SNMPv1 Network Management: Communication and Functional Models

OBJECTIVES

- *Communication model: Administrative and messages*
- *Administrative structure*
  - *Community-based model*
  - *Access policy*
  - *MIB view*
- *Message PDU*
- *SNMP protocol specifications*
- *SNMP operations*
- *SNMP MIB*
- *SNMP functional model*

We have covered the organization and information models of SNMPv1 in the previous chapter. In this chapter we will address the SNMPv1 communication and functional models. Although SNMPv1 does not formally define the functional model, applications are built in the community-based access policy of the SNMP administrative model.

5.1 SNMP COMMUNICATION MODEL

The SNMPv1 communication model defines specifications of four aspects of SNMP communication: architecture, administrative model that defines data access policy, SNMP protocol, and SNMP MIB. Security in SNMP is managed by defining community, and only members belonging to the same community can communicate with each other. A manager can belong to multiple communities and can thus manage multiple domains. SNMP protocol specifications and messages are presented. SNMP entities are grouped into an SNMP MIB module.

5.1.1 SNMP Architecture

The SNMP architectural model consists of a collection of network management stations and network elements or objects. Network elements have management agents built in them, if they are managed

elements. The SNMP communications protocol is used to communicate information between network management stations and management agents in the elements.

There are three goals of the architecture in the original specifications of SNMP [RFC 1157]. First, it should minimize the number and complexity of management functions realized by the management agent. Secondly, it should be flexible for future expansion (addition of new aspects of operation and management). Lastly, the architecture should be independent of architecture and mechanisms of particular hosts and gateways.

Only non-aggregate objects are communicated using SNMP. The aggregate objects are communicated as instances of the object. This has been enhanced in SNMPv2, as we shall see in the next chapter. Consistent with the rest of SNMP standards, ASN.1 transfer syntax and BER encoding scheme are used for data transfer SNMP.

SNMP monitors the network with the five messages shown in Figure 4.9; and we discussed them in Section 4.6. They comprise three basic messages: set, get, and trap. Information about the network is primarily obtained by the management stations polling the agents. The get-request and get-next-request messages are generated by the manager to retrieve data from network elements using associated management agents. The set-request is used to initialize and edit network element parameters. The get-response-request is the response from the agent to get and set messages from the manager. The number of unsolicited messages in the form of traps is limited to make the architecture simple and to minimize traffic.

There are three types of traps—generic-trap, specific-trap, and time-stamp, which are application specific. The generic-trap type consists of *coldStart*, *warmStart*, *linkDown*, *linkUp*, *authenticationFailure*, *egpNeighborLoss,* and *enterpriseSpecific*. The specific-trap is a specific code and is generated even when an *enterpriseSpecific* trap is not present. An example of this would be to gather statistics whenever a particular event occurs, such as use by a particular group. The time-stamp trap is the time elapsed between the last initialization or re-initialization of the element and the generation of the trap.

SNMP messages are exchanged using a connectionless UDP transport protocol in order to be consistent with simplicity of the model, as well as to reduce traffic. However, the mechanisms of SNMP are suitable for a variety of protocols.

5.1.2 Administrative Model

Although the topic of administrative models should normally be discussed as part of security and privacy under the functional model, at this point it helps to understand the administrative relationship among entities that participate in the communication protocol in SNMP. Hence, we will discuss it now.

In RFC 1157 the entities residing in management stations and network elements are called SNMP application entities. Peer processes, which implement SNMP, and thus support SNMP application entities, are termed protocol entities. We will soon discuss protocol entities in detail. First, let us look at the application entities.

We will refer to the *application entity* residing in the management station as the SNMP manager, and the application entity in the element as the SNMP agent. The pairing of the two entities is called an SNMP community. The SNMP community name, called the *community*, is specified by a string of octets. Multiple pairs can belong to the same community. Figure 5.1 shows multiple SNMP managers communicating with a single SNMP agent. While an SNMP manager is monitoring traffic on an element, another manager may be configuring some administrative information on it. A third manager can be monitoring it to perform some statistical study. We also have the analogous situation where a manager communicates with multiple agents.

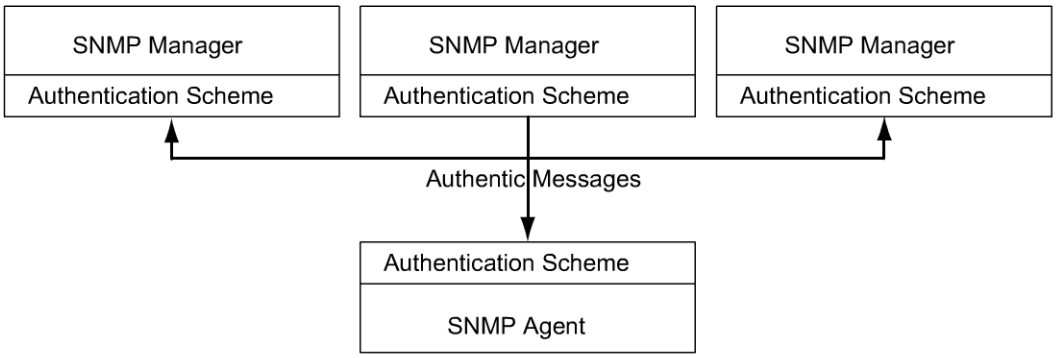

Figure 5.1 SNMP Community

With one-to-many, many-to-one, and many-to-many communication links between managers and agents, a basic authentication scheme and an access policy have been specified in SNMP. Figure 5.1 shows the *authentication scheme*, which is a filter module in the manager and the agent. The simplest form of authentication is the common community name between the two application entities. Encryption would be a higher level of authentication in which case both the source and the receiver know the common encryption and decryption algorithms.

The SNMP authorization is implemented as part of managed object MIB specifications. We discussed MIB specifications for managed objects in Chapter 4, and will discuss MIB specifications for SNMP protocol in Section 5.1.4. A network element comprises many managed objects—both standard and private. However, a management agent may be permitted to view only a subset of the network element's managed objects. This is called the community *MIB view*. In Figure 5.2 the SNMP agent has a MIB view of objects 2, 3, and 4, although there may be other objects associated with a network element. In addition to the MIB view, each community name is also assigned an *SNMP access mode*, either READ-ONLY or READ-WRITE, as shown in Figure 5.2. A pairing of SNMP MIB views with an SNMP access code is called a *community profile*.

A community profile in combination with the access mode of a managed object determines the operation that can be performed on the object by an agent. For example, in Figure 5.2, an SNMP agent with READ-WRITE SNMP access mode can perform all operations—get, set, and trap—on objects 2, 3, and 4. On the other hand, if the SNMP agent has READ-ONLY access mode privilege, it can only perform get and trap operations on objects 2, 3, and 4. Object 1 has a "not-accessible" access mode and hence no operation can be performed on it.

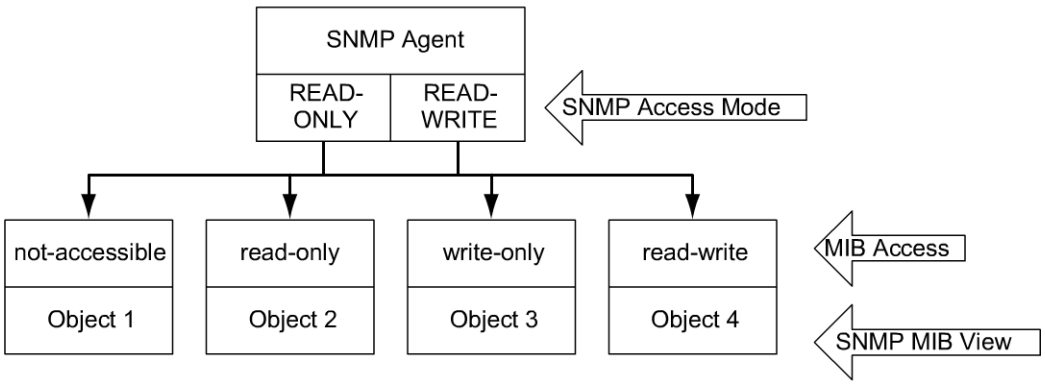

Figure 5.2 SNMP Community Profile

A practical application of the SNMP access policy can be envisioned in an enterprise management system of a corporation with headquarters in New York and domains or network sites in New York and San Francisco. Let Manager 1 and Community 1 be associated with San Francisco, and Manager 2 and Community 2 with New York. Let Manager 3 be the overall network management system, the Manager of Managers (MoM). Manager 1 manages Agents 1 and 2 associated with network elements in San Francisco. Manager 2 manages the New York network domain. Manager 1 does not have the view of New York and Manager 2 cannot perform operations on network elements belonging to the San Francisco domain. Manager 3 has both community names defined in its profile and hence has the view of the total enterprise network in New York and San Francisco.

The SNMP access policy has far-reaching consequences beyond that of servicing a TCP/IP-based Internet SNMP community. It can be extended to managing non-SNMP community using the SNMP proxy access policy. The SNMP agent associated with the proxy policy is called a proxy agent or commercially, a proxy server. The proxy agent monitors a non-SNMP community with non-SNMP agents and then converts objects and data to SNMP-compatible objects and data to feed to an SNMP manager.

Figure 5.4 shows an illustration of SNMP and non-SNMP communities being managed by an SNMP manager. A practical example of this would be a network of LAN and WAN. LAN could be a TCP/IP network with SNMP agents. WAN could be an X.25 network, which is not an Internet model, but can be managed by a proxy agent and integrated into the overall management system.

5.1.3 SNMP Protocol Specifications

Peer processes, which implement SNMP, and thus support SNMP application entities, are termed *protocol entities*. Communication among protocol entities is accomplished using messages encapsulated in a UDP datagram. An SNMP message consists of a version identifier, an SNMP community name, and a protocol data unit (PDU). Figure 5.5 shows the encapsulated SNMP message. The version and community names are added to the data PDU and along with the application header is passed on to the transport layer as SNMP PDU. The UDP header is added at the transport layer, which then forms the transport PDU for the network layer. The addition of the IP header to the Transport PDU forms the Network PDU for the data link layer (DLC). The network or DLC header is added before the frame is transmitted on to the physical medium.

An SNMP protocol entity is received on port 161 on the host except for trap, which is received on port 162. The maximum length of the protocol in SNMPv1 is 484 bytes (1,472 bytes now in practice). It

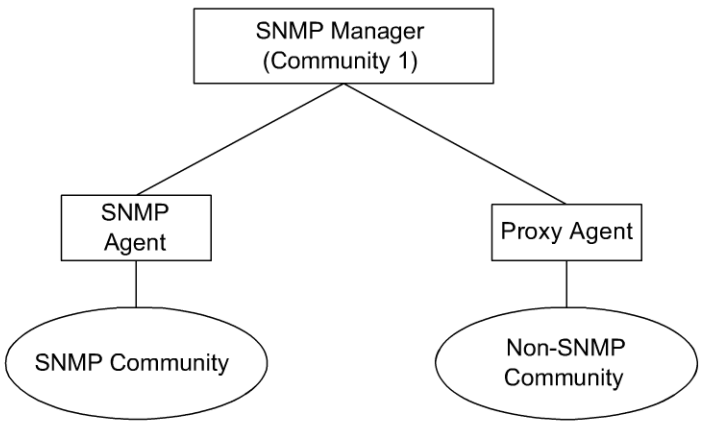

Figure 5.4 SNMP Proxy Access Policy

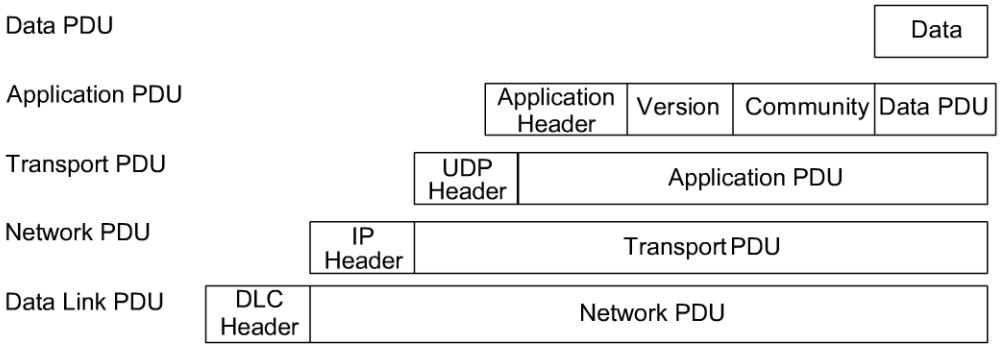

Figure 5.5 Encapsulated SNMP Message

is mandatory that all five PDUs be supported in all implementations: GetRequest-PDU, GetNextRequest-PDU, GetResponse-PDU, SetRequest-PDU, and Trap-PDU. One of these five data PDUs is the data PDU that we start with at the top in Figure 5.5. RFC 1157-SNMP Macro definition is given in Figure 5.6.

```
RFC1157 SNMP DEFINITIONS ::= BEGIN

IMPORTS

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

--top-level message

        Message ::=
                SEQUENCE {
                        version         -- version
                          INTEGER {             -1 for this RFC
                                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
                        get-next-request    GetRequest-PDU,
                        get-response        GetNextRequest-PDU,
                        set-request         GetResponse-PDU,
                        trap                SetRequest-PDU,
                        }                   Trap-PDU

-- the individual PDUs and commonly used data types will be defined later

END
```

Figure 5.6 RFC 1157-SNMP Macro

```
-- request/response information

RequestId ::=
        INTEGER

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

ErrorIndex ::=
        INTEGER

-- variable bindings

VarBind ::=
        SEQUENCE {
                name            ObjectName
                value           ObjectSyntax
        }

VarBindList ::=
        SEQUENCE OF
                VarBind
```

Figure 5.7 Get and Set Type PDU ASN.1 Construct [RFC 1157]

Basic operations of the protocol entity involve the following steps as a guide to implementation [RFC 1157]. The protocol entity that generates the message constructs the appropriate data PDU as an ASN.1 object. It then passes the ASN.1 object along with a community name and the transport addresses of itself and the destination (e.g., 123.234.245.156:161) to the authentication scheme. The authentication scheme returns another ASN.1 object (possibly encrypted). The protocol entity now constructs the message to be transmitted with the version number, community name, and the new ASN.1 object, then serializes it using the BER rules, and transmits it.

The reverse process goes on at the receiver. The message is discarded if an error is encountered in any of the steps. A trap may be generated in case of authentication failure. On successful receipt of the message, a return message is generated, if the original message is a get-request.

A managed object is a scalar variable and is simply called a variable. Associated with the variable is its value. The pairing of the variable and value is called *variable binding* or *VarBind*. The data PDU in the message contains a VarBind pair. For efficiency sake, a list of VarBind pairs can be sent in a message. The ASN.1 construct for get and set type of messages is shown in Figure 5.7 and a conceptual presentation in Figure 5.8. The *VarBindList* contains *n* instances of VarBind (pairs).

The PDU type for the five messages are application data types, which are defined in RFC 1157 as:

get-request	[0]
get-next-request	[1]

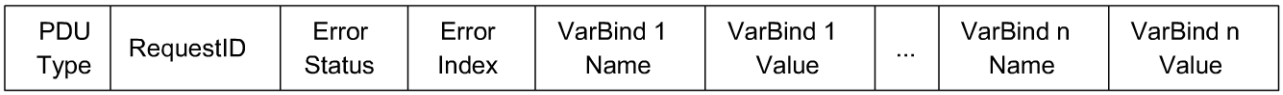

Figure 5.8 Get and Set Type PDUs

set-request	[2]
get-response	[3]
trap	[4]

In Figure 5.8 *RequestID* is used to track a message with the expected response or for loss of a message (remember UDP is unreliable). Loss-of-message detection is implementation specific, such as time out if no response is received for a request within a given time. A non-zero *ErrorStatus* is used to indicate that an error occurred. The convention is not to use 0 if no error is detected. *ErrorIndex* is used to provide additional information on the error status. The value is filled with NULL in those cases where it is not applicable, such as in get-request data PDU. Otherwise, it is filled with the *varBind* number where the error occurred; for example, 1 if the error occurred in the first *varBind*, 5 if the fifth *varBind* had an error and so on.

Figure 5.9 shows the structure for a trap PDU, which contains *n VarBind*s, i.e., *n* managed objects. The enterprise [RFC 1155] and agent-address pertain to the system generating the trap. The generic-trap consists of seven types as listed in Table 5.1. The integer in parenthesis associated with each name indicates the enumerated INTEGER. The specific-trap is a trap that is not covered by the *enterpriseSpecific* trap. Time-stamp indicates the elapsed time since last re-initialization.

Figure 5.9 Trap PDU

Table 5.1 Generic Traps

GENERIC-TRAP TYPE	DESCRIPTION (BRIEF)
coldStart(0)	Sending protocol entity is reinitializing itself; agent configuration or protocol entity implementation may be altered
warmStart(1)	Sending protocol entity is reinitializing itself; agent configuration or protocol entity implementation not altered
linkDown(2)	Failure of one of the communication links
linkUp(3)	One of the links has come up
authenticationFailure(4)	Authentication failure
egpNeighborLoss(5)	Loss of EGP neighbor
enterpriseSpecific(6)	Enterprise-specific trap

5.1.4 SNMP Operations

SNMP operations comprise get and set messages from the manager to the agent, and get and trap messages from the agent to the manager. We will now look at these operations in detail in this section.

GetRequest PDU Operation. Figure 5.10 shows a sequence of operations in retrieving the values of objects in a System group. It starts with the get-request operation using a GetRequest PDU from a manager process to an agent process and the get-response from the agent with a GetResponse PDU. The message from the manager starts from the left side and ends at the agent process on the right side of the figure. The message from the agent process starts on the right side of the figure and ends at the manager process on the left side of the figure. The sequence of directed messages moves with time as we move down the figure. Messages depicted represent the values of the seven objects in the System group.

The manager process starts the sequence in Figure 5.10 with a GetRequest PDU for the object *sysDescr*. The agent process returns a GetResponse PDU with a value "SunOS." The manager then sends a request for *sysObjectID* and receives the value "E:hp." The exchange of messages goes on until the value of 72 for the last object in the group *sysServices* is received.

GetNextRequest PDU Operation. A get-next-request operation is very similar to get-request, except that the requested record is the next one to the OBJECT IDENTIFIER specified in the request. Figure 5.11 shows the operations associated with retrieving data for the System group by the manager process using the get-next-request. The first message is a GetRequest PDU for *sysDescr* with the response returning the value "SunOS." The manager process then issues a GetNextRequest PDU with the OBJECT IDENTIFIER *sysDescr*. The agent processes the name of the next OBJECT IDENTIFIER *sysObjectID* and its value "E:hp." The sequence terminates when the manager issues a get-next-request for the object identifier next to *sysServices*, and the agent process returns the error message "noSuchName."

The System group example we just looked at is a simple case where all the objects are single-valued scalar objects. Let us now consider a more complex scenario of a MIB that contains both scalar and aggregate objects. A generalized case of a conceptual MIB comprising three scalar objects and a table is

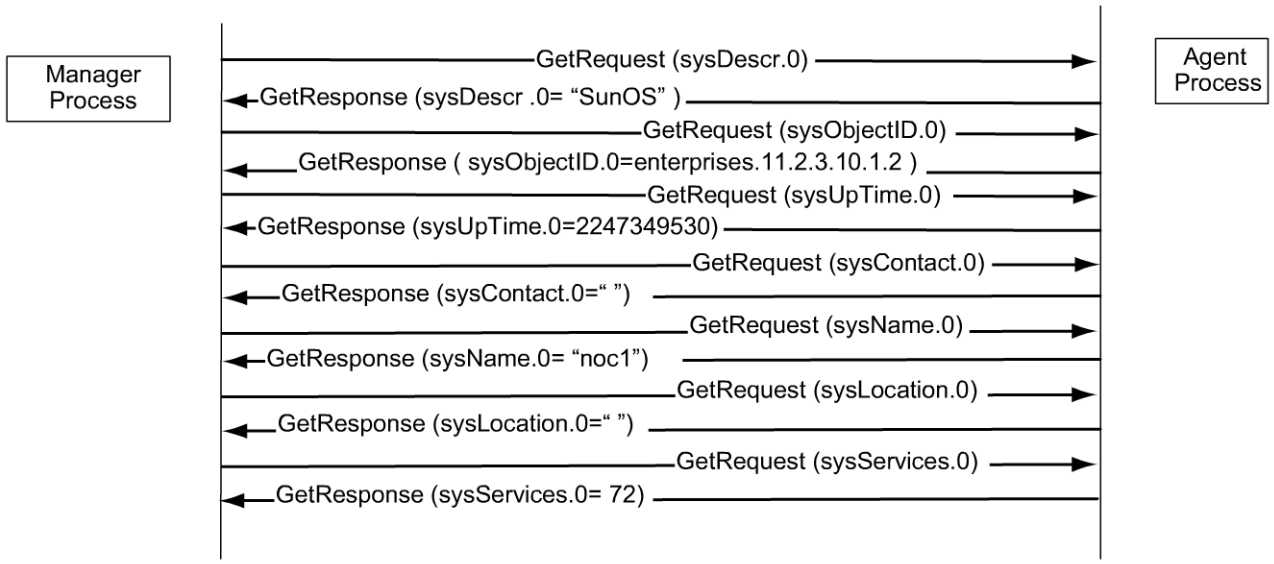

Figure 5.10 Get-Request Operation for System Group

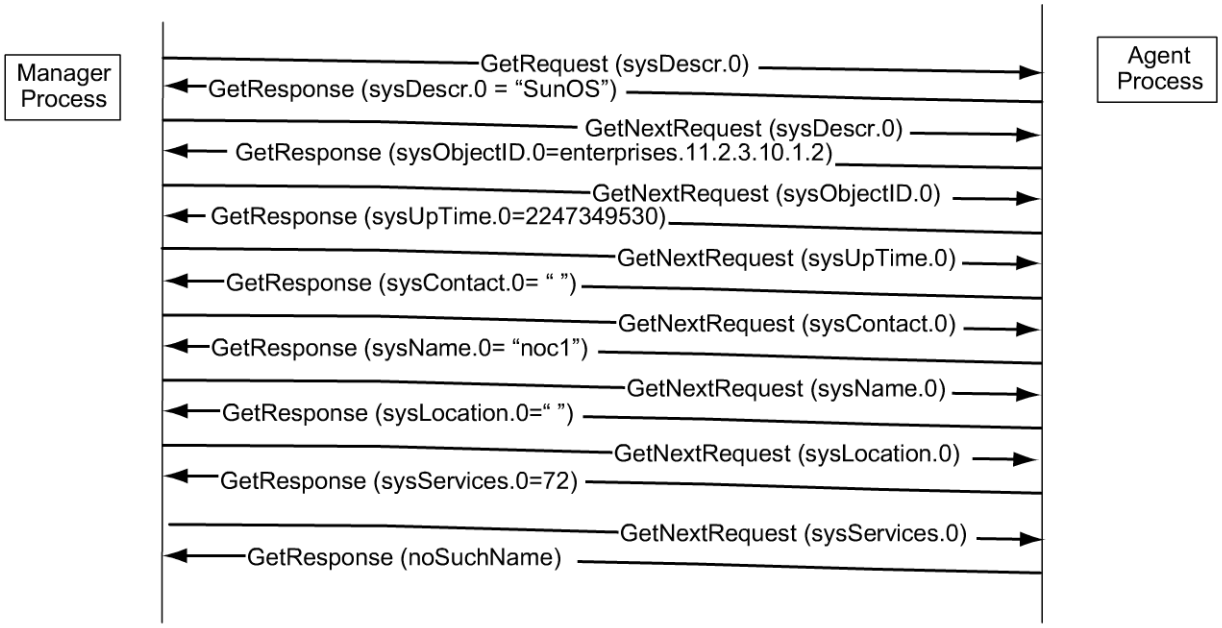

Figure 5.11 Get-Next-Request Operation for a System Group

shown in Figure 5.12. The first two objects A and B are single-valued scalar objects. They are followed by an aggregate object represented by the table T with an entry E and two rows of three columnar objects, T.E.1.1. through T.E.3.2. The MIB group ends with a scalar object Z.

Figure 5.13 shows the use of nine get-request messages to retrieve the nine objects. The left side of the figure shows the sequential operation for getting the MIB shown on the right side of the figure. The MIB shown is the same as in Figure 5.12, now drawn to follow the sequence of operations. We observe

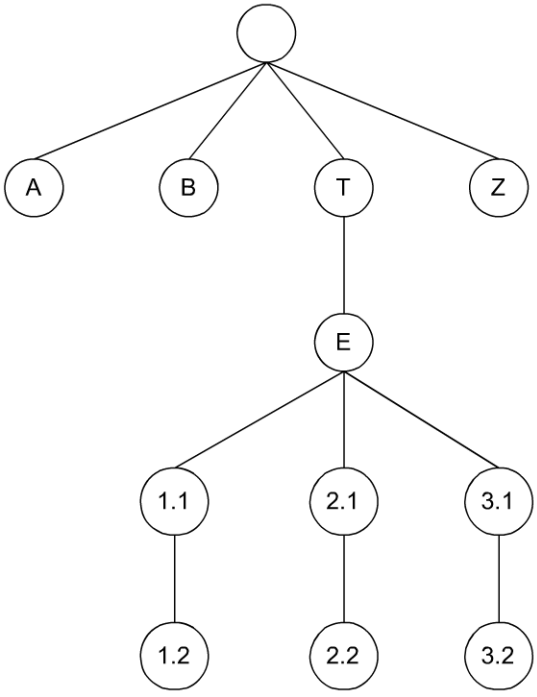

Figure 5.12 MIB for Operation Examples in Figures 5.13 and 5.15

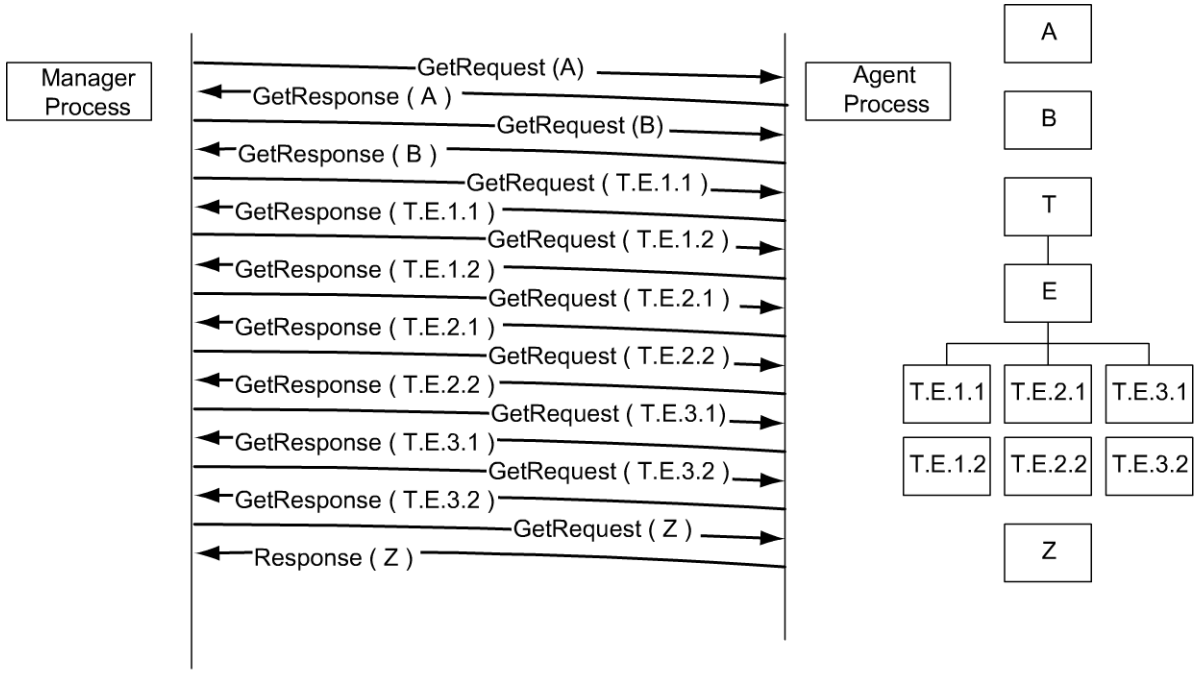

Figure 5.13 Get-Request Operation for a MIB in Figure 5.12

a few hidden assumptions in retrieving the data using the get-request operations. First, we need to know all the elements in the MIB including the number of columns and rows in the table. Second, we traversed the MIB from top to bottom, which is really from right to left in the MIB tree structure. Third, we retrieved the data in the table by traversing all the instances of a columnar object. The number of instances or rows in a table could be dynamic and is not always known to the management process. Thus, if the manager had issued a request for the object T.E.1.3 after acquiring T.E.1.2, it would have received an error message from the agent process. This is when get-next-request is very useful. However, we need to have a convention on the definition of what the next object in a MIB tree is, especially on the table representing an aggregate object. In SNMP, objects are retrieved using lexicographic convention. We will first explain what this convention is before using the get-next-request operation to retrieve the same MIB group data.

The increasing order of entity used in SNMP operations is in lexicographic order. Let us understand lexicographic order by considering a simple set of integers shown in Table 5.2. The left side is a sequence of numerically increasing integer numbers, and the right side is lexicographically increasing order for this sequence. We notice that in the lexicographic order, we start with the lowest integer in the leftmost character, which in our case is 1. Before increasing the order in the first position, we select the lowest integer in the second position from the left, which is 11. There are two numbers (1118 and 115) that start with 11. We anchor at 11 for the first two positions and then move on to select the lowest digit in the third position, which is 111. We then move to the fourth position and obtain 1118 as the second number. Now, return to the third position and retrieve 115 as the third number. Having exhausted 1s (ones) in positions two to four, select 2 for the second position, and retrieve 126 as the next number. We continue this process until we reach 9.

We will now apply the lexicographic sequence to ordering object identifiers in a MIB. Instead of each character being treated as a literal, we treat each node position as a literal and follow the same rules. An example illustrating this is given in Table 5.3. The MIB associated with this example is shown in

Table 5.2 Lexicographic-Order Number Example

NUMERICAL ORDER	LEXICOGRAPHIC ORDER
1	1
2	1118
3	115
9	126
15	15
22	2
34	22
115	250
126	2509
250	3
321	321
1118	34
2509	9

Figure 5.14. It can be noticed that the lexicographically increasing order of node traces the traversal of the tree starting from the leftmost node 1. We traverse down the path all the way to the leftmost leaf 1.1.5, keeping to the right whenever a fork is encountered. We then move up the tree and take a right on the first fork. This leads us to the leaf node 1.1.18. Thus, the rule at a forked node is to always keep to the right while traversing down and while going up. Thus, we are always keeping to the right if you imagine ourselves walking along the tree path and looking in the forward direction. We turn around when we reach a leaf.

Returning to get-next-request operation, the get-response message contains the value of the next lexicographic object value in each VarBind. If the request VarBind contains a scalar, non-tabular object,

Table 5.3 MIB Example for Lexicographic Ordering

1
1.1
1.1.5
1.1.18
1.2
1.2.6
2
2.2
2.10
2.10.9
3
3.4
3.21
9

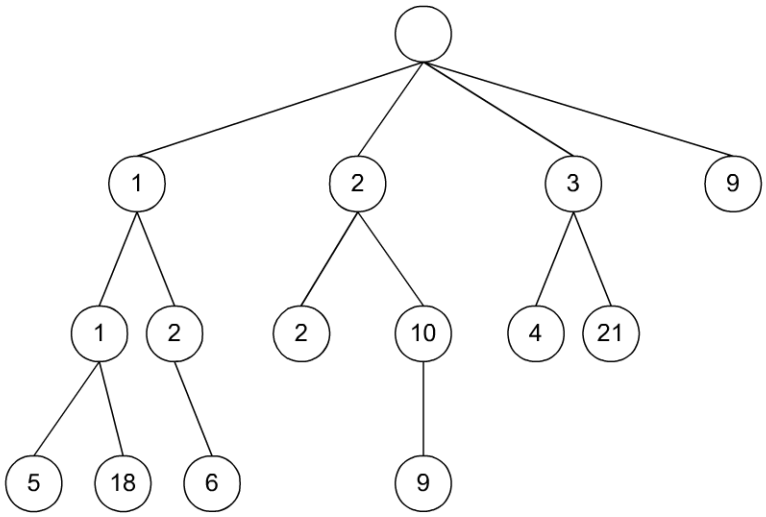

Figure 5.14 MIB Example for Lexicographic Ordering

the response contains the next scalar, non-tabular value, or the first columnar object value of a table, if it is the next lexicographic entity. Figure 5.15 shows the principle of operation of the functioning of get-next-request and response. We use the same MIB view that we had in Figure 5.12 using get-request operation. The manager process starts the operation with a get-request message for object A and receives the response with the value of A filled in. Subsequent requests from the manager are get-next-request type with the object ID of the just received ones. Responses received are the next object ID with its value. Operations continue until Z is received. The subsequent request receives a response with an error message "noSuchName."

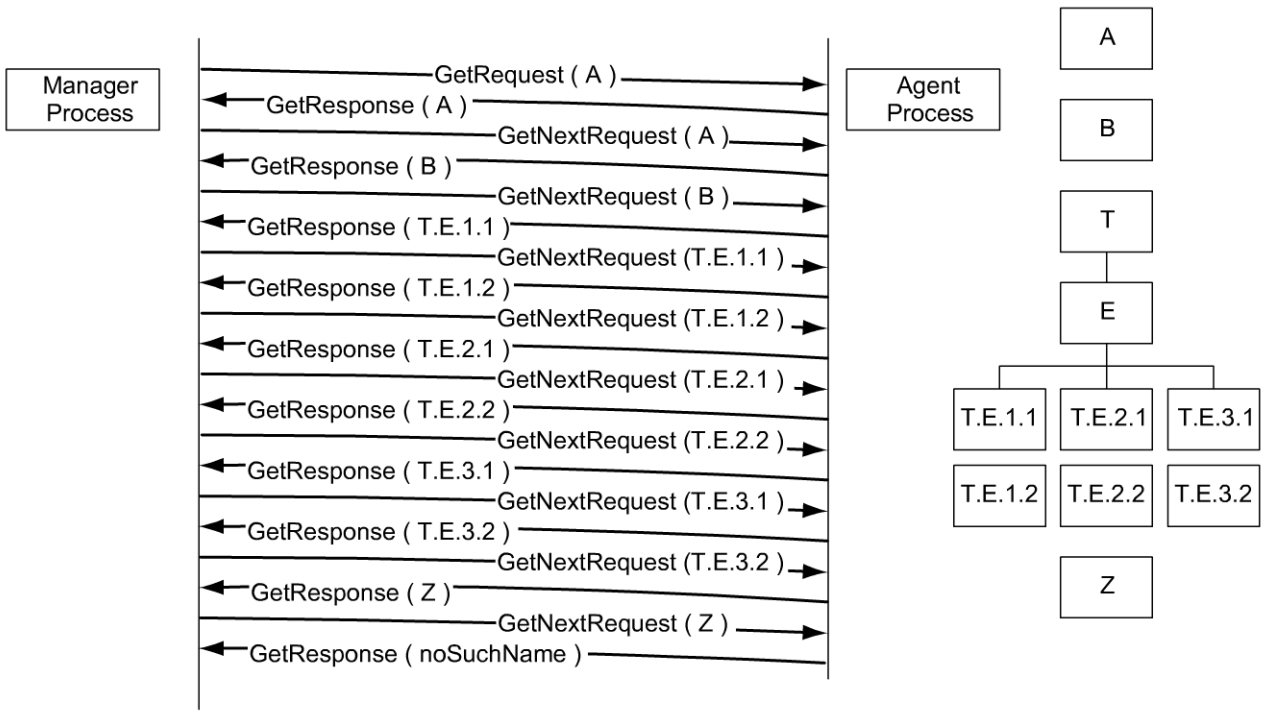

Figure 5.15 Get-Next-Request Operation for a MIB in Figure 5.12

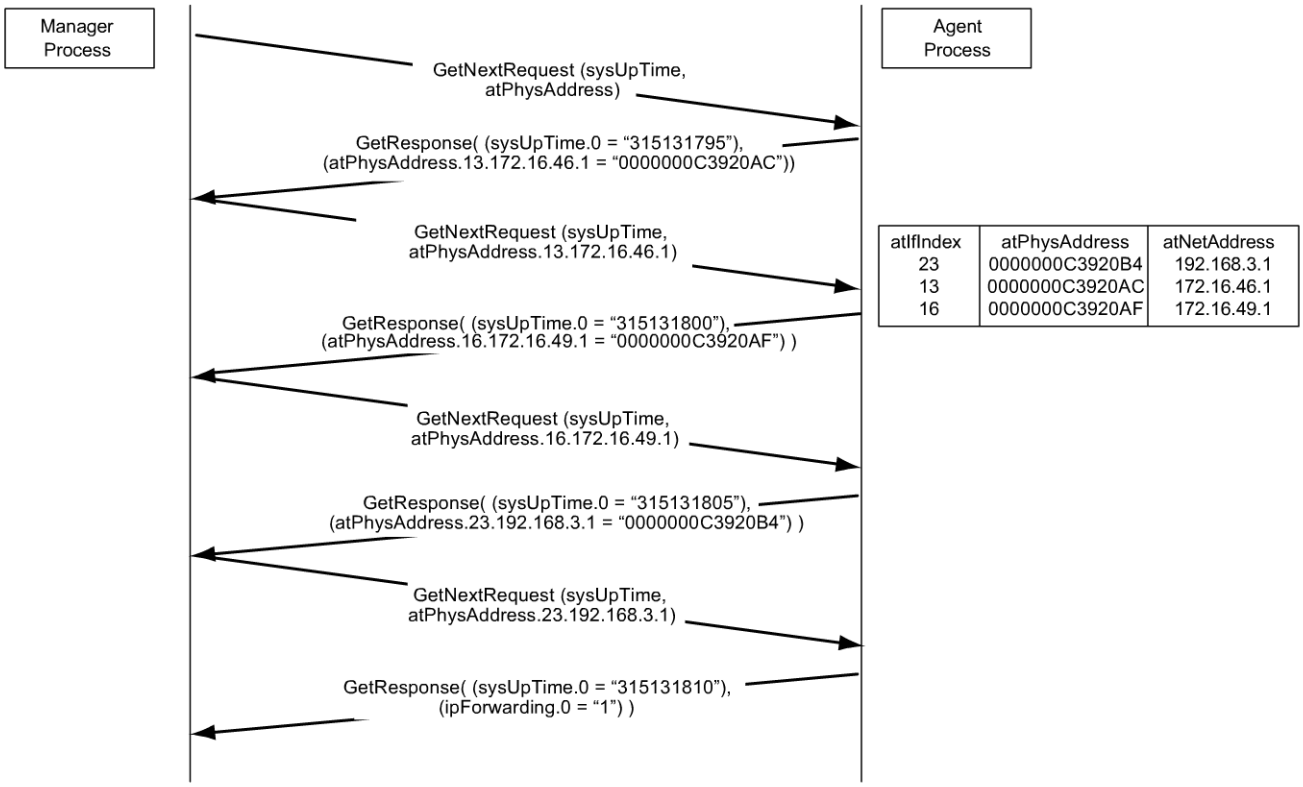

Figure 5.16 GetNextRequest Example with Indices

There are several advantages in using get-next-request. First, we do not need to know the object identifier of the next entity. Knowing the current OBJECT IDENTIFIER, we can retrieve the next one. Next, in the case of an aggregate object, the number of rows is dynamically changing. Thus, we do not know how many rows exist in the table. The get-next-request resolves this problem.

There is also another advantage of the get-next-request. We can use this to build a MIB tree by repeating the request from any node to any node. This is called MIB walk, and is used by a MIB browser in NMS implementation.

Figure 5.16 shows a faster method to retrieve an aggregate object. It shows an Address Translation table with a matrix of three columnar objects, *atIfIndex*, *atPhysAddress*, and *atNetAddress*. The objects *atIfIndex* and *atNetAddress* are the indices that uniquely identify a row. There are three rows in the table. If we use the get-next-request operation shown in Figure 5.15, it would take us ten message exchanges. The *VarBindList* comprises two VarBind name–value pairs, *sysUpTime* and *atPhysAddress*, suffixed with the values of *atIfIndex* and *atNetAddress*. Instead of issuing ten get-next-requests with a single VarBind in the message, the manager generates four GetNextRequest PDUs with a list of two VarBind fields. Although the Address Translation table is relatively stable, in general, a table is dynamic, and hence the time-stamp is requested by including *sysUpTime*.

In this method, the manager has to know the columnar objects of the table. The first query message retrieves the indices automatically. For the Address Translation table, the *atIfIndex* and *atNetAddress* are indices. This is shown in the request and response message OIDs. The first get-next-request message does not contain any operand value. The next three contain the value returned by the response. The fourth and last get-next-request brings the object, *ipForwarding*, which is the first element in the

IP group, which is the next group in Internet MIB. This is because all table entries in the Address Table have been retrieved. It is up to the manager process to recognize this and terminate the process. If the table contained more columns, the *VarBindList* could be expanded and values for all the objects in the next row obtained with each request.

There are more details to this PDU operation and the reader is referred to the references Perkins and McGinnis [1997], RFC [1905], and Stallings [1998].

SNMP PDU Format Examples. We will now look at the PDU for the System group example shown in Figure 5.10 using a sniffer tool. Sniffer is a management tool that can capture packets going across a transmission medium. We have used this tool to "sniff" some SNMP messages to display how messages actually look. We are presenting a series of messages that query a system for its system group data (Figure 5.17). This corresponds to the data shown in Figure 5.10. We then set the missing values for a couple of entities in the group (Figures 5.18 and 5.19) and finally reexamine them (Figure 5.20).

Figure 5.17(a) shows a GetRequest message for the system group values going from the manager, noc3.gatech.btc.gatech.edu (noc3, for short), to the agent, noc1.btc.gatech.edu (noc1, for short). The first line shows that it was sent at 13:55:47 from port 164 of noc1 to snmp port of noc3. The tool that was used has actually translated the conventional port number 161 to snmp. The community name is public and the GetRequest message is 111 bytes in length. The SNMP version number is not filled in.

```
13:55:47.445936 noc3.btc.gatech.edu.164 > noc1.btc.gatech.edu.snmp:
Community = public
GetRequest(111)
Request ID = 1
system.sysDescr.0
system.sysObjectID.0
system.sysUpTime.0
system.sysContact.0
system.sysName.0
system.sysLocation.0
system.sysServices.0
```

(a) Get-Request Message from Manager-to-Agent (Before)

```
13:55:47.455936 noc1.btc.gatech.edu.snmp > noc3.btc.gatech.edu.164:
Community = public
GetResponse(172)
Request ID = 1
system.sysDescr.0 =     "SunOS noc1 5.5.1 Generic_103640-08 sun4u"
system.sysObjectID.0 = E:hp.2.3.10.1.2
system.sysUpTime.0 = 247349530
system.sysContact.0 =     ""
system.sysName.0 =     "noc1 "
system.sysLocation.0 =     ""
system.sysServices.0 = 72
```

(b) Get-Response Message from Agent-to-Manager (Before)

Figure 5.17 Sniffer Data of Get Messages (Incomplete Data in Agent)

```
13:56:24.894369 noc3.btc.gatech.edu.164 > noc1.btc.gatech.edu.snmp:
Community = netman
SetRequest(41)
Request ID = 2
system.sysContact.0 = “Brandon Rhodes”

13:56:24.894369 noc1.btc.gatech.edu.snmp > noc3.btc.gatech.edu.164:
Community = netman
GetResponse(41)
Request ID = 2
system.sysContact.0 = “Brandon Rhodes”
```

Figure 5.18 Sniffer Data of Set-Request and Response for System Contact

```
13:56:27.874245 noc3.btc.gatech.edu.164 > noc1.btc.gatech.edu.snmp:
Co mmunity = netman
SetRequest(37)
Request ID = 3
system.sysLocation.0 = “BTC NM Lab”

13:56:27.884244 noc1.btc.gatech.edu.snmp > noc3.btc.gatech.edu.164:
Community = netman
GetResponse(37)
Request ID = 3
system.sysLocation.0 = “BTC NM Lab”
```

Figure 5.19 Sniffer Data of Set-Request and Response for System Location

```
14:03:36.788270 noc3.btc.gatech.edu.164 > noc1.btc.gatech.edu.snmp:
Community = public
GetRequest(111)
Request ID = 4
system.sysDescr.0
system.sysObjectID.0
system.s  ysUpTime.0
system.sysContact.0
system.sysName.0
system.sysLocation.0
system.sysServices.0
```

(a) Get-Request Message from Manager-to-Agent (After)

Figure 5.20 Sniffer Data of Get Messages (Complete Data in Agent)

```
14:03:36.798269 noc1.btc.gatech.edu.snmp > noc3.btc.gat ech.edu.164:
Community = public
GetResponse(196)
Request ID = 4
system.sysDescr.0 =      "SunOS noc1 5.5.1 Generic_103640 -08 sun4u"
system.sysObjectID.0 = E:hp.2.3.10.1.2
system.sysUpTime.0 = 247396453
system.sysContact.0 = "Brandon Rhodes"
system.sysName.0 = "noc1"
system.sysLocation.0 =  "BTC NM Lab"
system.sysServices.0 = 72
```

(b) Get-Response Message from Agent-to-Manager (After)

Figure 5.20 (continued)

The seven object IDs from *system.sysDescr.*0 to *system.sysServices.*0 all end with zero to indicate that they are single-valued scalar objects. The agent, noc1, sends a GetResponse message of 172 bytes with values filled in for all seven objects. The GetResponse message is shown in Figure 5.17(b). Notice that the values for *sysContact* and *sysLocation* in GetResponse are blank as they have not been entered in the agent. In addition, the request number identified in the GetResponse PDU is the same as the one in the GetRequest PDU.

Figure 5.18 shows the use of the SetRequest message to write the *sysContact* name in noc1 whose value is "Brandon Rhodes." Notice that the community name is changed to netman. The community of netman has the access privilege to write in noc1, and the object, *system.sysContact*, has the read-write access for the netman community. The agent, noc1, makes the change and sends a GetResponse message back to noc3. Figure 5.19 shows a similar set of messages for setting the entity *sysLocation* with the value "BTC NM Lab."

Figures 5.20(a) and (b) are a repetition of Figure 5.14 of the GetRequest and GetResponse messages. We now see the completed version of the system group data.

5.1.5 SNMP MIB Group

Figure 5.21 shows the MIB tree for the SNMP group, and Table 5.4 gives the description of the entities. Note that OID 7 and OID 23 are not used. The number of transactions in the description column in the table indicates ins and outs of the SNMP protocol entity. All entities except *snmpEnableAuthenTraps* have the syntax, Counter. The implementation of the SNMP group is mandatory—obviously!

5.2 FUNCTIONAL MODEL

There are no formal specifications of functions in SNMPv1 management. Application functions are limited, in general, to network management in SNMP and not to the services provided by the network.

There are five areas of functions (configuration, fault, performance, security, and accounting) addressed by the OSI mode. Some configuration functions, as well as security and privacy-related issues, were addressed as part of the SNMP protocol entity specifications in the previous section. For

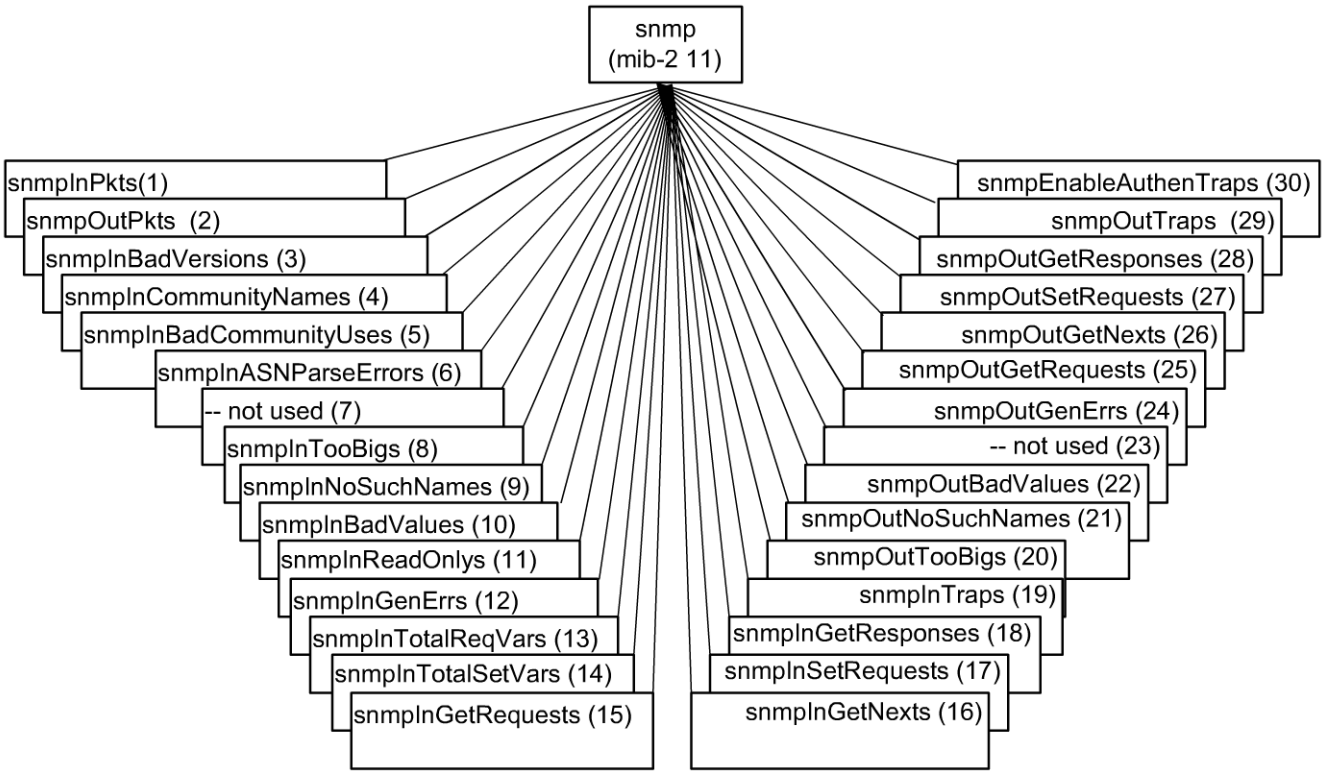

Figure 5.21 SNMP Group

example, the override function of traps is one of the objects in the SNMP group, which has the access privilege of read and write and hence can be set remotely. Security functions are built in as part of the implementation of the protocol entity. Community specifications and authentication scheme partially address these requirements.

The write access to managed objects is limited to implementation in most cases. Thus, configuration management in general is addressed by the specific network management system or by the use of console or telnet to set configurable parameters. We saw the use of the configuration management function in the examples shown in Figures 5.18 and 5.19.

Fault management is addressed by error counters built into the agents. They can be read by the SNMP manager and processed. Traps are useful to monitor network elements and interfaces going up and down.

Performance counters are part of the SNMP agent MIB. It is the function of the SNMP manager to do performance analysis. For example, counter readings can be taken at two instances of time and the data rate calculated. The intermediate manager/agent, such as RMON, can perform such statistical functions, as we will see in the next chapter.

The administrative model in protocol entity specifications addresses security function in basic SNMP.

The accounting function is not addressed by the SNMP model.

Table 5.4 SNMP Group

ENTITY	OID	DESCRIPTION (BRIEF)
snmpInPkts	snmp (1)	Total number of messages delivered from transport service
snmpOutPkts	snmp (2)	Total number of messages delivered to transport service
snmpInBadVersions	snmp (3)	Total number of messages from transport service that are of unsupported version
snmpInBadCommunityNames	snmp (4)	Total number of messages from transport service that are of unknown community name
snmpInBadCommunityUses	snmp (5)	Total number of messages from transport service, not allowed operation by the sending community
snmpInASNParseErrs	snmp (6)	Total number of ASN.1 and BER errors
	snmp (7)	Not used
snmpInTooBigs	snmp (8)	Total number of messages from transport service that have 'tooBig' errors
snmpInNoSuchNames	snmp (9)	Total number of messages from transport service that have 'noSuchName' errors
snmpInBadValues	snmp (10)	Total number of messages from transport service that have 'badValue' errors
snmpInReadOnlys	snmp (11)	Total number of messages from transport service that have 'readOnly' errors
snmpInGenErrs	snmp (12)	Total number of messages from transport service that have 'genErr' errors
snmpInTotalReqVars	snmp (13)	Total number of successful Get-Request and Get-Next messages received
snmpInTotalSetVars	snmp (14)	Total number of objects successfully altered by Set-Request messages received
snmpInGetRequests	snmp (15)	Total number of Get-Request PDUs accepted and processed
snmpInGetNexts	snmp (16)	Total number of Get-Next PDUs accepted and processed
snmpInSetRequests	snmp (17)	Total number of Set-Request PDUs accepted and processed
snmpInGetResponses	snmp (18)	Total number of Get-Response PDUs accepted and processed
snmpInTraps	snmp (19)	Total number of Trap PDUs accepted and processed
snmpOutTooBigs	snmp (20)	Total number of SNMP PDUs generated for which error-status is 'tooBig'
snmpOutNoSuchNames	snmp (21)	Total number of SNMP PDUs generated for which error-status is 'noSuchName'
snmpOutBadValues	snmp (22)	Total number of SNMP PDUs generated for which error-status is 'badValue'
	snmp (23)	Not used
snmpOutGenErrs	snmp (24)	Total number of SNMP PDUs generated for which error-status is 'genErr'
snmpOutGetRequests	snmp (25)	Total number of SNMP Get-Request PDUs generated
snmpOutGetNexts	snmp (26)	Total number of SNMP Get-Next PDUs generated
snmpOutSetRequests	snmp (27)	Total number of SNMP Set-Request PDUs generated
snmpOutGetResponses	snmp (28)	Total number of SNMP Get-Response PDUs generated
snmpOutTraps	snmp (29)	Total number of SNMP Trap PDUs generated
snmpEnableAuthenTraps	snmp (30)	Override option to generate authentication failure traps

Summary

All management operations are done using five messages in SNMPv1. They are get-request, get-next-request, set-request, get-response, and trap. The first three are sent from the manager to the agent and the last two are sent by the agent to the manager.

The SNMP communication model deals with the administrative structure and the five SNMP message PDUs. The administrative model defines the community within which messages can be exchanged. It also defines the access policy as to who has access privilege to what data. The five protocol entities are defined in ASN.1 format and macros. We learned SNMP operations by tracing messages exchanged between manager and agent processes. We then looked inside PDU formats for various messages to learn the data formats.

There is no formal specification for the functional model in SNMP management. However, management functions are accomplished by built-in schemes and managed objects. The administrative model in SNMP and the operations using managed objects are employed to accomplish various functions.

Exercises

1. Three managed hubs with interface id 11–13 (fourth decimal position value) in subnetwork 200.100.100.1 are being monitored by a network management system (NMS) for mean time between failures using the *SysUpTime* in *system* {*internet. mgmt.mib-2.system*} group. The NMS periodically issues the command get-request *object-instance community OBJECT IDENTIFIER* Fill the operands in the three set of requests that the NMS sends out. Use “public” for the *community* variable.
2. You are assigned the task of writing specifications for configuring SNMP managers and agents for a corporate network to implement the access policy. The policy defines a community profile for all managed network components where a public group (community name *public*) can only look at the System group, a privileged group (community name *privileged*) that can look at all the MIB objects, and an exclusive group (community name *exclusive*) that can do a read-write on all allowed components. Present a figure (similar, but not identical, to the flow chart shown in Figure 5.2) showing the paths from the SNMP managers to managed objects of a network component.
3. Fill in the data in the trap PDU format shown in Figure 5.9 for a message sent by the hub shown in Figure 4.2(a) one second after it is reset following a failure. Treat the trap as generic and leave the specific trap field blank. The only *varBind* that the trap sends is *sysUpTime*. (Refer to RFC 1157 and RFC 1215.)
4. An SNMP manager sends a request message to an SNMP agent requesting *sysUpTime* at 8:00 A.M. Fill in the data for the fields of an SNMP PDU shown in Figure 5.5. Please use “SNMP” for the application header, enumerated INTEGER 0 for version-1, and “public” for community name.
5. In Exercise 4, if the SNMP manager sent the request at 8:00 A.M. and the SNMP agent was reset at midnight after a failure, fill in the fields for the SNMP PDU on the response received.
6. An SNMP manager sends a request for the values of the *sysUpTime* in the System group and *if Type* in the Interfaces group for *ifNumber* value of 3. Write the PDUs with the fields filled in for

   (a) the get-request PDU, and
   (b) the get-response PDU with noSuchName error message for *ifType*

7. The following data response information is received by the manager for a get-request with a *varBindList*. Compose

   (a) the get-request PDU, and
   (b) the get-response PDU

OBJECT	VALUE
Error Status	Too big
Error Index	udpInErrors
udpInDatagrams	500,000
udpNoPorts	1,000
udpInErrors	5000
udpOutDatagrams	300,000

8. Draw the message sequence diagram similar to the one shown in Figure 5.10 for the hub example given in Figure 4.2(a). Assume that a separate get-request message is sent for each data value.
9. Repeat Exercise 7 with a VarBindList. Use the format of Figure 5.16.
10. For the UDP Group MIB shown in Figure 4.39, assume that there are three rows for the columnar objects in the *udpTable*. Write the OBJECT IDENTIFIER for all the objects in lexicographic order.
11. Draw the message sequence diagram for the following *ipNetToMediaTable* retrieving all the values of objects in each row with single get-next-request commands, similar to the one shown in Figure 5.16. The indices are *ipNetToMediaIfIndex* and *ipNetToMediaNetAddress*. Ignore obtaining *sysUpTime*.

ipNetToMedia IfIndex	IpNetToMediaPhys Address	ipNetToMediaNet Address	ipNetTo MediaType
25	00000C3920B4	192.68.252.15	4
16	00000C3920AF	172.16.49.1	4
9	00000C3920A6	172.16.55.1	4
2	00000C39209D	172.16.56.1	4

12. Compose data frames for SNMP PDUs for the example shown in Figure 5.16 for the following two cases:

    (a) The first GetNextRequest (*sysUpTime*, *atPhysAddress*) and the GetResponse.
    (b) The second GetNextRequest and GetResponse with values obtained in (a).

13. A data analyzer tool is used to look at a frame of data traversing a LAN. It is from the station *noc3* in response to a request from *noc1*. Use the following system status to answer this question.

    Version = 0
    Community = netman

OBJECT	VALUE	UNITS
Request ID	100	
Error Status	Too big	udpInErrors too high
Error Index	udpInErrors	
sysUpTime	1,000,000	hundredths of a second
udpInDatagrams	500,000	datagrams
udpNoPorts	1,000	datagrams
udpInErrors	5000	datagrams
udpOutDatagrams	300,000	datagrams

Compose the expected data frames for SNMP PDU types. Your frames should look like the ones shown in Figure 5.17.

(a) Get Request from the manager to the managed object.
(b) Get Response from the managed object to the manager.