable.

At least as far as regular information requests are concerned, the unreliable nature of UDP isn't a real problem. At worst, the management station issues a request and never receives a response. For traps, the situation is somewhat different. If an agent sends a trap and the trap never arrives, the NMS has no way of knowing that it was ever sent. The agent doesn't even know that it needs to resend the trap, because the NMS is not required to send a response back to the agent acknowledging receipt of the trap.

The upside to the unreliable nature of UDP is that it requires low overhead, so the impact on your network's performance is reduced. SNMP has been implemented over TCP, but this is more for special-case situations in which someone is developing
an agent for a proprietary piece of equipment. In a heavily congested and managed network, SNMP over TCP is a bad idea. It's also worth realizing that TCP isn't magic, and that SNMP is designed for working with networks that are in trouble—if your network never failed, you wouldn't need to monitor it. When a network is failing, a protocol that tries to get the data through but gives up if it can't is almost certainly a better design choice than a protocol that will flood the network with retransmissions in its attempt to achieve reliability.

SNMP uses the UDP port 161 for sending and receiving requests, and port 162 for receiving traps from managed devices. Every device that implements SNMP must use these port numbers as the defaults, but some vendors allow you to change the default ports in the agent's configuration. If these defaults are changed, the NMS must be made aware of the changes so it can query the device on the correct ports.

![TCP/IP communication model and SNMP](image)

**Figure 3-1. TCP/IP communication model and SNMP**
Figure 3-1 shows the TCP/IP protocol suite, which is the basis for all TCP/IP communication. Today, any device that wishes to communicate on the Internet (e.g., Windows NT systems, UNIX servers, Cisco routers, etc.) must use this protocol suite.

This model is often referred to as a protocol stack, since each layer uses the information from the layer directly below it and provides a service to the layer directly above it. When either an NMS or an agent wishes to perform an SNMP function (e.g., a request or trap), the following events occur in the protocol stack:

**Application:**

First, the actual SNMP application (NMS or agent) decides what it's going to do. For example, it can send an SNMP request to an agent, send a response to an SNMP request (this would be sent from the agent), or send a trap to an NMS. The application layer provides services to an end user, such as an operator requesting status information for a port on an Ethernet switch.

**UDP:**

The next layer, UDP, allows two hosts to communicate with one another. The UDP header contains, among other things, the destination port of the device to which it's sending the request or trap. The destination port will either be 161 (query) or 162 (trap).

**IP:**

The IP layer tries to deliver the SNMP packet to its intended destination, as specified by its IP address.
Medium Access Control (MAC):

The final event that must occur for an SNMP packet to reach its destination is for it to be handed off to the physical network, where it can be routed to its final destination. The MAC layer is comprised of the actual hardware and device drivers that put your data onto a physical piece of wire, such as an Ethernet card. The MAC layer also is responsible for receiving packets from the physical network and sending them back up the protocol stack so they can be processed by the application layer (SNMP, in this case).

This interaction between SNMP applications and the network is not unlike that between two pen pals. Both have messages that need to be sent back and forth to one another. Let's say you decide to write your pen pal a letter asking if she would like to visit you over the summer. By deciding to send the invitation, you've acted as the SNMP application. Filling out the envelope with your pen pal's address is equivalent to the function of the UDP layer, which records the packet's destination port in the UDP header; in this case it's your pen pal's address. Placing a stamp on the envelope and putting it in the mailbox for the mailman to pick up is equivalent to the IP layer's function. The final act occurs when the mailman comes to your house and picks up the letter. From here the letter will be routed to its final destination, your pen pal's mailbox. The MAC layer of a computer network is equivalent to the mail trucks and airplanes that carry your letter on its way. When your pen pal receives the letter, she will go through the same process to send you a reply.
3.2A SNMP Communities:

SNMPv1 and SNMPv2 use the notion of communities to establish trust between managers and agents. An agent is configured with three community names: read-only, read-write, and trap. The community names are essentially passwords; there's no real difference between a community string and the password you use to access your account on the computer. The three community strings control different kinds of activities. As its name implies, the read-only community string lets you read data values, but doesn't let you modify the data. For example, it allows you to read the number of packets that have been transferred through the ports on your router, but doesn't let you reset the counters. The read-write community is allowed to read and modify data values; with the read-write community string, you can read the counters, reset their values, and even reset the interfaces or do other things that change the router's configuration. Finally, the trap community string allows you to receive traps (asynchronous notifications) from the agent.

Most vendors ship their equipment with default community strings, typically public for the read-only community and private for the read-write community. It's important to change these defaults before your device goes live on the network. (You may get tired of hearing this because we say it many times, but it's absolutely essential.) When setting up an SNMP agent, you will want to configure its trap destination, which is the address to which it will send any traps it generates. In addition, since SNMP community strings are sent in clear text, you can configure an agent to send an SNMP authentication-failure trap when someone attempts to query your device with an incorrect community string. Among other things, authentication-
failure traps can be very useful in determining when an intruder might be trying to gain access to your network.

Because community strings are essentially passwords, you should use the same rules for selecting them as you use for UNIX or NT user passwords: no dictionary words, spouse names, etc. An alphanumeric string with mixed upper- and lowercase letters is generally a good idea. As mentioned earlier, the problem with SNMP's authentication is that community strings are sent in plain text, which makes it easy for people to intercept them and use them against you. SNMPv3 addresses this by allowing, among other things, secure authentication and communication between SNMP devices.

There are ways to reduce your risk of attack. IP firewalls or filters minimize the chance that someone can harm any managed device on your network by attacking it through SNMP. You can configure your firewall to allow UDP traffic from only a list of known hosts. For example, you can allow UDP traffic on port 161 (SNMP requests) into your network only if it comes from one of your network-management stations. The same goes for traps; you can configure your router so it allows UDP traffic on port 162 to your NMS only if it originates from one of the hosts you are monitoring. Firewalls aren't 100% effective, but simple precautions such as these do a lot to reduce your risk.

3.3A The Structure of Management Information:

So far, we have used the term "management information" to refer to the operational parameters of SNMP-capable devices. However, we've said very little about what management information actually contains or how it is represented. The
first step toward understanding what kind of information a device can provide is to understand how this data itself is represented within the context of SNMP. The Structure of Management Information Version 1 (SMIv1, RFC 1155) does exactly that: it defines precisely how managed objects are named and specifies their associated data types.

The definition of managed objects can be broken down into three attributes:

3.3.1 Name:

The name, or object identifier (OID), uniquely defines a managed object. Names commonly appear in two forms: numeric and "human readable." In either case, the names are long and inconvenient. In SNMP applications, a lot of work goes into helping you navigate through the namespace conveniently.

3.3.2 Type and syntax:

A managed object's datatype is defined using a subset of Abstract Syntax Notation One (ASN.1). ASN.1 is a way of specifying how data is represented and transmitted between managers and agents, within the context of SNMP. The nice thing about ASN.1 is that the notation is machine-independent. This means that a PC running Windows NT can communicate with a Sun SPARC machine and not have to worry about things such as byte ordering.

3.3.3 Encoding:

A single instance of a managed object is encoded into a string of octets using the Basic Encoding Rules (BER). BER defines how the objects are encoded and decoded so they can be transmitted over a transport medium such as Ethernet.
3.3.4 Naming OIDs:

Managed objects are organized into a tree-like hierarchy. This structure is the basis for SNMP's naming scheme. An object ID is made up of a series of integers based on the nodes in the tree, separated by dots (.). Although there's a human-readable form that's more friendly than a string of numbers, this form is nothing more than a series of names separated by dots, each of which represents a node of the tree. So you can use the numbers themselves, or you can use a sequence of names that represent the numbers.

In the object tree, the node at the top of the tree is called the root, anything with children is called a subtree, and anything without children is called a leaf node. For example, Figure 3-2's root, the starting point for the tree, is called "Root-Node." Its subtree is made up of ccitt(0), iso(1), and joint(2). In this illustration, iso(1) is the only node that contains a subtree; the other two nodes are both leaf nodes. Each
managed object has a numerical OID and an associated textual name. The dotted-decimal notation is how a managed object is represented internally within an agent; the textual name, like an IP domain name, saves humans from having to remember long, tedious strings of integers.

The directory branch currently is not used. The management branch, or mgmt, defines a standard set of Internet management objects. The experimental branch is reserved for testing and research purposes. Objects under the private branch are defined unilaterally, which means that individuals and organizations are responsible for defining the objects under this branch. Here is the definition of the internet subtree, as well as all four of its subtrees:

\[
\begin{align*}
\text{internet} & \quad \text{OBJECT IDENTIFIER ::= \{ iso org(3) dod(6) 1 \}} \\
\text{directory} & \quad \text{OBJECT IDENTIFIER ::= \{ internet 1 \}} \\
\text{mgmt} & \quad \text{OBJECT IDENTIFIER ::= \{ internet 2 \}} \\
\text{experimental} & \quad \text{OBJECT IDENTIFIER ::= \{ internet 3 \}}
\end{align*}
\]
<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>A 32-bit number often used to specify enumerated types within the context of a single managed object. For example, the operational status of a router interface can be <em>up</em>, <em>down</em>, or <em>testing</em>. With enumerated types, 1 would represent up, 2 down, and 3 testing. The value zero (0) must not be used as an enumerated type, according to RFC 1155.</td>
</tr>
<tr>
<td>OCTET STRING</td>
<td>A string of zero or more octets (more commonly known as bytes) generally used to represent text strings, but also sometimes used to represent physical addresses.</td>
</tr>
<tr>
<td>Counter</td>
<td>A 32-bit number with minimum value 0 and maximum value $2^{32} - 1$ (4,294,967,295). When the maximum value is reached, it wraps back to zero and starts over. It's primarily used to track information such as the number of octets sent and received on an interface or the number of errors and discards seen on an interface. A Counter is monotonically increasing, in that its values should never decrease during normal operation. When an agent is rebooted, all Counter values should be set to zero. Deltas are used to determine if anything useful can be said for successive queries of Counter values. A delta is computed by querying a Counter at least twice in a row, and taking the difference between the query results over some time interval.</td>
</tr>
<tr>
<td><strong>OBJECT IDENTIFIER</strong></td>
<td>A dotted-decimal string that represents a managed object within the object tree. For example, <code>1.3.6.1.4.1.9</code> represents Cisco System's private enterprise OID.</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>NULL</strong></td>
<td>Not currently used in SNMP.</td>
</tr>
<tr>
<td><strong>SEQUENCE</strong></td>
<td>Defines lists that contain zero or more other ASN.1 datatypes.</td>
</tr>
<tr>
<td><strong>SEQUENCE OF</strong></td>
<td>Defines a managed object that is made up of a SEQUENCE of ASN.1 types.</td>
</tr>
<tr>
<td><strong>IP Address</strong></td>
<td>Represents a 32-bit IPv4 address. Neither SMIv1 nor SMIv2 discusses 128-bit IPv6 addresses; this problem will be addressed by the IETF's SMI Next Generation (SMING) working group (see</td>
</tr>
<tr>
<td><strong>Network Address</strong></td>
<td>Same as the IP Address type, but can represent different network address types.</td>
</tr>
<tr>
<td><strong>Gauge</strong></td>
<td>A 32-bit number with minimum value 0 and maximum value <code>2^{32} - 1</code> (4,294,967,295). Unlike a Counter, a Gauge can increase and decrease at will, but it can never exceed its maximum value. The interface speed on a router is measured with a Gauge.</td>
</tr>
<tr>
<td><strong>TimeTicks</strong></td>
<td>A 32-bit number with minimum value 0 and maximum value <code>2^{32} - 1</code> (4,294,967,295). TimeTicks measures time in hundredths of a second. Uptime on a device is measured using this datatype.</td>
</tr>
<tr>
<td><strong>Opaque</strong></td>
<td>Allows any other ASN.1 encoding to be stuffed into an OCTET STRING.</td>
</tr>
</tbody>
</table>
internet as the OID 1.3.6.1, which is defined as a subtree of iso.org.dod, or 1.3.6 (the ::= is a definition operator). The last four declarations are similar, but they define the other branches that belong to internet. For the directory branch, the notation { internet 1 } tells us that it is part of the internet subtree, and that its OID is 1.3.6.1.1. The OID for mgmt is 1.3.6.1.2, and so on.

There is currently one branch under the private subtree. It's used to give hardware and software vendors the ability to define their own private objects for any type of hardware or software they want managed by SNMP. Its SMI definition is:

enterprises OBJECT IDENTIFIER ::= { private 1 }

The Internet Assigned Numbers Authority (IANA) currently manages all the private enterprise number assignments for individuals, institutions, organizations, companies, etc.

3.3.5 Defining OIDs

The SYNTAX attribute provides for definitions of managed objects through a subset of ASN.1. SMIv1 defines several datatypes that are paramount to the management of networks and network devices. It's important to keep in mind that these datatypes are simply a way to define what kind of information a managed object can hold. The types we'll be discussing are similar to those that you'd find in a computer programming language like C. Table 3-1 lists the supported datatypes for SMIv1.

The goal of all these object types is to define managed objects. MIB is a logical grouping of managed objects as they pertain to a specific management task,
vendor, etc. The MIB can be thought of as a specification that defines the managed objects a vendor or device supports. Both devices have different characteristics that require different management capabilities. Vendor-specific MIBs typically are distributed as human-readable text files that can be inspected (or even modified) with a standard text editor such as vi.

It's important to know how to read and understand MIB files. The following example is a stripped-down version of MIB-II (anything preceded by -- is a comment):

```
. The MIB imports the following items from RFC1155-

- Network Address
- IPAddress
- Counter
- Gauge
- TimeTicks

It also imports OBJECT-TYPE from RFC 1212, the *Concise MIB Definition*, which defines how MIB files are written. Each group of items imported using the IMPORTS clause uses a FROM clause to define the MIB file from which the objects are taken.

The OIDs that will be used throughout the remainder of the MIB follow the linkage section. This group of lines sets up the top level of the mib-2 subtree. mib-2 is defined as mgmt followed by .1. We saw earlier that mgmt was equivalent to 1.3.6.1.2. Therefore, mib-2 is equivalent to 1.3.6.1.2.1. Likewise, the interfaces group under mib-2 is defined as { mib-2 2 }, or 1.3.6.1.2.1.2.
After the OIDs are defined, we get to the actual object definitions. Every object definition has the following format:

<name> OBJECT-TYPE
SYNTAX <datatype>
ACCESS <either read-only, read-write, write-only, or not-accessible>
STATUS <either mandatory, optional, or obsolete>
DESCRIPTION
"Textual description describing this particular managed object."
::= { <Unique OID that defines this object> }

The first managed object in our subset of the MIB-II definition is ifTable, which represents a table of network interfaces on a managed device (note that object names are defined using mixed case, with the first letter in lowercase). Here is its definition using ASN.1 notation:

ifTable OBJECT-TYPE
SYNTAX SEQUENCE OF IfEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A list of interface entries. The number of entries is given by the value of ifNumber."
::= { interfaces 2 }

The SYNTAX of ifTable is SEQUENCE OF IfEntry. This means that ifTable is a table containing the columns defined in IfEntry. The object is not-accessible,
which means that there is no way to query an agent for this object's value. Its status is mandatory, which means an agent must implement this object in order to comply with the MIB-II specification. The DESCRIPTION describes what exactly this object is. The unique OID is 1.3.6.1.2.1.2.2, or iso.org.dod.internet.mgmt.interfaces.2.

Note that the name of the sequence (IfEntry) is mixed-case, but the first letter is capitalized, unlike the object definition for ifTable. This is how a sequence name is defined. A sequence is simply a list of columnar objects and their SMI datatypes, which defines a conceptual table. In this case, we expect to find variables defined by ifIndex, ifDescr, ifType, etc. This table can contain any number of rows; it's up to the agent to manage the rows that reside in the table. It is possible for an NMS to add rows to a table. Now that we have IfEntry to specify what we'll find in any row of the table, we can look back to the definition of IfEntry (the actual rows of the table) itself:

ifEntry OBJECT-TYPE
   SYNTAX IfEntry
   ACCESS not-accessible
   STATUS mandatory
   DESCRIPTION
   "An interface entry containing objects at the subnetwork layer and below for a particular interface."
   INDEX  { ifIndex }
   ::= { ifTable 1 }

ifEntry defines a particular row in the ifTable. Its definition is almost identical to that of ifTable, except we have introduced a new clause, INDEX. The index is a unique key used to define a single row in the ifTable. It's up to the agent to make sure the
index is unique within the context of the table. If a router has six interfaces, *ifTable* will have six rows in it. *ifEntry*'s OID is 1.3.6.1.2.1.2.2.1, or *iso.org.dod.internet.mgmt.interfaces.ifTable.ifEntry*. The index for *ifEntry* is *ifIndex*, which is defined as:

```
ifIndex OBJECT-TYPE
   SYNTAX INTEGER
   ACCESS read-only
   STATUS mandatory
   DESCRIPTION
      "A unique value for each interface. Its value ranges between
       1 and the value of ifNumber. The value for each interface
       must remain constant at least from one reinitialization of the
       entity's network-management system to the next reinitialization."
   ::= { ifEntry 1 }
```

The *ifIndex* object is read-only, which means we can see its value, but we cannot change it. The final object our MIB defines is *ifDescr*, which is a textual description for the interface represented by that particular row in the *ifTable*. Our MIB example ends with the END clause, which marks the end of the MIB. In the actual MIB-II files, each object listed in the *ifEntry* sequence has its own object definition. In this version of the MIB we list only two of them, in the interest of conserving space.
3.4A MIB-II:

MIB-II is a very important management group, because every device that supports SNMP must also support MIB-II. Therefore, we will use objects from MIB-II in our examples throughout this book. We won't go into detail about every object in the MIB; we'll simply define the subtrees. The section of RFC1213-MIB that defines the base OIDs for the mib-2 subtree looks like this:

```
mib-2     OBJECT IDENTIFIER ::= { mgmt 1 }
system    OBJECT IDENTIFIER ::= { mib-2 1 }
interfaces OBJECT IDENTIFIER ::= { mib-2 2 }
at        OBJECT IDENTIFIER ::= { mib-2 3 }
ip        OBJECT IDENTIFIER ::= { mib-2 4 }
icmp      OBJECT IDENTIFIER ::= { mib-2 5 }
tcp       OBJECT IDENTIFIER ::= { mib-2 6 }
udp       OBJECT IDENTIFIER ::= { mib-2 7 }
egp        OBJECT IDENTIFIER ::= { mib-2 8 }
transmission OBJECT IDENTIFIER ::= { mib-2 10 }
snmp      OBJECT IDENTIFIER ::= { mib-2 11 }
```

`mib-2` is defined as iso.org.dod.internet.mgmt.1, or 1.3.6.1.2.1. From here, we can see that the `system` group is `mib-2 1`, or 1.3.6.1.2.1.1, and so on. Figure 3-3 shows the MIB-II subtree of the `mgmt` branch.
Figure 3.-3. MIB-II subtree
<table>
<thead>
<tr>
<th>Subtree Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System</strong></td>
<td>1.3.6.1.2.1.1</td>
<td>Defines a list of objects that pertain to system operation, such as the system uptime, system contact, and system name.</td>
</tr>
<tr>
<td><strong>interfaces</strong></td>
<td>1.3.6.1.2.1.2</td>
<td>Keeps track of the status of each interface on a managed entity. The interfaces group monitors which interfaces are up or down and tracks such things as octets sent and received, errors and discards, etc.</td>
</tr>
<tr>
<td><strong>at</strong></td>
<td>1.3.6.1.2.1.3</td>
<td>The address translation (at) group is deprecated and is provided only for backward compatibility. It will probably be dropped from MIB-III.</td>
</tr>
<tr>
<td><strong>ip</strong></td>
<td>1.3.6.1.2.1.4</td>
<td>Keeps track of many aspects of IP, including IP routing.</td>
</tr>
<tr>
<td><strong>icmp</strong></td>
<td>1.3.6.1.2.1.5</td>
<td>Tracks things such as ICMP errors, discards, etc.</td>
</tr>
<tr>
<td><strong>tcp</strong></td>
<td>1.3.6.1.2.1.6</td>
<td>Tracks, among other things, the state of the TCP connection (e.g., closed, listen, synSent, etc.).</td>
</tr>
<tr>
<td><strong>udp</strong></td>
<td>1.3.6.1.2.1.7</td>
<td>Tracks UDP statistics, datagrams in and out, etc.</td>
</tr>
<tr>
<td><strong>egp</strong></td>
<td>1.3.6.1.2.1.8</td>
<td>Tracks various statistics about EGP and keeps an EGP neighbor table.</td>
</tr>
<tr>
<td><strong>transmission</strong></td>
<td>1.3.6.1.2.1.10</td>
<td>There are currently no objects defined for this group, but other mediaspecific MIBs are defined using this subtree.</td>
</tr>
<tr>
<td><strong>snmp</strong></td>
<td>1.3.6.1.2.1.11</td>
<td>Measures the performance of the underlying SNMP implementation on the managed entity and tracks things such as the number of SNMP packets sent and received.</td>
</tr>
</tbody>
</table>
3.5A SNMP Operations:

We've discussed how SNMP organizes information, but we've left out how we actually go about gathering management information. Now, we're going to take a look under the hood to see how SNMP does its thing.

The Protocol Data Unit (PDU) is the message format that managers and agents use to send and receive information. There is a standard PDU format for each of the following SNMP operations:

- get
- get-next
- set
- get-response
- trap

3.5.1 The get Operation:

The get request is initiated by the NMS, which sends the request to the agent. The agent receives the request and processes it to best of its ability. Some devices that are under heavy load, such as routers, may not be able to respond to the request and will have to drop it. If the agent is successful in gathering the requested information, it sends a get-response back to the NMS, where it is processed. This process is illustrated in Figure 3-4.
How did the agent know what the NMS was looking for? One of the items in the get request is a variable binding. A variable binding, or varbind, is a list of MIB objects that allows a request's recipient to see what the originator wants to know. Variable bindings can be thought of as $OID=value$ pairs that make it easy for the originator (the NMS, in this case) to pick out the information it needs when the recipient fills the request and sends back a response. Let's look at this operation in action:

```bash
$ snmpget cisco.ora.com public .1.3.6.1.2.1.1.6.0
system.sysLocation.0 = ""
```

Several things are going on in this example. First, we're running a command on a UNIX host. The command is called snmpget. Its main job is to facilitate the gathering of management data using a get request. We've given it three arguments on the command line: the name of the device we would like to query (cisco.ora.com), the read-only community string (public), and the OID we would like gathered (.1.3.6.1.2.1.1.6.0). If we look back at Table 2-2 we see that .1.3.6.1.2.1.1 is the system group, but there are two more integers at the end of the OID: .6 and .0. The .6 is
actually the MIB variable that we wish to query; its human-readable name is
sysLocation. In this case, we would like to see what the system location is set to
on the Cisco router. As you can see by the response (system.sysLocation.0 = ""), the
system location on this router currently is not set to anything. Also note that the
response from snmpget is in variable binding format, OID=value.

There is one more thing to look at. Why does the MIB variable have a .0
tacked on the end? In SNMP, MIB objects are defined by the convention x.y, where x
is the actual OID of the managed object (in our example, 1.3.6.1.2.1.1.6) and y is the
instance identifier. For scalar objects (that is, objects that aren't defined as a row in a
table) y is always 0. In the case of a table, the instance identifier lets you select a
specific row of the table; 1 is the first row, 2 is the second row, etc. For example,
consider the ifTable object we looked at earlier in this chapter. When looking up
values in the ifTable, we would use a nonzero instance identifier to select a particular
row in the table (in this case, a particular network interface).

The get command is useful for retrieving a single MIB object at a time. Trying
to manage anything in this manner can be a waste of time, though. This is where the
get-next command comes in. It allows you to retrieve more than one object from a
device, over a period of time.

3.5.2 The get-next Operation

The get-next operation lets you issue a sequence of commands to retrieve a
group of values from a MIB. In other words, for each MIB object we want to retrieve,
a separate get-next request and get-response are generated. The get-next command
traverses a subtree in lexicographic order. Since an OID is a sequence of integers, it's
easy for an agent to start at the root of its SMI object tree and work its way down until it finds the OID it is looking for. When the NMS receives a response from the agent for the get-next command it just issued, it issues another get-next command. It keeps doing this until the agent returns an error, signifying that the end of the MIB has been reached and there are no more objects left to get.

The get-next sequence returns seven MIB variables. Each of these objects is part of the \textit{system} group as it's defined in RFC 1213. We see a system object ID, the amount of time the system has been up, the contact person, etc.

Given that you've just looked up some object, how does get-next figure out which object to look up next? get-next is based on the concept of the lexicographic ordering of the MIB's object tree. This order is made much simpler because every node in the tree is assigned a number. To understand what this means, let's start at the root of the tree and walk down to the \textit{system} node.

To get to the \textit{system} group (OID 1.3.6.1.2.1.1), we start at the root of the object tree and work our way down. Figure 2-5 shows the logical progression from the root of the tree all the way to the \textit{system} group. At each node in the tree, we visit the lowest-numbered branch. Thus, when we're at the root node, we start by visiting \textit{ccitt}. This node has no nodes underneath it, so we move to the \textit{iso} node. Since \textit{iso} does have a child we move to that node, \textit{org}. The process continues until we reach the \textit{system} node. Since each branch is made up of ascending integers (\textit{ccitt}(0) \textit{iso}(1) \textit{join}(2), for example), the agent has no problem traversing this tree structure all the way down to the \textit{system}(1) group. If we were to continue this walk, we'd proceed to \textit{system.1} (\textit{system.sysLocation}), \textit{system.2}, and the other objects in the \textit{system} group. Next, we'd go to \textit{interfaces}(2), and so on.
3.5.3 The set Operation

The *set* command is used to change the value of a managed object or to create a new row in a table. Objects that are defined in the MIB as read-write or write-only can be altered or created using this command. It is possible for an NMS to set more than one object at a time.

It's similar to the other commands we've seen so far, but it is actually changing something in the device's configuration, as opposed to just retrieving a response to a query. If we look at an example of an actual set, you will see the command take place. The following example queries the `sysLocation` variable, then sets it to a value:
The first command is the familiar get command, which displays the current value of `sysLocation`. In one of the previous examples we saw that it was undefined; this is still the case. The second command is snmpset. For this command, we supply the hostname, the read-write community string (private), and the variable we want to set (`system.sysLocation.0`), together with its new value (s "Atlanta, GA"). The s tells snmpset that we want to set the value of `sysLocation` to a string; and "Atlanta, GA" is the new value itself. How do we know that `sysLocation` requires a string value? The definition of `sysLocation` in RFC 1213 looks like this:

`sysLocation` OBJECT-TYPE
  SYNTAX DisplayString (SIZE (0..255))
  ACCESS read-write
  STATUS mandatory
  DESCRIPTION
    "The physical location of this node (e.g., 'telephone closet, 3rd floor')."

::= { system 6 }The SYNTAX for `sysLocation` is DisplayString (SIZE (0..255)), which means that it's a string with a maximum length of 255 characters. The snmpset command succeeds and reports the new value of `sysLocation`. But just to confirm, we
run a final snmpget, which tells us that the set actually took effect. It is possible to set more than one object at a time, but if any of the sets fail, they all fail (i.e., no values are changed). This behavior is intended.
3. 1B. SNMP Agent Flowchart and Algorithm:

Figure 3.6 Agent Creation Flowchart

Algorithm:

1. Form a model of the managed resource. Define the data types needed for the agent.

2. Define the MIB for the objects to be managed.

3. The application or component being managed requires instrumentation that provides information in a form that can be accessed by the agent or subagent.

4. Define your environment.

5. Run the MIB compiler/code generator (imibgenall) on a selected table or MIB group in your MIB file
6. Flesh out the generated function skeletons by adding code to access the data in the component or application being managed.

7. Do stand-alone testing of the access functions.

8. Build the SNMP agent executable.

9. Test the SNMP agent using the SNMP test utilities.

10. Integrate your private MIB with an SNMP management platform and test the agent with the management system.

3.2b. SNMP Design/pseudo code:

typedef unsigned char ui8;
typedef int i32;
typedef unsigned int ui32;
typedef unsigned short ui16;

enum
{
    RD_ONLY = 0,
    WR_ONLY = 1,
    RDWR = 2,
    INT = 5,
    OCT_STRING = 6,
    PTR = 11,
    FUN_PTR = 12,
    myfun = 16,
    pktCount = 17,
    MAX_SUB_IDS = 20,
}
MAX_SNMP_SIZE = 4000,
MAX_STATIC_VBS = 10,
SHORT_MAX_SUB_IDS = 100

};

struct shObjld
{
    ui16 subidCnt;
    ui16 subids[SHORT_MAX_SUB_IDS];
};

struct objlface
{
    struct shObjld soid;
    ui32 accsPerm; // RD_ONLY, WR_ONLY, RD_WR
    ui32 objType; // INT, OCT_STRING, NULL_TERM_STRING
    ui32 accType; // PTR, FUNC_PTR
    void *objAccs;
};

struct objlface oitable[100];

struct snmpHdr
{


ui32 verType;
ui8 community[20];

};

struct snmp_struc
{
    struct snmpHdr hdr;
    ui32 pduType;
    ui32 reqId;
    ui32 errStat;
    ui32 errIndx;
    ui32 dynVbCnt;
    ui32 count;
};

struct snmp
{
    unsigned char reqid[10];
    unsigned char community[10];
    unsigned char pdutype[10];
    unsigned char objId[10];
    unsigned char errstatus[5];
    unsigned char errindx[10];
};
void *objAccsFun(void *);

data definitions

ui8 snmpmsg[MAX_SNMP_SIZE];

struct objiface oitable[100]=
{
    { { 1,2}, RD_ONLY, INT, PTR, &val0 },
    { { 2,13,2 }, RD_ONLY, OCT_STRING, FUN_PTR, &val1 },
    { { 3,1,2,3}, RD_ONLY, INT, PTR, &val2 },
    { { 4,1,2,3,4}, RD_ONLY, INT, PTR, &val3 },
    { { 5,1,2,3,4,5}, RD_ONLY, INT, PTR, &val4 },
    { { 6,1,2,3,4,5,6}, RD_ONLY, INT, PTR, &val5 },
    { { 7,1,2,3,4,5,6,7}, RD_ONLY, INT, PTR, &val6 },
    { { 8,1,2,3,4,5,6,7,8}, RD_ONLY, INT, PTR, &val7 },
    { { 9,1,8,9,4,5,6,2,19,23}, RD_ONLY, INT, PTR, &val8 },
    { { 10,10,4,5,8,3,7,4,7,2,1}, RD_ONLY, INT, PTR, &val9 },
    { { 11,10,11,3,6,4,5,7,3,4,5,9}, RD_ONLY, INT, PTR, &val10 },
    { { 12,8,8,8,12,3,7,8,9,3,4,5,7}, RD_ONLY, INT, PTR, &val11 },
    { { 13,9,7,5,3,5,2,5,2,3,8,5,5,6}, RD_ONLY, INT, PTR, &val12 },
    { { 14,12,4,6,3,5,9,3,5,2,5,7,8,14,9}, RD_ONLY, INT, PTR, &val13 },
    { { 15,4,8,2,8,5,7,0,2,5,3,4,8,9,6,9}, RD_ONLY, INT, PTR, &val14 },
};

struct snmp_struc snmpTemp;
Function descriptions

snmpTask()
{
    open socket
    bind it

    while(FOREVER)
    {
        wait for a snmp message by calling recvfrom(&snmp,)
        snmpDecodeMsg(SnmpMsg,&snmpsr);
        switch(snmpst.pduType)
        {
            case GET_REQ:
                stat=processGetRe(&snmpst);
                snmpEncodeMsg(snmpMsg,&snmpst);
                sendto(snmpMsg);
                case GET_NEXT_TEQ:
                    stat=processGetNextReq(&snmpst);
                case SET_REQ:
                    stat=processSetReq(&snmpst);
                }
    }
}

int searchOid(ui32 i2,ui32 i3);
int seareh

82
int encodeOidValue(ui32);
int encodeNextOid(ui32, ui32);
int encodeError(ui32 erind);
int decodeDataQ;
void dispOitable();
void checkPpts();

RFC1157-SNMP DEFINITIONS ::= BEGIN

IMPORTS
    ObjectName, ObjectSyntax, NetworkAddress, IpAddress, TimeTicks
    FROM RFC1155-SMI;

-- top-level message

Message ::= 
    SEQUENCE {
        version -- version-1 for this RFC
            INTEGER {
                version-1(0)
            },

        community -- community name
            OCTET STRING,

    }
data -- e.g., PDUs if trivial

ANY -- authentication is being used

}

-- protocol data units

PDUs ::= 

  CHOICE {
    get-request
      GetRequest-PDU,
    get-next-request
      GetNextRequest-PDU,
    get-response
      GetResponse-PDU,
    set-request
      SetRequest-PDU,
    trap
      Trap-PDU
  }

Before introducing the six pdutypes of the protocol, it is appropriate to consider some of the ASN.1 constructs used frequently:

-- request/response information
RequestID ::= INTEGER

ErrorStatus ::= INTEGER {
    noError(0),
    tooBig(1),
    noSuchName(2),
    badValue(3),
    readOnly(4),
    genErr(5)
}

ErrorIndex ::= INTEGER

— variable bindings

VarBind ::= SEQUENCE {
    name
        ObjectName,

    value
        85
}
ObjectSyntax

VarBindList ::= 
SEQUENCE OF 
VarBind

The GetRequest-PDU

The form of the GetRequest-PDU is:

GetRequest-PDU ::= 
[0] 
IMPLICIT SEQUENCE { 
request-id 
  RequestID, 

error-status  -- always 0 
  ErrorStatus, 
error-index  -- always 0 
  ErrorIndex, 
variable-bindings 
  VarBindList 
}

86
3.1C. Simulation Results:

Figure 3.7. Models and Sub-models of the Manager-Agent Paradigm

The Manager-Agent architectural relationship is detailed by several specific sub-models.

There is a close relationship between communications, information, and functional models.

As a part of testing the SNMP Agent testing, first we have to login system in the network as a root user then copy the Agent executable in the system and run the Agent program. In Linux following command used to perform this operation.
Agent(server)side:

root# scp username @ IP address :directory file path/file name .

After entering this it will ask the password

Password:******

After entering the password, then it will show the message permission granted.

Then run the Agent program.

It shows the message like this:

Agent is waiting for request:(This is running forever)

3.2 C MANAGER SIDE:

From SNMP manager side we have give the commands(get ,get-next set,get-response,trap )i.e the operations to be performed on the Agent by the manager .The next field specifies the version of the SNMP Agent .Here we are using the version 1(v1) and c specifies the community name .And then we have to give the Agent IP address .finally we have to give the object identifier of the system (device).This field gives the information about the device parameter and variables. The following command shows the format of the SNMP manager side.
User$ snmpcmd -vl -c public Agent IP OID

Optical Amplifier parameter testing:(Optical Amplifier MIB ONS 15501 MIB-1)

1) User $ snmpget -vl -c public 192.168.0.20 1.3.1.4.1.1869.11.11.2

Then Agent processes the request from the manager and gives the response to the manager, stores the values of variables and parameter in the MIB.

Following is the Agent response.

Ons 15501 PowerSupply 1Status OBJECT-TYPE
SYNTAX Ons 15501 AlaramStatus
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"Gives the status of power supply one."
::={ons 15501 Attr17}.

2) User $ snmpgetnext -vl -c public 192.168.0.20 1.3.1.4.1.1869.11.11.2
(next oid=(1.3.1.4.1.1869.11.11.3))

Following is the Agent response

Ons 15501 PowerSupply 2Status OBJECT-TYPE
SYNTAX Ons 15501 AlaramStatus
MAX-ACCESS read-only
STATUS current
DESCRIPTION

“Gives the status of power supply two.”

::={ons 15501 Attr18}.

3) User $ snmset -v1 -c public 192.168.0.20 1.3.1.4.1.1869.11.11.2 it will set the
power Supply1 Status.