Network Management Tools, Systems, and Engineering

OBJECTIVES

- *System utilities for management*
  - *Basic networking tools*
  - *SNMP tools*
  - *Protocol analyzer*
- *Measurement of network statistics*
  - *Traffic load*
  - *Protocol*
  - *Data*
  - *Error*
  - *Traffic monitoring using MRTG*
- *MIB engineering*
  - *Limitations of SMI and SNMP*
  - *Counters and rates*
  - *Object-oriented design*
  - *SMI tables*
  - *SMI actions*
  - *Guiding principles for MIB design*
- *Design considerations of NMS*
  - *System and user requirements*
  - *Local and remote management*
  - *Functional requirements*
  - *Architectural aspects*
  - *Key design considerations*
  - *Functional modules and managers*
  - *Root cause analysis*
  - *Distributed network management*
  - *Server and client platforms*
- *Network management systems*
  - *Display of functional views of network management*
  - *Examples of commercial and open source NMSs*
  - *System and applications management*
  - *Enterprise management system*
  - *Telecommunications management system*

Thus far we have covered SNMP standards for IP network management. This includes protocols and Management Information Bases (MIBs). In this chapter we will discuss assorted tools and techniques that can be used for the management of networks using SNMP (and other management protocols).

In Section 9.1 we start with commonly available utilities that can be used for management. This is followed by a discussion of tools for gathering network statistics in Section 9.2. Next, in Section 9.3 we examine the design of MIBs (MIB Engineering), which is important for any vendor of networking equipment. In Section 9.4 we turn to the design of a typical network management system (NMS) server for a large telecom network. In Section 9.5 we outline several commercial and free NMS products for different types of networks.

9.1 SYSTEM UTILITIES FOR MANAGEMENT

A significant amount of network management can be done using operating system (OS) utilities and some freely downloadable SNMP tools. These can be put together quickly using simple scripting languages such as Perl [Cozens; Lidie and Walsh, 2002]. Some of these tools are described below.

9.1.1 Basic Tools

Numerous basic tools are either a part of the OS or are available as add-on applications that aid in obtaining network parameters or in the diagnosis of network problems. We will describe some of the more popular ones here under the three categories of status monitoring, traffic monitoring, and route monitoring.

Network Status Tools. Table 9.1 lists some of the network status-monitoring tools that are available in the Linux/UNIX and Microsoft Windows (XP and Vista) environments. We use Linux and UNIX interchangeably in this section as the same set of basic utilities and tools are available on both. This also applies to Solaris, HP-UX, AIX, MacOS X, and Free BSD.

The command *ifconfig* on a UNIX system is used to assign an address to a network interface or to configure network interface parameters. Usually, one needs to be a super-user (su) in UNIX to set interface parameters. However, it can be used without options by any user to obtain the current configuration of the local system or of a remote system. In this and other commands, you may invoke *man command-name* to obtain the manual page of a command in a UNIX system. An example of *ifconfig* is shown in Figure 9.1.

Table 9.1 Status-Monitoring Tools

NAME	OPERATING SYSTEM	DESCRIPTION
ifconfig	Linux	Obtains and configures networking interface parameters and status
ping	Linux/Windows	Checks the status of node/host
nslookup	Linux/Windows	Looks up DNS for name–IP address translation
dig	Linux	Queries DNS server (supersedes nslookup)
host	Linux	Displays information on Internet hosts/domains

```
netman: ifconfig –a
lo0: flags=849<UP,LOOPBACK,RUNNING,MULTICAST> mtu 8232
        inet 127.0.0.1 netmask ff000000
hme0: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST>
        mtu 1500 inet 192.207.8.31 netmask ffffff00 broadcast 192.207.8.255
```

Figure 9.1 Example of Interface Configuration (*ifconfig*)

The option –a is for displaying all interfaces. As we can see, there are two interfaces, the loopback interface, lo0, and the Ethernet interface, hme0. Option –a provides information on whether the interface is up or down, what maximum transmission unit (mtu) it has, the Ethernet interface address, etc.

One of the most basic tools is *ping* (packet Internet groper). A frequent use of it is when an execution of a command on a remote station fails. As one of the first diagnostics, we want to ensure that the connection exists, for which the *ping* utility is executed to the remote host. If it fails, then there is a problem with the link. If it passes, then the trouble is with either the application or something else.

You have already used the *ping* tool in the Chapter 1 exercises. It is based on the ICMP echo_request message and is available in both UNIX and Microsoft Windows OSs. "Pinging" a remote IP address verifies that the destination node and the intermediate nodes have live connectivity and provides the round-trip delay time. By pinging multiple times, we can obtain the average time delay, as well as percentage loss of packets, which is a measure of throughput. This feature can be used to check the performance of the connection. There are numerous implementations of ping. Read the manual page for details on the specific implementation on each host.

The UNIX commands, *nslookup*, *host*, and *dig*, are useful for obtaining names and addresses on hosts and domain name servers. A domain name server provides the service of locating IP addresses. The *nslookup* tool is an interactive program for making queries to the domain name server for translating the host name to the IP address and vice versa. The command, by default, sends the query to the local domain name server, but any other server can also be specified. The command *dig* is a more powerful replacement for *nslookup*.

For example, the command *dig nimbus.tenet.res.in* on the host *beluga* yields the result shown in Figure 9.2.

An interpretation of the above result follows. The host, beluga, obtained the IP address of nimbus.tenet.res.in (203.199.255.4) from the domain's name server 203.199.255.3. This information is given in

```
[beluga]~> dig +nocomments nimbus.tenet.res.in
nimbus.tenet.res.in.  604800    IN    A     203.199.255.4
tenet.res.in.         604800    IN    NS    volcano.tenet.res.in.
tenet.res.in.         604800    IN    NS    lantana.tenet.res.in.
volcano.tenet.res.in. 604800    IN    A     203.199.255.3
;; Query time: 2 msec
;; SERVER: 203.199.255.3#53(203.199.255.3)
;; WHEN: Fri Mar  6 14:12:43 2009
;; MSG SIZE  rcvd: 149
[beluga]~>
```

Figure 9.2 Example of Network Address Translation (*dig*)

the first line of the output (the A-record). The *authority* for this information, i.e., the name server(s) for the domain, is given in subsequent NS-records. The name server that responded to this query is given in the comments at the end.

Instead of the host name, the IP address could also be used as the option parameter in the dig command. For example, the command dig +short –x 203.199.255.5 will return the information that this IP address is assigned to *lantana.tenet.res.in. dig* can give very verbose output. The options +*nocomments* and +*short* are used to suppress some of this.

dig or *nslookup* can help when one wants to find all the hosts on a local area network (LAN). This can be done using a broadcast ping on the LAN. We need to know the IP address to execute this command, which we may not know. However, if we know a host name on the LAN, we could obtain the IP address using *nslookup*.

The interactive command *host* can also be used to get the host name using the domain name server. However, with the appropriate security privilege, it can be used to get all the host names maintained by a domain name server. The *dig* and *host* utilities also provide additional data on the hosts, besides host names. Refer to the manual page for details.

Network Traffic-Monitoring Tools. Table 9.2 lists several traffic-monitoring tools. One of the tools is ping, which we have mentioned as a status-monitoring tool in Table 9.1. As we stated earlier, by repeatedly executing ping with a large repeat count (e.g., ping –c 100) and measuring how many of those were successfully received back, we can calculate the percentage of packet loss. Packet loss is a measure of throughput. The example in Figure 9.3 displays zero packet loss when five packets were transmitted and received. It also shows the round-trip packet transmission time.

Another useful tool, heavily based on ping, is called *bing* (point-to-point bandwidth ping). The *bing* utility determines the raw throughput of a link by calculating the difference in round-trip times for different packet sizes from each end of the link. For example, if we want to measure the throughput of a hop or point-to-point link between L1 and L2, we derive it from the results of the measurements of ICMP echo requests to L1 and L2. The difference between the two results yields the throughput of the link L1–L2. This method has the advantage of yielding accurate results, even though the path to the link L1–L2 from the measuring station could have a lower bandwidth than the link L1–L2 for which the measurement is

Table 9.2 Traffic-Monitoring Tools

NAME	OPERATING SYSTEM	DESCRIPTION
ping	UNIX/Windows	Used for measuring round-trip packet loss
bing	UNIX	Measures point-to-point bandwidth of a link
tcpdump	UNIX	Dumps traffic on a network
getethers	UNIX	Acquires all host addresses of an Ethernet LAN segment
iptrace	UNIX	Measures performance of gateways
ethereal, wireshark	Linux/Windows	Graphical tool to capture, to inspect, and to save Ethernet packets

```
netman: ping -s mit.edu
PING mit.edu: 56 data bytes
64 bytes from MIT.MIT.EDU (18.72.0.100): icmp_seq=0. time=42. ms
64 bytes from MIT.MIT.EDU (18.72.0.100): icmp_seq=1. time=41. ms
64 bytes from MIT.MIT.EDU (18.72.0.100): icmp_seq=2. time=41. ms
64 bytes from MIT.MIT.EDU (18.72.0.100): icmp_seq=3. time=40. ms
64 bytes from MIT.MIT.EDU (18.72.0.100): icmp_seq=4. time=40. ms

----mit.edu PING Statistics----
5 packets transmitted, 5 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 40/40/42
```

Figure 9.3 Example of Traffic Monitoring (*ping*)

made. However, there is a practical limit to it (about 30 times). In practice, this means that if you have a 64-kbps connection to the Internet, the maximum throughput of the link you can measure is 2 Mbps.

Other commands examine the packets that traverse the network to provide different outputs. The commands, *tcpdump* and *ethereal* (also known as *wireshark*), put a network interface in a promiscuous mode, in which raw data are gathered from the network without any filtering, and log the data. All of them could generate an output text file, associating each line with a packet containing information on the protocol type, length, source, and destination. Because of the security risk associated with looking at data in a promiscuous mode in these cases, the user ID is limited to super-user. An example of the output of *ethereal* is shown in Figure 9.4. Each line shows information about one packet. Several packets are Address Resolution Protocol (ARP) requests. In these cases, the source MAC address is shown. For ease of reading, the vendor ID of the MAC address is shown in text form, followed by the 24-bit serial number. The three lines in gray scale in the middle of the window show packets belonging to a single HTTP session over a transmission control protocol (TCP) connection. They capture the three-way handshake used by TCP to establish the connection. The first of these packets is a connection request from 10.6.21.59 to the HTTP port on server 10.94.3.215. The server responds with an SYN+ACK, and finally, the initiator ACKs to complete the connection establishment handshake.

For an example of *tcpdump* command output, please see the SNMP get-request and get-response PDUs shown in Figure 5.17 that were obtained using the *tcpdump* tool.

The command *getethers* discovers all host/Ethernet address pairs on the LAN segment (a.b.c.1 to a.b.c.254). It generates an ICMP echo_request message similar to ping using an IP socket. The replies are compared with the ARP table to determine the Ethernet address of each responding system.

The *iptrace* tool uses the NETMON program in the UNIX kernel and produces three types of outputs: IP traffic, host traffic matrix output, and abbreviated sampling of a pre-defined number of packets.

Network Routing Tools. Table 9.3 lists three sets of route-monitoring tools. The *netstat* tool displays the contents of various network-related data structures in various formats, depending on the options selected. The network-related data structures that can be displayed using *netstat* include the routing table, interface statistics, network connections, masquerade connections, and multicast memberships. The example of the routing tables obtained using *netstat* is given in Figure 9.5. It shows the ports associated with various destinations. *netstat* is a useful diagnostic tool for troubleshooting. For example, the routing table information shown in Figure 9.5 informs the network operator which nodes have been

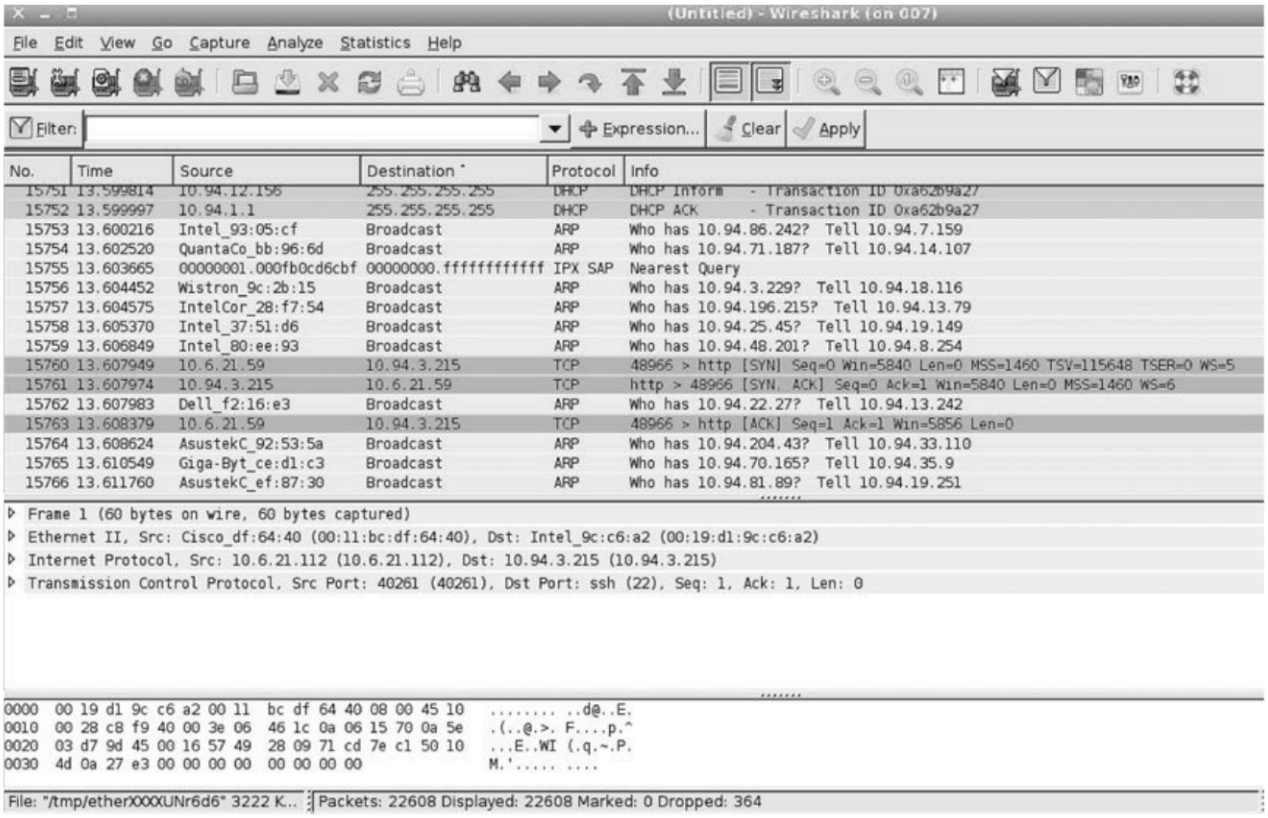

Figure 9.4 Traffic Monitoring using Ethereal (Wireshark)

Table 9.3 Route-Monitoring Tools

NAME	OPERATING SYSTEM	DESCRIPTION
netstat	UNIX	Displays the contents of various network-related data structures
arp rarp	UNIX, Windows	Displays and modifies the Internet-to-Ethernet address translation tables
traceroute tracert	UNIX/Windows	Traces the route to a destination with routing delays

active since the last purge of the table, which is typically in the order of a minute. The most frequently used options of netstat are: −r that obtains the contents of the routing tables; −i that prints the table of all networking interfaces; and −a that prints information about all active Internet connections and UNIX-domain sockets.

The *arp* tool displays and modifies the Internet-to-Ethernet address translation tables (ARP cache) used by the ARP. Some UNIX systems provide an additional tool for manipulating the contents of Ethernet-to-Internet address translation tables (RARP cache). The name of the tool is *rarp* and its use is similar to *arp*.

The third set of routing tools shown in Table 9.3 is *traceroute* (UNIX) or *tracert* (MS Windows), which is the basic tool used most extensively to diagnose routing problems. The tool discovers the route taken by packets from the source-to-destination through each hop. It is very useful in localizing the

```
netstat −r
Routing tables

Internet:
Destination        Gateway            Flags   Refs   Use      Netif  Expire
Default            gw.litech.net      UGC     44     541550   de0
172.16.15.1        gw.litech.net      UGH     0      0        de0
ah.litech.net      0:80:48:ee:74:b4   UHLW    9      2653683  de0    202
uucp.litech.net    uucp.litech.net    UH      0      0        lo0
sip-17.litech.net  big                UH      0      5551     ppp3
dip-244.litech.net gw.litech.net      UGH     0      2472     de0
univers-litech-gw  gw.litech.net      UGH     0      47       de0
194.44.232         gw.isr.lviv.ua     Ugc     0      171831   ppp9
OSPF-ALL.MCAST.NET localhost          UH      1      86491    lo0
OSPF-DSIG.MCAST.NE localhost          UH      1      25127    lo0
```

Figure 9.5 Routing Table using netstat−r

source of route failure. It is also useful in performance/packet delay evaluation as the result gives the delay time to each node along the route. *traceroute* is based on the ICMP time_exceeded error report mechanism. When an IP packet is received by a node with a time-to-live (TTL) value of 0, an ICMP packet is sent to the source. The source sends the first packet with a TTL of zero to its destination. The first node looks at the packet and sends an ICMP packet since TTL is greater than 0. Then, the source sends the second packet with a TTL larger than the TTL needed to get to the first node, and thus *traceroute* acquires the second node. This process continues until all the nodes between the source and the destination have been determined.

Figure 9.6 gives two sample traces taken close to each other in time between the same source *mani.bellsouth.net* and destination *mani.btc.gatech.edu*. We notice significant differences in the route taken, the delay times, and the number of hops. We would expect these differences since each packet in an IP network is routed independently. Each line shows the router that the packet traversed in sequential order. The three time counts on each line indicate the round-trip delay for each router on three attempts made from the source. We notice that there is a jump in the round- trip delay from 2–5 milliseconds to greater than 10 milliseconds when the packet crosses over from the local BellSouth network. In some lines, for example in lines 9 and 13 in Sample 1, one round-trip delay reads high, which could be attributed to the router being busy. In Sample 2, the second half of the route appears congested, indicating consistent large round-trip delays. We also notice that some of the routers respond with their IP address, while others do not. The lines that are marked with asterisks are responses from those routers, which have been administratively prevented from revealing their identity in their responses.

There are also Web-based *traceroute* and *ping* utilities available in some systems. The use of these tools significantly decreases the time necessary to detect and isolate a routing problem because the final decision is based on independent data obtained from various hosts on the net.

```
Tracing route to mani.btc.gatech.edu [199.77.147.96]
over a maximum of 30 hops:

 1    2 ms    3 ms    3 ms  bims008001.bims.bellsouth.net [205.152.8.1]
 2    4 ms    2 ms    3 ms  172.16.11.2
 3    5 ms    4 ms    3 ms  172.16.4.2
 4    5 ms    3 ms    3 ms  bims011033.bims.bellsouth.net [205.152.11.33]
 5    4 ms    4 ms    4 ms  205.152.13.98
 6    *       *       *     Request timed out
 7    5 ms    9 ms   12 ms  205.152.2.249
 8   33 ms   31 ms   31 ms  Hssi0-0-0.GW2.ATL1.ALTER.NET [157.130.65.229]
 9   68 ms   10 ms   11 ms  105.ATM3-0-0.XR1.ATL1.ALTER.NET [146.188.232.66]
10   11 ms   14 ms   12 ms  195.ATM12-0-0.BR1.ATL1.ALTER.NET
[146.188.232.49]
11   16 ms   14 ms   14 ms  atlanta1-br1.bbnplanet.net [4.0.2.141]
12   19 ms   15 ms   17 ms  atlanta2-br2.bbnplanet.net [4.0.2.158]
13   21 ms   56 ms  328 ms  atlanta2-cr99.bbnplanet.net [4.0.2.91]
14   17 ms   18 ms   17 ms  192.221.26.3
15   32 ms   20 ms   18 ms  130.207.251.3
16   20 ms   17 ms   17 ms  mani.btc.gatech.edu [199.77.147.96]

Trace complete
```

Sample 1

```
Tracing route to mani.btc.gatech.edu [199.77.147.96]
over a maximum of 30 hops:

 1    3 ms    3 ms    4 ms  bims008001.bims.bellsouth.net [205.152.8.1]
 2    3 ms    3 ms    2 ms  172.16.11.2
 3    5 ms    4 ms    4 ms  172.16.4.2
 4    5 ms    3 ms    4 ms  bims011033.bims.bellsouth.net [205.152.11.33]
 5    7 ms    4 ms    4 ms  205.152.13.98
 6    *       *       *     Request timed out
 7    9 ms    8 ms    9 ms  205.152.2.249
 8  228 ms  214 ms  191 ms  206.80.168.9
 9  230 ms  246 ms  234 ms  maeeast.bbnplanet.net [192.41.177.2]
10  243 ms  222 ms  212 ms  vienna1-nbr2.bbnplanet.net [4.0.1.93]
11  230 ms  213 ms  202 ms  vienna1-nbr3.bbnplanet.net [4.0.5.46]
12  247 ms  227 ms  236 ms  vienna1-br2.bbnplanet.net [4.0.3.149]
13  228 ms  235 ms  238 ms  atlanta1-br1.bbnplanet.net [4.0.2.58]
14    *     257 ms  238 ms  atlanta2-br2.bbnplanet.net [4.0.2.158]
15  225 ms  234 ms  233 ms  atlanta2-cr99.bbnplanet.net [4.0.2.91]
16  240 ms  229 ms  251 ms  192.221.26.3
17  235 ms  245 ms  225 ms  130.207.251.3
18    *     268 ms  243 ms  mani.btc.gatech.edu [199.77.147.96]

Trace complete
```

Sample 2

Figure 9.6 Examples of Route Tracing (*traceroute*)

9.1.2 SNMP Tools

There are several tools available to obtain the MIB tree structure, as well as its values from a network element. Each of these tools has several implementations. We will not go into specific implementations here, but will describe their functionality. You may obtain details on the use and options from the manual page describing the tool.

SNMP MIB tools are of three types: (1) SNMP MIB browser uses a graphical interface; (2) a set of SNMP command-line tools, which is primarily UNIX- and Linux/FreeBSD-based tools; and (3) Linux/FreeBSD-based tool, *snmpsniff*, which is useful to read SNMP PDUs.

SNMP MIB Browser. An SNMP MIB browser is a user-friendly tool that can be accessed from public software libraries or commercially purchased. All extract MIB-II of SNMPv1 and some can extract SNMPv2 MIB. Some can also acquire private MIB objects. You specify the host name or the IP address first. Then information on a specific MIB object, or a group, or the entire MIB can be requested, depending on the implementation. The response comes back with the object identifier(s) and value(s) of the object(s).

For example, using the graphical MIB browser *tkmib* (http://www.net-snmp.org) and requesting the variable *system.sysDescr* from host 10.65.0.32 yields the response shown in Figure 9.7. The MIB tree is shown in graphical form in the top window. By clicking on a leaf node in the tree and invoking a *get*,

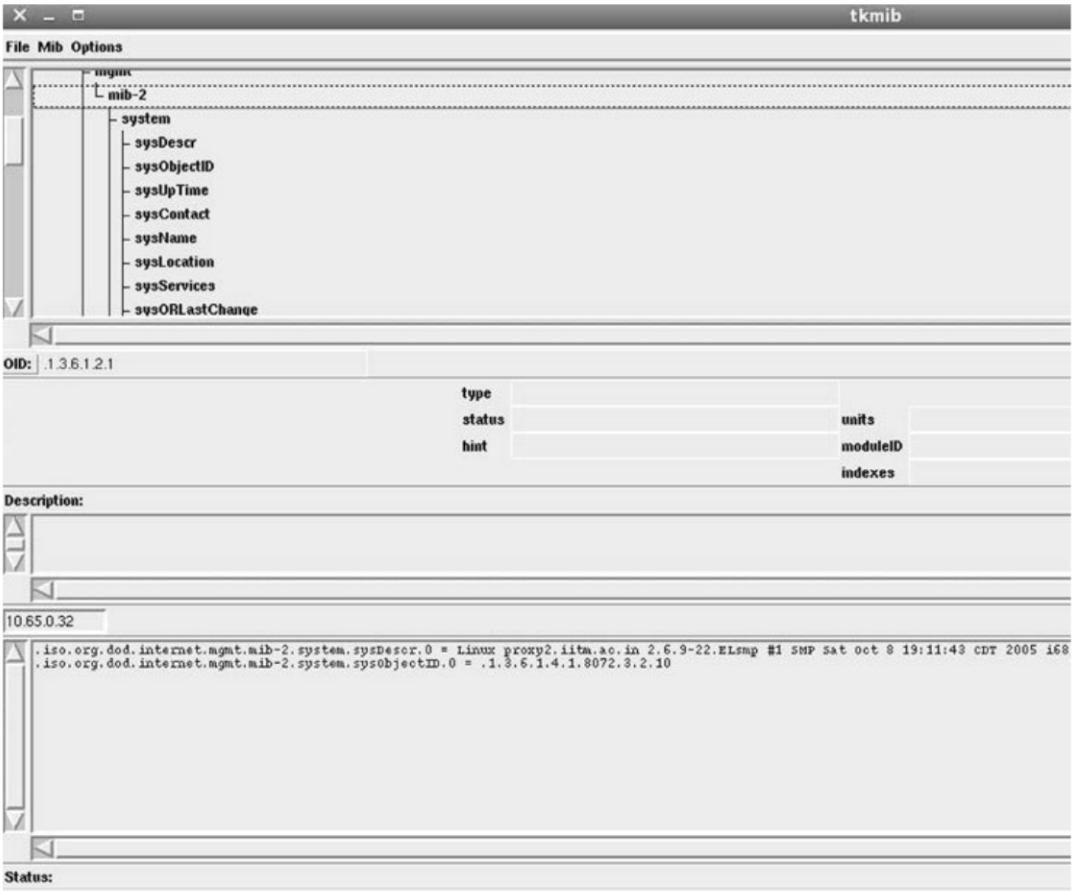

Figure 9.7 MIB Browser Example (*tkmib*) for a System Descriptor Object

```
199.77.147.182:

   sysDescr.0 : SunOS noc5 5.6 Generic_105181-03 sun4u
   sysObjectID.0 : 1.3.6.1.4.1.11.2.3.10.1.2
   sysUpTime.0 : 8d 22:21:53.74
   sysContact.0 :
   sysName.0 : noc5
   sysLocation.0 :
   sysServices.0 : 72
   sysORLastChange.0 : 0d 0:00:00.00
```

Figure 9.8 MIB Browser Example (Text Based) for a System Group

the response from the agent is shown in the bottom panel. We see that the node is a Linux-based proxy server running kernel 2.6.9-22 with symmetrical multiprocessor (SMP) support. Figure 9.8 shows the results obtained by using a text-based browser to retrieve the system group from host 99.77.147.182.

SNMP Command-Line Tools. There are several SNMP command-line tools available in the UNIX, Linux, or FreeBSD and Windows OS environments. Command-line tools generate SNMP messages, which are get, get-next, getbulk, set, response, and trap. Public domain software packages can be downloaded that are capable of operating either as an SNMPv1, SNMPv2, or SNMPv3 manager. A popular suite is available from *http://www.net-snmp.org.* The following commands are described in SNMPv1 format. An option of –v2 or –v3 is used to generate SNMPv2c or SNMPv3, respectively.

SNMP Get Command.

snmpget [options] host community objectID [*objectID*]

This command communicates with a network object using the SNMP *get-request* message. The *host* may be either a host name or an IP address. If the SNMP agent resides on the host with the matching *community* name, it responds with a *get-response* message returning the value of the *objectID*. If multiple *objectIDs* are requested, a *varBind* clause is used to process the message containing multiple object names (see Figure 4.48). If the get-request message is invalid, the get-response message contains the appropriate error indication.

For example,

snmpget 199.77.147.182 public system.sysdescr.0

retrieves the system variable system.sysDescr

system.sysdescr.0 = “SunOS noc5 5.6 Generic_105181-03 sun4u.”

The 0 at the end of the *objectID* indicates that the request is for a single scalar variable.

SNMP Get-Next Command.

snmpgetnext [options] host community objectID [objectID]

The command is similar to *snmpget* except that it uses the SNMP *get-next-request* message. The managed object responds with the expected get-response message on the *objectID* that is lexicographically next to the one specified in the request. This command is especially useful to get the values of variables in an aggregate object, i.e., in a table.

For example,

Snmpgetnext 199.77.147.182 public interfaces.ifTable.ifEntry.ifIndex.1 retrieves

Interfaces.ifTable.ifEntry.ifIndex.2 = "2."

SNMP Walk Command.

snmpwalk [options] host community [objectID]

The *snmpwalk* command uses *get-next-request* messages to get the MIB tree for the group defined by the *objectID* specified in the request. It literally walks through the MIB. Without the *objectID*, the command displays the entire MIB tree supported by the agent.

SNMP Set Command. The *snmpset* command sends the SNMP set-request message and receives the *get-response* command.

SNMP Trap Command. The *snmptrap* command generates a trap message. Some implementation handles only SNMPv1 traps and others handle SNMPv1, SNMPv2, and SNMPv3 and can be specified in the argument. Note that this acts as an SNMP agent

SNMP Sniff Tool. The SNMP Sniff tool, *snmpsniff*, is similar to the *tcpdump* tool and is implemented in Linux/FreeBSD environment. It captures SNMP packets going across the segment and stores them for later analysis.

9.1.3 Protocol Analyzer

The protocol analyzer is a powerful and versatile network management tool. We will consider it as a test tool in this section, and later on look at its use as a system management tool. It is a tool that analyzes data packets on any transmission line. Although it could be used for the analysis of any line, its primary use is in the LAN environment, which is what we will focus on here. Measurements using the protocol analyzer can be made either locally or remotely. The basic configuration used for a protocol analyzer is shown in Figure 9.9. It consists of a data capture device that is attached to a LAN. This could be a specialized tool, or either a personal computer or workstation with a network interface card. The captured data are transmitted to the protocol analyzer via a dial-up modem connection, a local or campus network, or a wide area network. The protocol analyzer analyzes the data and presents it to the user on a user-friendly interface.

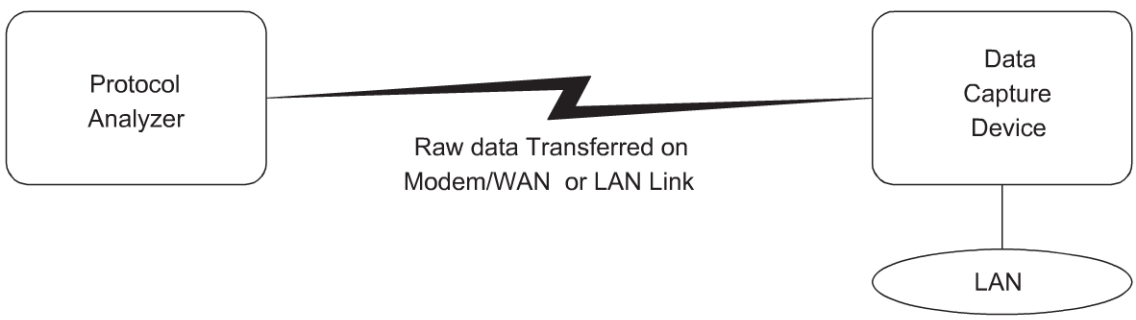

Figure 9.9 Basic Configuration of a Protocol Analyzer

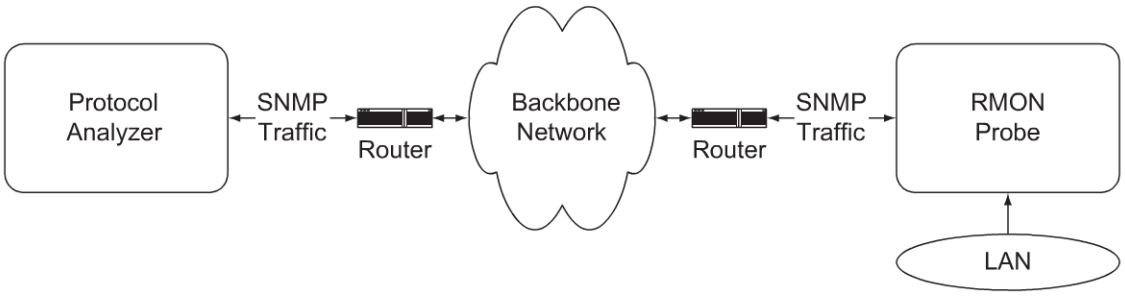

Figure 9.10 Protocol Analyzer with an RMON Probe

The protocol analyzers that are available in the commercial market are capable of presenting a multitude of results derived from the data. Contents of data packets can be viewed and analyzed at all layers of the OSI reference model. The distribution of various protocols at each layer can be ascertained. At the data link layer, besides the statistical counts, the collision rate can be measured for Ethernet LAN. At the transport layer, port information for different applications and sessions can be obtained. The distribution of application-level protocols provides valuable information on the nature of traffic in the network, which can be used for performance tuning of the network.

Numeous commercial and open-source protocol analyzers and sniffers are now available. Sniffer can be used as a stand-alone portable protocol analyzer, as well as on the network HP. NetMetrix protocol analyzer is a software package loaded on to a workstation. It uses LanProbe as the collector device, which can be configured also as an RMON probe. The communication between RMON and the protocol analyzer is based on the SNMP protocol, as shown in Figure 9.10.

A protocol analyzer functioning as a remote-monitoring analyzer collects data using an RMON probe. The raw data that are gathered are pre-analyzed by the RMON and transmitted as SNMP traffic instead of raw data in the basic configuration mentioned earlier. The statistics could be gathered over a time period for analysis or displayed on a real-time basis. In the promiscuous mode, the actual data collected by the probe could be looked at in detail, or statistics at various protocol layers could be displayed. The results are used to perform diagnostics on network problems, such as traffic congestion. They could also be used with the help of the tool to do network management functions such as traffic reroute planning, capacity planning, load monitoring, etc.

Using an RMON probe for each segment of the network and one protocol analyzer for the entire network, as shown in Figure 9.11, the complete network can be monitored. The RMON probe for each type of LAN is physically different. Even for the same type of LAN, we need a separate probe for each segment, which could be expensive to implement.

9.2 NETWORK STATISTICS MEASUREMENT SYSTEMS

One key aspect of network management is traffic management. We will consider performance management as one of the application functions when we deal with them in Chapter 11. However, we will first consider how the basic tools that we discussed in Section 9.1 are used to gather network statistics in the network at various nodes and segments. We will then cover an SNMP tool, Multi Router Traffic Grapher (MRTG), which can be used to monitor traffic.

One of the best ways to gather network statistics is to capture packets traversing network segments or across node interfaces in a promiscuous mode. We have learned that protocol analyzers do just that.

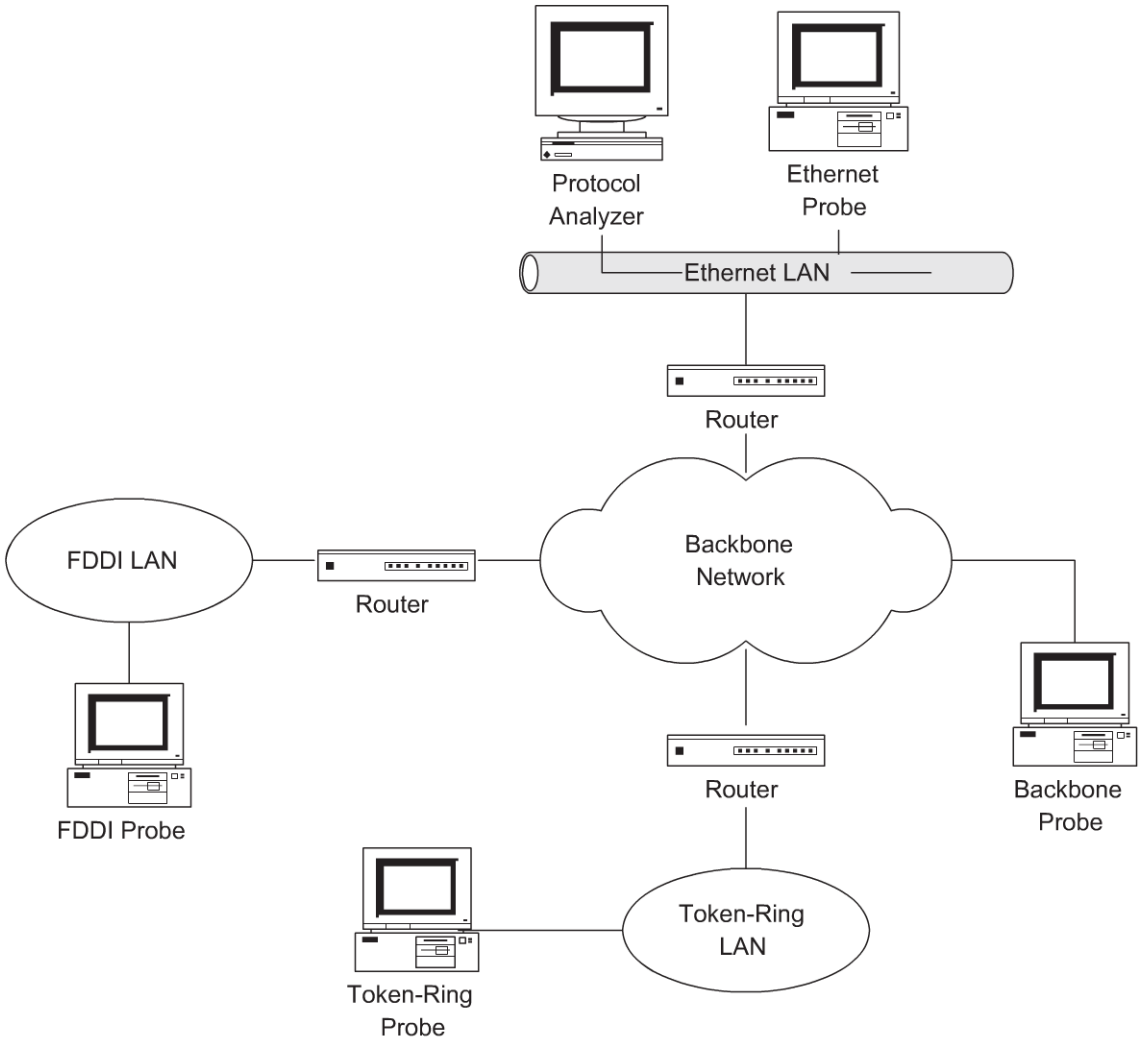

Figure 9.11 Monitoring of Total Network with Individual RMON Probes

Thus, they are good tools to gather network statistics. Another way to gather network statistics is to develop a simple application using a function similar to *tcpdump*, using a high-performance network interface card and processor, and analyze the data for the required statistics. After all, that is the basis on which protocol analyzers are built.

The RMON MIB that we studied in Sections 8.3 and 8.4, along with the SNMP communication protocol, provides a convenient mechanism to build network-monitoring systems. The configurations shown in Figure 9.10 and Figure 9.11 can be used as the network-monitoring system to gather various RMON objects. The RMON1 MIB groups and tables shown in Tables 8.2 and 8.3 are used to gather statistics at the data link layer in Ethernet and token-ring LANs. The RMON2 MIB groups and tables presented in Table 8.4 define parameters for higher-layer statistics.

9.2.1 Traffic Load Monitoring

Traffic load monitoring can be done based on the source, the destination, and the source–destination pair. We may want to balance the traffic load among the various LAN segments, in which case we need

to measure the total traffic in each network segment or domain. Data for traffic monitoring can be sampled at the data link layer using the RMON1 MIB history group. Traffic relevant to a host, either as source or as a destination, is available in the host group. Hosts can be ranked on the traffic load that they carry using the *HostTopN* group. In the absence of an RMON probe, there is no convenient way to measure traffic in a segment directly except to compute it externally knowing the hosts in the segment.

Load statistics in an IP network can also be obtained by measuring IP packets at the network layer level. The entities in the network layer host and the network layer matrix groups in RMON2 MIB can be used for this measurement. Figure 9.12, Figure 9.13, and Figure 9.14 show the load statistics measured in a Fiber Distributed Data Interface (FDDI) LAN segment using the NetMetrix protocol analyzer as the Load Monitor and FDDI probe.

In Figure 9.12 there are 1,609 sources that generated data packets. The top ten have been identified, with the highest entry being news-ext.gatech.edu. The entry LOW-CONTRIB is a combination of sources other than those specifically identified. Traffic is measured as the number of octets. Figure 9.13 presents similar statistical data on traffic that is destined to the hosts in the network segment. Figure 9.14 presents the top ten conversation pairs of hosts. Each line identifies the host-pairs, traffic from NetHost1 to NetHost2. Oct1to2 and Oct2to1 denote the traffic in octets from NetHost1 to NetHost2 and NetHost2 to NetHost1, respectively. For example, news-ext.gatech.edu transmitted 3K octets/second of outgoing traffic to and received 60K octets of incoming traffic from howland.erols.net.

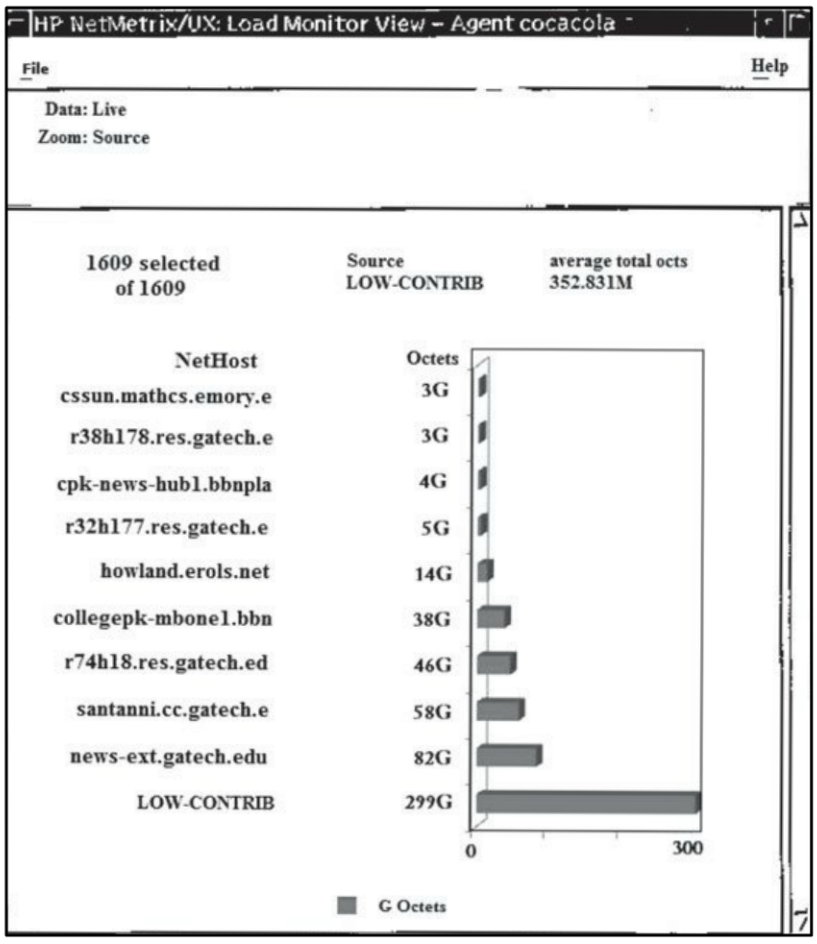

Figure 9.12 Load Statistics: Monitoring of Sources

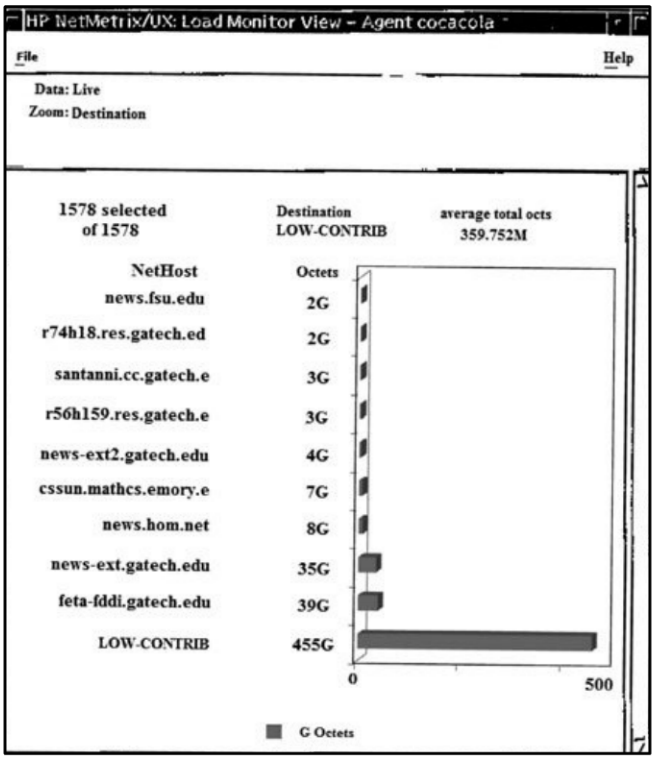

Figure 9.13 Load Statistics: Monitoring of Destinations

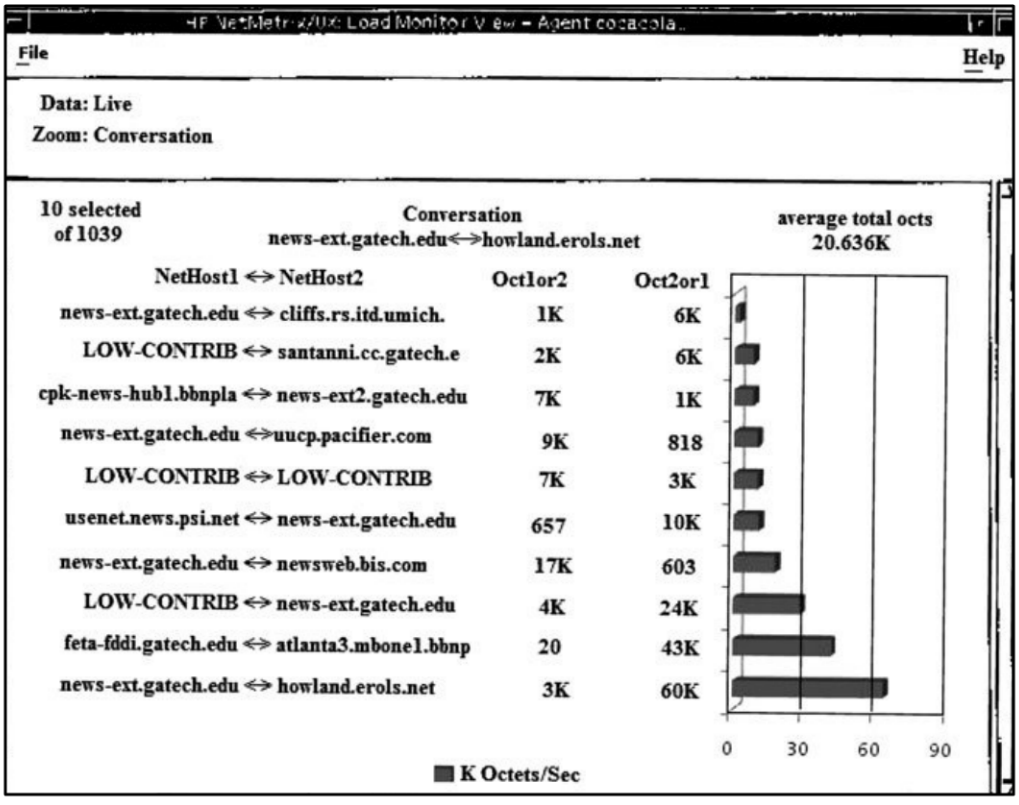

Figure 9.14 Load Statistics: Monitoring of Conversation Pairs

9.2.2 Protocol Statistics

Packets can be captured by data capture devices based on the filter set for the desired criteria. From the captured data, we can derive protocol statistics of various protocols at each layer of the OSI Reference Model. This is very useful at the application layer level. We can obtain the traffic load for different applications such as file transfer (FTP), Web data (HTTP), and news groups (NNTP). This information can be used for bandwidth management of real-time and non-real-time traffic.

Figure 9.15 shows the distribution of protocols at the data link (top left corner), network (top right corner), transport (bottom left corner), and application (bottom right corner) layers, obtained using NetMetrix LanProbe and a protocol analyzer. Data link and network layers show 100% LLC and IP protocols. The majority of the transport layer protocol packets belong to TCP and the next in order is UDP. The other category is undefined. At the application layer(s), the distribution contains HTTP (Web protocol), NNTP (news protocol), FTP-data, UDP-other, TCP-other, and undefined other. The Georgia Tech Internet backbone network in which the measurements were made carries a complex variety of protocol traffic including multimedia traffic and next generation Internet traffic.

9.2.3 Data and Error Statistics

Data and error statistics can be gathered directly from managed objects using the specifications defined in various MIB groups. The RMON statistics groups for Ethernet [RFC 1757] and token ring [RFC 1513] contain various types of packets and errors in the data link layer. Similar information is available on higher-level layers from specifications detailed in RMON2 [RFC 2021]. Information on statistics can also be gathered for the individual medium from the respective MIBs under the transmission group. For example, statistics on Ethernet can be derived from the Ethernet-like statistics group in Ethernet-like

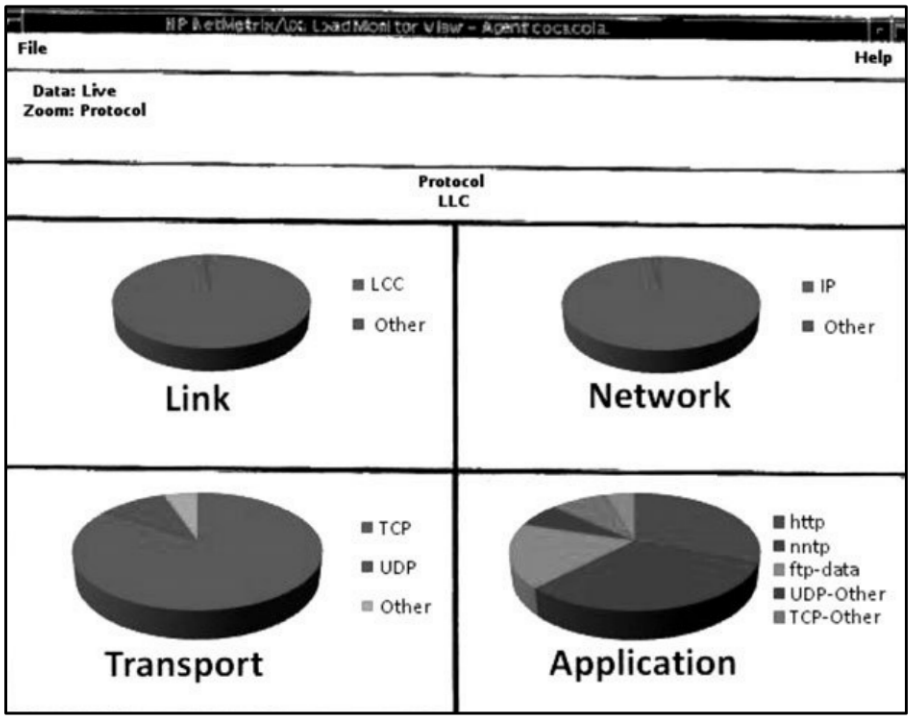

Figure 9.15 Protocol Distribution (NetMetrix)

interface types MIB [RFC 1284], token-ring statistics from IEEE 802.5 MIB [RFC 1748], and FDDI data from FDDI MIB [RFC 1285].

9.2.4 Using MRTG to Collect Traffic Statistics

The MRTG is a tool that monitors traffic load on network links [Oetiker and Rand, http://oss.oetiker.ch/mrtg]. It generates a live visual representation of traffic data by reading the SNMP traffic counters on routers and creates graphs that are embedded into Web pages. These can be monitored using any Web browser. Visual presentations of traffic data are presented as daily view, the last 7 days view, the last 4 weeks view, and the last 12 months view. The generic software can be implemented in either UNIX or Windows NT platform. An example of the views can be seen on http://switch.ch/network/operation/statistics/geant2.html.

9.3 MIB ENGINEERING

In the SNMP model of management, information about each network element is contained in its MIB (Chapter 4). The manager's view of the NE is defined by and is limited to its MIB. The ease of developing management applications depends on how closely the MIB matches the needs of the manager. In most cases, the MIB is hardcoded into the NE by the vendor. Thus, care must be taken in designing the MIB. This is the focus of *MIB engineering*.

There are some commonly used constructs in many MIBs. The use of these *idioms* makes it easier for the manager to understand and use the MIB. The SNMP protocol and structure of management information (SMI), by virtue of their stress on simplicity, have some limitations. Fortunately, there are work-arounds to enable the manager to accomplish complex tasks despite these limitations.

First we cover some basic principles, limitations, and idioms of SMI. We then take a number of frequently occurring requirements and show how to design MIBs for them.

9.3.1 General Principles and Limitations of SMI

SMI provides a very simple view of the network. Every element is completely defined by a set of variables (euphemistically referred to as *objects*), which may take on a limited number of data types. These include scalar or primitive types (Boolean, Integer, IP address, String, Counter, etc.) and three structured or constructed types (arrays, records, and sets).

An *array* is an ordered list of elements all being of the same type. Each element is identified by its index. A *record* is an ordered collection of elements of different types, with each element referred to by name. A *set* is an unordered collection of elements. The elements could either all be of the same type or of different types. E.g., the interfaces table, *ifTable*, is an array with one element for each interface in the node. Each element in the table contains a record with several fields describing the interface (see Figure 4.28).

As an example of a set, suppose the manager wishes to define a trap filter in an agent. The filter consists of a set of conditions that are AND'ed together. Only if all conditions are satisfied, the trap is forwarded. We could define:

```
TrapFilter ::= SET OF Conditions
```

The order of evaluation of the conditions does not matter.

There are restrictions on the construction of types:

- An array can contain a record and vice versa. An array can contain a record that itself contains other records. However, an array cannot contain a record that contains an array
- A record can be defined to hold related variables pertaining to one part of an NE (in one subtree of the MIB). Another record with identical fields but different names must be defined

The SNMP protocol allows a manager to get or set the value of a variable (see Figure 3.9). However, it does not permit a manager to perform an action such as resetting an interface or deleting a file. In object-oriented terms, an SNMP object has member variables and accessor methods, but does not have any general methods to perform actions.

The SNMP protocol supports request response transactions where each transaction can automatically either get or set a limited number of variables. The limit is imposed by the size of an SNMP PDU. It does not allow combining gets and sets in one transaction, nor does it allow a sequence of operations to form a session. SNMP transactions are very limited compared to database transactions. The latter allow a large sequence of select and update queries to be combined in a single transaction, which either succeeds completely or is not executed at all. In addition, access to the relevant parts of the database by other users can be prevented during the transaction.

9.3.2 Counters vs. Rates

Rates are central to network management. Performance is measured largely in terms of rates. For example, the throughput of a link is given in bits/second or packets/second, the throughput of a server in transactions/second, and user behavior in requests/hour.

Rates are also important in some aspects of fault management, specifically in determining when the load on the system is reaching its capacity and hence is liable to fail. For example, if a 1 Mb/s link is subjected to traffic at the rate of 0.98 Mb/s, congestion is very likely with the consequent increase in delays, packet loss, timeouts, and retransmissions. If the congestion persists, it could result in the failure of end applications or the routers at either end of the link. Proactive fault management attempts to spot congestion in its nascent stage and then take congestion avoidance measures to prevent failures. The onset of congestion is indicated by comparing the live throughput with predefined thresholds. The threshold depends on the capability of the router to tolerate temporary overload by means of buffering of packets. Suppose for a router with a 10-packet buffer we set the congestion threshold at 0.8 Mb/s; if the router has a 20-packet buffer, we might increase the threshold to 0.9 Mb/s. The threshold settings depend on the mix of packet sizes and other practical factors. In practice, the threshold is tuned by the operator based on experience.

The manager may also require the counter value. For example, if broadband subscribers are billed based on data transferred, the NMS manager would retrieve the counter of bytes transferred and pass it on to the Billing System.

A counter is a direct performance measure. On a network link, the network interface maintains several counters such as packets and bytes transmitted, packets and bytes received, errors, etc. Whenever a packet is processed, one or more counters are incremented appropriately. These counters are thus available to the NMS agent at no extra cost, are always up to date, and accurate. Hence, the MIB should include the counter. Be mindful that the counter reading wraps around and should be interpreted correctly.

A rate is a derived measure. To determine the transmit rate, we periodically read the transmit octets counter and calculate the rate:

$$\text{Transmit rate} = \frac{\text{transmit octets at } t_2 - \text{transmit octets at } t_1}{t_2 - t_1}$$

The rate depends on the averaging interval $t_2 - t_1$. For example, Figure 9.16 shows a 5-second burst of packets during which 4,000 octets are transmitted. The network is idle for some time thereafter.

Table 9.4 shows the rates calculated over various time intervals. Note that the rate varies quite drastically from 0 to 8,000 b/s. The interval 0–4 is especially troublesome. The counter is 3,000, but the actual number of octets transmitted is 3,500. Even if the Network Interface Card (NIC) has hardware to increment the counter for every octet, this would not be correct because we do not know whether the packet transmission is successful until the end of transmission.

Thus, we see that the derived measure of rate is imprecise and gives different results depending on the averaging interval. Different managers may require different averaging intervals and even one manager may require different averaging intervals at different times. For example, while projecting future growth in network traffic for capacity augmentation, the manager may want a 1-hour average. To monitor congestion in a link, the averaging may be done over 5-minute periods. While troubleshooting some problem, the manager may average over 30-seconds or less to see instantaneous fluctuations.

Normally, the MIB does not include rates. It is left to the manager to poll the counter at the desired periodicity and to compute the rate.

There are a few exceptions. The central processing unit (CPU) load of a server is usually computed using some OS-specific algorithm and the OS may not expose convenient counters. Hence, the agent simply makes available the rates computed by the OS, say the 30-second, 1-minute, and 5-minute load averages. The corresponding counters are not included in the MIB.

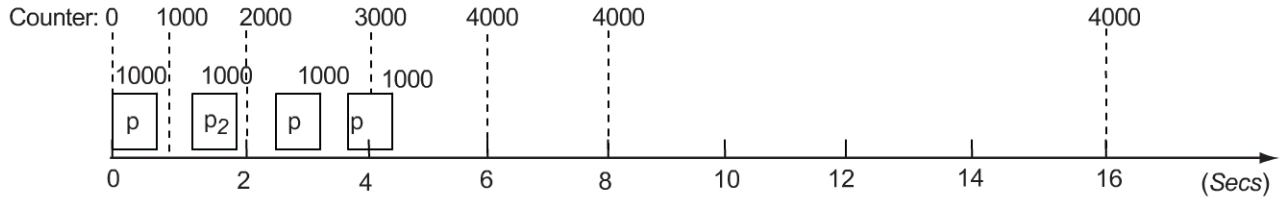

Figure 9.16 Burst of Activity on a Network Link

Table 9.4 Rates Calculated over Various Intervals

INTERVAL (SECONDS) t_1, t_2	COUNTER DIFFERENCE (OCTETS)	RATE (b/s)
0, 2	2,000 – 0	8,000
0, 4	3,000 – 0	6,000
0, 5	4,000 – 0	6,400
0, 10	4,000 – 0	3,200
0, 16	4,000 – 0	2,000
8, 12	4,000 – 4,000	0

9.3.3 Object-Oriented Approach to MIB Engineering

A network element has management parameters pertaining to its configuration, its operation, and its performance. In a complex device such as a server, a backbone router, or a telephone switch, the number of such parameters can easily be in the 1000s. For ease of comprehension, in an object-oriented design related variables are collected into classes. Similar classes are related to one another in an inheritance tree. Given a class definition, instances of the class can be created, referred to as objects [Bahrami, 1999].

SMI provides the MIB group that is a simple form of a class. A MIB group is just a subtree of the MIB tree. The name of the root node of the subtree is the name of the group. Consider MIB-II, which is the standard set of variables implemented by all SNMP nodes (see Section 4.7.4). These variables are grouped into a number of groups such as the system group, the interfaces group, the IP group, and the TCP group (see Figure 4.26).

In an object-oriented language such as C++ or Java, the language supports useful properties of the class. These include encapsulation (variables are private to a class), inheritance (one class inherits the variables and behavior of another class), and so on. We now show how these OO features can be used in information modeling, taking a router with network interfaces as an example.

In SMI, the analogous properties of MIB groups are more a matter of recommended practice. For instance, the TCP group has a count of the number of open connections, *tcpCurrEstab* (see Table 4.13). Normally, this is updated by the TCP code and this variable is considered to be encapsulated within the TCP group. However, it is quite possible for the interfaces code to examine the TCP headers as packets pass through the interface to detect the start/end of a session. The interfaces code could directly manipulate the *tcpCurrEstab* variable in the TCP group. This is a poor practice and is error-prone—a packet could be discarded by the IP layer before it reaches TCP. SMI does not prevent this practice.

Inheritance allows one class to derive some of its properties from a similar (usually more general) class. For instance, an interface has properties such as type, speed, address, and packets sent. An Ethernet interface shares these properties and also has additional properties such as number of collisions. An RS-232 interface has additional properties such as parity. So, it is desirable to derive Ethernet and RS-232 classes from a base interface class as depicted in Figure 9.17.

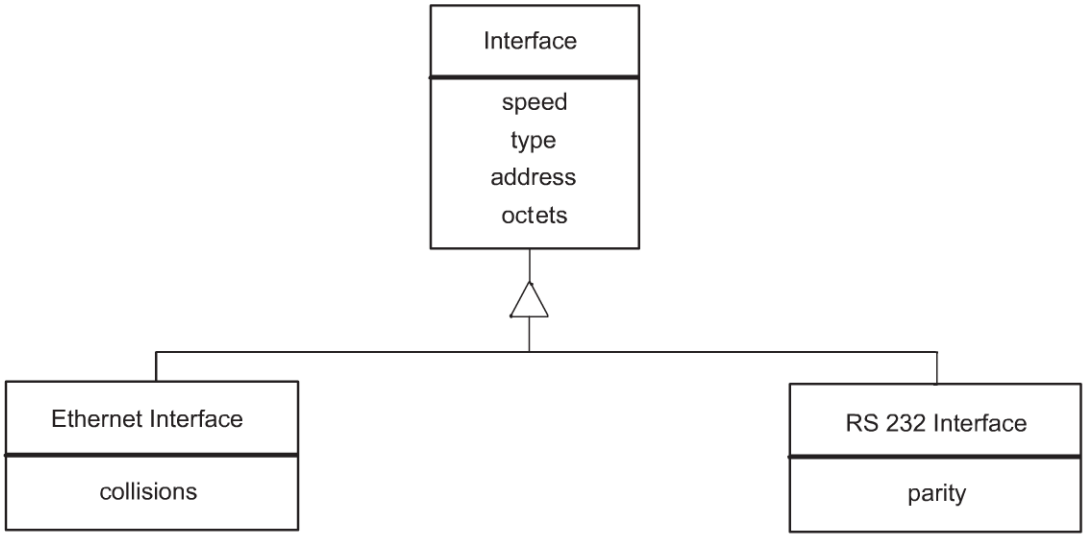

Figure 9.17 Inheritance Used to Define Various Network Interfaces

An Ethernet interface object has the properties speed, type, address, octets, and collisions, while an RS-232 object has the properties speed, type, address, octets, and parity.

Naming. An object-oriented language allows the reuse of the same name in different contexts without any confusion. For instance, given classes Router and Switch, there is no confusion in the use of the same member variable name *numInterfaces*: *Router.numInterfaces* and *Switch.numInterfaces* are clearly distinct. Similarly, grouping of classes into *packages* or *namespaces* permits reuse of class names.

SMI uses a globally unique hierarchy to classify names. Hence, the same name could be used in different subtrees (or groups). The system group in MIB-2 has a *sysDescr* variable. A vendor could have a *sysDescr* variable in its product-specific subtree. They are distinct as *mib-2.system.sysDescr* and *midas.corDECT.sysDescr*, for example.

There are limitations, however. The same name cannot be used twice in the same MIB file. Hence, it is conventional to have a unique prefix for all variables in a MIB group—"sys" for the system group, "if" for the interfaces group, and so on. Thus, we have *mib-2.system.**sysDescr*** and *mib-2.interfaces.ifTable.ifEntry.**ifDescr***.

Instantiation. Given a class, many objects of that class can be created or *instantiated*. This can be done statically at compile-time or dynamically at runtime as shown in the C++ code below.

```
Router r1, r2;      //Statically create 2 routers
Router *r3;
. . . .             // Dynamic creation of a 3rd router
if (condition) r3 = new Router;
```

One agent can in general have only one instance of a MIB subtree. In the special case of a subtree that is part of a table, the agent can have multiple instances. See, for example, *ifEntry* in the *interfaces.ifTable*. Even in this case, creation and deletion of rows in the table ("objects") are not as straightforward as the use of *new* and *delete* operators in C++. In SMIv1, row creation is ambiguous and deletion is not supported. In SMIv2, these operations require special fields to be added to the table (see Section 6.3.7).

Recommended Practices. As telecom devices are complex, the object-oriented design of a MIB is highly desirable. SMI has limited support for the object-oriented design of a MIB. Some commonly used practices are as follows:

1. Use of unique prefixes for all variables in a MIB subtree, such as "if" in the interfaces group, "tcp" in the TCP group, and so on. This makes names longer than necessary.
2. Extensive use of tables to allow variable numbers of objects of the same type, e.g., *ifTable, tcpConnTable*, etc.
3. Dynamic creation and deletion of rows (objects) require inclusion of special variables such as *rowStatus* in the row of a table and a variable such as *ifNumber* to count the number of active rows.
4. Copy-and-paste of certain blocks of MIB definition from one file (subtree) to another.

9.3.4 SMI Tables

Besides groups, in SMI, tables are the most commonly used construct for the aggregation of data. SMI tables were introduced in Section 4.7.3 using some examples from MIB-2. An SMI table consists of

three levels in the registration tree: table node, entry node, and leaf nodes containing columnar data (see Figure 4.22). For example, in the interface table we have the table node *ifTable*, the entry node *ifTable.ifEntry*, and the data nodes *ifTable.ifEntry.ifIndex.1, ifTable.ifEntry.ifDescr.1,* etc.

Since only a single entry node ever appears under the table node, it is a natural question whether this can be omitted with the data nodes directly under the table node. However, the entry node is essential because of the way SMI defines an index. The entry node describes a complete row and only the node describing a row can specify the index column (see Clause 4.1.6 in RFC 1212). Since the index is essential for specifying a leaf node in a table (one element of the table), every table must have an entry node.

We now turn to the selection of the index field. The index field must enable selection of a unique row. If two or more rows have the same index value, the manager can retrieve only one of the rows. This is a consequence of the form of the SNMP get request in which every data value is obtained by specifying its complete name. The variable binding allows only one data value for one variable name.

Sequence Number as Index. In many cases the rows in the table correspond to sequence-numbered physical entities. For example, a 24-port switch has ports numbered from 0 to 23; and in a router each interface occupies one physical, numbered slot. In such cases it is natural to use this sequence number as the index.

Name as Index. Consider a software application on a server. A unique index can be the name of the application suffixed with the version number. Recall that when a string is used as the index, the ASN.1 encoding of the string is suffixed to the object ID (OID) of the row object to specify a leaf node. Thus, the name of the variable holding the path name of the Apache Web server application in an *appTable* might be *appTable.appEntry.appPath. 'A'.'p'.'a'.'c'.'h'.'e'*. With the usual ASCII codes this would be *appTable.appEntry.appPath.65.112.97.99.104.101*.

OID (Variable Name) as Index. In some tables the index is an OID. Suppose we wish to maintain an inventory table of equipment in a MIB. Each item is identified by its vendor and brand name. Assume that each of these equipments is manageable by SNMP and has a proprietary MIB defined by the vendor. Each such MIB is contained in a MIB group noted in an OID that uniquely identifies the equipment. Consider the corDECT wireless exchange manufactured by Midas Communication Technologies. Midas has been assigned the enterprise node {1 3 6 1 4 1 3794} in the {private enterprises} subtree (see Figure 4.14). corDECT is the first product of Midas and hence has the OID {1 3 6 1 4 1 3794 1}. This OID, encoded in ASN.1, can be used as the index in the inventory table.

Multidimensional Tables. SMI supports only one index for a table. However, this index can be a concentration of several columns, effectively yielding a multidimensional table. An example is the *tcpConnTable* in which the index is the concatenation of local IP address, local port, remote IP address, and remote port. These four quantities together uniquely identify a TCP connection. Thus, the state of connection to a Web server would be given by the variable *tcpConnTable.tcpConnEntry.tcpConnState.10.6.21.7.20.27.10.6.21.1.0.80*. Here the host 10.6.21.7 has opened a connection to port 80 on the server 10.6.21.1. The local port is 20 * 256 + 27 = 5147.

Note that the *tcpConnTable* (as also most other such multidimensional tables) is sparse. The total number of possible index values is $2^{32}2^{16}2^{32}2^{16} = 2^{96} \approx 10^{32}$, a truly enormous number. Most hosts have only a few 10s to a few 100s of connections open at a time. Hence, the agent should use some space-efficient data structure such as a hash table or a linked list of linked lists [Horowitz, 1995].

9.3.5 SMI Actions

For managing a network element it is necessary to perform actions such as enable/disable an interface, delete temporary files on a sever, reboot a node, etc. An object in an object-oriented programming language has actions (member functions). SMI "objects" directly only contain data but do not have functions.

In a MIB, an action can be performed by defining appropriate semantics of a variable. The semantics is given by the DESCRIPTION field in the OBJECT-TYPE macro used to specify the variable. The manager can invoke the action by an SNMP set on this "action" variable.

There are two ways in which the agent can inform the manager of the result of the action (analogous to the return value of a function). If the action completes almost immediately, the agent sends the SNMP response only after the completion. The value of the variable in the response indicates the result of the action. An example is the deletion of a file.

Some actions may take an unpredictable amount of time, say formatting a disk. In this case, the agent initiates the action and sends the SNMP response immediately with the value indicating that the action is in progress. When the action completes, the agent updates the same or another variable that can be queried by the manager. An example of the latter is found in the *Interfaces* group in MIB-2. *iftable* has one column *ifAdminStatus*, which the manager sets to *up* or *down* to request a change of status. The actual status is available in *ifOperStatus*. Note carefully the DESCRIPTION fields in the definitions of the variables in Figure 9.18.

```
ifAdminStatus OBJECT-TYPE
        SYNTAX  INTEGER {
                up(1),      -- ready to pass packets
                down(2),
                testing(3)  -- in some test mode
              }
        ACCESS  read-write
        STATUS  mandatory
        DESCRIPTION
              "The desired state of the interface. The
              testing(3) state indicates that no operational
              packets can be passed."
        ::= { ifEntry 7 }

ifOperStatus OBJECT-TYPE
        SYNTAX  INTEGER {
                up(1),      -- ready to pass packets
                down(2),
                testing(3)  -- in some test mode
              }
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
              "The current operational state of the interface.
              The testing(3) state indicates that no operational
              packets can be passed."
        ::= { ifEntry 8 }
```

Figure 9.18 *ifTable* Variables for Changing the Status of an Interface

9.3.6 SMI Transactions

Suppose a manager needs to perform a complex task consisting of several steps. This may happen, for instance, if there is misrouting of packets to destination 69.96.172.9 by an IP router. The manager may go through the following steps to troubleshoot and fix the problem:

1. Fetch all entries in the routing table with the destination matching any prefix of 69.96.172.9 (such as 69.96.172.xxx,69.96.xxx.xxx) using SNMP Get.
2. Identify the incorrect entry.
3. Change the incorrect entries using SNMP set.
4. Test the new route using ping. If ping fails, repeat steps 1–3.

During this process the manager needs exclusive access to the routing table. If any other manager reads and modifies the same rows, confusion may result. What is required is a transaction that may include a sequence of Gets and Sets to a subtree of the MIB. The SNMP protocol does not support this. Each SNMP request is atomic and independent of all other requests. One request cannot contain both Get and Set. Only a small number of variables can be simultaneously accessed in a request, limited by the SNMP message size of 484 bytes (v1 and v2).

To implement complex transactions without modifying the SNMP standard, we need to add a few variables with special semantics (DECRIPTION field). Assuming that we need to provide transaction-based access to *aRoot* and its subtree, the extra variables are shown in Figure 9.19 and the corresponding declarations are shown in Figure 9.20. Note that aData1, aData2,… are nodes in the protected subtree. The nodes numbered >1000 are nodes added to support transactions.

To initiate a transaction, the manager attempts to set *aLock* to *true*. If the lock was already held by another manager, the set fails with an error. The manager periodically retries the set. When the set succeeds, the agent stores the manager's identification (typically, IP address and port number) in *aManagerInfo*. During the transaction, only this manager is permitted to access the subtree rooted in *aRoot*. In the normal course after completing its work, the manager ends its transaction by setting *aLock* to *false*.

Suppose the manager forgets to release the lock. After *aTimeout* elapses, the lock is automatically freed. There will be a default timeout, which can be modified by the manager, subject to a maximum to prevent hogging of the lock with starvation of other managers. Occasionally, an emergency situation may arise for which an authorized manager may need to preempt the lock. This is done by writing to *aForceUnlock*. The agent will accept the write and replace the current lock owner with the initiator of the forcible unlock, provided the initiator is in a list of authorized managers.

Thus, it is possible to provide complex transactions in which a manager has exclusive access to a subtree of an MIB during a session. This is accomplished within the constraints of SNMP and SMI

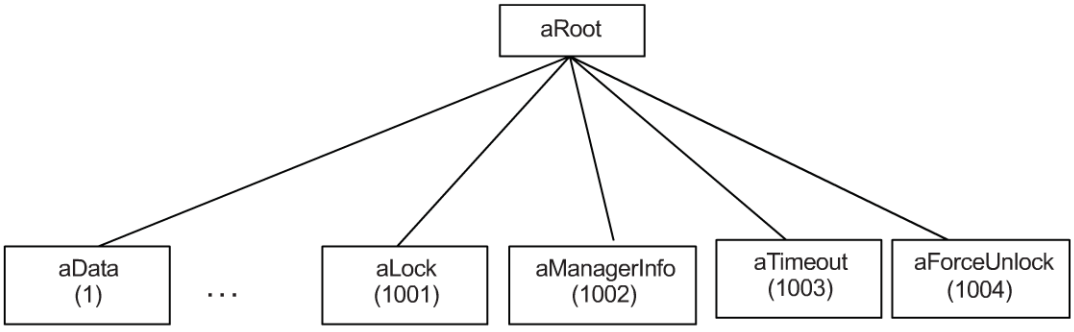

Figure 9.19 MIB Variables for Complex Transactions

```
aLock OBJECT-TYPE
        SYNTAX  Boolean
        ACCESS  read-write
        STATUS   optional
        DESCRIPTION
                "If it is false, set true succeeds and manager info is saved. If
                true, set true by owner succeeds and resets timer. If true, set
                true by other manager fails with badValue(3) error."
        ::= { aRoot 1001 }

aManagerInfo OBJECT-TYPE
        SYNTAX  ManagerInfo
        ACCESS  read-only
        STATUS   optional
        DESCRIPTION
                "Contains the id of the manager who most recently succeeded in
                acquiring aLock."
        ::= { aRoot 1002 }

aTimeout OBJECT-TYPE
        SYNTAX  Time Ticks
        ACCESS  read-write
        STATUS   optional
        DESCRIPTION
              "aLock will be reset aTimeout Ticks after being set."
        ::= { aRoot 1003 }

aForceUnlock OBJECT-TYPE
        SYNTAX  Boolean
        ACCESS  read-write
        STATUS   optional
        DESCRIPTION
           "If set by an authorized manager, alock is reset."
        ::= { aRoot 1004 }
```

Figure 9.20 Declarations for Figure 9.19

by defining a few auxiliary variables. To cater to specific needs, the set of auxiliary variables can be changed, for instance to define separate read and write locks, etc.

9.3.7 Summary: MIB Engineering

In this section we first outlined some of the limitations of SMI and SNMP. We discussed the support provided by SMI for the object-oriented design of complex MIBs. Finally, we presented MIB engineering techniques for handling tables, performing actions, and supporting complex transactions. To learn more about MIB engineering, the next step is to read some of the available MIBs for devices similar to the one at hand and to follow the patterns therein. The guiding principle in MIB engineering is that *the MIB should facilitate the common needs of most managers.*

9.4 NMS DESIGN

We now turn to the design of an NMS server for a large telecom or enterprise network. We first outline the requirements of such an NMS. This determines the architecture. Next, we go into details of the important components of the server.

9.4.1 Functional Requirements

The characteristics of a telecom or large enterprise network were discussed in Chapter 1. Telecom networks today provide a variety of services including voice, data, Internet, and video. A large telecom network serves 10s to 100s of millions of subscribers. For example, India, which has the fastest growing-telecom base in the world, had 413 million subscribers in February 2009 (http://www.telecomindiaonline.com). Its two largest service providers were Bharti Airtel with 90 million and BSNL with 80 million subscribers. India's telecom base is expected to double in the next 4–5 years.

The NMS is expected to support the full range of fault, configuration, accounting, performance, and security (FCAPS) functionality (Section 3.10). Based on these, the key requirements are given below.

Scalability. Typically, such a network has a large number of manageable devices including switches, servers, routers, base stations, multiplexers, etc. These may range in number in the thousands in a large enterprise or Tier-3 telecom network to several million in the largest Tier-1 telecom networks. In most networks, the number of elements grows with time. By scalable we mean that the same NMS can be deployed in networks of different sizes merely by changing the hardware platform. For example, if dual-processor servers with a 2-GB RAM and a 200-GB hard disk are sufficient for a network with N elements, a network with $10N$ elements may require a 16-processor server with a 32-GB RAM and a 1-TB disk. This requires a software design that can exploit the concurrency of the 16-processor server.

Scalability is important to the network administrator. Having invested in one NMS and having developed experience in using it, the administrator would expect the same product to be useable as the network grows in size. For the NMS vendor, scalability is important for a different reason. Developing an NMS is expensive, requiring several years of development by a large team of software engineers. Having invested so much, the vendor would like to maximize its revenue by selling the same product to many customers.

Heterogeneity. As the network evolves over many years, it encompasses a diversity of technologies, vendors, and types of equipment. These may support different management protocols such as SNMPv1, SNMPv2, SNMPv3, CMIP, etc. Some devices may use proprietary protocols. The NMS should easily accommodate this diversity without increasing the complexity faced by the operator.

Geographic Spread. We are interested in managing networks with a wide geographic spread, which has three implications. First, the WAN links used to connect the NMS to distant NEs may have a limited bandwidth, say 64 Kb/s or less, compared to 100 Mb/s or more for a 100m LAN connection. This bandwidth may have to be shared with subscribers' traffic. Hence, the amount of management traffic generated must be limited. Second, the end-to-end delay for an IP packet may be 10s or 100s of milliseconds or even several seconds compared to a submillisecond on a LAN. Third, long-distance links are likely to fail from time to time resulting in temporary disconnection of some parts of the network.

Bursty Load. Some activities such as polling *ifInOctets* for performance reports are done at regular intervals. These present a steady, predictable load to the NMS. Other activities such as handling faults are unpredictable. They are characterized by long periods of quiet with short bursts of activity.

Figure 9.21 shows the number of events generated in a live telecom network, measured at 15-minute intervals during a 24-hour period. The minimum number of events during an interval is 10, the maximum is 947, and the average is 153. See Jagadish and Gonsalves [2009a] for a more extensive discussion of observed fault characteristics of a telecom network.

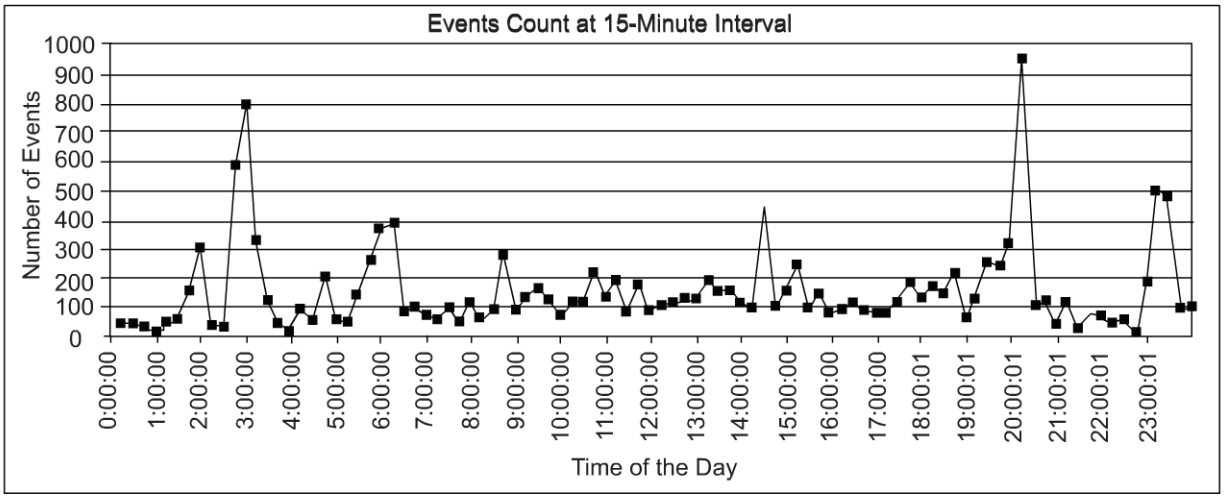

Figure 9.21 Occurrence of Fault Events in a Typical Telecom Network during a 24-Hour Period

It is generally too expensive to dimension the server hardware for the extreme peak load. Hardware is dimensioned well above the average load, but below the peak. NMS software must be designed to handle peak overload conditions gracefully. It may temporarily delay or discard less important events, but it should not fail altogether.

Real-Time Response. Fault notifications require responses within a short time. This may range from under 1 second to 10s of seconds depending on the nature of the event. The response may be an automatic control action by the NMS, or it may involve manual intervention by an operator.

Batch Processing. Performance monitoring, especially for capacity planning, requires a periodic analysis of large volumes of polled data. While imposing a heavy load on the server hardware, this should not be allowed to degrade the real-time response.

Diverse Users. The NMS has a variety of users. These include:

1. *Administrators*: These are experienced and privileged staff who have a high level of responsibility for the management of the network. They may perform sensitive tasks such as bringing a major switch online or offline, reformatting the disk in a server, granting permission to other NMS users. If a mistake is made in any such task, it could have serious repercussions on the operation of the network, even leading to considerable financial loss.
2. *Operators*: Many staff perform routine operational tasks that affect management aspects of the network to a lesser extent. These include generating traffic reports, diagnosing and repairing an individual port in a switch, answering customer queries, etc.
3. *Subscribers/Clients*: Customers of the network may be given access to see the health of their access to the network, to see reports of their SLA agreement such as bandwidth use, etc. They may be able to configure some parameters of their service, e.g., whether or not an email warning is sent when they near their monthly download limit. They cannot change anything else in the network or view other customers' information.

Clearly the NMS must have support for the creation of a large numbers of users, each having different privileges. The menus and commands that are available to each user should be limited to those that are necessary for the role of that user.

Local and Remote Management. Much of network management is performed remotely from the *Network Operations Center (NOC).* However, there are users who need to use the NMS from other locations. Subscribers may be located anywhere in the network. A technician, who is replacing a faulty hardware unit, may need to login to the NMS from the field location to divert traffic from the faulty unit, configure the replacement unit, and restore traffic routing. Accountants and top management may want to see reports on network resource use, expenses, and generation of revenue from their offices.

Ease of Use. Network administration is faced with two challenges: one is the increasing complexity of network equipment and the other the high attrition rate of technical staff. Hence, it is important for the NMS to have an easy-to-use UI. Increasingly, this is graphical (GUI) and may be browser based. At the same time, experienced users may prefer cryptic commands and keyboard shortcuts that enable them to work faster than with a point-and-click interface.

Security. There are two aspects to security; the first is the security of the network and network elements. This includes providing secure access to network elements for privileged administrators and operators, preventing and detecting malicious attacks such as the denial of service attacks, and monitoring and ensuring the confidentiality of sensitive traffic on the network.

The second aspect is the security of the NMS itself. The NMS perforce has to have access to every network element. It can perform arbitrary configuration of network elements. It stores vast amounts of commercially sensitive data about the network operation. As such, it introduces a single point of vulnerability to the network: a person who can gain unauthorized access to the NMS may well have complete unfettered freedom to tamper with the entire network!

Data Management. With tens of thousands to millions of network elements, each being polled regularly for several variables, the volume of data collected can be very large. If lost, these data can never be recovered. Careful backup of data is essential.

These data need to be kept online for 3–12 months and archived offline for several years. In some countries, preservation of certain network data for several years is mandated by laws and regulations.

For a telecom network with 10 million subscribers, the typical volume of data collected is shown in Table 9.5. (Details of the calculations are given in the exercises at the end of this chapter.)

Note that the disk space requirement is 2–3 times the data volume shown in the table. This is to account for index files created by the database management system (DBMS), temporary copies of data made by the NMS applications, and temporary working space.

9.4.2 Architecture of the NMS Server

We now examine the architecture and major design aspects of the NMS server in light of the requirements discussed in Section 9.4.1. In many cases there is no one correct decision, only design trade-offs where different designers may prefer different choices. This is the nature of real-world design. In these cases, we discuss the range of good choices.

Table 9.5 Typical Data Storage Requirement for a 10m Subscriber Network

PERIOD	1 DAY	1 WEEK	1 MONTH	1 YEAR
Data Volume	2 GB	15 GB	60 GB	750 GB

The NMS server has to handle diverse tasks with very different characteristics. These include polling of performance parameters at regular internals, real-time response to unexpected events, and a good user interface (UI). It has to deal with a plethora of device types. It is a large and complex software application and we hence adopt a *Modular architecture* [Bahrami, 1999]. A set of closely related functions is grouped into a module. These functions interact closely and use shared data structures. The different modules interact relatively infrequently.

A typical architecture for an NMS server is shown in Figure 9.22. This section draws many lessons from the CygNet NMS [Gonsalves, 2000; www.nmsworks.co.in], which has been deployed in several major telecom and enterprise networks.

A number of modules are grouped around a central *managed object database* (*MDB*). The *MDB* contains the configuration information of each managed object. A managed object is typically an NE (router) or a subsystem of an NE (interface of a router). The *MDB* contains the events and performance data of each managed object (MO). The *MDB* must be designed for efficient access to large amounts of data. It contains a code to validate the data and maintain consistency.

The *Discovery Module* is responsible for automatically detecting the presence of new NEs in the network. If these are of interest to the administrator, the discovery module adds corresponding MOs to the *MDB*. The discovery module is a part of the *Configuration Manager*, which also has provision for the manual addition of MOs. The discovery module also detects changes to the configuration of an NE and updates the *MDB* accordingly.

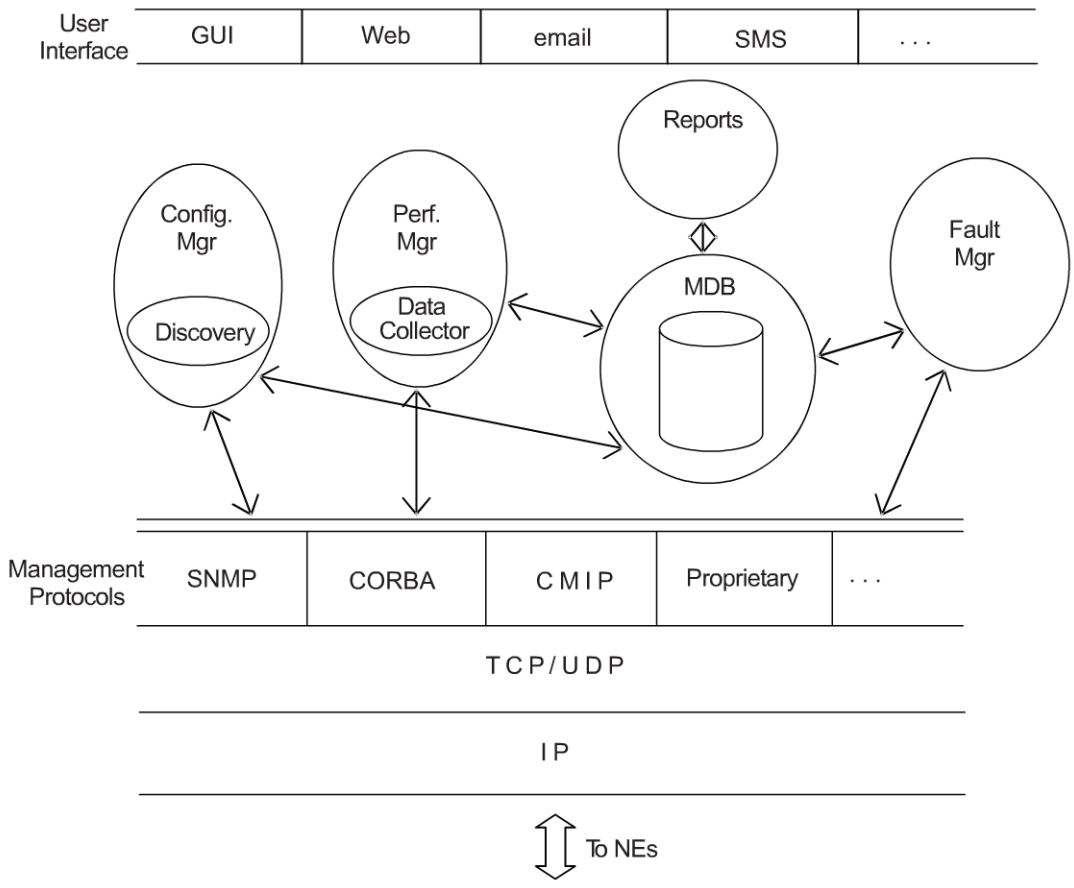

Figure 9.22 Architecture of an NMS Server

The *Fault Manager* (FM) receives notifications of events in NEs. It may also infer faults by analysis of data. For instance, by comparing measured throughput on a link with link capacity, it can detect a congestion fault. The FM notifies the operator through various means such as text, graphics, audio, etc. It may also take automatic corrective action to resolve a fault.

One of the important functions of the *Performance Manager* (PM) is data collection. This is done by periodically polling NEs for relevant performance data. Note that some of these data may be used by the FM as described above. The other important function of the PM is data analysis. Reports are generated to identify bottlenecks, analyze trends in order to plan capacity upgrades, and to tune the network and systems. Anomalies in performance trends may also indicate security problems such as denial of service attack.

Each module has two logical layers. The lower or *core* layer contains the business logic of the module. For example, the logic of sending periodic poll requests, processing the response, and updating the MDB is in the core layer of the PM. The upper layer contains the UI. Typically, this is a graphical user interface (GUI). It may also use text, audio, video, SMS, and so on.

Functional modules communicate with the NEs through the protocol layer. This layer may support a number of common management protocols such as various versions of SNMP, XML, CMIP, and also proprietary protocols.

9.4.3 Key Design Decisions

In this section we describe key design decisions motivated by the requirements in Section 9.4.1. This is followed by a detailed design of the functional modules.

The NMS server has to perform many functions. It deals with real-world entities such as switches, base stations, routers, servers, subscribers, and operators. It needs to handle the properties of these entities and evenly relate to them. It is natural to use an *object-oriented design* [Bahrami, 1999]. Software objects correspond one-to-one to physical objects. This simplifies design, makes it easier to adopt the design to new requirements, and results in more robust software. Most NMSs are written in C++ or Java.

The design needs to be **scalable**. That is, the same software should be able to cater to a network of 100, 1,000, or 1,000,000 elements. The design approach that accomplishes this using the choice of hardware and software is illustrated in Figure 9.23. Scalability can be achieved to a limited extent by using more powerful hardware. Suppose that 100 NEs could be managed using a server with a 1-GHz CPU (Figure 9.23(a)). Replacing the server with a 3-GHz CPU may increase the capacity to 300 NEs (Figure 9.23(b)). Only limited scalability can be achieved by this approach: today, 3 GHz is the practical limit of CPU speeds.

To further increase the capacity, we may consider a server with multiple CPUs sharing memory (a shared memory multiprocessor (SMP)). Such SMPs are available inexpensively with up to eight CPUs. However, a sequential program can use only one CPU. To use multiple CPUs simultaneously, a concurrent software design is needed. This is achieved by using multiple *threads of execution* [Tanenbaum, 2007]. Each thread runs on one CPU and can access the same shared memory. Thus, by redesigning our software to use threads, we could manage $100 \times 3 \times 8 = 2{,}400$ NEs with an 8-CPU 3-GHz server (Figure 9.23(c)). These capacity increases are the ideal. In practice, the actual increase in capacity will be lower as one thread may wait for another one, or threads conflict to access the same variables.

Beyond a point, an SMP becomes prohibitively expensive. To further increase the capacity, we need to again redesign our software to enable it to run on a cluster of SMPs interconnected by a high-speed

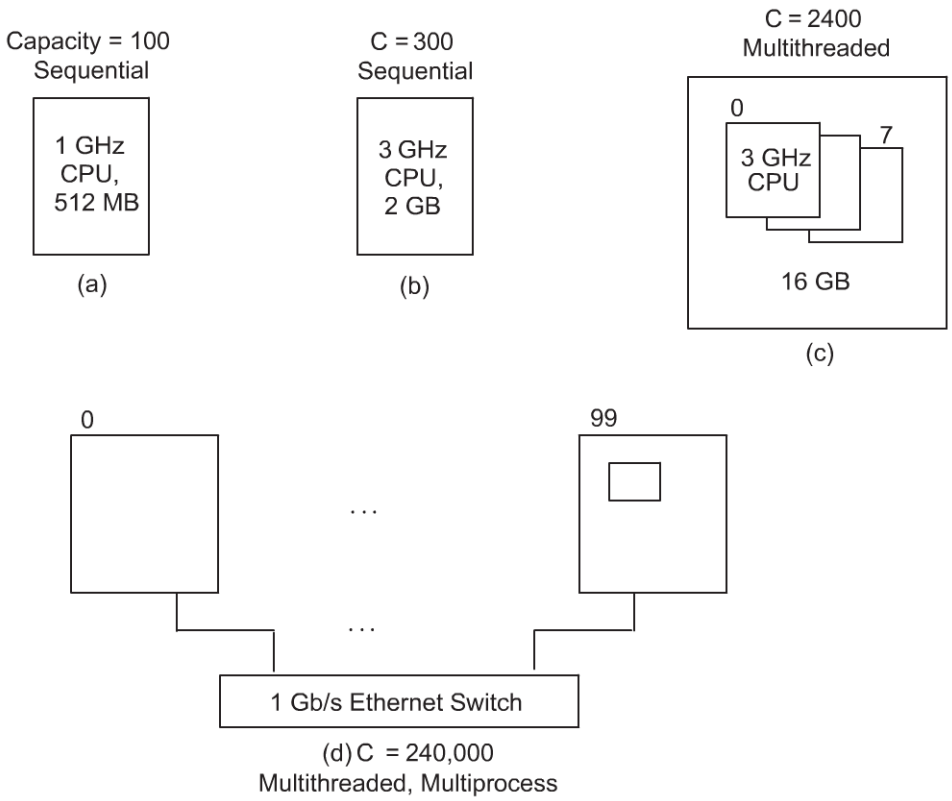

Figure 9.23 Scalable Design Using Threads and Processes

LAN. The software needs to be structured as cooperating *processes* that communicate with one another using interprocess communication (IPC) such as TCP/IP, RPC, Java RMI, or XML/HTTP [Tanenbaum, 2007]. With 100, 8-CPU SMPs connected by a 1-Gb/s Ethernet, the capacity of our server could scale up to 2,400 × 100 = 240,000 NEs (Figure 9.23(d)).

Note that the numbers of NEs above are the managed NEs only. The total number including unmanaged PCs, telephones, etc., may be one to three orders of magnitude larger. The actual capacity also depends on the polling interval, etc., so the numbers in Figure 9.23 are indicative only.

The server needs to handle event notification in real time. It also periodically needs to generate reports. To assure that real-time processing is not delayed by periodic batch processing, the former is given higher priority. If the software is structured as *threads and processes*, priorities can be assigned to these and the OS will ensure the real-time response.

With a large number of NEs and data saved online for months to facilitate analysis, the NMS has to deal with enormous volumes of data. Most of the data are structured with well-defined fields such as <NE id, time stamp, OID, value> for a performance parameter. Appending incoming data to a flat file is very efficient. However, retrieval based on criteria is very inefficient. All major NMSs use an R-DBMS for data storage. Examples of commercial R-DBMS products are Oracle, DB2, and SQL server. Open-source MYSQL and PostgreSql products are also commonly used.

We next consider the UI. To minimize training costs and to allow non-technical people (such as accountants and top management) to use the NMS, it is normal to have a *graphical user interface* (GUI). Nowadays, the GUI is often *browser based*. This further reduces the learning curve and ensures that the GUI will run on a wide range of desktop PCs.

The NMS needs to present a large amount of information to the operator. A well-designed GUI can use color-coded symbols and icons to present this information without confusion. Color codes can be used to highlight fault information that needs urgent attention. A judicious use of audio further improves the UI. Network administration is a 24 × 7 activity. Many NMS servers have the ability to notify the operator via SMS, pager, or telephone call with text-to-speech synthesis.

9.4.4 Discovery Module

In a large telecom network, the details of the NEs to be managed need to be entered into the NMS database. Doing this manually is a daunting and error-prone task. The goal of the discovery module is to automate this task. Initially, the discovery module tries to find manageable elements in the network and add their details to the NMS. Subsequently, it periodically checks each NE for any changes in this configuration. The discovery module also tries to determine the topology of the network.

Discovery supplements, but does not replace, manual configuration. For example, the NMS could discover that a router has ten interfaces. Polling intervals for the interface traffic depend on the importance of the connected links. Discovery may set a default polling interval of 15 minutes. An administrator may set the intervals for important interfaces to 5 minutes. For a backup link, the administrator may change the interval to 1 hour.

The strategy used in the discovery process is to check a range of IP addresses for the presence of NEs using the simplest and most generic possible techniques. When an NE is found, the discovery process attempts to find out the type of NE and its characteristics by using more elaborate and specific techniques.

Configuring Discovery. Discovery is controlled by a configuration file, which sets a number of parameters. A basic set of parameters is shown in Table 9.6. Many more parameters could be specified.

Discovery Procedure. The flow chart of the discovery procedure is shown in Figure 9.24. "Check if node present" is normally done using ping. As ping is unreliable, 2–3 ping requests are sent. In some locations, ping may not work because its ICMP echo request and echo response packets are blocked by firewalls. In such cases, attempting to open a TCP connection to a port may be used. This is more expensive in terms of network bandwidth and CPU resources on both ends.

Table 9.6 Basic Discovery Parameters

PARAMETER	VALUE	DESCRIPTION
IP addresses	10.0.0.1–10.0.0.254, 192.168.0.0/24	A range or list of IP addresses
Wait interval	10 seconds	Waiting time between discovery of successive IPs to minimize load on the network
SNMP version	v1	v1, v2c, or v3
SNMP community	"public"	A commonly used value
Discover types	Router, server, switch	Only elements of these types are added to the MDB
Ignore types	PC, UPS	Elements of these types are not added to the MDB

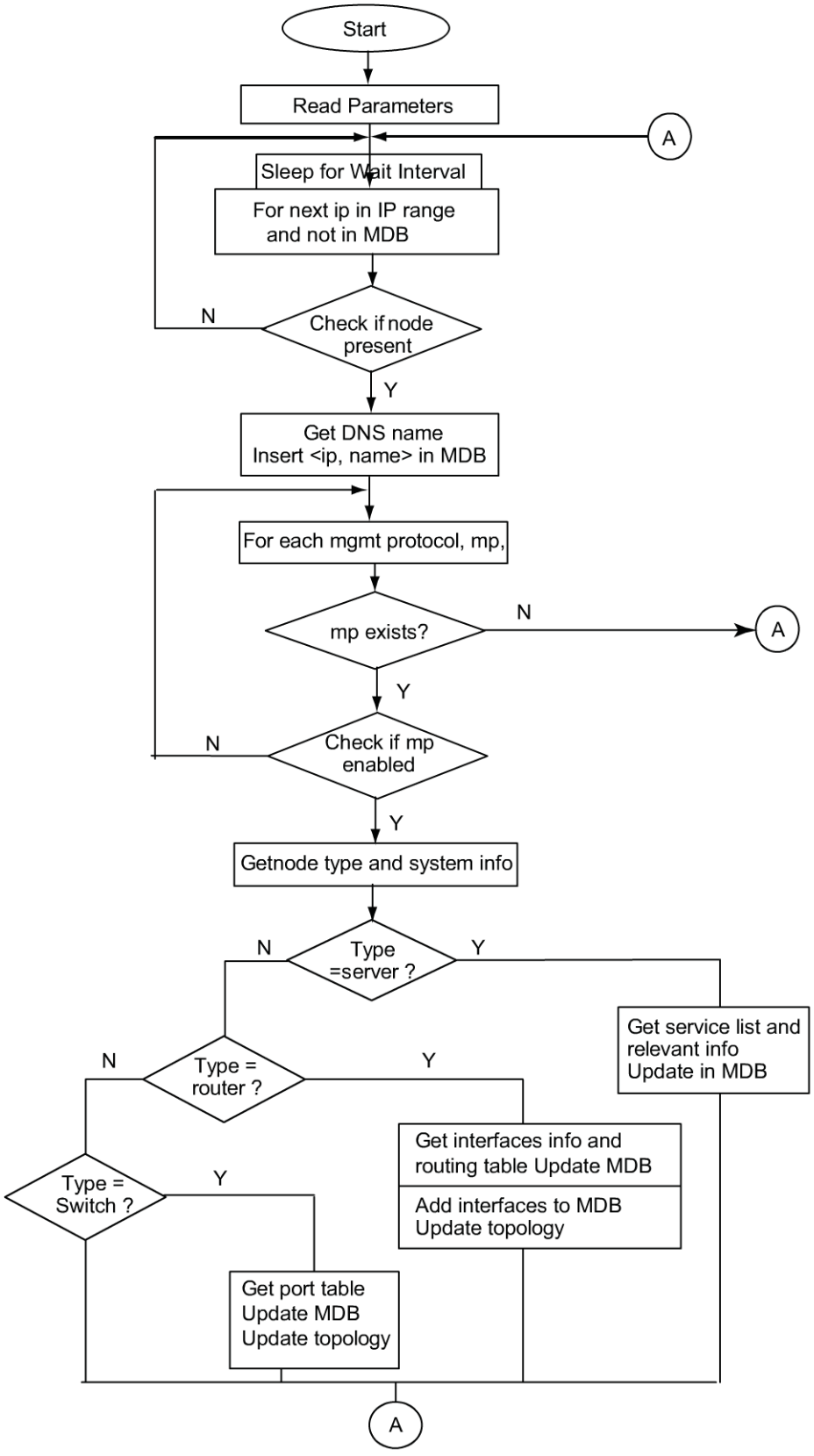

Figure 9.24 Discovery Procedure

```
SNMPget(ip, v3, auth info, sysObjectId)
If response then node is v3
Else SNMPget(ip, v2c, community, sysObjectId)
If response then node is v2c
Else SNMPget(ip, v1, community, sysObjectId)
If response then node is v1
If node is v1, v2c or v3 then
Lookup sysObjectId to determine type of node
Based on node type, use SNMPget, SNMPgetnext, SNMPgetbulk to retrieve
relevant OIDs and tables.
```

Figure 9.25 Discovery of an SNMP Node

Inserting a newly discovered element into the MDB usually also involves configuring polling for OIDs dependant on the type of NE, configuring a trap receiver and other event detectors, and adding an icon to the pictorial map of the network. Finally, an administrator may be notified to manually approve and possibly modify the parameters and settings for the new NE.

Checking for which management protocol is enabled is done by sending a series of requests to glean information. For example, a typical sequence for SNMP is shown in Figure 9.25. In the steps that check for the SNMP version, it may be necessary to check for several possible authentication parameters. For example, suppose the read community for PCs in a network is "public," for routers it is "x375tz," and for servers it is "private." For every IP, each of these communities is tried until one is accepted by the agent.

Rediscovery. Periodically, the NMS may check NEs that are in the MDB for any change in their configuration. This process is referred to as rediscovery. Some changes may be detected by the NMS during normal operations. For instance, if the PM is polling an interface on a router and that interface is permanently disconnected. Polls will fail and the FM will show the interface status as down. The operator who disconnected the interface will notice this and delete the interface from the MDB.

Suppose, however, that a new interface is connected to a router. This will not be detected by the polling process. Similarly, new NEs may be connected to the network. A server that was serving as an email relay may be redeployed as a Web server.

Rediscovery follows a simpler flow chart compared to discovery. We already know the management protocol and authentication information and the configuration. Rediscovery mainly involves fetching the configuration information from the NE and comparing it with the details in the MDB. Any new services or subsystems detected are updated in the MDB and notified to an administrator for review. Any removal of existing services or interfaces is presented to an administrator for confirmation before deletion from the MDB.

Topology. The topology of a network depends on the interconnections between routers and switches. Hubs are usually not SNMP-manageable and are not included in topology discovery. During the discovery process, the discovery module determines the type of each NE and hence identifies the routers and switches. It then obtains topology information from the MIB of the router or switch (last two items in Figure 9.25).

In a router, every active interface has an IP address. This is given by the *ipAdEntAddr* MIB variable (Table 4.8). For each interface, the neighbor's IP address can be found in the *ifForwardNextHop* MIB variable (Table 4.11). The variables *ipAdEntIfIndex* and *ipForwardIfIndex* in these two tables form the

link. The *ifType* and *ifSpeed* variables in the interface table give the type and speed, respectively, of the link between this router and the neighboring router. Next, the discovery process is applied to each of the neighbors to further extend the topology. Note that to avoid discovering a very large number of nodes, many of which may be outside the domain of the NMS server, discovery may be limited to a certain number of loops. It is also usually limited to the ranges of IP addresses that belong to the organization.

When discovery encounters a layer 2 switch, it can find the number of ports from the *dot1dBaseNumPorts* MIB variable [RFC 1493]. If there are several switches in the network, the spanning tree table (*dot1dStpPortTable*) can be used to find the topology of the extended LAN. Finally, discovery can find the MAC address of machines connected to each port using the forwarding database (*dot1dTpFdbTable*).

Figure 9.26 shows an example topology and information about the network that could be obtained automatically through discovery. The discovery process has created a simple map with each link labeled with its speed and type. The width of the lines is indicative of the link speed. Each interface is labeled with its IP address. The layout of the nodes and links is arbitrary. After discovery, the NMS administrator can manually reposition nodes and links to reflect their geographical positions.

Efficiency Issues. In Figure 9.24 we showed a sequential algorithm that checks one IP address at a time. For any IP that is not in use, the process waits for three pings to timeout, say $3 \times 10 = 30$ seconds. Assume that for each address in use the discovery takes 5 seconds.

Table 9.7 shows the total time taken for discovery for different networks with varying percentages of IPs in use. In the first row, we have one Class C network that has a total of about 250 assignable addresses. Of these, varying percentages are actually used. Likewise, in the second row we have ten Class C networks. Consider the first cell in the table. The total discovery time is calculated as:

Discovery time = Number of IPs in use x success time + number of IPs not in use × timeout time

$= 250 \times 0.05 \times 5 + 250 \times 0.95 \times 30$ seconds

$= 62.5 + 7{,}125$ seconds ≈ 2.0 hours

The other cells are calculated similarly.

It is clear from Table 9.7 that sequential discovery is practical only for very small networks. During discovery, the discovery module spends most of the time waiting for the response or timeout. The CPU time for each IP that is tested may be at most a few milliseconds, while the network delay may be 100s of milliseconds (success) to 10s of seconds (failure).

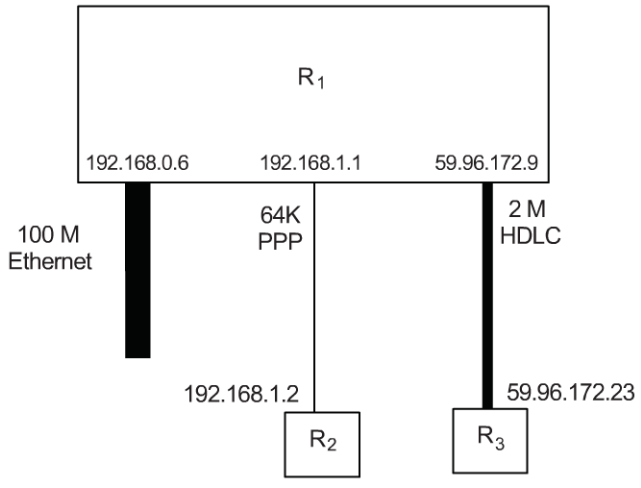

Figure 9.26 Example of Topology Discovery of WAN Links

Table 9.7 Time to Complete Discovery Sequentially

TOTAL ASSIGNABLE ADDRESSES \ % ADDRESSES IN USE	5%	20%	50%	100%
1 Class C ~ 250	2.0 hr	1.7 hr	1.2hr	0.3 hr
10 Class C ~ 2,500	20.0 hr	17.4 hr	12.2 hr	3.5 hr
1 Class B ~ 65,000	*21.6 dy*	*18.8 dy*	*13.2 dy*	*3.8 dy*
16 Class B ~ 1,000,000	*332.8 dy*	*289.4 dy*	*202.5 dy*	*57.9 dy*

Note: Numbers in italics are days, normal font in hours

Hence, it is feasible to have concurrent testing of several IPs without overloading the NMS server. The discovery module could transmit messages to a number of IPs in quick succession without waiting for replies. Later, when each reply is received, it is matched with the corresponding request and processed appropriately.

Suppose 100 IPs are tested concurrently. The time to complete discovery decreases dramatically as shown in Table 9.8. For small networks, discovery takes a few seconds to a few minutes. For a large network, the time ranges from an hour to 3 days. By increasing the concurrency, we could reduce these times further. For a network with 5% of 1,000,000 IPs in use, i.e., 50,000 NEs, 3 days is acceptable. Keep in mind that the administrator needs to verify and perhaps correct the discovered elements, which would take much longer than 3 days!

Thread Pool or Worker Pool A convenient method of implementing concurrent discovery is by using a pool of worker threads (Figure 9.27). The discovery controller creates *tasks* from the range(s) of IPs to be discovered, each task containing one or a few IPs. It adds these tasks to the *Task Queue*. Next, it creates a number of *worker threads*. Each of the workers picks up a task from the Task Queue when it is free. It executes the sequential discovery algorithm shown in Figure 9.24 for each of the IPs in the task. Whenever a worker completes discovery of the IPs in one task, it takes another task from the queue. Thus, all workers are kept busy until the task queue is empty.

Note that the Task Queue is a shared data structure and must be accessed by only one worker at a time. This exclusive access can be implemented using semaphores or other mutual exclusion mechanisms [Tanenbaum, 2007]. The Task Queue is accessed twice per task, once to add the task and once to remove it. If access to the Task Queue is very frequent, the mutual exclusion overhead could become a bottleneck. To avoid this, the controller normally bundles several IPs in one task. E.g., with 1,000,000 addresses and 1 IP per task, the number of accesses to the Task Queue is 2,000,000. If 250 IPs are bundled per task, the number of tasks is reduced to 4,000 and the number of accesses to only 8,000.

The optimum number of worker threads depends on the range of IPs, the hardware configuration of the NMS server, and the load generated by other modules. For flexibility, upper and lower limits on the number of workers can be configured in the discovery parameters. The controller then adjusts the actual number in the pool dynamically within these limits.

9.4.5 Performance Manager

The PM has two major functions. The first is data collection, which is an online activity. The PM continually monitors network elements for interesting performance-related statistics. These may be processed

Table 9.8 Time to Complete Discovery with Ten Concurrent Threads

% ADDRESSES IN USE / TOTAL ASSIGNABLE ADDRESSES	5%	20%	50%	100%
1 Class C ~250	72 s	63 s	44 s	13 s
10 Class C ~2,500	719 s	625 s	438 s	125 s
1 Class B ~65,000	*5.2 hr*	*4.5 hr*	*3.2 hr*	*0.9 hr*
16 Class B ~1,000,000	*79.9 hr*	*69.4 hr*	*48.6 hr*	*13.9 hr*

Note: Numbers in italics are hours, normal font in seconds

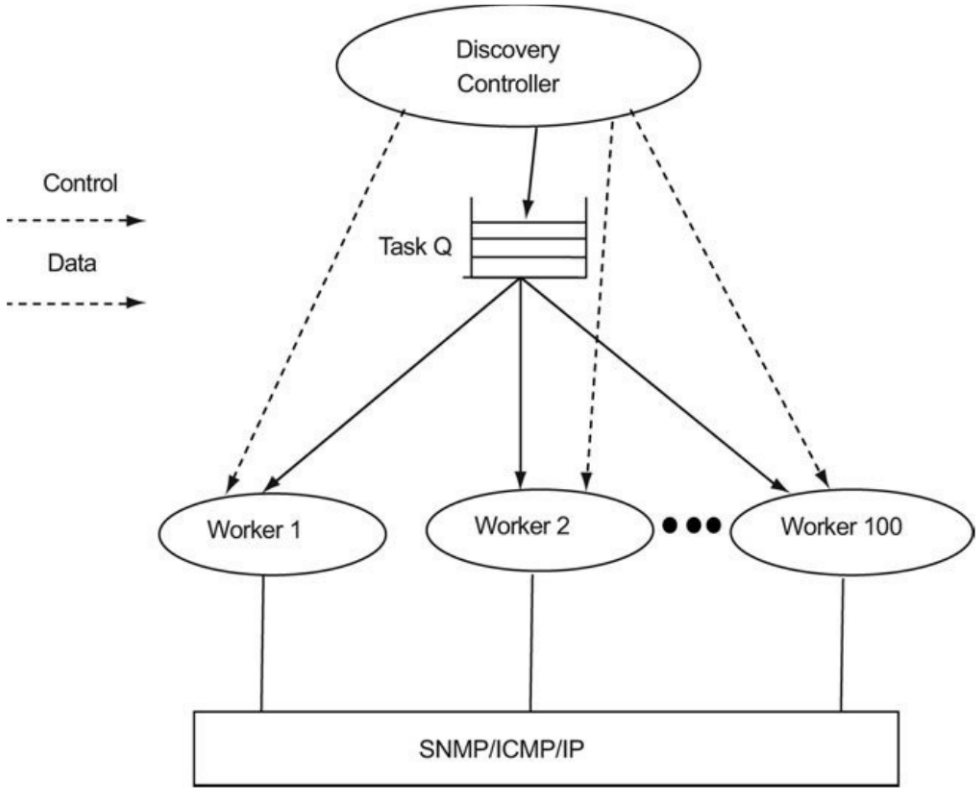

Figure 9.27 Worker Pool Design for Concurrent Discovery

and are stored in a database. Example statistics include traffic on a link (*ifInoctets* and *ifOutoctets*), rate of transactions hitting a server, and CPU utilization of a server.

The other major function is offline *analysis and report generation* based on the collected data. A common requirement is analysis of the utilization of a network link with projection of further growth in traffic. This is used to plan capacity upgradation before the link becomes a bottleneck.

Data Collection. The PM collects data for several reasons. Data may be used for analysis of trends in order to plan capacity upgradation. In this case, the data are not needed immediately. Hence, data collection could be done *offline*. The agent or a lower-level manager polls the variables of interest and

accumulates the values in a local file. Periodically, say once a day, a *bulk transfer* of the file to the NMS server is done. We refer to the lower-level manager as the *local data collector* (LDC).

Data collection may be done in order to detect congestion or to implement quality of service (QoS). In these cases, the NMS needs to take immediate action in case of a fault. Hence, *online* data collection is required. Here, the polling is done from the server directly to the agent.

In both cases, polled data are stored in the same database table. Each poll record contains at least the following fields:

- **NE:** identification of the agent, typically the DNS name or the IP address
- **Variable name:** OID or other name
- **Value:** collected value
- **Timestamp:** time of polling

We now consider some important issues in offline and online data collection.

Offline Data Collection. Let us consider the fields in the poll record. The NE *name* used by the LDC and the manager may be different. For example, the NE may have a local IP address that is used by the LDC for its polling, which is put into the local file. When the file is transferred to the manager, the manager needs to modify this to the name used by it for the NE, which may be a global address.

The *timestamp* inserted by the LDC is based on its clock. There may be a skew between this clock and the manager's clock. Since LDCs often run on low-end devices, the clock skew could be very large. The manager needs to estimate the skew and adjust the timestamps accordingly. The situation is made more difficult if the LDC reboots in the middle of the day and skews changes.

Finally, the manager needs to ensure that there are no *duplicate* records or lost records. After transferring the day's file, the LDC should delete it and start a fresh file. However, suppose the LDC does not delete the transferred records. The manager needs to check for duplicates by inserting a unique sequence number in every record.

Online Data Collection. For real-time performance analysis, the PM polls each NE directly for variables of interest. Some issues need to be addressed:

Avoid overloading the server: The PM may be polling several variables in a large number of NEs, e.g., 1,000 routers each having 20 interfaces. There are four variables of interest in each router. Thus, there are 80,000 polls. If polls are synchronized and each is done with a separate packet, the burst of 80,000 replies within a short span could overload the server.

The CPU load depends on the number of packets and the number of variables. In fact the per-packet overhead due to interrupts, etc., is often high. We can reduce the number of packets by combining a number of variables in one SNMP get request. For example, an SNMP get response with one variable may be 65 B long. If we pack four variables for one interface into one packet, the length may increase only to 140 B (compared to $4 \times 65 = 260$ B for 4 packets). Second, the data collector can stagger polls to avoid bursts of packets.

Avoid overloading the network: Suppose that ten of these routers are in a distant part of the network that is connected to the NMS by a 64 kb/s link. Suppose the NMS polls all variables every 30 seconds and it uses one packet of 200 B for each interface. The resultant traffic on the link is $(10 \times 20 \times 200)/30$ B/second = 10.67 kb/s. About 16% of the link bandwidth is used for polling only. In addition, traps, accounting, and configuration messages will consume more bandwidth. This can seriously affect subscriber traffic on this link. One solution is to configure different polling rates. Only a few important

interfaces are polled every 30 seconds. Other interfaces may be polled once in 5 minutes or 15 minutes. This will reduce polling traffic overhead to a negligible level.

Avoid overloading the agent: A single CPU in a low-end NE often performs the real functions of the NE (routing, switching, printing, etc.) and processing of manager requests. Too many manager requests could overload the CPU and cause its real functions to suffer. Mitigation techniques mentioned above will help this problem also.

Poll Configuration. Different metrics vary at different rates. For a fast-varying metric, say traffic on a backbone link, it may be necessary to poll the counters frequently, say once in 30 seconds–2 minutes. The free disk space on a server varies slowly. We may poll this only once an hour.

Hence, the data collector should allow the poll period to be specified independently for each variable on each NE. There may be several variables being polled on one NE, possibly at different periods. To reduce the CPU and network load, the data collector tries to club several polls into one network packet, e.g., one SNMPget-request command could accommodate 10–20 OIDs and their values.

Dynamic poll periods: Suppose a link is being polled at 15-minute intervals. If it experiences congestion, it is useful to increase the rate of polling to 5 minutes or even 1 minute. It is possible for the data collector to support such dynamic poll periods. However, conditions for changing the poll period are usually related to faults, accounting, etc. Hence, the better design is to have the FM, as part of its handling of the congestion fault, to re-configure the polling period for the variable of interest in the data collector.

A novel design optimizes both the accuracy of the collected data and keeps the network traffic and server load within limits [Jagadish and Gonsalves, 2009b]. The agent or LDC polls data at a high rate and pushes it to the NMS at a possibly lower rate. The LDC evaluates the error between the data that reach the manager and the actual data. It adapts the rate of pushing to the manager to keep the error within a specified accuracy objective. In addition, the NMS sends feedback to the LDC on its load and the importance of this NE in relation to other NEs. This factor, which changes dynamically depending on the faults in the network, is used to further adjust the rate of push.

Concurrency: In a large network, the data collector could be polling several variables in each of 1000s of NEs for a total of 10,000s of polls. If the average polling period is say 300 seconds, the aggregate rate of polling could be 100/second. This could easily overload the CPU. Hence, the data collector for a large network should be multithreaded and possibly multiprocessed. The *worker pool design* introduced in the discovery module (refer Section 9.4.4) is appropriate here also.

Overrun: Ideally each variable is polled at precise intervals. For example, if a variable is configured for a 2-minute polling interval, the polls should take place as shown in Figure 9.28.

In reality, polls may get delayed for a number of reasons. Owing to network loss/congestion, the response to one poll may reach only after the next poll time. The CPU may be overloaded, especially due to a burst of faults. Fault handling is usually a higher priority than polling.

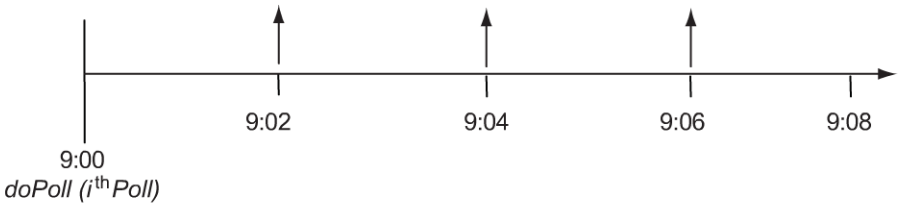

Figure 9.28 Ideal Polling at 2-Minute Intervals

A poll that is scheduled for 9:02, may actually be executed at 09:02:25. In case of severe congestion, the poll may be delayed to 09:04:45, i.e., after the next scheduled poll time. Suppose polls are scheduled at precisely 2-minute intervals, with one poll object being created for each poll. Under overload, a backlog of poll objects could build up in memory. This may slow down the system, further increasing the overload, and affecting the critical fault handling. If the overload persists, the NMS server could even crash.

The solution is to schedule only one poll at a time for each variable. When a poll completes, the next poll is scheduled at the first nominal poll time in the future. In the above example, if the first poll completes at 09:01, the second poll is scheduled at 09:02. However, if the first poll is delayed and completes only at 9:02:23, the second poll is scheduled at 09:04. This tactic of skipping a poll in case of overload ensures that the data collector does not further overload the system. The pseudo-code for static and load-sensitive polling is given in Figure 9.29 and Figure 9.30, respectively.

Database Schema. Polled data are stored in a database for ease of access and analysis. In a large network, with data stored for months or even years, the database size can become very large (see Table 9.9). The database schema need to be carefully designed to ensure good performance even with these very large sizes.

The minimum information stored for each poll of a variable is: NE name, Timestamp, Variable name, and Variable value.

The performance of report generation depends on the indices in the poll records. Of the four fields, it is normal to select records by timestamp (show traffic on 20/06/2008), by NE (show transactions on *www.server.com*), and by variable (show *ifInOctets* on several links). It is rare to select by value. Hence, tables should be indexed on the first three fields.

Assuming 4 B per field and three indices, the size of each poll record is 28 B. To account for the indices and temporary copies, we assume 28 × 2 = 56 B per record.

Assuming 10,000 NEs, 10 variables per NE, and an average poll period of 300 seconds, we have an aggregate polling rate of 10,000 × 10/300 = 333 polls/second. Database sizes over various periods are shown in Table 9.9.

```
doPoll (ith poll)
{
poll for OID
process value
schedule (i+1)th poll at tstart + i × pollInterval
}
```

Figure 9.29 Static Polling can Cause System Overload

```
doPoll (ith poll)
{
poll for OID
process value
let k = (tnow - tstart)/pollInterval
        //k is the index of the next nominal poll
Schedule poll at tstart+k × pollInterval
}
```

Figure 9.30 Load-sensitive Polling Algorithm

Table 9.9 Typical Growth of Collected Data

	1 DAY	1 WEEK	1 MONTH	1 YEAR
Number of records	29 m	201 m	863 m	10,501 m
Database size	1.6 GB	11.3 GB	48.3 GB	588 GB

A good DBMS is designed to perform well with the size of one table being up to a few million records and occupying 10s–100s MB of space. Beyond these limits, performance degrades and it is desirable to split the data into multiple tables. On the other side of the coin, having multiple tables makes programming more complex. It is easier to select relevant records from a single table than from a set of tables. DBMS performance also begins to degrade if there are 100s of simultaneously open tables.

The polled data can be divided into multiple tables in several ways:

- One table per NE
- One table per variable (or OID)
- One table per day

Let us examine the performance implications of each of these designs. We consider the number of open tables, the size of each table, the time for insert (done for every poll), and the performance for reports (typically done in batch mode). We also consider the safety of the data.

With one table/NE, the number of open tables is very large as all NEs are being polled, by say 10,000 open tables. The size of each table is modest and insert is fast as there is no conflict between multiple polls. Reports that analyze data across many NEs are tedious to develop. However, reports for one NE access only one table and hence may run efficiently.

Considering one table per variable, the number of tables is large but less than in the above case. The tables are of moderate size resulting in good performance. Insert is fast. Almost any report would need to access many tables, resulting in access conflicts and poorer performance.

One table/day results in only a few 100 tables. Further, all inserts are done in the current day's table only. Most reports analyze past data and hence there is no conflict between such batch reports and the data collector's inserts. A few online reports access the current day's table and cause occasional conflict.

Because of the structure of a DBMS table, even a few bytes of data corrupted in a table or its index could result in loss of the entire table. Hence, having a single table is a poor choice. One table per day is perhaps the best design. Loss of one table does not lead to loss of all information about some important NE. With one table per NE, we could lose all data collected about the most important NE!

Conclusions. Table 9.10 summarizes the different designs and their implications on performance and data safety. For a small network, a single table is a good choice. For large networks, one table/day is often the best choice, though some administrators may prefer one table/NE.

Note that in the 1-table/NE, the NE name is the table name and hence need not be a column (and index). Hence, the space per record is 20 B. Similarly, in the 1 table/variable case, the variable name is the table name.

Some NMS platforms give the administrator the option of configuring the number of tables for data collection and how the polled data are split among these tables. Once the choice is made it is difficult to change as the code for reports depends on the schema. The choice should be made with due diligence.

Table 9.10 Comparison of Different DBMS Schema Designs

	1 TABLE	1 TABLE/ NE	1 TABLE/OID	1 TABLE/DAY
Space/record	28 B	20 B	20 B	20 B
Number of open tables	1	Very large	Large	Few
Insert	Slow	Fast	Fast	Fast
Reports	1 table, conflict with insert	1 table, less conflict	Several tables, some conflict	1 table, conflict for online reports, no conflict for offline reports
Data safety	Poor	Good	Good	Very good

9.4.6 Fault Manager

Fault management is the most important function of the NMS: if the network has faults, performance and accounting become almost irrelevant. Because of the unpredictable nature of faults and their tendency to occur in bursts, the design of the fault management module is especially complex. If it is not done well, the NMS server itself could fail when the network experiences a burst of faults.

We first define the terms used in fault management. We then trace the path of a fault through the system. Finally, we touch on some advanced topics such as root cause analysis (RCA).

Definitions. A *fault* is a problem in a network element or network link. Examples are: a system goes down due to power failure; a link fails due to a cable cut; a disk gets full; and congestion on a link when traffic exceeds some threshold.

When a fault occurs, the NMS may detect it as an *event*. This may be done in different ways by a variety of *event detectors*. One form of event detector is a trap receiver that handles notifications sent by the NE. Another form of event detector is the poll manager detecting that an element is down when a poll fails. This is notified to the FM.

A single fault may result in several events in the NMS. For example, as SNMP traps are unreliable, the agent may send the same trap repeatedly to ensure that the manager receives it. These redundant events are filtered out by the *event correlator*.

When an event first occurs indicating the presence of a new fault in the network, if the fault is of interest to the operator, an *alarm* is created as the manifestation of this fault. The alarm should remain in the NMS as long as the fault persists. The alarm contains all the information about the fault and the actions taken by the operator or the NMS in response. Ideally, for every fault of interest, there should be exactly one alarm in the NMS.

The alarm is brought to the attention of the operator via an *alarm indication*. The indication may be graphical (change of color of an icon on the screen), audio, an SMS, an entry in a log file, etc. The type of indication depends on the importance and urgency of the fault.

Sometimes, one fault may result in events that cannot be easily correlated. For example, if a link fails, the routers on both ends of the link may independently report the fault. This results in two alarms being created in the NMS. Such situations are detected by the *alarm correlator*, which is a high-level analog of the event correlator.

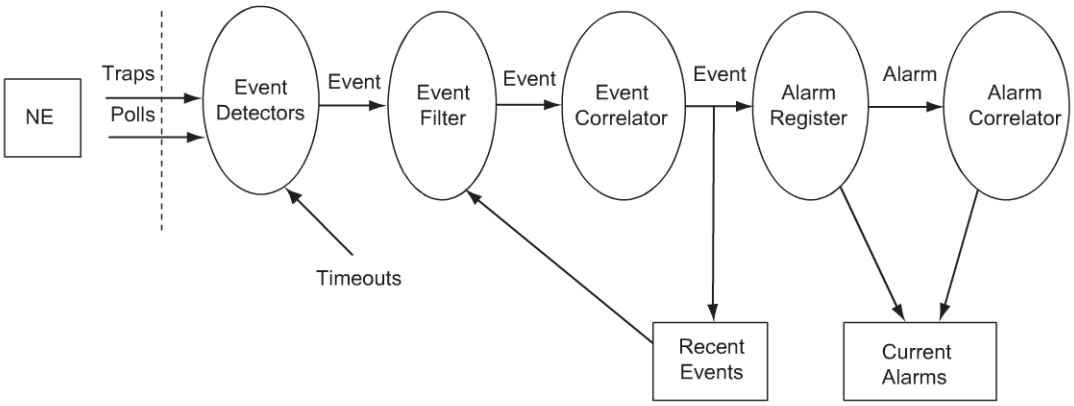

Figure 9.31 Path of an Event through the Fault Manager

Path of an Event in the FM. The first step in fault management is detection of the fault event by the NMS (Figure 9.31). This is done by one or more of a variety of **Event Detectors** in the FM. Different types of events and the corresponding event-detection mechanisms are given below:

1. Notification: This detector receives notification (such as SNMP traps) sent by the NE and converts them into event objects in the NMS.
2. Poll failure: When a status or performance poll request sent by the data collector in the poll manager fails, it indicates a fault in the NE. This is informed to the FM, again by creating an event object.
3. Performance threshold crossing: When the data collector polls for traffic variables such as *ifIn Octets* and *ifOutOctets*, it computes the corresponding rates. If any rate crosses a threshold, a thre-shold event is generated.
4. Internal escalation: When a fault occurs in an important element, the NMS may set a time limit during which the fault must be attended to. If this time limit is exceeded, an event is generated.

The next step is **Event Filtering**. An NE may generate events that are not of interest to the NMS. For example, a router may periodically send a notification about interfaces that have not been configured. Any such event that matches one of a set of conditions configured by the administrator is discarded by the event filter module.

Next, the event is processed by the **Event Correlator**. The event correlator first checks if this event is a duplicate of an event in the *Recent Events Table*. The event may be an exact duplicate (e.g., a router sends a link-down trap every 10 seconds as long as the link is down). The event may be closely similar to recent events (e.g., a link-down trap is received and soon thereafter polling for *ifInOctets* on that interface fails). In either case the duplicate event is discarded.

Since duplicate events may be separated in time, the event correlator must examine all events within a time window. Say, if the polling interval is 300 seconds, the time lag between the trap and the poll failure may be up to 300 seconds. In this case, the event correlator may examine events in a time window of 500 seconds. Of course, using too large a time window would result in the event correlator loading the NMS server unnecessarily.

The event correlator also has to consider the possibility that the link state has changed three times instead of just once. Consider the following sequence of events:

1. 10:31:05: Router7, If3, Link down, Detector: Trap.
2. 10:34:32: Router7, If3, Link down, Detector: Status Poll.

Clearly both events are caused by the same physical fault in Interface 3 of Router7. The first is generated by the trap receiver, the second by the data collector's status polling.

All the fields of the two events match except the timestamps and the event detector. Hence, the event correlator discards event (2). Next consider this sequence:

1. 10:43:08: Router7, If3, Link down, Detector: Trap.
2. 10:44:15: Router7, If3, Link up, Detector: Trap.
3. 10:46:10: Router7, If3, Link down, Detector: Status Poll.

If we simply compare fields as in the previous case, we would conclude that event (3) is a duplicate of event (1) and is to be discarded. However, a closer analysis indicates that the link went down, then came up within a minute and again went down a second time. The second link-down trap evidently did not reach the NMS. Event (3) should be accepted.

These examples indicate that the event correlator should search backwards in the Recent Events Table until one of three conditions is met:

1. It finds a matching event for the same NE and subsystem (such as interface): the new event is discarded.
2. It finds a different event for the same NE and subsystem: the new event is accepted.
3. It reaches the beginning of the time window without finding an event for this NE and subsystem: the new event is accepted.

An event that passes through the event filter and event correlator is now converted into an alarm by the **Alarm Registration** module. The FM has now recognized the existence of a new fault in the network. This alarm will be notified to the operator by suitable alarm indications. The alarm object is used to keep track of actions taken by the operator and NMS until the fault is rectified (discussed below). It is possible that one network fault results in the creation of two or more seemingly unrelated alarm objects. The **Alarm Correlator** and **Root Cause Analysis** modules address this problem. These are discussed at the end of this section.

Alarm Indications. The FM indicates the existence of an alarm to the operator in one or more ways. These include:

- *Visual*: the icon for the concerned NE changes color and may start flashing. A pop-up window may also be used to display information about the alarm
- *Audio*: A tone or other audible indication is played. Using text-to-speech synthesis (TTS), information about the alarm can be conveyed without the operator having to come near the display
- *SMS, phone call*: if the operator is not in the NOC, the NMS could send an SMS or dial out to the operator's phone and play a message (using TTS). This is useful outside normal working hours
- *Email*: if the fault does not require urgent attention, the NMS could send an email to the operator
- *Log*: in all cases, the FM writes a message to a non-volatile log on disk for post-mortem analysis. The log message contains at least a timestamp, name/address of the NE, previous state, new fault state, and descriptive text

For each fault, which one or more of these indications are used depends on the nature of the fault, the severity of the fault, the importance of the faulty subsystem and NE, and other faults that are currently pending.

Consider a router with two links, one is a leased line that connects the campus to the Internet, the other connects to a dial-up modem as a fallback in case the leased line fails. Now, suppose that a periodic

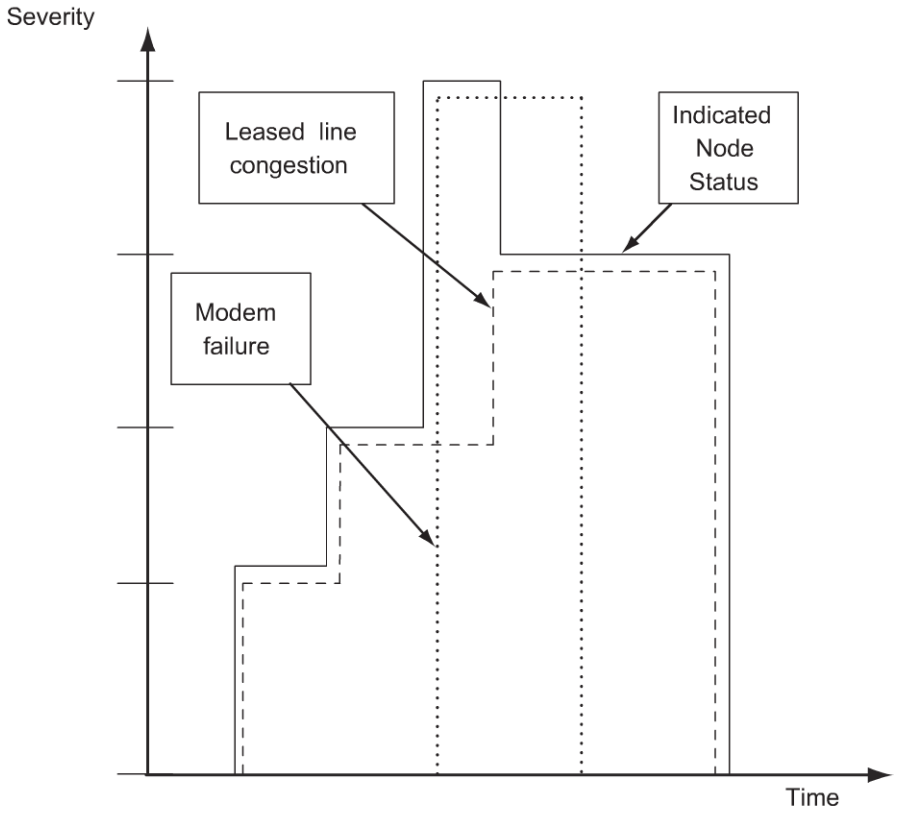

Figure 9.32 Alarm Indication for a Router with Two Simultaneous Faults

self-test of the modem fails—an alarm of critical severity (red indicated by dotted line) is generated (Figure 9.32). Soon after, the leased line experiences congestion of major severity (blue indicated by dashed line). If the color of the router symbol reflects the highest severity of any subsystem, the operator will learn of the modem failure (unimportant as it is not being used) and the congestion on the leased line (quite important) will be hidden. It is clear that the visual alarm indication should reflect the high priority alarm rather than the high severity one, shown by the black line.

Since these priorities are very much dependent on the network, the design of the FM should allow these to be configured by the administrator. Typically, the following configurability is supported:

1. For each <NE, subsystem, event>, set the severity.
2. For each <NE, subsystem, event>, set the priority.
3. For each <NE, subsystem, event, severity>, set the alarm indication(s).
4. For each <NE, subsystem, event, priority>, set the alarm indication(s).

Of course, for ease of use, a good NMS will have pre-configured defaults that satisfy most cases. This configuration of alarms is usually done through a GUI.

Alarm Finite State Machine. As we have already seen, a fault has a lifetime during which it goes through several stages. These include initial occurrence, the operator noticing the fault, corrective action being taken, and clearing of the fault. The fault is represented in the NMS by an alarm object. This alarm object must mirror the physical reality. The most natural design to represent this behavior is a *finite state machine* (FSM).

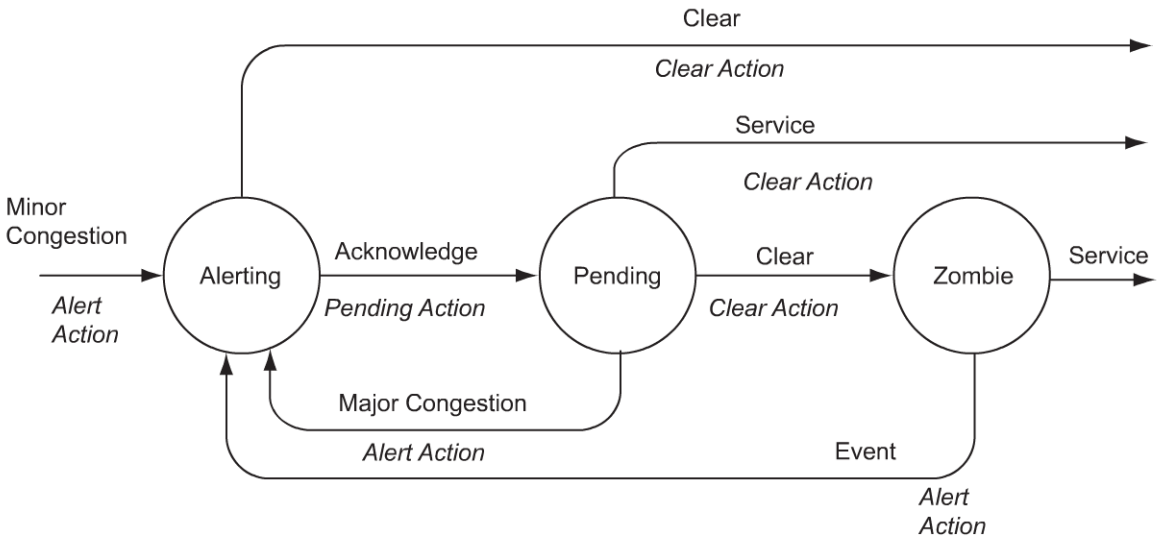

Figure 9.33 Alarm Finite State Machine

An FSM has *states, events,* and *actions*. The object remains in a state for some period of time. During this time it may perform some action continuously. The FSM makes a transition to another state only when an *event* occurs. The transition is (almost) instantaneous and is usually accompanied by some *action*.

Let us consider a typical FSM for a link congestion fault (Figure 9.33). The states are shown by labeled circles. Events are the labels above the transition arcs. Actions are shown in italics below the arcs.

When a minor congestion event is detected by the FM, it creates a new alarm object and puts it in the alerting state. The alert action taken on the transition is to change the color of the icon on the screen (say, to orange), start it flashing, and perhaps start playing an audio message. The flashing and audio playback continue as long as the alarm is in the alerting state to catch the operator's attention.

The next event is the operator *acknowledging* the new alarm. The action taken is to stop the flashing and audio playback. The alarm then moves to the pending state. The operator now starts to take some corrective action. There are several possible events that could occur next.

1. Service: The operator completes the corrective action and indicates that the fault has been serviced. The FM restores the icon color to normal (green) and resolves the alarm object from its action list. Note that the alarm object is usually kept in a history table in the DB for future auditing and analysis.
2. Clear: The fault may clear on its own; say a temporary burst of noise on a link due to operating of welding machines in the vicinity. In this case the icon is also restored to normal. However, the alarm object is kept in memory in a zombie state. This is to give the operator a chance to return to the console and indicate the corrective action taken if any. That is, the operator should service the zombie alarm.
3. Major congestion: The fault may worsen before it is corrected, which typically happens with congestion. The initial event that caused creation of the alarm object may have been a minor congestion, say link utilization exceeding 90%. Before congestion control measures take effect, utilization reaches 95% and a major congestion event is generated. This causes the alarms to go each to the alerting state with the color changing from orange to red.

Table 9.13 Transition Table for Alarm FSM in Figure 9.33

	INITIAL	ALERTING	PENDING	ZOMBIE
Minor congestion	*Alert action* Alerting state	*Null*	*Null*	*Alert action* Alerting state
Major congestion	*Alert action* Alerting state	*If alarm severity is Minor then: Alerting action* Alerting state	*If alarm severity is Minor then: Alerting action* Alerting state	*Alert action* Alerting state
Acknowledge	NA	Pending action Pending state	NA	NA
Service	NA	NA	Clear action Closed state	Clear action Closed state
Clear	Null	*Clear action* Closed state	*Clear action* Zombie state	*Null*

Null: ignore the event
NA: not applicable—this <state, event> pair should never occur. Usually indicates a software bug and should be logged and reported to the software developers.

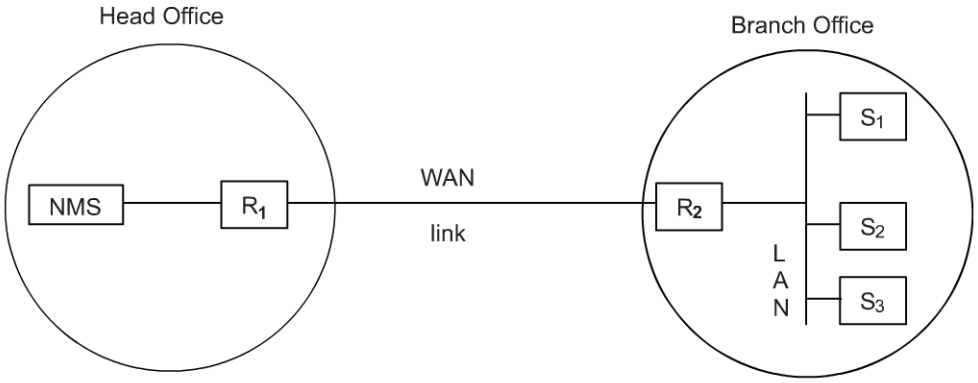

Figure 9.34 Part of an Enterprise Network

We have seen earlier in this section some simple methods to filter and correlate repeated or duplicate events. After the remaining events are registered as alarms, the alarm correlator searches for multiple alarms that arise from the same physical fault. It then suppresses all but one of these to avoid overloading and confusing the operator.

Several correlation techniques are commonly used. All have intelligence or reasoning behind them. The reasoning methods distinguish one technique from another. We will discuss these approaches in Chapter 11.

Similar to the case with event correlation, related alarms may be registered at different times. The range in delays depends on the polling interval and could be several seconds to 10s of minutes. Any alarm correlator could adapt to this delay in two ways: (1) It may favor immediate indication, in which case the operator may initially see the less important alarm, which is replaced shortly thereafter by the more important one. (2) It may opt for consistency in UI and delay indication of any alarm until the end of the correlation window. This could affect fault rectification time and should be limited.

The reasoning-based approaches described in Section 11.4 require a significant amount of operator intervention to ensure that the associated library has sufficient, but not too much, knowledge. Here we

describe an efficient *fully automatic* method for detecting the one alarm out of a group of alarms that represents the real fault, i.e., the **root cause** alarm. This method is based only on the topology and state information in the NMS database and uses a novel graph algorithm [Bhattacharya *et al*., 2005]. It is implemented in the CygNet NMS, which is described in Section 9.5.4.

The goal of RCA is to show only one primary alarm for one fault and to suppress secondary alarms. This can be accomplished by modeling the network as a graph in which each NE is a node and considering the path taken from the NMS to reach each node. If node A can be reached only via node B, A is said to be dependant on node B. If there are alarms from both A and B, the alarm from B is the root cause and the alarm from A is suppressed. In the branch office example in Figure 9.34, nodes R_2, S_1, S_2 , and S_3 are dependant on node R_1. Thus, the RCA algorithm presents only the alarm in R_1 to the operator. As the other nodes are not reachable, their status is shown as unknown (grey color) rather then up (green) or down (red).

A dependency relation can be defined on the network elements based on various criteria. One criterion we consider here is *reachability* from the NMS. The status of network element A depends on the status of network element B if A can only be reached via B. The algorithm has two parts. The first part forms a reachability graph from the physical topology. It also takes the full list of status events from the NMS as its input. From this list it determines the status of individual elements in the network. After the execution of the first part, the output is the real status of each element in the network. The real status can be *Up*, *Down*, or *Unknown*. An element with real status *Down* is the root cause for itself (and an alarm will be generated for this element). Nodes that are reachable only via a *Down* node have their status set to *Unknown*.

The FM can also optionally generate alarms for *Unknown* elements. It can indicate for each *Down* node the set of *Unknown* nodes that depend on it. This may be useful for the operator to decide on the order in which to attend to each of several *Down* nodes.

The second part of the algorithm takes each *Unknown* node U and determines the *Down* node D that is on the path from the NMS to node U. This *Down* node D is the root cause for U. The algorithm determines the set of *Unknown* nodes {U} that are dependant on each *Down* node D. The number of nodes in this set gives the operator an indication of how serious the failure of node D is.

The RCA algorithm is efficient. In the worst case, it takes O(*e*) time where *e* is the number of edges (links) in the network graph.

In a large network with 1,000s of links or more, it is not feasible to invoke the RCA algorithm for every alarm that is registered. A convenient strategy that can be adopted is to maintain a count of the number of alarms registered since the RCA algorithm was last invoked. When this count exceeds some threshold, the RCA algorithm is run. Since different alarms have different importance, it is preferable to use a weighted count. The weight can depend on the NE and component that generated the alarm, and the severity of the alarm. In addition, there is a maximum delay after which the RCA algorithm is run independent of the number of registered alarms (provided there is at least one pending alarm).

For example, assume that weights are assigned as shown in Table 9.14 and the threshold is 16. If 2 critical alarms occur, the RCA algorithm is triggered. However, it takes 8 consecutive minor alarms or 16 warning alarms to trigger the RCA algorithm.

9.4.7 Distributed Management Approaches

A small network can be managed conveniently by a single NMS. All desired functions are performed by this NMS. In a large network, a single NMS may not be able to handle the load of managing the

Table 9.14 Sample Weights based on Alarm Security for RCA Invocation

Threshold = 16

SEVERITY	WEIGHT
Critical	8
Major	4
Minor	2
Warning	1

entire network. Especially in a geographically distributed network, the WAN links may have relatively low bandwidth and could become bottlenecks. Hence, it is desirable to have several managers. These managers may all perform similar functions, or they may be functionally specialized. For example, one manager could perform only fault management, while another manager is used only for performance monitoring.

A distributed management application consists of several manager applications running on different management stations. Each manager performs its management functions either by directly interacting with agents or indirectly via other lower-level managers. Distributed management can be classified into categories ranging from centralized management at one end of the spectrum, through weakly distributed, strongly distributed, to cooperative management at the other end [Meyer, 1995].

In the centralized and weakly distributed paradigms, there is a well-defined hierarchy of management stations, and a manager can delegate tasks only to those strictly below it in the hierarchy. In the strongly distributed and cooperative paradigms, horizontal delegation is also allowed.

Another issue is the selection of a suitable management technology to implement the paradigm and whether delegation of management tasks is static or dynamic. For weakly distributed systems, it is customary to have static code running at various lower-level managers. Each manager knows the system being managed and is statically configured to manage it. A lower-level manager can execute a predefined task when requested by a higher-level manager.

The dynamic delegation of management functions to intermediate-level managers allows more flexibility [Meyer, 1995]. Management operations can be instantiated by higher-level managers by downloading or transferring code to remote management stations on the fly.

Multitier Architecture. In choosing a multitier architecture, there is a trade-off between flexibility and ease of implementation and deployment [Vanchynathan *et al*., 2004]. A strongly distributed or cooperative architecture permits almost unlimited flexibility, while a weakly distributed (hierarchical) architecture is much easier to implement correctly and efficiently. In particular, with the restricted communication and links between managers, it is straightforward to ensure consistency of data that may be replicated in several managers. This is much harder to achieve with strongly distributed designs. Further, most network operators and enterprises have well-defined hierarchical structures for their networks and their personnel.

Topology. We partition the set of elements to be monitored into groups of manageable size based on some criterion, say geographical. A manager process monitors each group and these processes constitute the lowest layer of the management architecture (Figure 9.35).

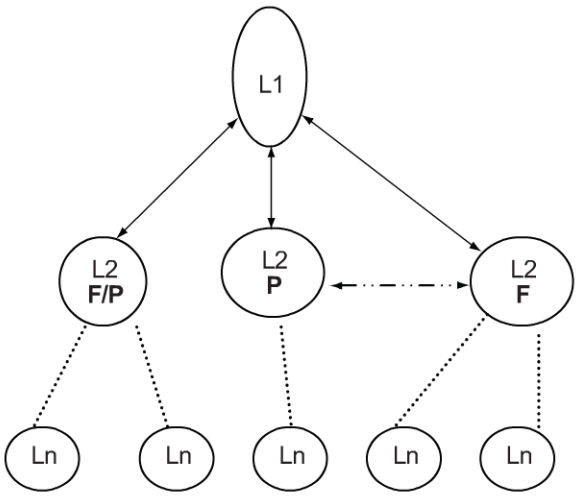

Figure 9.35 Hierarchical Multitier Management

One or more managers that run management processes at higher levels in the hierarchy are a parent of each of these management stations. This concept can be extended to as many layers as required.

To ensure predictable behavior, each cluster of network elements can have at most one parent, i.e., exactly one manager. If we represent each management station by a vertex and draw a directed edge (i, j) between two vertices i and j, if i is managed by j, the topology of the resulting network corresponds to a directed acyclic graph (DAG) [Horowitz, 1995].

9.4.8 Server Platforms

The NMS server requires two software platforms, namely the OS and the DBMS. The choice of these is important as they affect the development effort, performance, security, reliability, and cost of the NMS server. Once the OS and DBMS are chosen, the server design and implementation tend to become specific to these. Changing these at a later date is costly and time-consuming. The platform must be stable, of modest cost, and with a guarantee that it will be supported for the foreseeable future.

A detailed comparison of the available platforms is beyond the scope of this book. We would like to mention the trend of open-source platforms that can compete with commercial platforms in all respects. While in the past the designer of a large and complex software application such as an NMS server was constrained to run it on one of the commercial platforms, today, increasingly we find designers opting for open-source platforms.

Operating System (OS). The OS must support threads, processes, and shared-memory multiprocessors. It must support very large RAM and disk and a variety of peripherals. It must have a good programming interface for access to and control of its functions. It must efficiently support a variety of programming and scripting language chosen by the designers. It must have available a range of third-party software tools, components, and libraries.

Several popular server operating systems today are Linux, various flavors of UNIX (Sun Solaris, IBM AIX, HPUX, MacOS X, FreeBSD) and Microsoft Windows. All of these meet the requirements stated above though to varying degrees. Linux, FreeBSD, and NetBSD are popular open-source server OSs. The others are commercial (Sun Solaris has an open-source version available).

Database Management System (DBMS). The DBMS must efficiently support very large amounts of data. This includes hundreds of tables and 100s of millions of records. The DBMS must support transactions. It must have good administrative tools for creating indexes, recovering from table corruption, taking backups, and especially performance tuning.

As with the OS, the designer has a choice of several high-quality DBMS platforms, both commercial and open source. The viable open-source platforms are MySQL and PostgreSQL. Commercial platforms include Oracle, IBM DB2, Microsoft SQL Server, and Sybase.

9.4.9 NMS Client Design

We now briefly touch on the design of the client application that enables users to access the functions of the NMS. Recall the requirements for diverse users, ease of use and local/remote management capabilities described in Section 9.4.1. Since different users may have to login to the NMS from widely dispersed locations, they need to have the client application software running on their local terminal. Nowadays, the terminal is usually a PC, which is capable of doing substantial processing involved in the UI. This includes providing a window manager, rendering of graphics, generation of audio, etc. Broadly speaking, there are three approaches to the client application: a dumb terminal, a rich client, and a browser-based client. These are elaborated below.

Terminal Client. A "dumb" character-oriented terminal is used to login directly to the NMS. Terminal emulation software such as *xterm* and *putty* may be run on a PC or server and connected to the NMS server via telnet or ssh over a TCP/IP connection. All software functions are performed fully on the server, including editing of commands typed by the user and rendering (color, boldface) of the text on the screen. Terminal clients are useful when the user is at a remote location with access to only low-end terminals, or they are sometimes preferred by veteran administrators and operators for reasons of familiarity. Almost any terminal, PC, or server is capable of being used as a terminal client as is.

Rich Client. In this case, the client PC runs a special client application that is designed to work with the NMS server. The client application is usually GUI based. Besides generic GUI functions such as windows, icons, menus, and a pointer, it may include significant NMS functionality. For example, given a table with the details of the NEs, the client application may be responsible for generating a geographical map and overlaying on it icons for each NE and network link. Likewise, the client may allow the user to sort lists and tables on various criteria without the intervention of the server.

A rich client is usually written in an object-oriented language with good GUI support. Perhaps the most popular language for rich NMS clients is Java, with C++ and C# also being used.

While the rich client can give a very powerful UI and nearly instantaneous response to many user commands, it suffers from two drawbacks, the first being portability. The client application may run on only one OS and sometimes only one version of that OS. Thus, the user has to ensure that his/her PC has the right OS. If the client application is to run on a variety of OS platforms, the NMS vendor has to invest a substantial amount in software development and testing on each of these platforms. This has to be repeated with every new release of the client application.

A second and more serious problem is compatibility with the NMS server. As new versions of the server and client are released, perhaps independently of each other, the user has to ensure that the right version of the client application is installed on the client PC. In some cases, the wrong version simply does not work, which is an annoyance. In other cases, the wrong version appears to work but due to subtle incompatibilities, it performs the wrong functions. For instance, when the server indicates to the client that NEs have failed, the

client may display them in green (normal) instead of red due to an incompatibility. This is much more serious as the operator will ignore the fault.

The situation is aggravated when an operator needs to login to several NMS servers in different regional networks. The servers, from the same vendor, may be of different versions. The operator's PC will need to have several versions of the client application installed and the operator would have to run the corresponding client application depending on which NMS server s/he is working with!

Browser or Web Client. The browser or Web client promises to combine the advantages of the terminal and rich clients without their disadvantages. It is rapidly becoming *de facto* for NMS clients (and for many other applications also). The browser client provides all its functionality through the GUI of a Web browser such as Firefox, Internet Explorer, or Safari. (Firefox is open source and runs on almost any OS platform, Internet Explorer runs only on Microsoft Windows, and Safari is supplied with MacOS X. The NMS server includes a Web server to which the user connects via the browser. The NMS server throws up Hyper Text Markup Language (HTML) pages to the browser. For a better user experience, the HTML pages may include some client-side processing through Javascript. Network maps and icons are usually provided using Ajax.

Since there is no software installed on the client PC, there are no issues of version incompatibility between the client and the server. Any client-side NMS processing is done through Javascript code that is downloaded in the HTML page. It is not stored on the disk of the client PC (though it may be cached temporarily to enhance performance).

Today, browsers are ubiquitous. Almost any PC has a good, recent browser installed. Many mobile phones, including some low-end models, can run a browser, though the small screen limits the amount of information that can be presented. Almost everyone who uses a PC and the Internet is familiar with the use of a browser. Hence, the training effort for new users is greatly reduced.

Owing to differences in the way in which different browsers render an HTML page, the problem of portability to different browsers is still an issue, albeit a much smaller one than that of portability of rich client applications. For instance, a data entry form in which the labels and boxes are aesthetically placed in one browser may have some of these GUI components overlapping in another browser. So, the server needs to determine which browser the user is using and feed HTML pages that are tuned to the peculiarities of that browser. Likewise, Javascript code that works on one browser may not work on another browser.

Fortunately, the incompatibilities between browsers are decreasing and are well-documented. Also, there are a variety of open source and proprietary code libraries available that permit the developer to write code that works equally well across a variety of browsers.

9.4.10 Summary: NMS Design

Starting from the requirements for management of a telecom network or a large enterprise network, we derived the architecture for an NMS server. After some general design decisions, we discussed the design of the main modules in an NMS server, the discovery module, PM, and the FM.

The FM is the most complex of the modules in an NMS. This is because it has to deal in real time with a wide range of events that occur at unpredictable times. Rapid fault detection and rectification is the key to the operation of a network. The FM is especially relied upon when there are serious faults in the network. At such a time, the FM would experience a burst in processing requirements. The design of the FM needs to be especially careful to ensure that the NMS itself does not fail due to overload during such periods of severe network problems.

We traced the path of an event through the FM. We explained different methods used to indicate the fault to the operator and the various alarm states. We explored assorted techniques for the correlation of

seemingly unrelated events and alarms. These range from the simple matching of fields in event records, to sophisticated AI and graph algorithms.

Finally, we discussed approaches to distributed management. This enables scaling to very large networks. It also mirrors the structure of typical large organizations.

9.5 NETWORK MANAGEMENT SYSTEMS

So far we discussed the use of simple system utilities and tools for management. This was followed by a detailed examination of the design of a high-end NMS server. In the rest of this chapter we will describe several commercial and open-source NMSs. We start with the management of networks, and then cover management of systems and applications. This is followed by enterprise management and telecommunications network management. Finally, we describe approaches for distributed management.

9.5.1 Network Management

A network consists of routers, switches, and hubs connected by network links. Servers, workstations, and PCs are connected to LANs in the network. Various access technologies may be used. In network management, we are primarily interested in the health and performance of the routers, switches, and links. We may also monitor the health of servers.

The first task involved in network management is the **configuration** of the above network elements, their agents, and the NMS itself. This includes discovery of network elements and the topology of the network.

Daily users of NMSs are people in the NOC who do not have the same engineering background as those who designed and implemented the systems. Therefore, ease of use is an important factor in the selection of NMSs. For example, an operator does not constantly sit in front of a monitor and watch for failures and alarms. Thus, when an alarm goes off, it should attract the attention of the operator visibly, audibly, or both. It should present a global picture of the network and give the operator the ability to "drill down" to the lowest level of component failure by successive point-and-click operations of the icon indicating an alarm.

Figure 9.36, Figure 9.37, and Figure 9.38 show hierarchical views of a network testbed at the NMSLab, IIT Madras, which were captured with a CygNet NextGen NMS. (The IP addresses of the nodes have been changed for security reasons.) Figure 9.36 shows the global view with network segments and numerous domains behind routers and gateways. Figure 9.37 is obtained by clicking the mouse (also colloquially referred to as drilling down) on the network segment icon 192.168.9.0 in Figure 9.36 and shows the nodes that are part of that private LAN. The LAN has a switch (SW-11) connected to two routers RTR-1, and RTR-2), and a few other IP hosts. The port details on the switch SW-11 are obtained by drilling down on the SW-11 icon in Figure 9.37. The result shown in Figure 9.38 contains 12 ports, some of which are free.

Next, the **fault management** capability of the NMS must support monitoring of the health of the NEs and links. In an IP network, this is usually done using ICMP ping and SNMP get messages. The NMS reports faults to the operator in a variety of ways depending on the nature, severity, and importance of the fault. It may assist in the rectification of the fault.

The NMS must support **performance management** especially of the expensive WAN links. Planning and management reports keep upper-level management apprised of the status of the network and system operations. Reports in this category include network availability, systems availability, problem reports, service response to problem reports, and customer satisfaction.

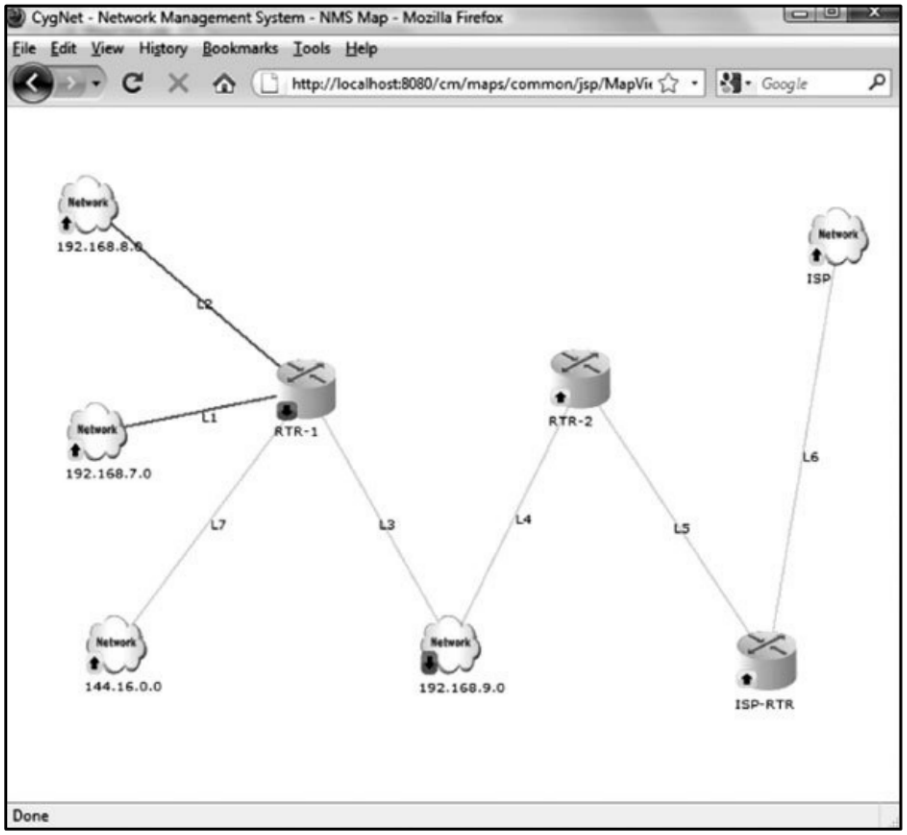

Figure 9.36 Global View of Network

Performance management provides traffic trend reports to enable the network administrator to identify bottlenecks. The administrator can take action to alleviate the bottleneck, such as re-routing traffic via an alternate route, or changing priorities for different classes of traffic. Performance reports help the administrator see long-term traffic trends in order to plan capacity expansion in a timely and cost-effective manner. Trends in traffic should address traffic patterns and volume of traffic in internal networks, as well as external traffic.

Accounting management is probably the least developed function of network management applications. Accounting management could include individual host use, administrative segments, and external traffic.

Accounting of individual hosts is useful to identify some of the hidden costs. For example, the library function in universities and large corporations consumes significant resources and may need to be accounted for functionally. This can be done by using the RMON statistics on hosts.

The cost of operations for the Information Management Services department is based on the service that it provides to the rest of the organization. For planning and budget purposes, this may need to be broken into administrative group costs. The network needs to be configured so that all traffic generated by a department can be gathered from monitoring segments dedicated to that department.

External traffic for an institution is handled by service providers. The tariff is negotiated with the service provider based on the volume of traffic and traffic patterns, such as peak and average traffic. An internal validation of the service provider's billing is a good practice.

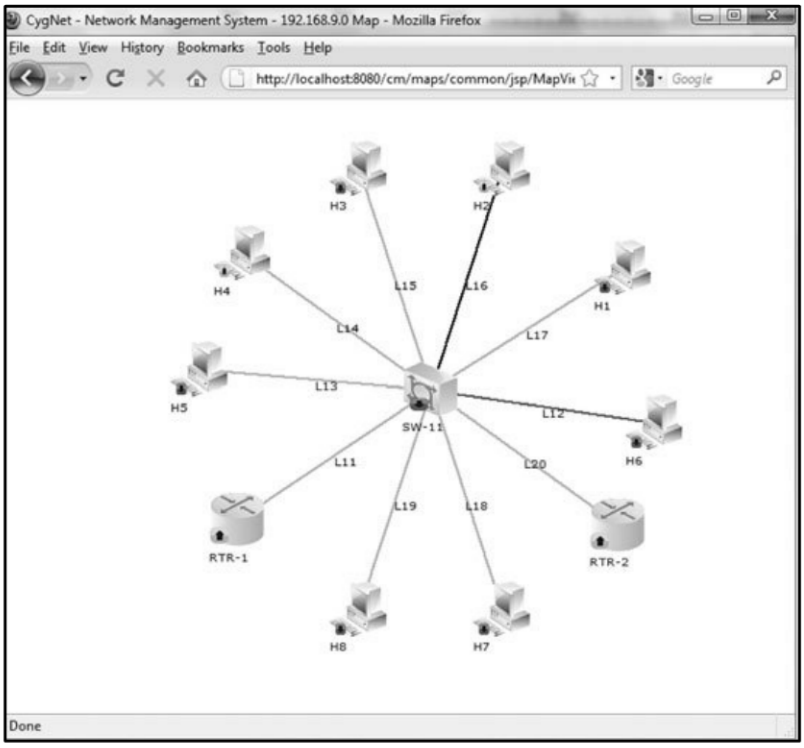

Figure 9.37 Domain View

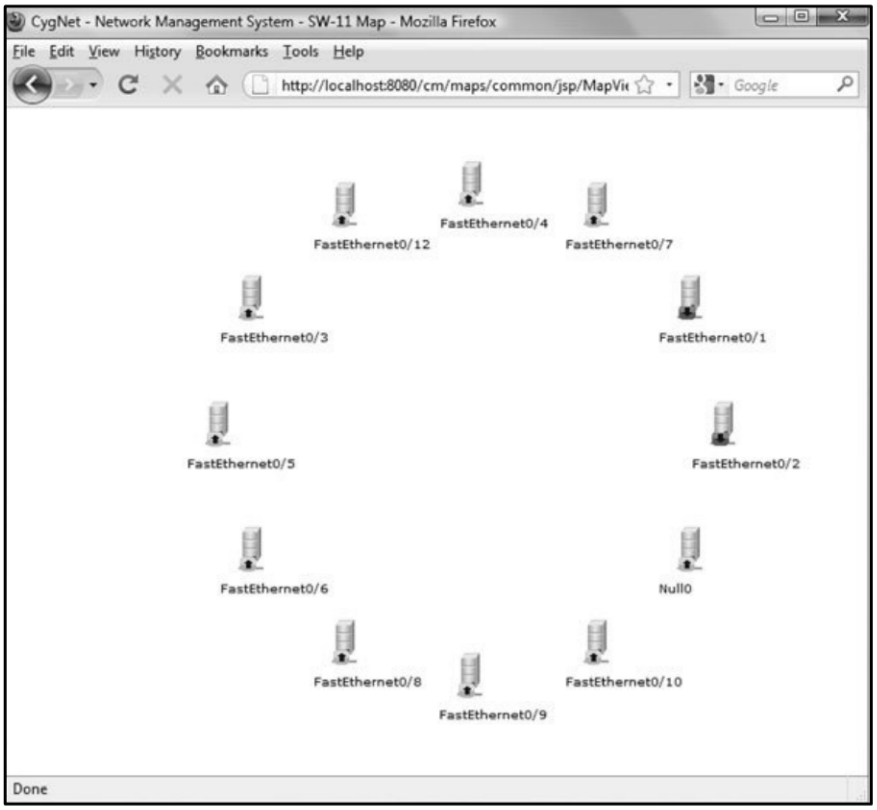

Figure 9.38 Interfaces View

Security management is both a technical and an administrative issue in information management. It involves securing access to the network and the information flowing in the network, access to data stored in the network, and manipulating the data that are stored and flowing across the network. The scope of network access not only covers enterprise intranet network, but also the Internet that it is connected to.

Security management also covers security of the NMS itself. The NMS database contains a wealth of often confidential information about the organization. This must be made available to authorized personnel, but kept away from all others. Likewise, a user of the NMS could reconfigure NEs throughout the network. Hence, login to the NMS must be carefully controlled.

Thus, network management involves the complete **FCAPS** spectrum of application functions defined in Chapter 3. Configuration, fault, and performance management are found in almost every network management deployment. In some cases, the NMS is also used for security and accounting management.

OpenNMS. This is an open-source NMS (http://www.opennms.org), which claims to be the first project aiming to build a complete enterprise-class open-source NMS. OpenNMS is written largely in Java and has a browser-based UI. It is primarily used for managing SNMP devices. It is used to manage small networks of under 25 NEs to large networks with over 80,000 NEs.

The major functional areas covered by OpenNMS are autodiscovery, status polling of NEs, performance polling, and fault management. Reports are provided using the JFreeChart package. Much of the operation of OpenNMS can be customized by the user by means of filters. A filter consists of rules, each of which is essentially a simplified form of an SQL statement. It configures how the component of the NMS behaves. For example, the notification rules control whether or not a received event triggers a notification to the user. Filters can be added to control autodiscovery, to specify the list of IP interfaces that are included in data collection, polling, etc.

Unlike many commercial NMS products, OpenNMS does not have a graphical map for the display of the NEs and their status. The designers of OpenNMS believe that seasoned network administrators prefer to see event lists. OpenNMS provides event lists grouped in various convenient ways. On the main WebUI page there is a "real-time console" (RTC) that reflects the status of categories of devices. These categories reflect groups of devices like database servers, Web servers, etc. However, anything in the database can be used to create a custom category list, and grouping devices by location, building, vendor, IP range, etc., is very common. The categories list follows a basic tenet of OpenNMS; once configured, it should be simple to use and as automated as possible. As new devices are added, the categories automatically update. There is no need for manual customization, such as would be required with a useful map. (There is a group of developers working on a map, so this may become available in due course.)

By default, the FM receives SNMP traps. Other event detectors can be configured through XML configuration files. When an event is accepted, it results in a notification to the user, which can be on-screen, via email, SMS, etc.

In keeping with the philosophy of being utilitarian, OpenNMS has a variety of reports. Most are simple tables with some graphs. They lack the frills that may be attractive to a novice, but contain a wealth of information in a format that is easy for a network administrator to comprehend.

With OpenNMS, the user has the ability to do comprehensive monitoring of a network at the price of just the server hardware. The NMS is customizable by the network administrator without any programming. As with any open-source product, the enterprise has the comfort that it could always get unusual customizations done as the source code is freely available. Currently, OpenNMS supports F, P, and part of C in FCAPS. With

the on-going efforts of the developer community, it is likely that other functional areas will be covered in future. Commercial support is also available from the OpenNMS Group for enterprises that need it.

9.5.2 System and Application Management

Network management addresses only the managing of a network (i.e., managing the transport of information). System management deals with managing system resources, which complements network management. For example, *ping* is used to test whether a host is alive. However, we want to know more about the use of system resources on the host, such as the amount of CPU use on the host or whether a specific application is running on the host.

Historically, enterprises had two separate and distinct organizations. The telecommunications department took care of communications, giving rise to network management. The management of information systems (MIS) department took care of the computers, a task that involved system management. However, in the current distributed environment of client/server architecture, the computing depends heavily on the communication network and the distinction has disappeared. System and network management now form a single umbrella, headed by a chief information officer. System and network management are beginning to be considered together as a solution for information management issues and problems. System management tools, which used to be custom developed, are currently available as commercial systems and are being integrated with network management.

System management tools monitor the performance of computer systems. Some parameters that can be monitored are (1) CPU use, which measures the number of processes that are running and the resource consumption of each; (2) status of critical background processes (called *daemons* in UNIX terminology); and (3) application servers such as the Simple Mail Transfer Protocol (SMTP), File Transport Protocol (FTP), and Domain Name Server (DNS). System management may also include backup of server databases, desktop workstations, and operations support systems that support operations such as help desk and trouble ticket tracking.

Several UNIX-based tools can be used to monitor systems. Such tools are constantly evolving and are updated on Web sites that are used to track them. http://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html is a site active from 1996 that provides a comprehensive list of commercial and open-source network management tools.

High-End System Management. The Computer Associates (CA) Unicenter TNG and Tivoli Enterprise Manager TME 10 are two integrated systems solutions available commercially [ZDNet]. Both solutions offer features that can be classified as high end and thus require the vendor's ongoing active participation. They meet the requirements of large enterprises, particularly as they are offered as integrated solutions, which we will discuss in Section 9.5.3.

Low-End System Management. System management of hosts can be accomplished by installing simple and free public domain software and configuring it to local system management.

Nagios. This is a fairly comprehensive open-source NMS (http://www.nagios.org). The project has been active for over 10 years so it is stable. The design is scalable and extensible.

Nagios provides the basic network management features. These include status and performance polling, fault handling, and configuration management. The FM can indicate faults via a graphical map,

color-coded event lists, and email, pager, and cellphone. It allows the administrator to schedule the downtime of hosts, servers, and the network. The reporting feature supports capacity planning.

Unlike other NMSs, Nagios does not have built-in mechanisms for polling. Instead, it relies on external plugins. These can be executables or scripts and hence give substantial flexibility. Nagios by default has support for managing routers, switches, and other IP network components; Windows, Linux, UNIX, and Netware servers; network printers; publicly available services such as HTTP, FTP, SSH, etc. With its extensible design, Nagios has more than 200 community-developed plugins available.

It has a rich UI, which includes a map (unlike OpenNMS described in Section 9.5.1). Sample screenshots are available at http://www.nagios.org/about/screenshots.php. The UI is customizable to each user. This facilitates specialization and also helps maintain security of sensitive information.

As with OpenNMS, Nagios is a good choice for a small organization that cannot afford a big-budget commercial NMS. It is also sufficiently comprehensive and scalable that even large organizations use it. The Nagios Web site claims that it can handle networks of over 100,000 NEs. Given the high overhead of the external polling mechanism, the server hardware would be very expensive if all the 100,000 NEs are polled.

Big Brother. A well-known low-end system management product is Big Brother [MacGuire S]. Although it has very serious limitations in terms of both platforms and functionality, it may be adequate for small and medium-sized company networks.

Big Brother is an example of software that can be implemented with relative ease to manage system resources and is Web based. A central server on a management workstation runs on a UNIX platform and clients on managed objects. Big Brother is written in C and UNIX shell scripts and hence can be run on multiple platforms. The central management station presents the status of all systems and applications being monitored in a matrix of colored cells, each color designating a particular status. It supports pager and email functions that report the occurrence of alarms. Software can be downloaded and modified to meet local requirements for the operations, services, and applications to be monitored.

Big Brother performs the dual function of polling clients and listening to periodic status reports from clients that are UNIX based. Polling checks the network connectivity to any system. Client software periodically wakes up, monitors the system, transmits information to the central server, and then goes back to sleep. Textual details can be obtained on the exact nature of a problem. The systems are grouped for ease of administration.

9.5.3 Enterprise Management

We will next describe the two commercially available integrated solutions to system and network management. The two solutions are offered by two vendors, Computer Associates and Tivoli, the latter has been acquired by IBM. Both partners with several NMS vendors provide integrated solutions.

Computer Associates Unicenter TNG. The CA Unicenter TNG framework [CA] provides infrastructure to support integrated distributed enterprise management. It is based on a client/server architecture having an agent in each host and a centralized workstation. CA provides Unicenter TNG agents that can run on a large number of platforms; and the list continues to increase as the customer base diversifies. Besides TCP/IP and SNMP, the TNG framework supports numerous other network protocols, accommodating a varied enterprise environment.

CA describes Unicenter TNG as a framework comprising three components: Real World Interface, object repository, and distributed services. The Real World Interface presents a visual depiction of managed objects from different user perspectives. Both two-dimensional and three-dimensional presentations are available. The object repository is a management information database that includes multiplicity of data, such as managed objects, metadata class definitions, business process views, policies, topology, and status information. The distributed services link elements at the communications level.

Because of the TNG agents running in the hosts, during autodiscovery the system discovers not just host identifications, but also details of the processor, disk, and other components of the system. The discovered components are presented in a Real World Interface that presents a unified GUI at the higher levels. An object repository stores all the autodiscovered information and any other management information needed to support the Real World Interface. System and network management views are extended to business process views, whereby objects related to an administrative group or functional group can be presented. This approach makes operations easier for human personnel because they do not have to know the technical details behind the operations they are performing.

Event management in the TNG framework includes a rule-based paradigm that correlates events across platforms and presents the resultant alarms at the central console. These events include standard SNMP traps. Standard drilling through layers to detect the lowest level component failure is built in as desktop support.

An additional feature of the TNG framework is calendar management, which provides a shared calendar so that all personnel can view each other's activities. The calendar handles both one-time and periodic activities. Operations such as triggering an alarm pager or email can be programmed according to weekly and weekend shift schedules.

Some of the other notable features of the TNG framework are backup and disaster recovery, customized and canned report generation, and virus detection—all of which can be programmed as part of calendar management activities. In addition to these standard features, numerous optional modules, such as advanced help desk management, Web server management, and software delivery are available.

Tivoli Enterprise Manager. Tivoli's management framework, originally named the Tivoli TME 10 framework, provides system and network management and is in the same class as the Unicenter TNG framework. Tivoli has changed the TME 10 from a two-tier, client/server architecture to a three-tier architecture and has renamed it Tivoli Enterprise Manager. Tivoli claims that the new architecture has increased the capability of handling from 300 to 10,000 managed nodes. The extended three-tier architecture also has a gateway as a middle layer that is designed to handle as many as 2,000 agents. The agent module has been redesigned to consume fewer resources.

Tivoli merged with IBM, allowing it to integrate the features of its systems with IBM's NetView NMS. NetView performs the network management function, and the complementary features of the Tivoli management applications perform the system management functions. The platform is object oriented and uses the standard CORBA-compliant object model for use with diverse and distributed platforms. This feature is somewhat similar to the Sun Enterprise Manager [Sun Enterprise], which also uses the CORBA-compliant OSI object model and standard CMIP protocol. In the management framework, Tivoli management agents reside in the managed host applications. Although the TME Enterprise system does not use SNMP as the management protocol, NetView handles SNMP traps.

The Tivoli Enterprise Manager monitors network, systems, and applications in a distributed architecture, as in most other systems. Event management has a built-in rule-based engine that correlates events and diagnoses problems. It further has an automated or operator-initiated response mechanism to

correct problems wherever possible, using a decision support system. Service desk technology is integrated with network and system management technology in the service-level management framework.

Tivoli service management comprises problem management, asset management, and change management. The problem management module tracks customer requests, complaints, and problems. The asset management module handles inventory management. The change management module incorporates and manages business change processes.

The Tivoli Enterprise Manager framework contains an applications manager module (global enterprise manager) that coordinates business applications residing in diverse multiple platforms. An API is provided to permit third-party vendors to integrate their application software into the Tivoli Enterprise Manager framework. The applications manager also measures the response and throughput performance of applications.

The security management module provides encryption and decryption capabilities (optional). A software distribution module automates software distribution and updates. Operations can be scheduled by the workload scheduler module.

9.5.4 Telecommunications Management Systems

Unlike an enterprise network, a telecom network provides commercial services to subscribers. The operator has legal responsibilities to the subscribers. Telecom operators are usually subject to stringent regulations.

Telecommunications network management includes monitoring and managing the network with certain distinctive features. Apart from supporting standard management features such as low level, as well as consolidated, fault and performance management, there are certain features that are specific to telecom:

1. A typical telecommunications network often includes equipment acquired over several years, and a telecommunications NMS must be capable of managing these heterogeneous systems from a single point.
2. Most telecom devices that implement the ITU-T standard protocols (such as CMIP/CMISE) differ in details and this makes development of the management application more complex.
3. Some devices have only an EMS, which often has a proprietary interface, and the NMS must communicate with the EMS rather than with the device agent directly. Thus, often the NMS manages some NEs via an EMS and some directly.
4. Typical telecom networks include multivendor multitechnology equipment, supporting diverse protocols. SDH and related transport technologies (SONET, DWDM, etc.) continue as the de facto technology for delivering reliable and scalable andwidth services over the last decade; and service providers have large amounts of deployed fiber that has been laid out over a period of time.
5. Provisioning of bandwidth is a common and important requirement in telecom networks. Provisioning bandwidth in optical networks in an optimal manner without causing fragmentation of the available bandwidth is a challenge. For example, if there is a requirement to provision an E1 circuit (2 Mbps), it would not be desirable to "break" an STM1 link (64 Mbps) to do the provisioning.
6. A related requirement is the maintenance of the latest inventory in the database so that when there is an incoming bandwidth-provisioning request, reliable and fresh information about the availability of bandwidth can be accessed. Usually, this information is maintained and updated manually. However, a telecom network is a dynamically changing entity, so automatic discovery

with periodic rediscovery to synchronize the inventory database with the actual network is becoming a critical requirement for efficient and optimal management of the network. While autodiscovery of IP networks using ping and/or SNMP is a standard feature in data NMS, the discovery of circuits in telecom networks is a far less straightforward issue.

7. A typical telecom provider assures customers of quality of service via service level agreements (SLAs). In a telecom environment this includes parameters such as call completion rate for voice calls, signal strength on wireless links, call completion for data calls, etc. The management system must provide the support for SLA management.
8. RCA to diagnose the actual source of a problem is another important feature to be supported.
9. Networks tend to be large, often with hundreds of thousands of NEs, and have a wide geographic spread. The telecom operator may have a hierarchical organization with different administrators responsible for different regions of the network. The NMS architecture must be scalable and should support the hierarchy of the operator.

In this section we describe CygNet, a commercial telecom NMS deployed at leading telecom service providers in India. CygNet is the flagship product of NMSWorks Software Pvt. Ltd., a technology company that is a part of the TeNeT (Telecommunications and Networks) group of the Indian Institute of Technology Madras, India.

CygNet NMS is a multivendor multitechnology management system that has been designed to meet the needs of telecommunication management. CygNet architecture accommodates most of the needs of a telecommunications management system.

Architecture of the CygNet Core. Figure 9.39 depicts the architecture of the core CygNet NMS. We describe the major components below.

Managed Element Modeling, Storage, and Depiction The elements server is responsible for this function. Network elements that are managed by the CygNet NMS are stored in the topology (topo) database. Elements to be managed by the CygNet NMS can either be added manually or by automatic discovery. A discovery configurator allows the operator to specify a range of IP addresses for discovering

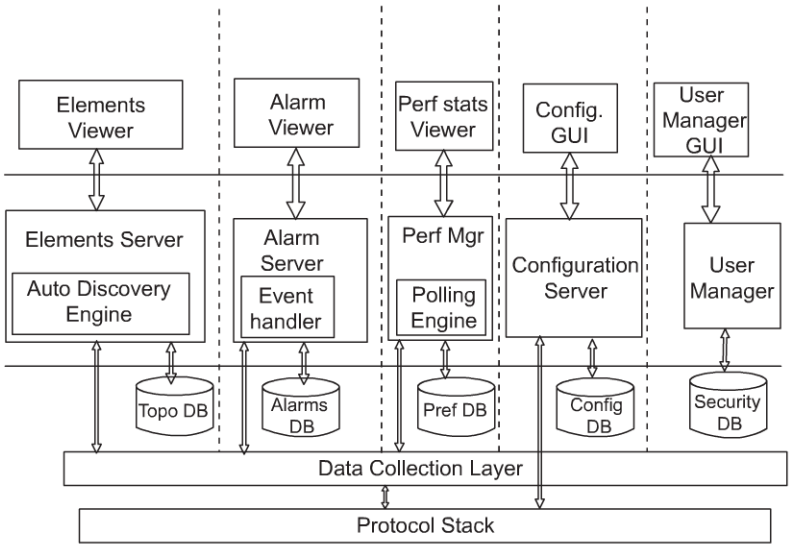

Figure 9.39 Architecture of the CygNet Core

elements, the protocol to be used for discovery, and other such details. A map-based GUI displays the managed elements and the topology of the network, color coded to reflect the status.

Fault Management System The CygNet alarms server is responsible for detecting a problem, correcting it, and restoring the system to normal operation at the earliest. **Fault detection** is done by polling, as well as receiving notifications. Alarms are generated in response to fault conditions in the network, stored in the database, and notifications (visual, audio, email, SMS) sent to capture the attention of concerned persons. The **fault reporting feature** ensures that information regarding a fault condition is disseminated in a meaningful manner to the concerned operators. GUI-based views of current, as well as historical faults, are available. The **fault escalation feature** allows the fault resolution time to be minimized and fault escalation using notifications (traps or TMF 814 notifications), email, or SMS. Multiple events associated with a common fault in a single network element are correlated and forwarded as one consolidated event or alarm by the RCA module described in Section 9.4.6. This minimizes the occurrence of event storms that would result in degraded levels of service.

Performance Management System The function of the performance management module is to manage the performance of individual network elements, links, and services. The PM module performs the following three major functions:

1. Data collection and storage in database.
2. Graphical views of real-time performance statistics.
3. Historical reports of collected data from database.

Configuration Management System The configuration management module in CygNet allows the operator to directly manage and configure network elements using the protocol supported by the network element. Configuration information regarding various network elements is stored in the configuration database.

User Manager This is the authentication and authorization component. CygNet identifies several classes of users with different privileges and access rights. All users have to be authenticated before they log on to CygNet. Role-based access control, idle time-out, and user audit trail are some of the features that this module supports.

Architecture of CygNet as a Telecom NMS. CygNet supports a layered architecture depicted in Figure 9.40 that enables it to be customized for various specific requirements. The custom components

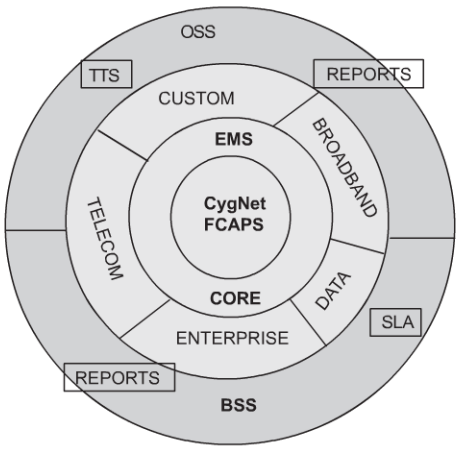

Figure 9.40 CygNet as a Telecom NMS

can be added as overlays to the core product. This also allows CygNet to scale to very large telecom networks.

Telecommunication-specific Subsystems. The CygNet design supports most of the features required for a telecom NMS specified at the beginning of this section. Here we describe some aspects of the design that enable efficient telecom management and also certain components (Mediation Server and Optical Transport Network Management System (OTNMS)) that are customized for this purpose.

CygNet has been designed so that it can be deployed either in a centralized manner or as a multitiered system. This provides a good scalability option when a very large and widely distributed network needs to be managed. Network management traffic can be reduced if lower-level management stations transfer management data to the central server periodically. This deployment strategy also avoids a single point of failure (Section 9.4.7, Figure 9.35).

At the lowest level, the CygNet local manager runs lightweight processes that predominantly collect data. The collected data are analyzed for performance and fault conditions and summarized reports are sent to the central management station. Figure 9.41 depicts this configuration. CygNet supports standard interfaces (TMF 814, SNMP, FTP, SCP (Secure Copy Protocol)) between the local NMS and central NMS. [Vanchynathan *et al*., 2004].

CygNet supports TMF 814, an NMS–EMS interface defined by the TM Forum, and increasingly adopted as the standard for management of optical transport networks. (TMF 814 is discussed in more detail in Chapter 16. This allows management of newer optical network devices, since many vendors include support for TMF 814 in their EMSs.

There are usually legacy NEs in a network and these either do not have an EMS at all, or even if there is an EMS, it does not support TMF 814. In order to integrate such equipment, the *CygNet Mediation Server* designed for multiple protocol support is used (Figure 9.41). The mediation server of CygNet is a middleware component that is between the NMS (or other OS) and the network. It serves to hide the heterogeneity in EMS interfaces from and provide a uniform TMF 814 interface towards the NMS. The lowermost adaptation layer allows plug-ins to convert proprietary protocols to the format of TMF 814. The middle pre-processing layer performs functions such as event correlation, suppression, filtering, caching, etc. It is a major enabler in bridging legacy and new networks and expedites integration of new equipment into CygNet.

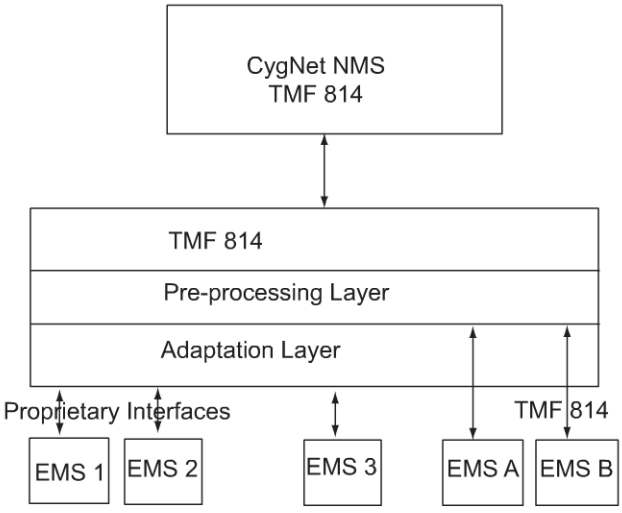

Figure 9.41 CygNet Mediation Server

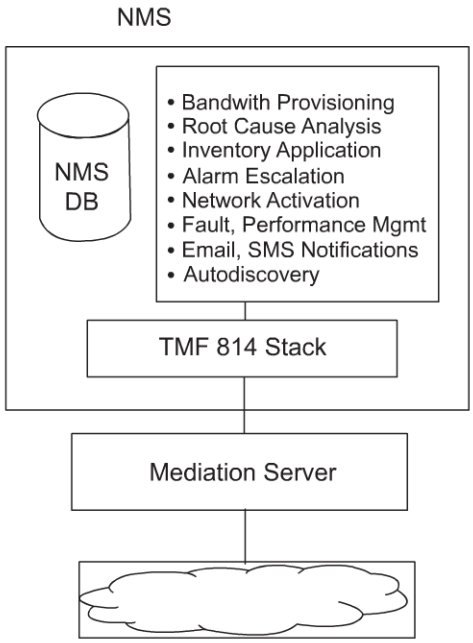

Figure 9.42 Architecture of the CygNet OTNMS

CygNet Optical Transport NMS (OTNMS). This CygNet component is customized for end-to-end management and bandwidth provisioning of optical transport networks. The functional architecture of OTNMS has been designed with a view to making the OTNMS an all-in-one solution for optical network management (Figure 9.42).

It supports automatic, dynamic provisioning of bandwidth services using a path computation algorithm designed to satisfy a maximum number of service requests while optimizing bandwidth use and avoiding fragmentation [Madanagopal, 2007].

The availability of reliable inventory is central to the success of automatic provisioning. The resource inventory database in CygNet stores network and computing equipment, logical resources, and topology. It keeps track of the physical configuration of the network, equipment inventory (cards, ports, etc.), physical connectivity, and logical connectivity of the different network layers. Data in the resource inventory are queried by the provisioning system to satisfy service requests and understand where capacity is available. However, in a typical telecom network, the inventory maintained by the NMS can become out of date for many reasons, including manual network intervention, equipment upgrade, network maintenance, card failures, unaccounted deactivations, etc. Once the inventory is out of date, any design based on this inventory system is unreliable. The CygNet OTNMS supports automatic discovery of NEs, as well as circuits of optical networks with periodic rediscovery. The autodiscovery feature ensures that the inventory can automatically synchronize itself with the network. This requires automatically uploading physical, logical, and service information and correlating it to the normalized internal object models of the inventory database.

The CygNet OTNMS has support for SLA-related performance data collection and storage, and display of SLA reports and threshold alarms to indicate SLA violations. It also uses an RCA module to pinpoint the actual cause of a failure.

Summary

In this chapter we listed a number of utilities that are available on commonly used operating systems such as Linux, UNIX, and Windows. These are invaluable tools in the repertoire of any network manager, and support a significant amount of troubleshooting and traffic monitoring. Some of these tools are based on SNMP, others use assorted protocols such as ICMP or proprietary messages over TCP. We discussed techniques used for monitoring of statistics and the use of MRTG for collecting router traffic statistics.

Focusing on SNMP-enabled devices, the vendor needs to design an MIB that supports remote management of the device. This topic of MIB Engineering was covered in Section 9.3. Several common MIB designs were presented.

Section 9.4 dealt with the design of an NMS server for a large telecom or enterprise network. Motivated by the requirements, we presented a generic architecture. We then covered the detailed design of the important architectural blocks, especially discovery, fault, and performance management. This section draws significantly on our experience in the design of a commercial NMS, viz., CygNet.

Finally, we presented several NMS products, both commercial and open source, which are commonly used for the management of different types of networks and services.

Exercises

1. Execute the commands *nslookup* and *dig* on a host IP address and analyze the results.
   (a) Compare the two results for the common information.
   (b) What kind of additional information do you get from *dig*?
2. Use *dig* to determine the IP address of www.tenet.res.in (or the DNS name that your instructor provides you with).
3. Use *dig* to list all the hosts associated with the domain *tenet.res.in* (or the DNS name that your instructor provides you with).
4. Using *dig*, determine the domain name that corresponds to the IP address 203.199.255.5 (or the one that your instructor provides you with).
5. Ping an international site 100 times and determine the delay distribution and packet loss. Repeat at different times of day and night. Explain any variations.
6. Using *tcpdump* or *wireshark* on an Ethernet interface on a host, capture ten IP packets. Examine their headers and contents.
7. In diagnosing poor network performance—for example, delays—you need to know where the bottleneck is. Use *traceroute* to an international site on another continent and isolate the delay in the path.
8. From a workstation in a segment of your institute's network, discover all other workstations in your segment using a network tool. Substantiate your results with the data gathered by some other means.
9. As a network manager, you are responsible for the operation of a network. You notice heavy traffic in a host that is on a TCP/IP network and want to find out the details.
   (a) What basic network monitoring tool(s) would you use?
   (b) What would you look for in your results?
10. Using an SNMP tool, determine which of the nodes given by your instructor has the longest and shortest up time. Substantiate your result with the gathered data.
11. The function *snmpWalk(subtreeRoot)* returns the values of all the visible OIDs in the subtree rooted in *subtreeRoot*. Using the standard SNMPv1 messages, devise an algorithm for *snmpWalk()*.
12. A switch has several ports each having a different speed. Write a complete SMI definition for the table of port speeds. Assume that the switch OID is {experimental 7}.

13. A node may have several hard disks. Write an SMI definition for the important parameters (brand, type, capacity) of such a set of disks.
14. It is desired to provide username/password protection to a MIB subtree {*enterprises Midas(3794) corDECT(1) config(2)*}. What are the MIB variables that need to be added for this purpose? Draw the MIB subtree and for each variable give its ASN.1 macro definition.
15. It is desired to provide access through SNMP to the database for an NMS course. Design a MIB rooted in {*enterprises NMSWorks(15760) course(10)*} to hold information including the start and end dates of the course, the number of students registered, and the list of reference textbooks. Draw a pictorial representation of the subtree. Write the ASN.1 macro definitions of each node.
16. It is desired to store the roll numbers and final marks for students in an NMS course in an SMI table. Using the nodes {*enterprises NMSWorks (15760) course(10)*} as the base:
    (a) Pictorially draw the subtree of the management information tree.
    (b) Write the complete SMI macro definition for each new node.
17. For a certain trap generated by an agent, it is desired to provide an acknowledgement to the agent that the manager has received the trap. Using only SNMPv1 and SMIv1 without any changes, devise a mechanism to accomplish this.
18. Design a MIB variable *rebootNode*, which a manager can use to reboot the NE. The manager needs assurance that the node has been rebooted. Explain carefully the semantics and write the ASN.1 declaration of the variable.
19. It is desired to manage a traffic light using SNMP. The controller needs to know the count of the number of vehicles that have passed, and be able to change the light between red and green. Sketch the MIB subtree {traffic} for this purpose. For each node in this subtree, write the complete ASN.1 macro.
20. A single-threaded discovery module discovers the subnet 192.168.9.xxx. This subnet contains 20 NEs of which four are routers each having ten interfaces. Each network request from the NMS takes 2 seconds round-trip time. Each failure has a timeout of 30 seconds. Approximately how long will the discovery process take?
21. Write a pseudocode for topology discovery (similar to Figure 9.26). Assume that routers are SNMPv1-enabled with a known community string.
22. With a 1-GHz CPU, the average CPU time taken for status polling is: poll manager 0.6 ms, SNMP stack 0.8 ms. When the poll manager detects a change of status, it informs the FM and writes to the DBMS. The CPU times taken are: FM 1 ms, DBMS 5 ms. Assume that 10% of the polls detect a change of status, and that the CPU power increases linearly with the CPU clock speed.
    (a) What is the minimum CPU speed to handle 1,000 polls/second?
    (b) If the PM, FM, and SNMP are run on one CPU and DBMS on a second CPU, what are the minimum speeds of these two CPUs for 1,000 polls/second? Assume 20% increase in all CPU requirements due to the inter-CPU communication overhead.
23. A data collector polls for 100 OIDs, each on a different NE. When there is a response, the round-trip delay is 1 second per poll. When there is a fault, the timeout is 30 seconds. Assume that 20% of the polls suffer a timeout.
    (a) What is the minimum polling period if the data collector is single threaded?
    (b) What is the minimum number of threads needed to achieve a polling period of 30 seconds?
24. An NMS manages 1,000 large NEs and 100,000 small NEs. For each of the large NEs, it polls 100 OIDs with a polling interval of 5 minutes. For each of the small NEs, it polls two OIDs once an hour.

(a) Estimate the volume of data to be stored in the polled data tables (excluding indexes and temporary space) per day, per week, per month, and per year.
(b) Assume that the network consists of 20 regions each containing 1/20th of the NEs. The following database designs are being considered: one table/NE, one table/region, one table/whole network. In each case, a new table may be created every day, every week, every month.
(c) Draw a table with columns labeled day, week, and month, and one row for each database design. In each cell of the table, enter the number of records/table and the number of tables if data are kept online for 6 months.
(d) The vendor of the DBMS recommends that one table should not contain more than 500 million records and that the DB should not contain more than 1,000 tables. Mark the designs that are feasible according to these criteria. Which one of these designs do you prefer? State your reasons.

25. A telecom network that used to serve 100 million subscribers has 1 million NEs. Suppose the poll manager polls 1% of the NEs for 100 OIDs each at a 5-minute interval and 20% of the NEs for 10 OIDs each at a 15-minute interval. The remaining NEs are polled only for status once an hour.
    (a) Compute the disk space requirements for 1 day, 1 week, 1 month, and 1 year. Assume that the disk space requirement is three times the table size to allow for temporary files
    (b) Suppose the PM also polls the status of each subscriber unit (such as phone, ADSL modem, etc.) once a day. What is the additional disk space requirement?
26. An NMS is connected to a remote network by a 64 kb/s link. There are 1,000 NEs in the remote network, each having 10 OIDs of interest. Assuming 5 OIDs in each SNMP get-request, neatly sketch a curve showing the bandwidth used as a function of the polling period, p. What value of p will result in 20% utilization of the link?
27. In Exercise 26, assume that a regional NMS in the remote network does the polling and once an hour does a bulk file transfer of all the data values only to the central NMS. What value of p will result in 20% utilization of the link? Assume that scp is used and achieves a compression ratio of 50%.
28. An NMS is connected to a remote network by a 64 kb/s link. The NEs in the remote network generate 50 faults/second. Of these, 1% are critical and the others are minor. What fraction of the link bandwidth is used if:
    (a) All fault notifications are sent using SNMP traps?
    (b) Critical fault notifications are sent using SNMP traps, and minor fault notifications are sent once a day using scp? Assume that a trap is of size 100 bytes, and that the information about a fault occupies 8 bytes in a file. Neglect all other overheads.
29. SNMPv1 defines communication only between the manager and the agent. In a network with multiple management servers, communication between them may be necessary. For example, when the operator at one server acknowledges an alarm, the alarm indication should also stop at the other server. Define a mechanism using only SNMPv1 for this purpose.
30. We wish to have two management servers each with a copy of the database for redundancy. There are two approaches.
    (a) Each server independently polls the agents and updates its copy of the database.
    (b) The active server polls the agents and then passes the data onto the passive server.

    What are the pros and cons of these approaches? Consider criteria such as load on the agents, network traffic, and loss of data in case of server failure. The consistency of views is an important consideration.