27.1.5 Accounting Management

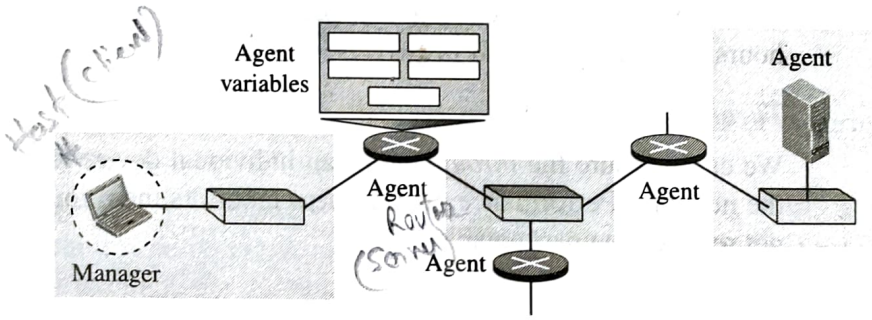
Accounting management is the controlling of users' access to network resources through charges. Under accounting management, individual users, departments, divisions, or even projects are charged for the services they receive from the network. Charging does not necessarily mean cash transfer; it may mean debiting the departments or divisions for budgeting purposes. Today, organizations use an accounting management system for the following reasons:

- It prevents users from monopolizing limited network resources.
- It prevents users from using the system inefficiently.
- Network managers can do short- and long-term planning based on the demand for network use.

27.2 SNMP

Several network management standards have been devised during the last few decades. The most important one is **Simple Network Management Protocol (SNMP),** used by the Internet. We discuss this standard in this section. SNMP is a framework for managing devices in an internet using the TCP/IP protocol suite. It provides a set of fundamental operations for monitoring and maintaining an internet. SNMP uses the concept of manager and agent. That is, a manager, usually a host, controls and monitors a set of agents, usually routers or servers (see Figure 27.2).

Figure 27.2 *SNMP concept*



SNMP is an application-level protocol in which a few manager stations control a set of agents. The protocol is designed at the application level so that it can monitor devices made by different manufacturers and installed on different physical networks. In other words, SNMP frees management tasks from both the physical characteristics of the managed devices and the underlying networking technology. It can be used in a heterogeneous internet made of different LANs and WANs connected by routers made by different manufacturers.

27.2.1 Managers and Agents

A management station, called a *manager*, is a host that runs the SNMP client program. A managed station, called an *agent*, is a router (or a host) that runs the SNMP server program. Management is achieved through simple interaction between a manager and an agent.

The agent keeps performance information in a database. The manager has access to the values in the database. For example, a router can store in appropriate variables the number of packets received and forwarded. The manager can fetch and compare the values of these two variables to see if the router is congested or not.

The manager can also make the router perform certain actions. For example, a router periodically checks the value of a reboot counter to see when it should reboot itself. It reboots itself, for example, if the value of the counter is 0. The manager can use this feature to reboot the agent remotely at any time. It simply sends a packet to force a 0 value in the counter.

Agents can also contribute to the management process. The server program running on the agent can check the environment and, if it notices something unusual, it can send a warning message (called a *Trap*) to the manager.

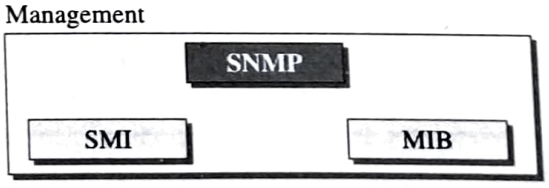
In other words, management with SNMP is based on three basic ideas:

1. A manager checks an agent by requesting information that reflects the behavior of the agent.
2. A manager forces an agent to perform a task by resetting values in the agent database.
3. An agent contributes to the management process by warning the manager of an unusual situation.

27.2.2 Management Components

To do management tasks, SNMP uses two other protocols: **Structure of Management Information** (SMI) and **Management Information Base** (MIB). In other words, management on the Internet is done through the cooperation of three protocols: SNMP, SMI, and MIB, as shown in Figure 27.3.

Figure 27.3 *Components of network management on the Internet*



Let us elaborate on the interactions between these protocols.

Role of SNMP

SNMP has some very specific roles in network management. It defines the format of the packet to be sent from a manager to an agent and vice versa. It also interprets the result and creates statistics (often with the help of other management software). The

packets exchanged contain the object (variable) names and their status (values). SNMP is responsible for reading and changing these values.

SNMP defines the format of packets exchanged between a manager and an agent. It reads and changes the status of objects (values of variables) in SNMP packets.

Role of SMI

To use SNMP, we need rules for naming objects. This is particularly important because the objects in SNMP form a hierarchical structure (an object may have a parent object and some child objects). Part of a name can be inherited from the parent. We also need rules to define the types of objects. What types of objects are handled by SNMP? Can SNMP handle simple types or structured types? How many simple types are available? What are the sizes of these types? What is the range of these types? In addition, how are each of these types encoded?

SMI defines the general rules for naming objects, defining object types (including range and length), and showing how to encode objects and values.

We need these universal rules because we do not know the architecture of the computers that send, receive, or store these values. The sender may be a powerful computer in which an integer is stored as 8-byte data; the receiver may be a small computer that stores an integer as 4-byte data.

SMI is a protocol that defines these rules. However, we must understand that SMI only defines the rules; it does not define how many objects are managed in an entity or which object uses which type. SMI is a collection of general rules to name objects and to list their types. The association of an object with the type is not done by SMI.

Role of MIB

We hope it is clear that we need another protocol. For each entity to be managed, this protocol must define the number of objects, name them according to the rules defined by SMI, and associate a type to each named object. This protocol is MIB. MIB creates a set of objects defined for each entity in a manner similar to that of a database (mostly metadata in a database, names and types without values).

MIB creates a collection of named objects, their types, and their relationships to each other in an entity to be managed.

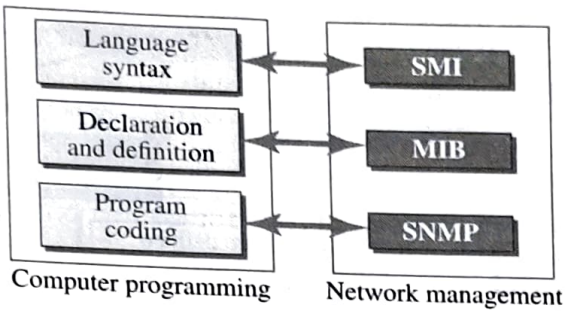
An Analogy

Before discussing each of these protocols in more detail, let us give an analogy. The three network management components are similar to what we need when we write a program in a computer language to solve a problem. Figure 27.4 shows the analogy.

Syntax: SMI

Before we write a program, the syntax of the language (such as C or Java) must be predefined. The language also defines the structure of variables (simple, structured, pointer, and so on) and how the variables must be named. For example, a variable name

Figure 27.4 *Comparing computer programming and network management*



must be 1 to *n* characters in length and start with a letter followed by alphanumeric characters. The language also defines the type of data to be used (integer, real, character, etc.). In programming, the rules are defined by the syntax of the language. In network management, the rules are defined by SMI.

Object Declaration and Definition: MIB

Most computer languages require that objects be declared and defined in each specific program. Declaration and definition create objects using predefined types and allocate memory location for them. For example, if a program has two variables (an integer named *counter* and an array named *grades* of type char), they must be declared at the beginning of the program:

```
int counter;
char grades [40];
```

MIB does this task in network management. MIB names each object and defines the type of the objects. Because the type is defined by SMI, SNMP knows the range and size.

Program Coding: SNMP

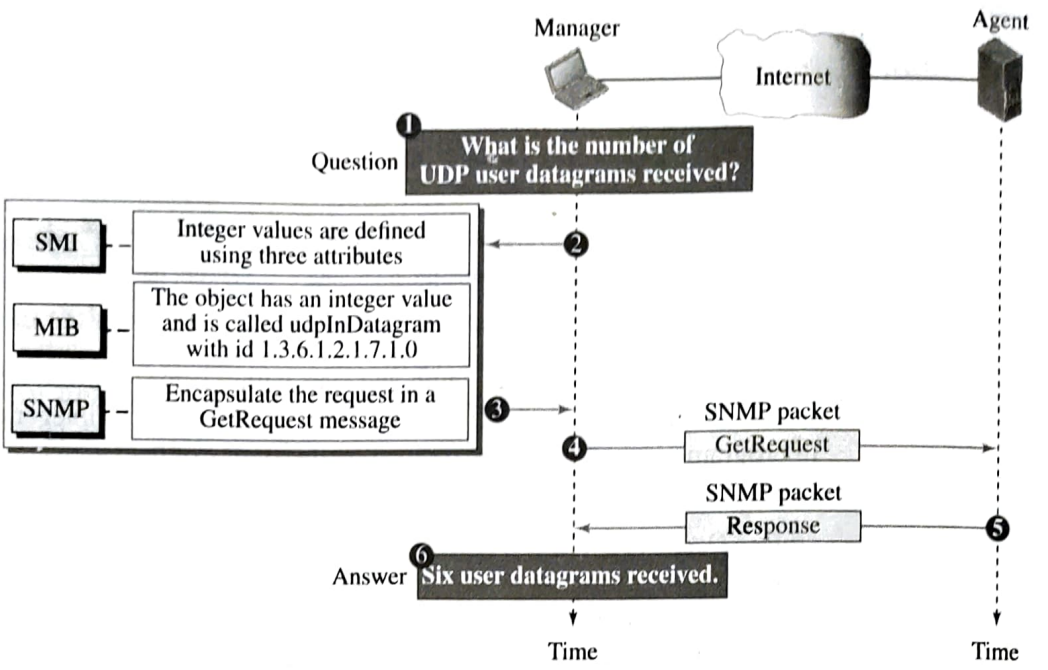
After declaration in programming, the program needs to write statements to store values in the variables and change them if needed. SNMP does this task in network management. SNMP stores, changes, and interprets the values of objects already declared by MIB according to the rules defined by SMI.

27.2.3 An Overview

Before discussing each component in more detail, let us show how each of these components is involved in a simple scenario. This is an overview that will be developed later, at the end of the chapter. A manager station (SNMP client) wants to send a message to an agent station (SNMP server) to find the number of UDP user datagrams received by the agent. Figure 27.5 shows an overview of steps involved.

MIB is responsible for finding the object that holds the number of UDP user datagrams received. SMI, with the help of another embedded protocol, is responsible for encoding the name of the object. SNMP is responsible for creating a message, called a GetRequest message, and encapsulating the encoded message. Of course, things are more complicated than this simple overview, but we first need more details of each protocol.

Figure 27.5 *Management overview*



27.2.4 SMI

The Structure of Management Information, version 2 (SMIv2) is a component for network management. SMI is a guideline for SNMP. It emphasizes three attributes to handle an object: name, data type, and encoding method. Its functions are:

- To name objects.
- To define the type of data that can be stored in an object.
- To show how to encode data for transmission over the network.

Name

SMI requires that each managed object (such as a router, a variable in a router, a value, etc.) have a unique name. To name objects globally, SMI uses an **object identifier,** which is a hierarchical identifier based on a tree structure (see Figure 27.6).

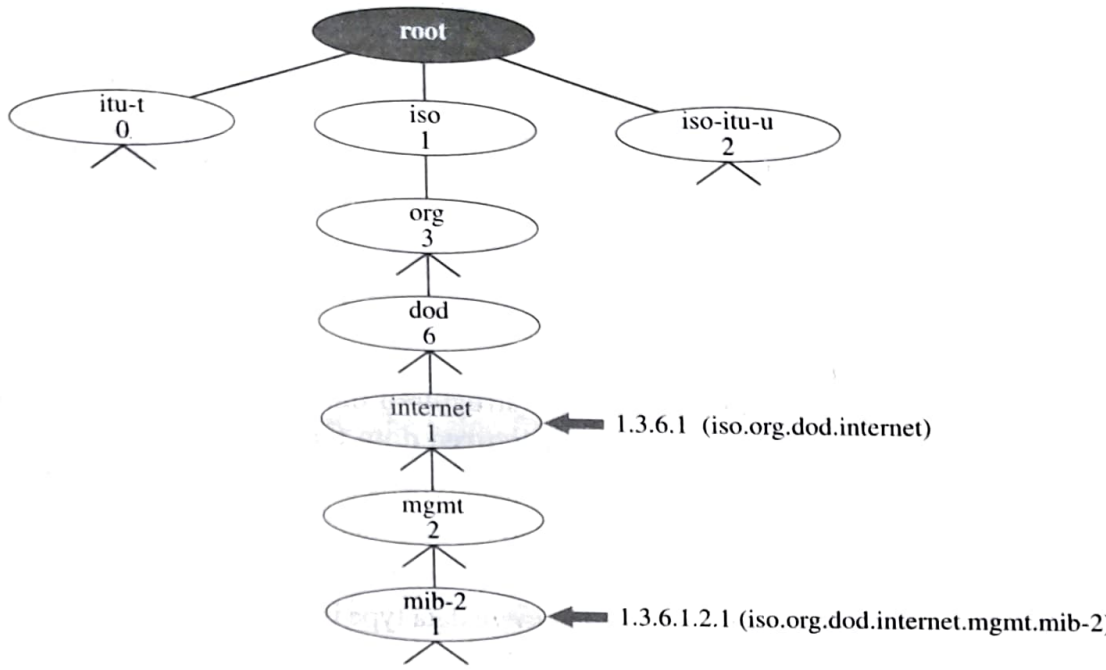
The tree structure starts with an unnamed root. Each object can be defined using a sequence of integers separated by dots. The tree structure can also define an object using a sequence of textual names separated by dots.

The integer-dot representation is used in SNMP. The name-dot notation is used by people. For example, the following shows the same object in two different notations.

iso.org.dod.internet.mgmt.mib-2 ↔ **1.3.6.1.2.1**

The objects that are used in SNMP are located under the *mib-2* object, so their identifiers always start with 1.3.6.1.2.1.

Figure 27.6 *Object identifier in SMI*



Type

The second attribute of an object is the type of data stored in it. To define the data type, SMI uses **Abstract Syntax Notation One (ASN.1)** definitions and adds some new definitions. In other words, SMI is both a subset and a superset of ASN.1. (We discuss ASN.1 in the last section of this chapter.)

SMI has two broad categories of data types: *simple* and *structured*. We first define the simple types and then show how the structured types can be constructed from the simple ones.

Simple Type

The **simple data types** are atomic data types. Some of them are taken directly from ASN.1; some are added by SMI. The most important ones are given in Table 27.1. The first five are from ASN.1; the next seven are defined by SMI.

Table 27.1 *Data types*

Type	*Size*	*Description*
INTEGER	4 bytes	An integer with a value between -2^{31} and $2^{31}-1$
Integer32	4 bytes	Same as INTEGER
Unsigned32	4 bytes	Unsigned with a value between 0 and $2^{32}-1$
OCTET STRING	Variable	Byte-string up to 65,535 bytes long
OBJECT IDENTIFIER	Variable	An object identifier
IPAddress	4 bytes	An IP address made of four integers
Counter32	4 bytes	An integer whose value can be incremented from zero to 2^{32}; when it reaches its maximum value it wraps back to zero

Table 27.1 *Data types (continued)*

Type	*Size*	*Description*
Counter64	8 bytes	64-bit counter
Gauge32	4 bytes	Same as Counter32, but when it reaches its maximum value, it does not wrap; it remains there until it is reset
TimeTicks	4 bytes	A counting value that records time in 1/100ths of a second
BITS		A string of bits
Opaque	Variable	Uninterpreted string

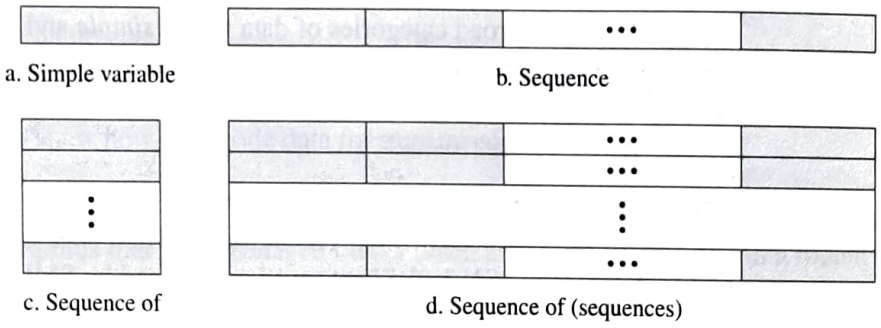
Structured Type

By combining simple and structured data types, we can make new structured data types. SMI defines two **structured data types:** *sequence* and *sequence of*.

- ***Sequence.*** A *sequence* data type is a combination of simple data types, not necessarily of the same type. It is analogous to the concept of a *struct* or a *record* used in programming languages such as C.
- ***Sequence of.*** A *sequence of* data type is a combination of simple data types all of the same type or a combination of sequence data types all of the same type. It is analogous to the concept of an *array* used in programming languages such as C.

Figure 27.7 shows a conceptual view of data types.

Figure 27.7 *Conceptual data types*

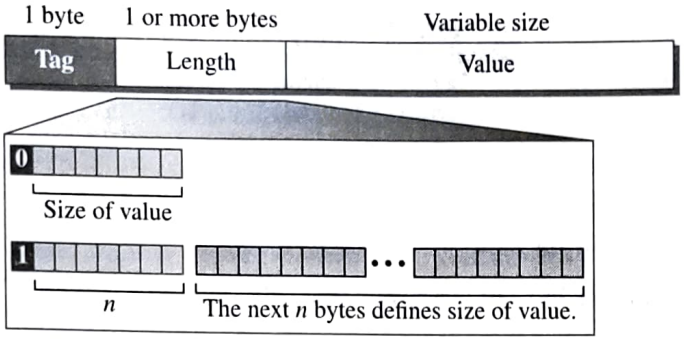


Encoding Method

SMI uses another standard, **Basic Encoding Rules (BER),** to encode data to be transmitted over the network. BER specifies that each piece of data be encoded in triplet format: tag, length, and value (TLV), as illustrated in Figure 27.8.

The tag is a 1-byte field that defines the type of data. Table 27.2 shows the data types we use in this chapter and their tags in hexadecimal numbers. The length field is 1 or more bytes. If it is 1 byte, the most significant bit must be 0. The other 7 bits define the length of the data. If it is more than 1 byte, the most significant bit of the first byte must be 1. The other 7 bits of the first byte specify the number of bytes

Figure 27.8 *Encoding format*



needed to define the length. The value field codes the value of the data according to the rules defined in BER.

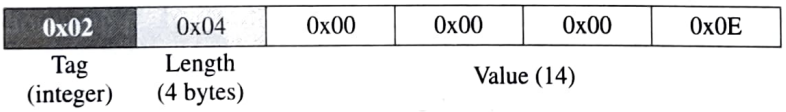
Table 27.2 *Codes for data types*

Data Type	*Tag (Hex)*	*Data Type*	*Tag (Hex)*
INTEGER	**02**	IPAddress	**40**
OCTET STRING	**04**	Counter	**41**
OBJECT IDENTIFIER	**06**	Gauge	**42**
NULL	**05**	TimeTicks	**43**
SEQUENCE, SEQUENCE OF	**30**	Opaque	**44**

Example 27.1

Figure 27.9 shows how to define INTEGER 14. The size of the length field is from Table 27.1.

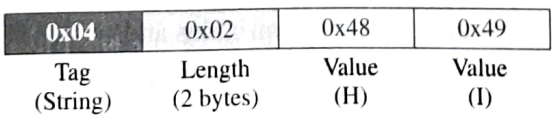
Figure 27.9 *Example 27.1: INTEGER 14*



Example 27.2

Figure 27.10 shows how to define the OCTET STRING "HI."

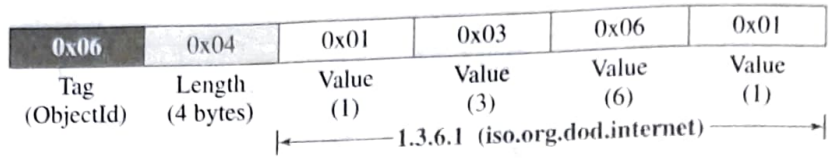
Figure 27.10 *Example 27.2: OCTET STRING "HI"*



Example 27.3

Figure 27.11 shows how to define ObjectIdentifier 1.3.6.1 (iso.org.dod.internet).

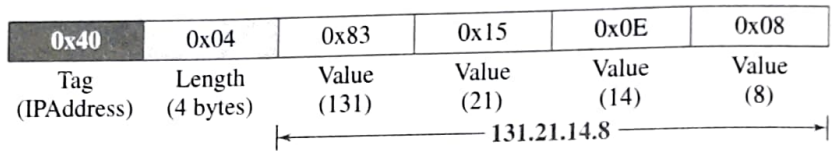
Figure 27.11 *Example 27.3: ObjectIdentifier 1.3.6.1*



Example 27.4

Figure 27.12 shows how to define IPAddress 131.21.14.8.

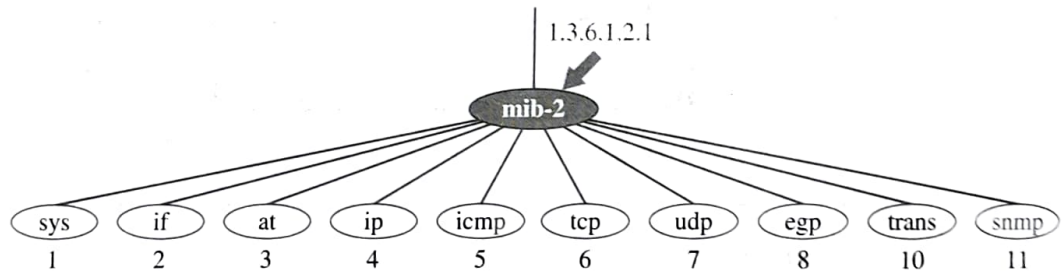
Figure 27.12 *Example 27.4: IPAddress 131.21.14.8*



27.2.5 MIB

The Management Information Base, version 2 (MIB2) is the second component used in network management. Each agent has its own MIB2, which is a collection of all the objects that the manager can manage. (See Figure 27.13.)

Figure 27.13 *Some mib-2 groups*



The objects in MIB2 are categorized under several groups: system, interface, address translation, ip, icmp, tcp, udp, egp, transmission, and snmp (note that group 9 is deprecated). These groups are under the mib-2 object in the object identifier tree. Each group has defined variables and/or tables.

The following is a brief description of some of the objects:

- **sys** This object (*system*) defines general information about the node (system), such as the name, location, and lifetime.
- **if** This object (*interface*) defines information about all of the interfaces of the node including interface number, physical address, and IP address.

- **at** This object (*address translation*) defines the information about the ARP table.
- **ip** This object defines information related to IP, such as the routing table and the IP address.
- **icmp** This object defines information related to ICMP, such as the number of packets sent and received and total errors created.
- **tcp** This object defines general information related to TCP, such as the connection table, time-out value, number of ports, and number of packets sent and received.
- **udp** This object defines general information related to UDP, such as the number of ports and number of packets sent and received.
- **egp** These objects are related to the operation of EGP.
- **trans** These objects are related to the specific method of transmission (future use).
- **snmp** This object defines general information related to SNMP itself.

Accessing MIB Variables

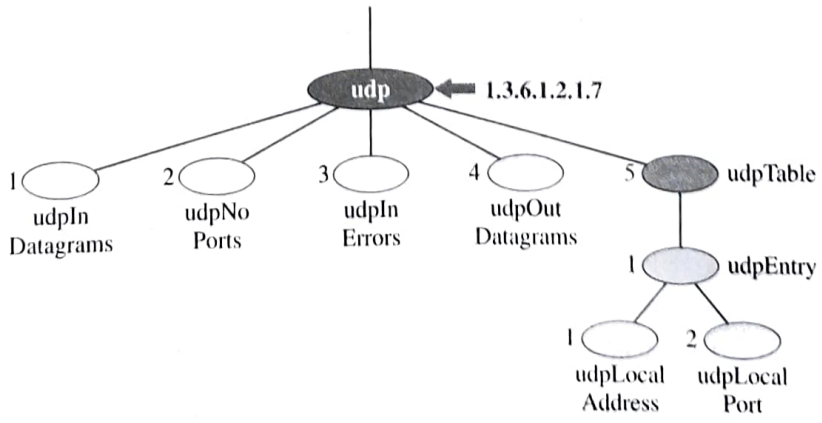
To show how to access different variables, we use the udp group as an example. There are four simple variables in the udp group and one sequence of (table of) records. Figure 27.14 shows the variables and the table. We will show how to access each entity.

Simple Variables

To access any of the simple variables, we use the id of the group (1.3.6.1.2.1.7) followed by the id of the variable. The following shows how to access each variable.

udpInDatagrams	→	**1.3.6.1.2.1.7.1**
udpNoPorts	→	**1.3.6.1.2.1.7.2**
udpInErrors	→	**1.3.6.1.2.1.7.3**
udpOutDatagrams	→	**1.3.6.1.2.1.7.4**

Figure 27.14 *udp group*



However, these object identifiers define the variable, not the instance (contents). To show the instance, or the contents, of each variable, we must add an instance suffix. The instance suffix for a simple variable is simply a zero. In other words, to show an instance of the above variables, we use the following:

udpInDatagrams.0 → 1.3.6.1.2.1.7.1.0
udpNoPorts.0 → 1.3.6.1.2.1.7.2.0
udpInErrors.0 → 1.3.6.1.2.1.7.3.0
udpOutDatagrams.0 → 1.3.6.1.2.1.7.4.0

Tables

To identify a table, we first use the table id. The udp group has only one table (with id 5), as illustrated in Figure 27.15. So to access the table, we use the following:

udpTable → 1.3.6.1.2.1.7.5

However, the table is not at the leaf level in the tree structure. We cannot access the table; we define the entry (sequence) in the table (with id of 1), as follows:

udpEntry → 1.3.6.1.2.1.7.5.1

This entry is also not a leaf and we cannot access it. We need to define each entity (field) in the entry.

udpLocalAddress → 1.3.6.1.2.1.7.5.1.1
udpLocalPort → 1.3.6.1.2.1.7.5.1.2

These two variables are at the leaf level of the tree. Although we can access their instances, we need to define *which* instance. At any moment, the table can have several values for each local address/local port pair. To access a specific instance (row) of the table, we add the index to the above ids. In MIB, the indexes of arrays are not integers (unlike most programming languages). The indexes are based on the value of one or more fields in the entries. In our example, the udpTable is indexed based on both the local address and the local port number. For example, Figure 27.16 shows a table with four rows and values for each field. The index of each row is a combination of two values. To access the instance of the local address for the first row, we use the identifier augmented with the instance index:

udpLocalAddress.181.23.45.14.23 → 1.3.6.1.2.7.5.1.1.181.23.45.14.23

27.2.6 SNMP

SNMP uses both SMI and MIB in Internet network management. It is an application program that allows:

- A manager to retrieve the value of an object defined in an agent.
- A manager to store a value in an object defined in an agent.
- An agent to send an alarm message about an abnormal situation to the manager.

Figure 27.15 *udp variables and tables*

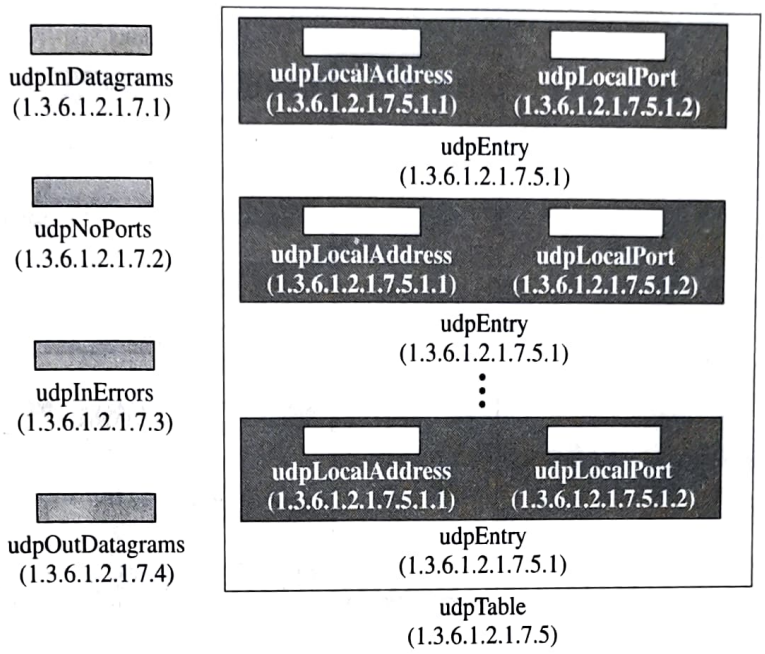
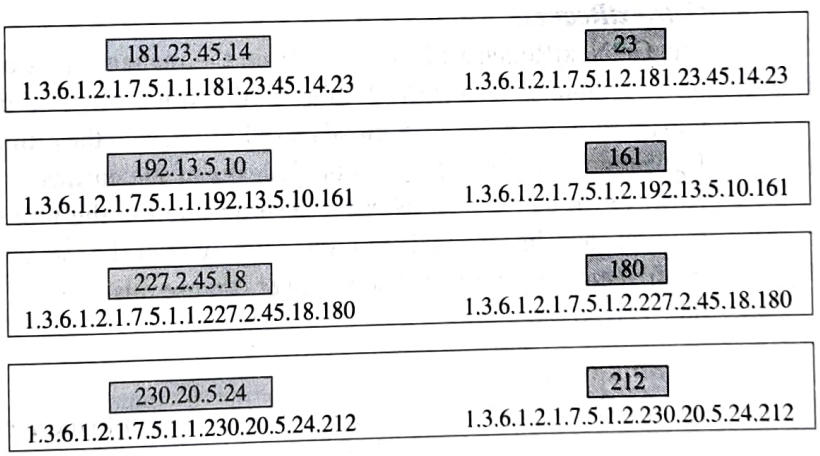


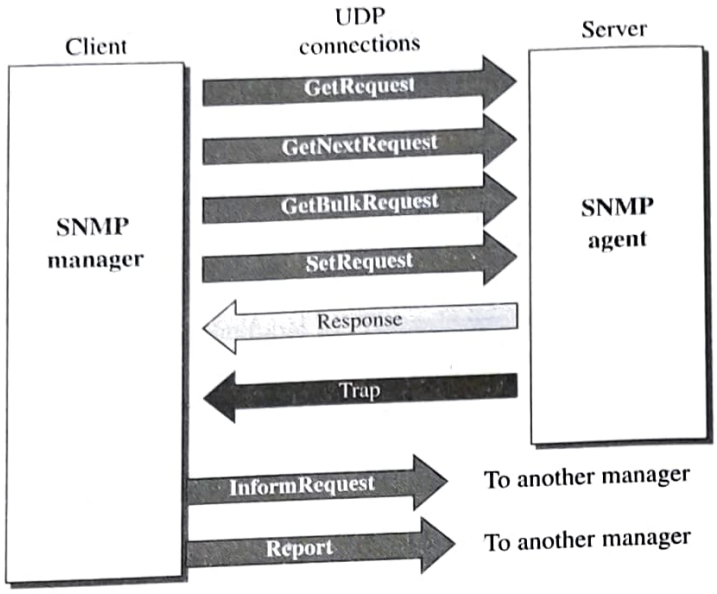
Figure 27.16 *Indexes for udpTable*



PDUs

SNMPv3 defines eight types of protocol data units (or PDUs): *GetRequest*, *GetNext-Request*, *GetBulkRequest*, *SetRequest*, *Response*, *Trap*, *InformRequest*, and *Report* (see Figure 27.17).

Figure 27.17 *SNMP PDUs*



GetRequest
The GetRequest PDU is sent from the manager (client) to the agent (server) to retrieve the value of a variable or a set of variables.

GetNextRequest
The GetNextRequest PDU is sent from the manager to the agent to retrieve the value of a variable. The retrieved value is the value of the object following the defined ObjectId in the PDU. It is mostly used to retrieve the values of the entries in a table. If the manager does not know the indexes of the entries, it cannot retrieve the values. However, it can use GetNextRequest and define the ObjectId of the table. Because the first entry has the ObjectId immediately after the ObjectId of the table, the value of the first entry is returned. The manager can use this ObjectId to get the value of the next one, and so on.

GetBulkRequest
The GetBulkRequest PDU is sent from the manager to the agent to retrieve a large amount of data. It can be used instead of multiple GetRequest and GetNextRequest PDUs.

SetRequest
The SetRequest PDU is sent from the manager to the agent to set (store) a value in a variable.

Response
The Response PDU is sent from an agent to a manager in response to GetRequest or GetNextRequest. It contains the value(s) of the variable(s) requested by the manager.

Trap

The **Trap** (also called *SNMPv2 Trap* to distinguish it from SNMPv1 Trap) PDU is sent from the agent to the manager to report an event. For example, if the agent is rebooted, it informs the manager and reports the time of rebooting.

InformRequest

The InformRequest PDU is sent from one manager to another remote manager to get the value of some variables from agents under the control of the remote manager. The remote manager responds with a Response PDU.

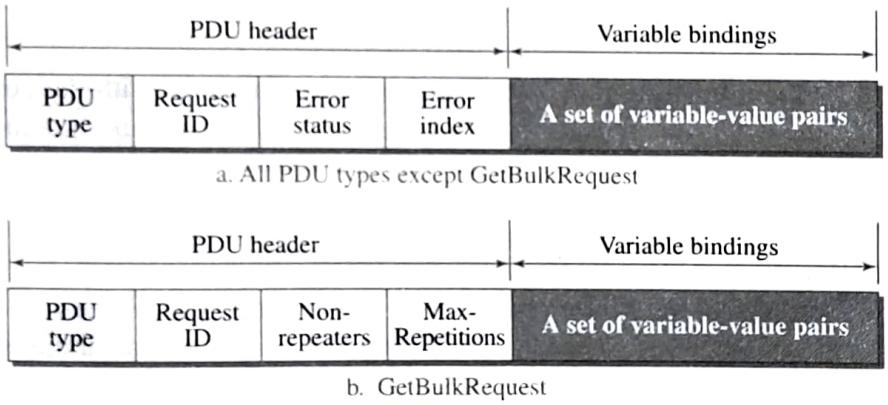
Report

The Report PDU is designed to report some types of errors between managers. It is not yet in use.

Format

The format for the eight SNMP PDUs is shown in Figure 27.18. The GetBulkRequest PDU differs from the others in two areas, as shown in the figure.

Figure 27.18 *SNMP PDU format*



The fields are listed below:

- **PDU type.** This field defines the type of the PDU (see Table 27.3) .

Table 27.3 *PDU types*

Type	*Tag (Hex)*	*Type*	*Tag (Hex)*
GetRequest	**A0**	GetBulkRequest	**A5**
GetNextRequest	**A1**	InformRequest	**A6**
Response	**A2**	Trap (SNMPv2)	**A7**
SetRequest	**A3**	Report	**A8**

- **Request ID.** This field is a sequence number used by the manager in a request PDU and repeated by the agent in a response. It is used to match a request to a response.

- **Error status.** This is an integer that is used only in response PDUs to show the types of errors reported by the agent. Its value is 0 in request PDUs. Table 27.4 lists the types of errors that can occur.

Table 27.4 *Types of errors*

Status	*Name*	*Meaning*
0	noError	No error
1	tooBig	Response too big to fit in one message
2	noSuchName	Variable does not exist
3	badValue	The value to be stored is invalid
4	readOnly	The value cannot be modified
5	genErr	Other errors

- **Non-repeaters.** This field is used only in a GetBulkRequest PDU. The field defines the number of non-repeating (regular objects) at the start of the variable-value list.
- **Error index.** The error index is an offset that tells the manager which variable caused the error.
- **Max-repetitions.** This field is also used only in a GetBulkRequest PDU. The field defines the maximum number of iterations in the table to read all repeating objects.
- **Variable-value pair list.** This is a set of variables with the corresponding values the manager wants to retrieve or set. The values are null in request PDUs.

Messages

SNMP does not send only PDUs, it embeds each PDU in a message. A message is made of a message header followed by the corresponding PDU, as shown in Figure 27.19. The format of the message header, which depends on the version and the security provision, is not shown in the figure. We leave the details to some specific text.

Example 27.5

In this example, a manager station (SNMP client) uses a message with a GetRequest PDU to retrieve the number of UDP datagrams that a router has received (Figure 27.20).

There is only one Varbind sequence. The corresponding MIB variable related to this information is udpInDatagrams with the object identifier 1.3.6.1.2.1.7.1.0. The manager wants to retrieve a value (not to store a value), so the value defines a null entity. The bytes to be sent are shown in hexadecimal representation.

The Varbind list has only one Varbind. The variable is of type 06 and length 09. The value is of type 05 and length 00. The whole Varbind is a sequence of length 0D (13). The Varbind list is also a sequence of length 0F (15). The GetRequest PDU is of length ID (29).

Note that we have intended the bytes to show the inclusion of simple data types inside a sequence or the inclusion of sequences and simple data types inside larger sequences. Note that the PDU itself is like a sequence, but its tag is A0 in hexadecimal.

Figure 27.19 *SNMP message*

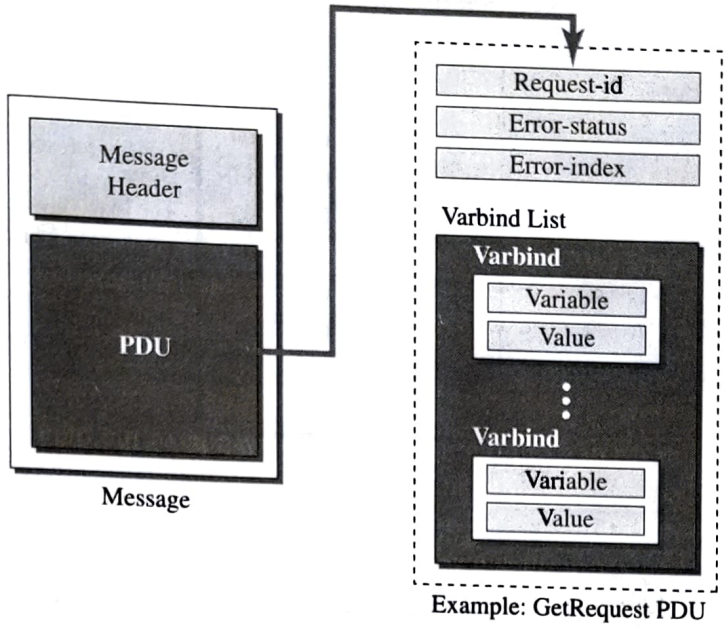


Figure 27.20 *Example 27.5*

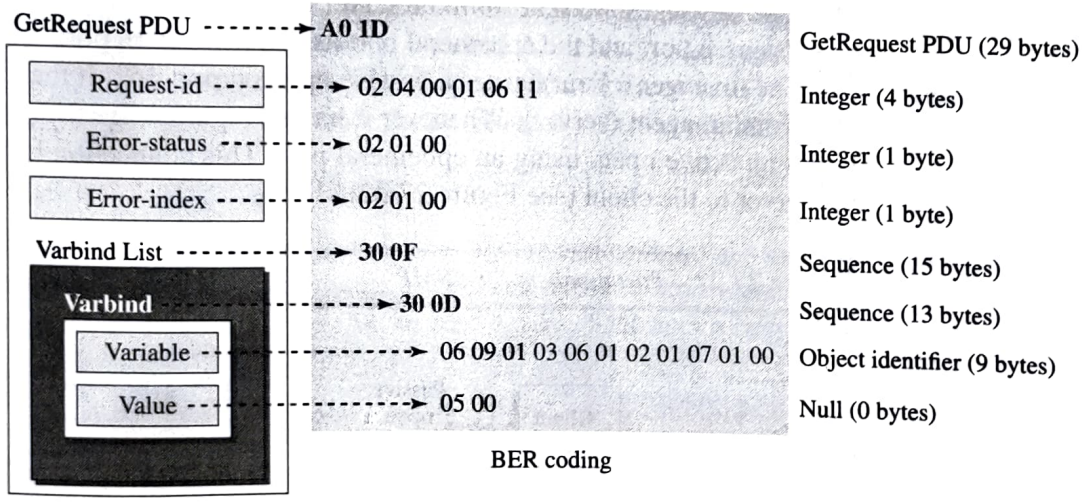


Figure 27.21 shows the actual message sent. We assume that the message header is made of 10 bytes. The actual message header may be different. We show the message using rows of 4 bytes.The bytes that are shown using dashes are the ones related to the message header.

Figure 27.21 *Actual message sent for Example 27.5*

30	29	--	--
--	--	--	--
--	--	--	--
A0	1D	02	04
00	01	06	11
02	01	00	02
01	00	30	0F
30	0D	06	09
01	03	06	01
02	01	07	01
00	05	00	

Message

Note:
The byte values are in hexadecimal.

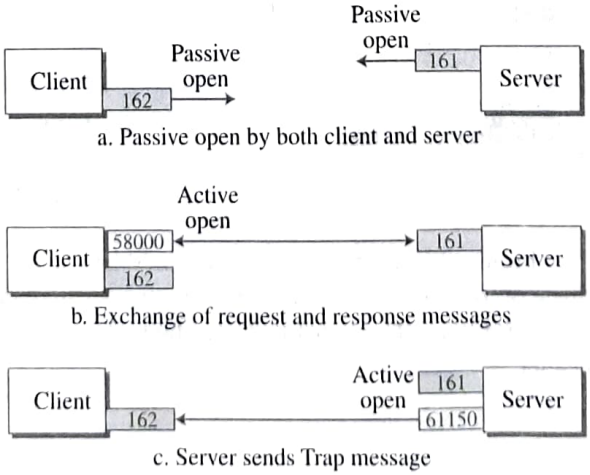
UDP Ports

SNMP uses the services of UDP on two well-known ports, 161 and 162. The well-known port 161 is used by the server (agent), and the well-known port 162 is used by the client (manager).

The agent (server) issues a passive open on port 161. It then waits for a connection from a manager (client). A manager (client) issues an active open using an ephemeral port. The request messages are sent from the client to the server using the ephemeral port as the source port and the well-known port 161 as the destination port. The response messages are sent from the server to the client using the well-known port 161 as the source port and the ephemeral port as the destination port.

The manager (client) issues a passive open on port 162. It then waits for a connection from an agent (server). Whenever it has a Trap message to send, an agent (server) issues an active open, using an ephemeral port. This connection is only one-way, from the server to the client (see Figure 27.22).

Figure 27.22 *Port numbers for SNMP*



The client-server mechanism in SNMP is different from other protocols. Here both the client and the server use well-known ports. In addition, both the client and the server are running infinitely. The reason is that request messages are initiated by a manager (client), but Trap messages are initiated by an agent (server).

Security

SNMPv3 has added two new features to the previous version: security and remote administration. SNMPv3 allows a manager to choose one or more levels of security when accessing an agent. Different aspects of security can be configured by the manager to allow message authentication, confidentiality, and integrity.

SNMPv3 also allows remote configuration of security aspects without requiring the administrator to actually be at the place where the device is located.

27.3 ASN.1

In data communication, when we send a continuous stream of bits to a destination, we somehow need to define the format of the data. If we send a *name* and a *number* in a single message, we need to tell the destination that, for example, the first 12 bits define the name and the next 8 bits define the number. We will have more difficulty when we send a complex data type such as an array or a record. For example, in the case of an array, if we send 2000 bits in a message, we need to tell the receiver that it is an array of 200 numbers each of 10 bits or it is an array of 10 numbers each of 200 bits.

A solution is that we separate the definition of data types from the sequence of bits transmitted through the network. This is done through an abstract language that uses some symbols, key words, and atomic data types and lets us make new data types out of the simple types. The language is called Abstract Syntax Notation One (ASN.1). Note that ASN.1 is a very complex language used in different areas of computer science, but in this section, we only introduce the language as much as needed for the SNMP protocol.

27.3.1 Language Basics

Before we show how we can define objects and associated values, let us talk about the language itself. The language uses some symbols and some key words and defines some primitive data types. As we said before, SMI uses a subset of these entities in its own language.

Symbols

The language uses a set of symbols, given in Table 27.5. Some of these symbols are single characters, but some are pairs of characters.

Table 27.5 *Symbols used in ASN.1*

Symbol	*Meaning*	*Symbol*	*Meaning*
::=	Defined as or assignment	..	Range
|	Or, alternative, or option	{ }	Start and end of a list
–	Negative sign	[ ]	Start and end of tag
—	The following is a comment	( )	Start and end of a subtype