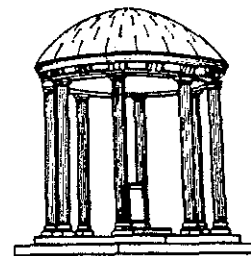Providing Common Access Mechanisms for
Dissimilar Network Interconnection Nodes

TR91-013

February, 1991

John Menges

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

UNC is an Equal Opportunity/Affirmative Action Institution.

Providing Common Access Mechanisms for

Dissimilar Network Interconnection Nodes

by

John E. Menges

A thesis submitted to the faculty of The University of North Carolina at Chapel Hill
in partial fulfillment of the requirments for the degree of Master of Science in the
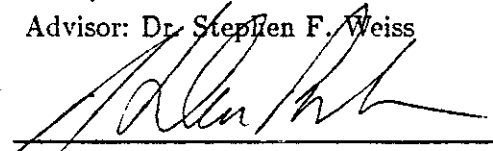Department of Computer Science.

Chapel Hill

1990

Approved by:

_____

Advisor: Dr. Stephen F. Weiss

_____

Reader: Dr. J. Dean Brock

_____

Reader: Dr. Daniel A. Pitt

John E. Menges. Providing Common Access Mechanisms for Dissimilar Network Interconnection Nodes (Under the direction of Stephen F. Weiss.)

ABSTRACT

Over the past several years, thousands of Local Area Networks (LANs) around the world have been interconnected to form huge computer networks. These interconnections between LANs have been made using a variety of different classes of Interconnection Nodes (INs) made by many different vendors. Management of these essentially similar INs has been more difficult than necessary because of the dissimilar methods required to manage the various IN types.

This thesis describes the design and implementation of access mechanisms that make it possible to manage the similar aspects of the various IN types in a common manner, without sacrificing the type-specific functions of the various proprietary management systems.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF FIGURES

Chapter I

An Introduction to the Problem

With the advent of large computer networks comes the need to manage these networks. Network management involves both maintaining adequate data transmission capabilities in the face of growing and changing needs and keeping the network operating smoothly and/or restoring it to operation quickly in the event of hardware and software failures.

There are two major aspects of network management: Local Area Network (LAN) management and LAN interconnection management. LAN management focuses on individual Local Area Networks, while LAN interconnection management involves only the portion of the network that ties LANs together into a larger network. For example, monitoring individual packets on a LAN to isolate a protocol bug is a LAN management function, while monitoring the utilization of a serial link between a campus and a regional network in order to ensure that the link has sufficient capacity is a LAN interconnection function. In this paper I will address only LAN interconnection management.

The reader of this paper is presumed to be familiar with computer science and networking concepts. The first chapter of [Tan88] is a good introduction to networking.

A large portion of the interconnection management problem involves knowing what is going on at those points where LANs are interconnected and controlling the configuration and state of the various classes of devices that interconnect these LANs. These devices, which I shall call Interconnection Nodes (INs), can be categorized into a small number of major classes, depending on the layer of the ISO reference model[1] into which they naturally fit. For example, repeaters operate at the physical layer, bridges at the link layer, and gateways and routers at the network layer.

In general, the higher up the ISO reference model an IN belongs, the more configuration and state information there is to maintain and monitor. A repeater could count how many

---

[1]The International Organization for Standardization (ISO) Reference Model of Open Systems Interconnection is 7-layer conceptual model for communications protocols. See [Com88, Chapter 10] for a discussion of protocol layering, including the ISO model.

packets it has transferred in each direction over each interface and (in the case of Ethernet) how many collisions it has detected, but since it operates at the physical layer, it could do little else. There are also few if any conceivable options for configuring an IN at this level. A bridge, operating at the link layer, could keep track of how many of its received packets were forwarded, filtered, and ignored, how many of each major (link-layer) type of packet have been processed in each of these categories, over which interface each host address is reachable (using link-layer addresses), how many packets were sent to or received from each host, etc. Configuration information might consist of custom packet filters, the number of active interfaces, or interface types and speeds. A gateway or router might maintain packet counts broken down by major and minor protocol type, protocol error counts, routing failures, and so forth. The configuration of an IN at this level involves such things as maintaining routing tables and distance metrics, alternative routes, choice of dynamic route computation protocols, and much more.

Though INs within a particular class (e.g., bridges) may be designed by different vendors and may perform somewhat differently depending on their specific intended application, they tend to have a high degree of commonality among themselves as to the types of configuration options and monitoring information needed to manage them. Even between classes of INs, there is significant overlap. For instance, the number and types of interfaces, interface speeds, and per-interface packet counts might apply to all classes.

Until recently, each vendor has provided its own set of data structures for management-related configuration, state, and history information, and has assigned its own semantics to these data structures. Often these structures vary from model to model and from revision to revision, even among products from the same vendor. Each vendor also supplies its own command language for retrieving management information and setting configuration parameters, and again this often varies from model to model and revision to revision. Some of these command languages are human-oriented, while others are computer-oriented. Figure 1.1 shows how to get link level statistics from a LANmark Ethernet bridge. Figure 1.2 shows how to get the same type of information from a 3Com IB/1 bridge.

Finally, the underlying protocols and physical media over which management functions are performed vary greatly. Some INs can only be managed via out-of-band, low-speed serial ports. Some (e.g. bridges) can be managed in-band, but since they operate below the network level, they do not necessarily use the same transport protocols as higher level INs

```
SELECT COMMAND => dgn
DIAGNOSTIC NAME or ? or Return=END.......... => lndg
TEST NUMBER or ?............................. => 3
SPECIFY PORT # or L or P or ?............... => 320
SPECIFY CURRENT STATISTICS TYPE  or R = RETRIEVE SAVED STAT or ? => 1
*** LAYER STATISTICS    DIU5                            08/25/89   09:28:40
   PORT  320      LN   0


                      transmitted    received
PACKETS:                45515205     54556166
BROADCAST PACKETS:        931364     11256085
VIRTUAL CHN DATA:              0            0
VIRTUAL CHN PROTOCOL:          0            0
PACKETS DISCARDED:           161        51984
ELAPSED TIME:    022.00:22:34



CLEAR STATISTICS?   Y = YES, N = NO..........N => @

SELECT COMMAND =>
```

Figure 1.1: Getting Link-Level Statistics from an IBX LANmark Bridge

```
# show netstatistics 0
                         Network 0 Statistics
RcvPackets RcvBytes    MCPkt BCPkt Crc   Frame Long  LostP
---------- ----------  ----- ----- ----- ----- ----- -----
1356334    103941443   19282 29529 0     0     0     0
XmtPackets XmtBytes    MCPkt BCPkt Defer 1Coll MColl LColl EColl NetErr
---------- ----------  ----- ----- ----- ----- ----- ----- ----- ------
502272     54303486    39    41377 387   15    9     0     0     0
# show netstatistics 1
                         Network 1 Statistics
RcvPackets RcvBytes    MCPkt BCPkt Crc   Frame Long  LostP
---------- ----------  ----- ----- ----- ----- ----- -----
4230131    487520021   64843 64897 1450  1529  0     0
XmtPackets XmtBytes    MCPkt BCPkt Defer 1Coll MColl LColl EColl NetErr
---------- ----------  ----- ----- ----- ----- ----- ----- ----- ------
830216     62838838    11403 10125 13590 2666  1121  0     0     0
#
```

Figure 1.2: Getting Link-Level Statistics from a 3Com IB/1 Bridge

on the same network are able to route. (For example, a bridge might use Xerox Network Services (XNS) for network management, while the larger network may be connected via Internet Protocol (IP) routers.)

In some cases these differences are essential, but in other cases they are accidental. Accidental differences in management mechanisms unnecessarily complicate the management of large computer networks, particularly networks that involve many different classes, models, and revisions of INs from various vendors. Even if management mechanisms were common to all the INs within each administrative domain, differences in mechanisms between domains limits management tool sharing across domains.

These problems have been recognized by the Internet[2] and ISO communities, and network management standards are emerging to address them. There are, however, large networks already in place utilizing many INs that do not now and will never support the emerging network management standards. Furthermore, much of this equipment has a long lifetime and will be in use for many years to come. This thesis addresses the problem of integrating such existing INs into network management systems utilizing the new standards. The software developed for this project provides a mechanism for accessing INs that do not support standard management protocols.

First, an overview of this paper is in order. This chapter introduces the problem to be solved. Chapter 2 discusses what is *above* the access mechanisms developed for this project, i.e., the network management tools that will use this software to gain access to INs in the network. Chapter 3 describes the target network *below* this software, the North Carolina Data Network (NCDN) and the INs that tie it together. Chapter 4 outlines the constraints on the design of this project imposed by the desired tools and the target network. Chapter 5 discusses the standards environment in which we are operating. Chapter 6 gives an overview of the design of the project, while Chapters 7 through 10 detail the design of the individual components. Chapter 11 evaluates the resulting system to see how well it conforms to its requirements and gives suggestions for improvements. Chapter 12 discusses potential future projects and applications. Finally, the appendix contains the source code. The reader can gain a good overview of this project by reading chapters 1 through 6, 11, and 12. Chapters 7 through 10 and the appendices are essential reading for a maintainer of the software, particularly if the reader will be adding device drivers.

---

[2]For an introduction to the Internet, see [Com88, Chapter 1].

Chapter II

An Overview of IN Management Tools

This chapter puts the project into perspective by giving an overview of the types of tools that are necessary to properly manage LAN interconnections, and the capabilities we require of a network management system supporting these tool types.

2.1 Classes of Tools

The types of tools necessary for managing INs can be grouped into five major classes, as follows:

1. **Display Only** tools are able to obtain information from an IN or a set of INs and display this information, but are not able to change the configuration or state of an IN.

2. **Display and Control** tools have the same capabilities as *Display Only* tools, but in addition are capable of changing the configuration of an IN (e.g., its routing tables) or its state (e.g., by rebooting).

3. **Monitoring and Alerting** tools monitor a set of INs, detect abnormal situations, and alert the appropriate operations personnel when necessary. They might also be able to take corrective action on their own.

4. **Statistical** tools collect information from INs and store it into a database for later perusal or report generation.

5. **Direct Interaction** tools provide an escape hatch for situations where the set of available tools falling into the above classes do not cover all possible information gathering and control functions. They allow direct interaction with an IN via its proprietary command language.

INs **Tools** **User Interfaces**
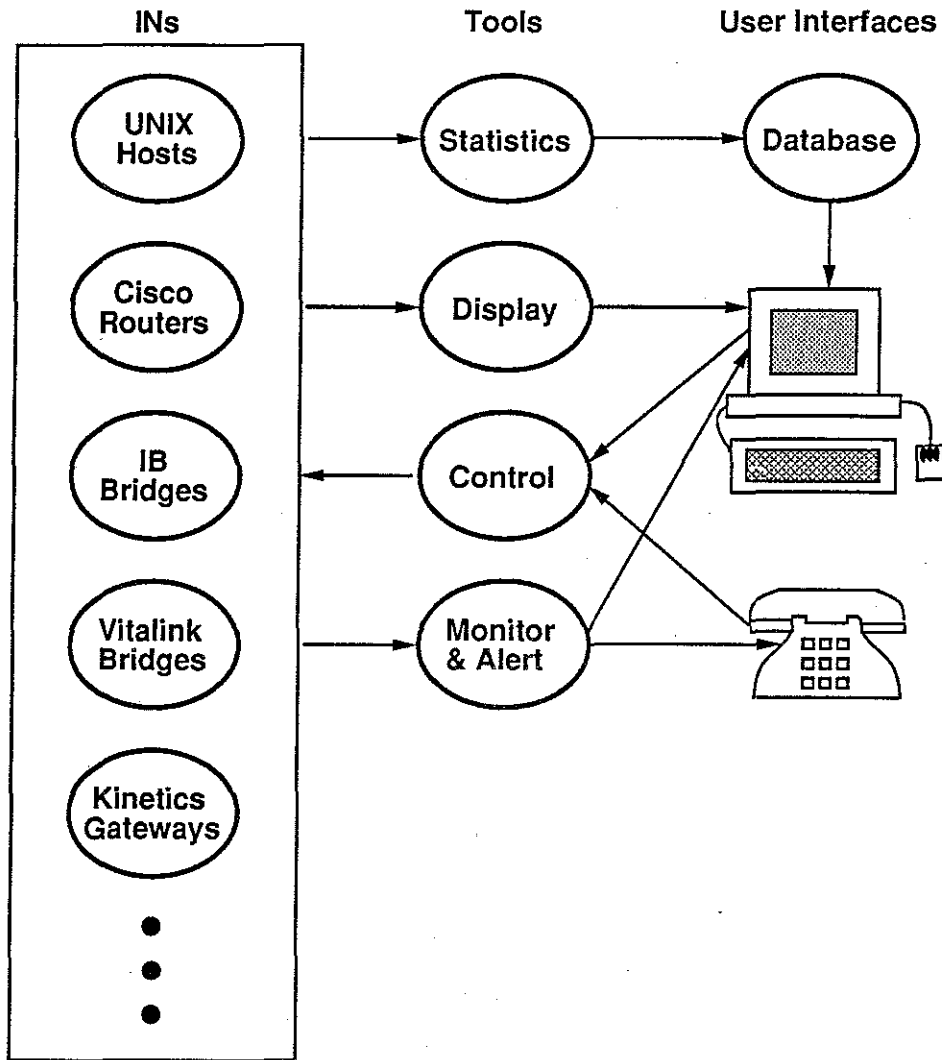
Figure 2.1: An Overview of Network Management Tools

Note that the boundaries between these tool types are flexible. For example, one might imagine a *Statistical* tool that also detects abnormalities and functions as a *Monitoring and Alerting* tool as well, or a *Display and Control* tool that incorporates a *Direct Interaction* feature.

We will now take a more detailed look at each of the above-mentioned tool types. Refer to Figure 2.1 throughout this discussion.

2.2 Display Only Tools

There is a wide variety of potential *Display Only* tools. The form a given tool might take would depend on the intended users of the tool, the types of INs being monitored, the number of INs being monitored simultaneously, the type of information to be displayed, and the capabilities of the available display hardware.

Users of a distributed computing system may wish to know the overall status of a network so that they can determine which resources are currently available to them and which are unavailable due to network failure. They also may wish to know how fast data is flowing at various points in the network, to help them decide which of a set of redundant resources can be accessed most efficiently. This suggests graphical tools that display the overall network or a portion of it in logical or geographical form, with reachability and data flow rate information encoded using colors, line styles, and gauges.

Of course, the more reliably the network satisfies the user's needs, the less the user will want to use such a tool. Thus, the availability to network management personnel of good network management tools may obviate the need for such tools by the user. (Who but the telephone company has a need to monitor the telephone network?) One can hope that these tools will become unnecessary within a short time. Then the users can forget about the network and concentrate on their work (using the network).

In addition to the capabilities required of user-oriented displays, network operations personnel will need tools which are capable of displaying much more information, such as the details of the configuration and state of particular INs, link utilization graphed dynamically over a period of time, error counts on particular interfaces, and so forth. They may also wish to trap and display individual packets or events for diagnostic purposes.

Some displays, such as network maps, might be able to treat all INs alike. Others will need to distinguish between different classes of INs, so that the various types of data structures can be displayed appropriately. For example, bridge displays might present link-level routing tables, while gateway displays might present network-level routing tables. Some displays will be oriented toward the network as a whole or a portion of it, while others will focus on one IN at a time.

The capabilities of the display hardware will greatly affect the form taken by a particular display tool. At the low end, line oriented tools can be envisioned for simple queries or reports, such as the value of a particular counter in a particular IN. These could run on

7

devices as simple as hardcopy consoles or even printers. Full screen oriented displays can be imagined for displaying limited amounts of information at a time about a single IN or a small set of INs. Figure 2.2 shows an example of such a tool. These tools might be useful to network operations personnel working from home, where more elaborate display hardware is not available. Finally, more sophisticated tools are possible where window-oriented bitmapped displays with mouse and keyboard interaction are available.

Once access to IN information is widely available, there is much room for brainstorming and experimentation with different methods of displaying this information. As we shall soon see, the types of anticipated display applications affect the design of the underlying access mechanisms.

2.3 Display and Control Tools

Display and Control tools would be used by network engineers and operations personnel to monitor the network, diagnose and fix problems, and change the configuration of the network. They would have all the capabilities of *Display Only* tools, but would additionally be able to change IN configurations and states. Most likely, these tools would present different capabilities to different classes of users depending on how much control of the network each class of users is allowed. For example, operators may be allowed to reboot INs but not change their configuration, while engineers may be able to do both.

Configuration changes would involve items such as configuring the IN software to match its intended use (e.g., indicating the types and number of interfaces a particular IN has), manually maintaining certain routing table entries, modifying bridge filters, changing passwords, etc. State changes might involve rebooting an IN, temporarily disabling a link, or clearing its event counters.

It is important that all INs within an administrative domain be easily accessible to the appropriate maintenance personnel from at least one central location. Access from all points on the network is preferable if this can be done without posing a security threat. Performing similar actions on different types of INs should be possible using similar mechanisms, to reduce the training requirements and frustration of operations personnel. For example, in a graphical tool, selecting a **REBOOT** menu item and then clicking on a box representing a particular IN and confirming the choice in a dialogue box might be *the* method of rebooting

Bridge Status/Traffic Monitor

Bridge Name/Address	Interface Number	Bridge Status	Input Pkts/Sec	Err/Sec	Output Pkts/Sec	Err/Sec
compcenter	0	Up	359	0	129	0
08000200A165	1		199	1	124	0
biomedical	0	Up	189	0	63	0
080002008935	1		301	1	47	0
chemistry	0	Up	49	0	65	0
08000200AF21	1		175	0	49	0
compsci	0	Up	109	0	87	0
08000200A608	1		262	1	51	0
physics	0	Up	4	0	36	0
08000200016C	1		130	0	4	0
geography	0	Down				
08000200D17B	1					
history	0	Up	145	4	99	0
08000200D13D	1		284	0	74	0
medschool	0	Up	16	0	87	0
080002010829	1		258	6	16	0
radiology	0	Up	351	4	68	0
080002010F56	1		292	1	79	0

Figure 2.2: A Simple Full Screen Display Only Tool

an IN, regardless of type This is illustrated in Figure 2.3.

Elaborate tools using bitmapped displays and mouse and keyboard input certainly have great value in simplifying the management of a large network. In most environments today, however, it is also necessary to have tools with the same basic capabilities as these full-blown management tools, but that have significantly reduced hardware requirements. Parallel tools must therefore be available for managing the network via, e.g., ASCII terminals over low speed telephone lines. These would be used by operations personnel at remote locations if, for example, they are replacing some broken hardware at an arbitrary location in the network, or are working from home outside normal working hours.

2.4   Monitoring and Alerting Tools

Many networks are large enough to require sophisticated network management tools but not large enough to justify employing network operations personnel to do nothing but monitor the network looking for problems. Thus, tools that monitor the network automatically and alert the appropriate personnel when some abnormal situation arises can be important in maintaining the reliability of a network. The goal is to have tools that reliably detect network problems as soon as the users do, if not before. In some cases, problems can be resolved before users are aware that there was a problem. This capability can be extremely valuable, both to the users of a network and to those charged with its maintenance.

Tools in this class can vary greatly in sophistication. Very simple tools that poll INs to see if they are up and send a message to an operator's screen or even an electronic mail message to his mailbox when one goes down can have dramatic effects on the reliability of a network. Much more sophisticated tools are easily imagined. The sophistication of such a tool is a function of what is being monitored, how intelligent the problem detection, analysis, and recovery methods are, and how appropriate personnel are notified when this is necessary.

What is monitored will ultimately place an upper limit on what problems can be detected, how much analysis can be performed, and what actions can be taken when a problem is detected. A simple monitor might just poll each IN in an administrative domain periodically using a simple query. The information collected will only indicate which INs are up (and healthy enough to answer the query) and which are down (determined by timing out waiting

10
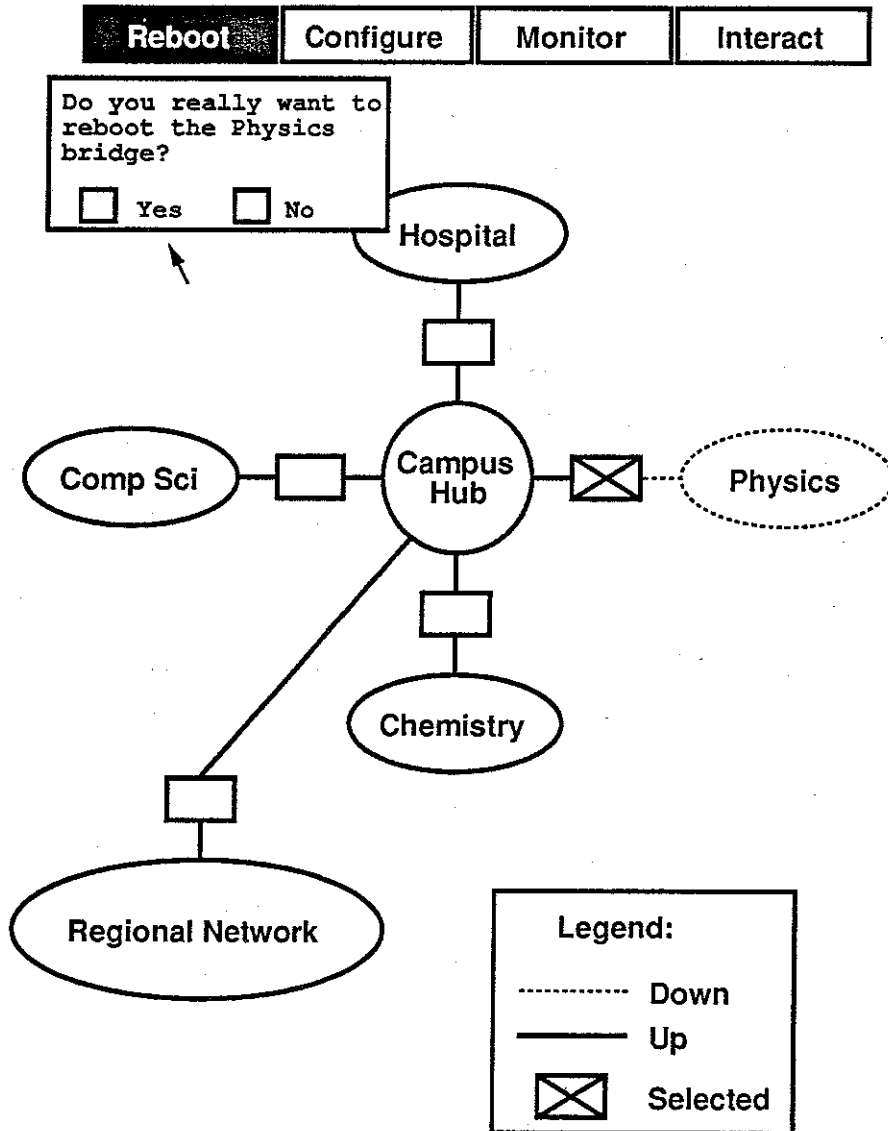
Figure 2.3: A Simple Display and Control Tool for a Bitmapped Display

for a response). This is enough to show which areas of the network are reachable. At first glance, it appears that this mechanism would only check reachability of the portion of the network *between* LANs, not LANs at the *edge* of the network. However, as we shall see, host computers that are not INs can be queried using the same network management tools as are used for INs, so information regarding the reachability of any portion of the network can be obtained using this simple mechanism. More reliable and informative results can be obtained by querying each IN to determine if *it* thinks it and its interfaces are up. (An IN may be able to answer simple queries, but may know its interfaces are down.) If packet counts are monitored, such anomalies as zero packets being transferred over a supposedly active interface, or abnormally high traffic rates, can be detected. If error counts are monitored, network *degradation* can be detected as well as network *failure*. Finally, if configuration parameters are monitored, changes in the configuration of an IN can be detected and logged. If the IN itself is capable of detecting problems, it can notify a network management tool without waiting to be polled. (Such a notification is called a *trap*.) To be entirely sure of the network status, however, some polling is necessary. This is because an IN may be too sick to know that it is sick and to be able to tell somebody.

Problem detection can be as simple as noticing when the status of an IN goes from *up* to *down*, or receiving an IN's alert that it has problems. More elaborate problem detection mechanisms usually involve maintaining some idea of what is normal and comparing collected information against this idea of normalcy. When certain predefined thresholds are exceeded, a flag goes up saying something is wrong. For example, if error and packet counts are retrieved periodically, the monitor can detect when the ratio of errors to packets processed is too high. The monitor's idea of normalcy and appropriate thresholds would typically be determined by network engineers and stored in a database, but one can imagine a monitor that determines normalcy automatically by observing the network in action. One example of the utility of this type of monitor is if one wishes to detect when one type of packet abruptly dominates a portion of the network for an extended period of time. This situation can signal a host gone haywire or an ill-conceived configuration change.

Once it has been determined that something is wrong with the network, the network management system has the option of simply reporting the symptom(s) (there may be more than one detected symptom of a single problem), or attempting to analyze the symptoms to determine their cause. This analysis can be *passive* or *active*. *Passive* analysis seeks

to determine what is wrong based solely on symptoms reported by the detection phase of the normal monitoring process. A network management system employing *active* analysis, on the other hand, is capable of requesting additional information from the INs when a symptom is detected, and using this information to further aid its analysis. In either case, the purpose of such analysis is to give operations personnel a clearer idea of what is wrong than is otherwise possible. As an example of when this might be useful, consider a tree-structured network with a monitoring agent at the root. If an IN midway between the root and the leaves of the network goes down, the symptoms of this failure are that all the INs from the failed IN toward the leaves will be detected as being down, as they are unreachable. Without analysis, this single failure would be reported as a multitude of individual failures, and it would be left to the operator to figure out the relationship between the INs that are reported as being down. On the other hand, if the analysis portion of the monitoring agent knows the structure of the network, it can perform this analysis and simply report the single failed node as being down. This type of analysis can greatly simplify network management and speed recovery. It can, however, be extremely difficult to design effective analysis logic for all but the most simple failure modes. It should also be noted that drawing false conclusions can be worse than not attempting to analyze the symptoms in the first place.

Once an a monitoring agent has determined what it thinks is wrong, it can simply report its findings to an operator, or it can attempt to resolve the problem on its own. Such automatic recovery efforts are most likely to be used as a temporary measure to keep a network going in the presence of certain intermittent problems with simple recovery procedures, until the source of the problem can be isolated and the problem can be fixed. For example, if a particular model of IN is known to have a software bug that causes INs of this type to occasionally stop forwarding packets, it may be possible to reboot an IN in this state to recover the network. This action could be taken automatically and immediately, rather than awaiting human interaction (which could come hours or even days later, if the problem occurs at night or on a weekend). But care must be taken not to go overboard with automatic recovery procedures, as they can easily cause more problems than they solve. All recovery procedures undertaken should be reported to an operator, or at least logged so that an operator can later determine what actions have been taken.

Finally, when a problem occurs, it is often necessary to alert the appropriate personnel so that the problem can be resolved by human interaction. Before alerting takes place, it

13

must be determined who should be alerted, and by what method they are to be notified. Determining who should be alerted can involve some analysis of the problem, and will likely involve the use of a database. For example, the alerting system may need to determine in which of several administrative domains an IN belongs, and then which personnel are responsible for this administrative domain. Or, it may use its analysis of the *seriousness* of a problem to decide what level of expertise is required of the operations personnel it must alert. In any case, once it has determined whom to alert, the monitoring agent will need to determine which of several methods of notification it should use. Possible notification methods include:

1. Making some visual change in a computer display. For example, on a graphical display, one might turn a box representing an IN from green to red.

2. Sounding an audible alarm, such as the bell on a terminal or workstation. This might be done in conjunction with the above-mentioned visual change.

3. Sending a message to a terminal or workstation window where the appropriate person is logged in. One may first need to find out where he is logged in, or one might assume some standard location.

4. Sending electronic mail. This is likely to be slower in reaching the target person, but might be more reliable in the sense that it gets a message to the one place he is certain to check within a reasonable period of time.

5. Calling via telephone. Systems now exist that will allow a person to be called on the telephone and given an arbitrary spoken message describing the network problem.

The first two methods are oriented toward notifying personnel whose job it is to be monitoring continual displays of network status, and requires them to be near the display station. The last three are methods used to track down operations personnel who are not likely to be near such a display station at all times. Method 5 has some important advantages. It is a good method for reaching people off-hours, when they are not likely to be near *any* terminal or workstation, much less the management display station. Of course, some limits may need to be placed on when the alerting system is allowed to call, perhaps as a function of the urgency of the problem. Even during normal working hours, telephone notification has several advantages. First, it allows the alerting system to easily gain immediate feedback

14

as to whether or not the intended target was reached. The telephone-calling system may require the entry of an identifying code via telephone buttons before a message is delivered. If the line is busy, the phone is not answered, or the identifying code is not entered, it may attempt to call another person to deliver the message. Secondly, it uses an independent network (the public telephone system) for notification, which may be important in the presence of a network failure that prevents the delivery of the message announcing the failure.

2.5 Statistical Tools

Statistical network management tools gather information from INs on a periodic basis and store this information in a database, where it can later be accessed by report generation programs and perused by management personnel using a database query language. The data collected would typically include uptime information and traffic statistics broken down various ways (e.g. by protocol and by interface). It might also include relatively stable configuration information, stored in the form of a single *snapshot* (typically the current configuration) and *configuration change* information from which a history of IN configurations can be constructed.

This type of tool is useful for long-range planning and detecting changes in usage patterns over time. The data collected can help in the decision processes leading to configuration changes and equipment purchases and can help justify proposed changes to higher-level management.

2.6 Direct Interaction Tools

In some situations, the available general-purpose *Display and Control* tools available do not cover the whole ground when it comes to information gathering and configuration and state control capabilities. Vendors often supply some means of direct control of an IN via a command language. In some cases, this access is obtained using a fairly standard mechanism, such as **telnet**. In other cases, a proprietary protocol is used. Sometimes such direct access can only be obtained via out-of-band serial lines, while at other times access can be obtained through the IN's normal data interfaces. While such *Direct Interaction* tools do not lend themselves to management mechanisms that are common among INs of different types and from different vendors, access to these interactive interfaces is often required. This class of

tools could be considered a special case of the *Display and Control* class, but because of its lack of generality I list it separately.

2.7 Summary

As can be seen from the above discussion, there are many different network management tools that might be useful in managing a large computer network. These tools can vary widely with respect to intended users, functionality and sophistication, but they can be classified into a relatively small number of types. As we shall see in Chapter 4, this classification can help us to assess the demands such tools might make on the underlying IN access mechanisms. In Chapter 5 we will see that a rather simple access mechanism can be the foundation upon which a wide range of tools and tool systems are built. But first, let us take a look at the network on which these tools will be used.

Chapter III

The North Carolina Data Network

The North Carolina Data Network (NCDN) is a Wide Area Network (WAN) that links together Local Area Networks (LANs) across the state of North Carolina. It is part of the Southeastern Universities' Research Association Network (SURAnet), which, in turn, is part of the Internet. The participating institutions are primarily universities and research institutions.

This network provides a vital data communications mechanism to the participating institutions. It binds the participating institutions together with each other and with institutions around the country and the world via electronic mail, distributed file systems and database systems, file transfer capabilities, and much more. It allows researchers to share software, collaborate on research, publish results, share solutions, debate issues, etc. In short, it enables them to work together efficiently.

Fifteen major geographically separated locations across the state are connected via microwave or leased land lines, at data rates from 56 Kbps to 3.088 Mbps. Buildings on institutional campuses are typically connected via INs linked by broadband or fiber, and LANs within a single building are usually connected using local INs. Data rates through INs range between 1 and 10 Mbps in both cases. Various transmission protocols are used on the NCDN, including TCP/IP, DECnet, XNS, and AppleTalk. TCP/IP dominates, as it is available for most operating systems. It is also the only transport protocol that is routed between the NCDN and the SURAnet and Internet.

The Microelectronics Center of North Carolina (MCNC) owns, operates, and maintains a large portion of this network, especially the inter-institutional network. The MCNC-owned portion of the NCDN is called CONCERT (COmmunications for North Carolina Education, Research, and Technonogy). Figure 3.1 gives an overview of the CONCERT network, and Figure 3.2 gives more detail about the particular LANs and INs and how they are interconnected. The work described in this thesis was supported by MCNC for their
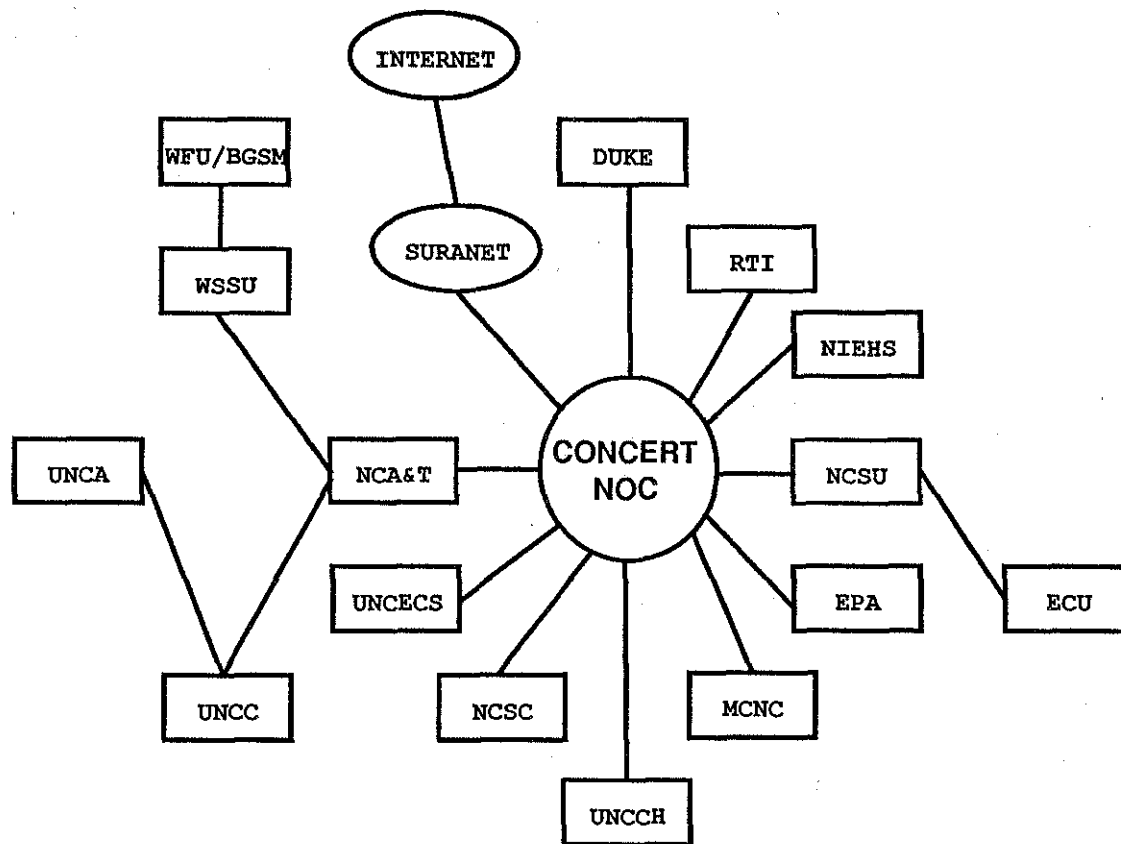
Figure 3.1: Overview of the CONCERT Portion of the NCDN

Figure 3.2 CONCERT Network Detail

use at the hub of the network, located on their premises. It is also being made available to all the participating institutions for use from within their administrative domains.

LAN interconnections in the NCDN are made using a variety of IN types from various vendors. At present, there are about 135 INs, broken down as follows:

1	Proteon 4200 Router (Gateway to the SURAnet and Internet)
21	Cisco Routers
11	Kinetics AppleTalk/Ethernet Gateways
6	UNIX hosts operating as IP Gateways
28	3Com IB/1 Bridges (Ethernet to Broadband)
4	3Com IB/2 Bridges (Ethernet to Ethernet)
7	3Com IB/3 Bridges (Ethernet to High Speed Serial)
11	VitaLink TransLAN Bridges
7	Intecom IBX LANmark Bridges
17	IBM PC-compatibles running Novell Netware Gateway software
9	Digital LANbridge 100s
4	Gatorbox AppleTalk/Ethernet Gateways
4	Retix Ethernet Bridges
1	AT&T Commview Gateway
1	Digital Decnet/SNA Gateway
1	Cabletron NB20E Ethernet Bridge
1	Synoptics 3323 Bridge
1	Network Systems Corporation EN643 Router

Of these IN types, only the Proteon and Cisco routers support any standard network management protocol as they come from the vendor. They both support the Internet standard, called the Simple Network Management Protocol (SNMP) [CFSD88]. (SNMP and other standards will be discussed in Chapter 5.) SNMP support is also available from third parties for the Kinetics gateways and UNIX[1] hosts. The other INs support various proprietary network management protocols. Thus, ignoring the work described in this thesis, only about 29% of the INs in the NCDN can support a standard management protocol. This percentage is sure to grow, however, as newer INs are installed that support SNMP (or

---

[1]UNIX is a trademark of Bell Laboratories

another standard)[2].

Work already completed on this project extends SNMP support to all the 3Com, Vi-taLink, and LANmark bridges, bringing the SNMP-supporting fraction to about 71%. Drivers can easily be added to obtain support for most of the remaining IN types.

Many different operating systems are used in the NCDN, including UNIX, VMS, VM, DOS, OS/2, etc. Network management software might be run under any of these operating systems, but if one needs to be chosen based on widest use, UNIX is the most likely candidate.

Although the NCDN is vital to the work of the participating institutions, and is in use 24 hours a day, 7 days a week, the network is not yet large enough to justify constant human surveillance even during normal working hours. Without automatic monitoring systems, a problem in the network often exists for hours before someone who can do something becomes aware of it. Even when an appropriate person is aware of a problem, network management tools currently in widespread use are extremely primitive, causing diagnosis to be slow and painful in many cases. Network failures are frequent, and users are all too aware of the possibility of the network being down when they want to use it. We must move toward a network that, like telephone and electrical service, has few enough outages that users are *surprised* when something does not work. A good network management system can help us approach this goal.

---

[2]Since the time this information was compiled (Summer 1989) vendors have provided SNMP support for many more of the above types. In most cases, however, SNMP support requires a hardware upgrade to existing INs.

Chapter IV

Design Constraints

In this chapter we will take a look at the design constraints on this project that are imposed by the desired tools and the target network discussed in the previous two chapters. First, we will take a look at how the various tool types discussed in Chapter 2 affect the design of a network management system. Then we will see what demands the target network and the INs that link it together (described in Chapter 3) make on the design.

Many of the numerical goals derived in this chapter are based on arbitrarily chosen usage patterns and acceptable levels of performance. I have used the experienced gained during several years of network management to arrive at these figures, but others might easily have drawn somewhat different conclusions. I believe meeting the requirements I set forth in this chapter will result in a good management system for most existing networks.

4.1   Requirements Imposed by Desired Tools

In Chapter 2 I identified five classes of network management tools. Let us take a look at each tool class in turn, identifying the requirements each imposes on our system.

4.1.1   Display Only Tools

In the discussion of *Display Only* tools, I mentioned that general users of the computing system who are making non-trivial use of the network (a large and growing proportion of all users) might want to be able to run tools that display the status of the network, including reachability information and some measure of the busyness of various portions of the network. One possibility mentioned is a graphical display with a logical or geographical view of the network, indicating reachable and unreachable portions of the network by color or drawing style, and some sort of meter indicating how busy each portion of the network is. This type of tool alone places a great burden on the network management system. At times

when some major portion of the network is down, there might be literally hundreds of users running such a tool to find out what is wrong. A poorly designed network management system might make matters worse by increasing network traffic and the processing burden of the INs all over the network whenever some portion of the network experiences a failure or degradation in performance.

Such a tool imposes three requirements on a network management system:

1. **Caching.** Users must not be allowed to access INs directly, or the load imposed by these tools will increase linearly with the number of users using the tool at any given moment. There mut be an intermediate agent that collects information from the INs at a reasonable rate, caches this information, and hands it off to user tools on request.

2. **Distribution of Caching Agents.** Even if there is a single caching agent that insulates INs from excessive requests, traffic to and from this caching agent can overload INs. Thus, caching agents must be distributable. Caching agents should be able to share information with each other, so that only the caching agents within a particular administrative domain are querying the INs within that domain.

3. **Standard Access to Caching Agents.** The transport protocols used to access the caching agents must be available on a wide variety of operating systems, so as to allow widespread use of *Display Only* tools.

How often might one need to query each IN for such an application? A user who has just lost a connection to a remote machine is likely to pull up a network display immediately. If the display does not reflect the problem within 10 to 15 seconds, the user is likely to start using other methods to figure out what is wrong. Querying 10 INs in a 10 second period would require 1 query per second (per user). An indication of when the network is back up is less critical. Perhaps 1 or 2 minutes would suffice for this.

The information requirements of a user-oriented display are minimal. Normally, operational status and per-interface packet counts are sufficient. The display portions of *Display and Control* tools for the use of network operators have greater requirements. These will be discussed in the next section.

4.1.2 Display and Control Tools

Network operations and engineering personnel have a need to monitor and control INs at every level of detail. They also need to manage different types of INs from different vendors. They must be alerted when things go wrong, and must be able to request detailed information when diagnosing a problem. Once the cause is found, they should be able to control the INs as well, in order to fix the problem. These needs suggest the following management system characteristics:

1. **Full Capabilities.** Standard network management tools should have every management capability the vendor provides through any other mechanism, so that the operator does not have to use one mechanism to gain one capability and another mechanism to gain another.

2. **Commonality.** INs that have a common capability (e.g., rebooting) should be controlled in an identical manner. Tool writing can be facilitated by achieving this commonality below the tool level.

3. **Security.** The need for full remote control of an IN makes good security mechanisms necessary, to avoid unauthorized access to INs.

A good diagnostic system will need to be able to access a few INs at a time at a relatively high query rate. A particular piece of information may need to be retrieved from several INs once per second or so. If several items of information are needed each second, this may involve several queries of an IN per second while such diagnosis is going on. Thus, query turnaround times should typically be less than one second.

4.1.3 Monitoring and Alerting Tools

Ideally, a *Monitoring and Alerting* tool should be able to alert a continuous display within 10 to 15 seconds of a network failure, so that operations personnel watching for a problem are notified promptly and can take action immediately. It might even be possible to fix some problems before connections time out, if a fix is known, fast, and non-destructive of connections. It is also advantageous for operators to know about a problem by the time a user calls to report it. This gives the users confidence that things are being taken care of, and makes them less likely to call and bother operations personnel who are trying to figure out what is wrong.

24

Such speedy notification can place a heavy, steady demand on both the INs themselves and on the LANs carrying management information to and from INs. If the INs themselves are capable of alerting a monitoring tool when certain problems arise, this can reduce the need for frequent polling. Some forethought, knowledge of the network topology, and added complexity in the monitoring agent can also help. For example, it may not be necessary to poll every IN for operational status. If those at the far reaches of the network are polled successfully, the INs in-between must be working. As a last resort, it may also be necessary to trade away some alerting promptness in order to keep network congestion low.

Monitoring for operational status should be frequent, but monitoring for system degradation and long-term problems can be less so. Looking for increased error counts or excessive traffic might require only one or two polls per IN per hour.

The information requirements of *Monitoring and Alerting* tools are typically very small. Operational status and packet and error counts are all that are required in most cases. Occasional dumping of configuration information may also be useful. An intelligent analysis capability might require more information, but still less than a *Display and Control* tool would need.

4.1.4   Statistical Tools

Statistical tools can have large information requirements, but typically have low polling rate and turnaround time requirements. One might occasionally wish to gather statistics every 10 minutes or so for about a week, to analyze how the network changes throughout a day or a week. Normally, however, daily statistics are more than sufficient for planning, analyzing trends, and detecting slowly degrading components. This daily poll may, however, dump large portions of the state of each IN on the network, in order to do configuration change detection and logging.

4.1.5   Direct Interaction Tools

Direct interaction tools impose a different sort of burden on the network management system. They require some common mechanism for gaining interactive access to a wide variety of IN types using varying physical media and transport protocols. Some means of bridging the gap between these proprietary interaction mechanisms and a common user interface using a ubiquitous transmission protocol is required.

25

4.1.6 Tool Complexity

While we are talking about tools, it should be noted that network management tools can be complicated, and therefore time-consuming and expensive to design, code, debug, and maintain. If we use standards wherever possible, this burden can be greatly diminished by sharing tools. A very large community of potential tool writers and users exists on the Internet. It should be a requirement of our system that we use the same standards as this community, so that we can use their tools and they can use ours.

4.2 Requirements Imposed by the Target Network

The target network itself imposes a number of requirements on our network management system, because of its topology, administrative organization, and component IN types.

The topology of the NCDN has little redundancy at present. Thus, unless there is some back-door for gaining management access, tools using the network itself to gain access will only be able to tell how the network looks from the perspective of the point in the network where information is gathered and cached. It might be useful for such caching agents to be able to exchange information via the public telephone network or some other alternate network. Access to the caching agents themselves may be a problem. There should at least be one within each administrative domain, so that operators in each domain will be able to see what is going on.

As noted in Chapter 3, those INs in the NCDN that support any non-proprietary network management protocol at all, support SNMP. This is also the Internet standard. SNMP, then, is the obvious choice for a standard protocol. (Standards will be further discussed in the next chapter.) We need a system that will allow those devices that do not support SNMP to look as if they do. I also noted in Chapter 3 that there are many different types of INs that do not support SNMP. Our design must make it easy to add new types of devices to bring them into the SNMP fold.

Finally, the INs in the network must not be overburdened by a network management system. While network management is important, and some network resources will need to be spent on providing good management capabilities, we must place some upper limits on how much of our network resources we will allow network management to consume. These upper limits will have to be determined empirically after the management system is

in place, but we need initial approximations to guide our design. Intuitively, a proportion of any resource higher than about 5% dedicated to network management seems excessive under normal circumstances. During diagnosis, a proportion higher than about 15% might under some circumstances skew our results too much to allow us to understand what is going on while the system is operating normally. Querying a *particular* IN more than a few times a second also seems both unnecessary and unacceptable.

4.3 Summary

Before moving on, let us summarize the conclusions drawn in this chapter. Listed below are the demands that tools and the target network make on our management system. For clarity, they are reorganized and restated from the discussion above.

1. Caching agents must be used to collect information from INs, cache it, and deliver it to tools.

2. Caching agents must be fast enough to handle about one hundred requests per second.

3. Caching agents must be distributable among administrative domains, and should share information with each other. It should also be possible for them to communicate with each other using a second, independent network.

4. Communication with caching agents must be done over standard transport protocols, to facilitate wide distribution of tools.

5. It should be possible to use standard management tools to perform any management task on an IN that can be performed via another means.

6. The management protocol should mask accidental differences in the way identical management functions are performed on different IN types, below the tool level.

7. The management protocol should have authorization mechanisms capable of preventing unauthorized IN access.

8. The network management system should allow INs to be managed using imported tools that conform to current Internet standards.

9. The network management system must not use more than about 5% of any particular network resource (LAN or IN) under normal circumstances, and not more than about 15% while diagnosis is taking place. No IN should be queried more than a few times per second in any case.

10. Round-trip time for a query of an IN must be less than one second.

11. Operational status queries (queries for up or down status) must be particularly undemanding on INs and LANs, as they are executed frequently.

12. Interactive access to INs is required, and should use a common interface at the user level and standard transport protocols at the tool/network interface.

13. Our system must make it as easy as possible to add drivers that will give us access to more IN types.

Now that we know the requirements for our network management system, we can take a look at some standard solutions that will help us meet our requirements. We will do so in the next chapter.

Chapter V

Network Management Standards

In the previous chapter we established the need to use network management standards.
There are two reasons why this is important:

1. Newly purchased INs are likely to support a standard network management system.
   When they do, and when the standard they support is the one (or one of the ones)
   we are already using, these new INs fit trivially into our overall network management
   system.

2. Tool writing is time-consuming and expensive. If we follow standards, we are more
   likely to be able to share tools and thereby decrease tool writing costs.

In this chapter, we will take a look at what standards are now available and which are on the
horizon. I will then discuss how to fit our network management system into the standards
environment in which we operate.

5.1 ISO Standards

The International Organization for Standardization (ISO), in an effort to promote inter-
operability among various types of computing systems, is developing standards for network
protocols. As a part of this effort, the ISO is developing a network management system stan-
dard called Common Management Information Services (CMIS) [ISO90b]. CMIS defines the
management capabilities of network elements and the management objects (counters, tables,
etc.) needed to support these capabilities. It defines which management objects are to be
maintained for the network entity as a whole and for each protocol layer. It also specifies
the internal interfaces with the various layers that the management system uses to gain
access to these objects, and the external interface to managing entities. The external inter-
face standard is called the Common Management Information Protocol (CMIP) [ISO90a],

the protocol to be used for communicating management information between the managed and managing entities. CMIP is intended to run on top of ISO transport protocols. CMIS supports both simple objects such as counters and addresses, and aggregate objects such as sequences and tables. Management object values are encoded using Abstract Syntax Notation One (ASN.1) [ISO90c], an ISO encoding standard.

CMIS is expected to become the Internet network management standard in the long run, but at present it is still under development and unavailable for use.

5.2 Internet Standards

The Internet currently uses the Internet Protocol (IP) as its underlying transport protocol, with datagram and session services built on top of IP. The datagram service is called the User Datagram Protocol (UDP), and the session service is called the Transmission Control Protocol (TCP). The Internet Activities Board (IAB) has stated that it intends to convert to the ISO protocols when they are well defined and stable. This includes transport protocols to replace UDP, TCP, and IP, as well as higher-level protocols to perform such functions as file and mail transfer, interactive sessions, etc. CMIS/CMIP will also become the network management standard for the Internet [Cer88].

In the mean time, however, since CMIS/CMIP is unavailable, the Internet has had to define its own intermediate standard for network management to fill the gap in time. This intermediate standard is the Simple Network Management Protocol (SNMP). The SNMP standard was published in August 1988, and was essentially an improved (but not upwardly compatible) version of the previous standard, the Simple Gateway Management Protocol (SGMP), which was published in November 1987. SNMP will be discussed in greater detail in the following sections.

The IAB is also working on a transition plan for converting to CMIS/CMIP. One of the results of this effort is called CMOT, which stands for **CMIP O**ver TCP/IP [WB89]. CMOT is intended to ease the transition to ISO standards by making it possible to access the CMIS network management system running on a network entity supporting TCP/IP (and possibly ISO) transport protocols, using TCP/IP as a transport mechanism. It works by injecting a Lightweight Presentation Protocol (LPP) between CMIS and TCP/IP on the managed and managing entities, to allow the CMIS management agent and managing

30

client to use the standard ISO presentation service interface and have calls to this interface mapped into calls to the TCP layer for actual delivery.

Another part of the transition plan is to separate the management protocol and object definitions from object type definitions, the encoding method for object instance values, and the object naming scheme. The Structure of Management Information (SMI) [RM88] thus defines object types, how they are to be encoded, and how they are to be organized into a naming plan. The hope is that the SMI will remain relatively stable, as the objects managed and the management protocol change to follow protocol migration to ISO standards. The Management Information Base (MIB) [MR88] defines the objects to be supported by TCP/IP-based network entities. An overview of the SMI and MIB are given in the next two sections, to lay the groundwork for the subsequent discussion of SNMP.

5.3   The Structure of Management Information (SMI)

The Structure of Management Information (SMI) specifies the types of objects that are used for network management, how objects of each of these types are to be represented when transmitted between managed and managing entities, and the overall naming system used to identify specific object instances.

A few of the basic object types defined in the current SMI are listed below, as examples:

- **IpAddress.** - A four-byte Internet Protocol address.

- **Counter.** - A non-negative integer that monotonically increases until it reaches some maximum value, after which it wraps around to zero. This might be used for packet counts.

- **Gauge.** - A non-negative integer that may increase or decrease. This might be used to count active sessions or indicate queue lengths.

- **TimeTicks.** - A non-negative integer that represents absolute time.

Aggregate object types are also included in the SMI, for representing lists and tables composed of the basic types.

The values of object instances of these types are to be encoded using ASN.1 when transmitted between managed and managing entities. ASN.1 is the ISO standard encoding

31

scheme mentioned above; it will also be used to encode object instance values in CMIP. Associated with each of the object types is a particular ASN.1 type (such as **INTEGER**).

A network management system refers to objects by name. The name of an object is called its *object identifier*. The name space of all object identifiers forms a tree, as illustrated in Figure 5.1. An object identifier is a sequence of integer node numbers, starting one level below the root (which is implied), and specifying which node at each successive level leads to the leaf representing an object instance. Each object identifier also has associated with it an ASCII name, with mnemonic strings representing the integer node numbers at each level, separated by delimiters. I will use "." as a delimiter in this paper.

The SMI also defines the top levels of the name tree. (Lower levels are defined by individual MIBs.) At the top level, ISO has been assigned a node named "iso" (node #1). Other organizations also have top-level nodes. The ISO has designated one of its subtrees for other national or international organizations, and calls it "org" (3). One of these organizations is the U.S. Department of Defense, which is assigned node "dod" (6). The Internet Activities Board (IAB) administers a subnode of the dod node, named "internet" (1). Thus, all object identifiers with which we are concerned will begin with "iso.org.dod.internet" (1.3.6.1). Four nodes are currently defined by the IAB to reside under this node:

- **directory (1)** - For future OSI directory use.

- **mgmt (2)** - For objects defined in IAB-approved documents (MIBs). The first published MIB revision has been assigned a subnode, number 1. Subsequent revisions will be given sequentially higher subnode numbers. The current MIB will always have subnode name "mib."

- **experimental (3)** - For Internet experiments.

- **enterprises (4)** - For objects defined unilaterally. A subnode of this node, "private" (1) is used for objects that are specific to a single vendor's equipment. Each vendor would have a subnode of the "private" node, under which it is free to define its own MIB.

Thus, all of the object names with which we will be dealing will reside in one of two subtrees, "iso.org.dod.internet.mgmt.mib" (1.3.6.1.2.1, under MIB revision 1), or "iso.org.dod.internet.enterprises.private" (1.3.6.1.4). The former subtree is for objects defined across vendor boundaries, and the latter for objects defined by specific vendors.
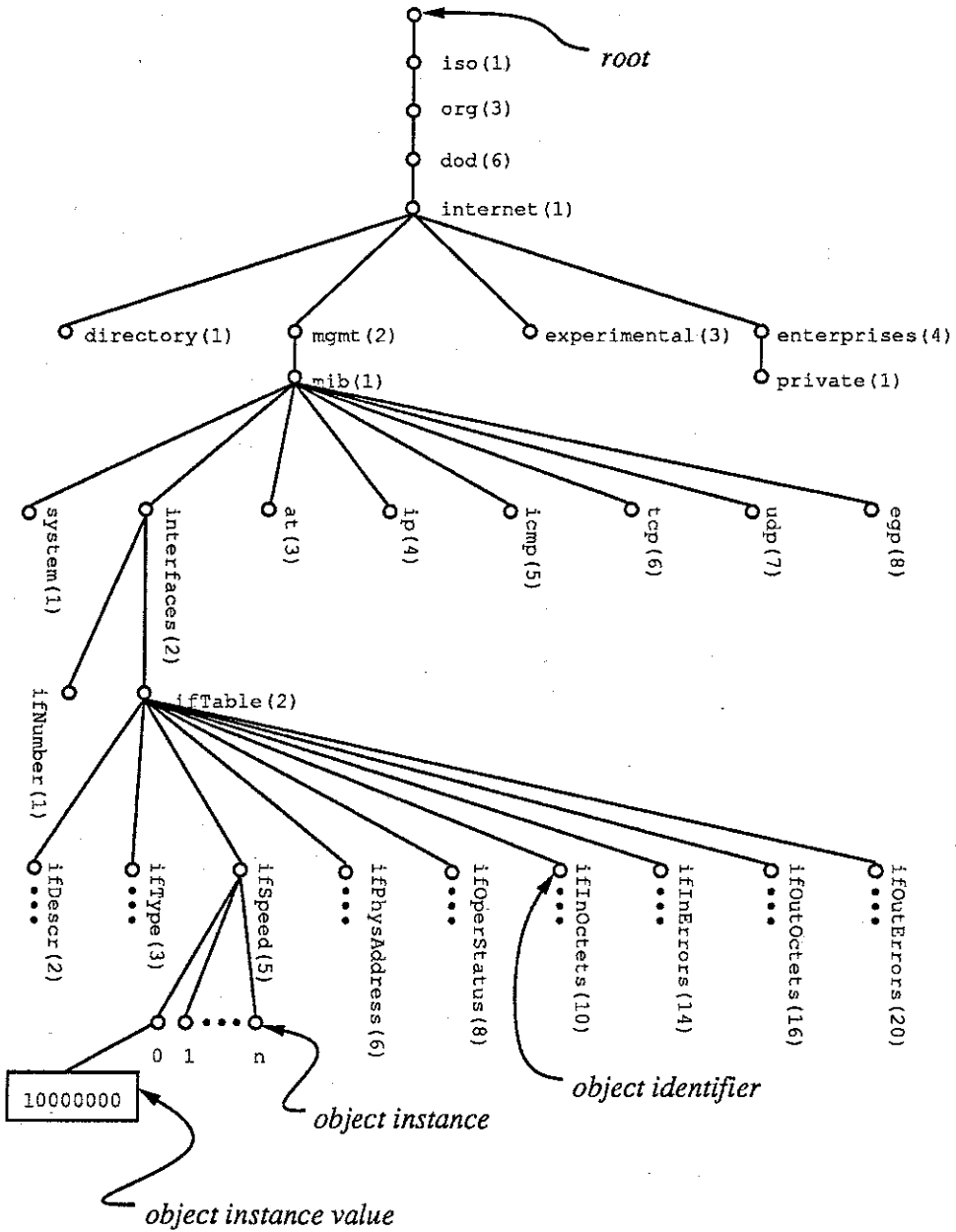
Figure 5.1: An Abbreviated View of the Current IAB MIB

5.4 Management Information Bases (MIBs)

A Management Information Base defines those objects that must be supported by a particular type of IN, along with the names and types (and resultant encodings) of these objects. Note that a particular instance of an object is identified by an (object name, instance) pair, so an object name may refer to more than one object. (How object instances are identified is a function of the management protocol. I will discuss how SNMP identifies object instances in the next section.) For example, a particular IN might have two interfaces. Assume an object of name X of type *counter* that is defined to be the number of input packets seen on a particular interface since boot time (or counter rollover). Assume also that the interfaces are numbered "1" and "2." The *object name* X is shared by two *object instances*, or *variables*, which we may denote by the ordered pairs (X, 1) and (X, 2). Each of these variables identifies the input packet counter on its respective interface.

I will not attempt to define all the object names in the current IAB MIB, but will describe the top-level subnodes of the MIB and a few objects so that the reader can get a feel for what types of objects might be used for network management, how these objects are organized, and what objects must be supported by which types of IN.

The current MIB (revision 1) is divided into eight subtrees. They are defined as follows:

- **system** (1) - System-wide variables, such as a textual description of the the system, and how long it has been since the system was last rebooted.

- **interfaces** (2) - Interface-specific objects, such as the operational status of an interface and non-protocol-specific packet counts.

- **at** (3) - Objects comprising a translation table mapping network addresses into physical addresses.

- **ip** (4) - Objects pertaining to the Internet Protocol (IP), such as counters indicating how many IP packets were discarded due to various errors or were delivered successfully, and the IP routing table.

- **icmp** (5) - Objects pertaining to the Internet Control Message Protocol (ICMP), such as counters for various types of ICMP packets.

- **tcp** (6) - Objects pertaining to the Transmission Control Protocol (TCP), such as objects that give information about current TCP connections.

- **udp** (7) - Objects pertaining to the User Datagram Protocol (UDP), such as UDP packet and error counters.

- **egp** (8) - Objects pertaining to the Exterior Gateway Protocol (EGP). (EGP is a routing table maintenance protocol by which gateways inform each other of routing changes.) These include objects that define a table of EGP-speaking neighbor gateways.

Which objects must be supported by a given IN type? The rule is that if a particular top-level subtree of the MIB is applicable to an IN, then it must include all the objects defined in that subtree of the MIB. The first two subtrees, "system" and "interfaces," are applicable to all IN types, so all objects under these subtrees must be supported by all INs. The "ip" subtree, on the other hand, would only be supported by gateways and routers that route IP packets. Note that although bridges and repeaters also forward IP packets, they do not distinguish between IP and other protocols, and therefore do not have the information necessary to support the objects in the "ip" subtree.

It will be instructive to look at some of the objects in one of the subtrees, to get an idea of how the MIB is organized. The "interfaces" subtree contains one object of which there is only one instance, "...interfaces.ifNumber." This is an integer indicating the number of interfaces the IN supports, and this determines how many instances there are of the per-interface objects. The per-interface object names all begin with "...interfaces.ifTable.ifEntry." A few of them are listed below, with brief descriptions.

- **ifDescr** - type Octet String - A string of text containing information about the interface, such as the name of the manufacturer, the product name, and the hardware version.

- **ifType** - type Integer - An integer indicating to which of a predefined set of interface types (e.g. *Ethernet*) an interface belongs. The set of possible types is defined in the MIB, and includes an "other" option for types not listed.

- **ifSpeed** - type Gauge - The interface's bandwidth in bits per second.

- **ifPhysAddress** - type Octet String - The physical address (e.g., Ethernet address) of the interface.

- **ifOperStatus** - type Integer - The operational status (up or down) of the interface.

35

- **ifInOctets** - type Counter - The number of bytes received on this interface.

- **ifInErrors** - type Counter - The number of times an error was detected while receiving a packet on this interface.

- **ifOutOctets** - type Counter - The number of bytes sent over this interface.

- **ifOutErrors** - type Counter - The number of times an error was detected while sending a packet over this interface.

Now that we have an idea of what the SMI and MIB are, it is time to take a look at the SNMP management protocol itself, which uses the SMI and MIB.

5.5   The Simple Network Management Protocol (SNMP)

As stated earlier, the Simple Network Management Protocol (SNMP) is the network management protocol intended to fill the need for network management on the Internet until such time as the Internet converts to the ISO CMIS/CMIP network management system and protocol. SNMP is designed to be what its name implies, *simple*. Particular emphasis is given to ensuring that as little logic as possible resides in the INs themselves. This is for two reasons. First, if SNMP is to be widely (and quickly) implemented by vendors, the demands it imposes on INs must be minimal. Secondly, a simple, basic object-manipulation capability in INs frees up those who write management tools to write tools as simple or as elaborate as they wish, without having the structure or complexity of their tools dictated by the management structure of the INs they are managing.

One consequence of the requirement that SNMP be simple is that SNMP only understands basic object types (such as *Integer* or *String*), and, unlike the SMI it uses, is not capable of handling aggregate data types such as sequences or tables. Each object must be retrieved or modified individually. This means, for example, that a row of a table can neither be read nor modified as a unit. While this contributes to IN simplicity, it causes functional difficulties that are difficult if not impossible to resolve at the tool level. For example, how would one add an entry (a whole row, that is) to a routing table? Also, would it not be more efficient to request a whole table at a time, or at least a whole row at a time, rather than requesting the object at each (row, column) element of a table individually, and would not the results have more integrity in the face of changing tables? It is this author's opinion that

36

SNMP has some serious deficiencies in this area. But CMIP does handle aggregate objects, and perhaps SNMP is good enough for the interim. In fact, its simplicity may well be what has made it possible for SNMP to be widely deployed soon enough to be an effective interim solution.

As I mentioned in the previous section, it is the job of the management protocol to specify how object instances are delineated. SNMP specifies an object instance via a simple extension of the SMI's object naming convention. An object instance is denoted by an object name concatenated with an instance qualifier. For example, if the object name is "...interfaces.ifTable.ifEntry.ifDescr," the object that contains the textual description of the second interface would be named "...interfaces.ifTable.ifEntry.ifDescr.2." Since all instance specifications are numeric (though they need not consist of a *single* integer, as in the example above), the underlying node numbers are obtained trivially from the instance name. Thus, the above-mentioned variable would be specified numerically as (1.3.6.1.2.1.2.2.1.2.2). The last node number, a "2," indicates the interface number. Objects names that may only have one associated instance are given instance number "0," so "...interfaces.ifNumber" would be specified numerically as (1.3.6.1.2.1.2.1.0).

SNMP provides for two methods of obtaining information from an IN. An IN can alert a management tool of a reboot or link status change (from up to down, or vice-versa) or other critical event asynchronously, using a *trap* message. Alternatively, a management tool can send a *get* message to an IN, indicating the variables for which values are requested, and the IN will reply with a *response* message containing both the variable names requested and the values of the associated variables. SNMP also gives a tool the capability of sending a *set* message to an IN to change the values of variables. The IN changes the value of the each variable and then returns a *response* message showing the new values of the changed variables. Of course replies may also include error indications if the requested action cannot be performed. I will not go into the detail of SNMP message contents or formats in this paper See [CFSD88] for details.

There are two types of SNMP *Get* messages. A simple *get* message specifies explicitly which object instances are being requested. If a requested instance does not exist, an error indicator is returned. A variation of the *get* message, the *getnext* message, specifies partial or complete object or instance names. The IN returns, for each of these names, the fully qualified name of the *next* object instance that it supports (in lexicographic order by the

numeric version of the object instance name), along with its value. This makes it easier to traverse tables, or get all the values of all the instances associated with an object name. It also eases the management of INs of different types or supporting different MIB revisions, which may not all support the same objects.

Each SNMP *get* or *set* message contains a *community ID* that indicates what set of capabilities the tool user claims. Each IN maintains a set of communities and an associated set of capabilities. A capability set indicates, for each object, whether a given community has *read-only*, *write-only*, or *read-write* permission with respect to that object (or none of the above). Write-only access deserves some explanation. While the current MIB does not list any such objects, the SMI allows the definition of objects that cause an action to be performed when set. For example, there might be a *reboot* object that, when set, reboots the IN. There would be no point in *reading* such an object. Such objects are not yet defined because the security mechanisms of SNMP are not yet very good, and providing such capabilities might be dangerous.

SNMP provides hooks for implementing authentication procedures to ensure that a tool user actually belongs to the claimed community. For example, the message formats contain an *authentication* section for authentication information. However, no particular authentication mechanism is a part of the SNMP standard, and most vendors have implemented null authentication procedures. This means that if a user knows a community name he can do whatever that community is allowed to do with a particular IN. Keep in mind that the community name is part of the message transmitted over the network, and it would be easy to obtain such community names by snooping somewhere along the network between the IN and the management tool. For this reason, most IN vendors allow their INs to be set up to specify that certain community names (those with dangerous powers) may only be used from hosts with specified IP addresses. This at least makes it harder to gain unauthorized access to an IN.

Current implementations of SNMP are built on top of UDP, though other transport protocols may be used. One consequence of building on top of UDP is that message delivery is not guaranteed. This, once again, simplifies IN management code, as message acknowledgements and retries do not have to be implemented. (In fact, an IN may ignore a request it considers unimportant, if it is too busy to answer it.) It does, however, complicate tool writing, as timeouts and retries must be implemented there. Each SNMP message contains

a unique ID (specified by the initiating process, which is the tool except in the case of traps), and matching responses contain this same ID, so a tool can send out multiple requests to the same IN and can sort out the responses even if some are missing or responses do not arrive in the same order as the corresponding requests were sent.

It should be noted that, while SNMP's primary purpose is to manage INs, there is nothing about SNMP that restricts its use to *interconnection* nodes. Any network node has at least one network interface. Computers attached to a network usually also support some higher-level protocols like IP, TCP, and UDP. So do many terminal servers, printers, and other special purpose nodes. Thus, both the MIB and SNMP are applicable to any network node to some degree. SNMP, then, could potentially be used to manage all the nodes on a network.

5.6 Conclusion

In this chapter I have discussed the various current and upcoming network management standards for the Internet. SNMP is the method of choice for this project because it is currently available, and newer INs in the NCDN come with SNMP support.

Thus far in this paper I have discussed various introductory topics. I have described the problem that needs solving, i.e., the provision of common access mechanisms to dissimilar network interconnection nodes. I have also given an overview of network management tools and their requirements, and have taken a look at the target network and how it affects the design. Finally, I have reviewed the standards environment within which we must work.

The remainder of this paper discusses how I have gone about solving the stated problem. In the next chapter, I give an overview of the solution. Following that, are detailed descriptions of the various components of this solution. Finally, I will evaluate the solution by comparing it to the stated requirements, and will give suggestions for improvements and future developments.

Chapter VI
Design Overview

In this chapter I will give an overview of my solution to the problem of providing common access mechanisms to dissimilar network interconnection nodes. In the process, I will give particular attention to the identification of, and justification of, the major design decisions affecting the result.

6.1 Build, Buy, or Wait?

The first question that needs to be addressed is whether to build our own management system, buy (or acquire) what someone else has built, or wait for someone else to do it for us. We did some of each. Choosing an established network management standard enabled us to purchase or acquire all our network management tools and to make use of new ones as they become available. During the time that the work described in this paper was underway, upgrades for SNMP support became available for some of the IN types already in use. Unfortunately, most of these upgrades involved replacing hardware components (e.g., adding memory or upgrading CPU speed), a much more expensive solution than the one described in this paper. Other IN types already in use will never support a network management standard. We needed access mechanisms for those IN types to make our network management system as complete as possible. Thus, we purchase and acquire where possible and economically feasible, build to fill in the gaps, and wait for native management support for some of the devices to which we have temporarily gained access with our own system.

6.2 Protocol, Operating System, and Language Choices

I chose a transport protocol, operating system, and programming language for implementing my software based on the single criterion of facilitating as widespread use of it as possible

40

within the NCDN. I chose TCP/IP as the transport protocol for inter-process communication because of its ubiquitous nature among the different operating systems in the network. TCP/IP is so widely available that it is unlikely that any other transport mechanism will be added to the system. The choice of operating system is less obvious, but still rather easy. UNIX predominates, especially among those hosts likely to be running network management tools. This choice is also less critical than the others, as my choice of language makes it easy to port the software to other operating systems. My language of choice is C. Because of its widespread availability among operating systems, this greatly aids portability over other choices. A close second would have been something like C++, whose object-oriented programming support would have made implementation easier, but whose relative lack of availability would have hindered portability somewhat.

6.3   Defining the Access Mechanisms

Specifically what access mechanisms are we attempting to provide to all INs, and just how common are they? In order to answer these questions, we must look back to the requirements we identified in chapter 4. Briefly, the two relevant requirements are:

1. A single protocol for monitoring and controlling INs, having the characteristic that if we want to perform an identical function on two different INs (regardless of type, as long as the function is applicable to both), we use identical methods on both.

2. Some back-door method of accessing INs interactively, wherever the manufacturer provides such a capability.

Refer to Figure 6.1 during the following discussion.

   The first requirement can be met by somehow providing a mapping between the SNMP protocol and various proprietary access mechanisms provided by the vendors, whenever an IN doesn't support SNMP directly. I will call a process performing this function an *SNMP Proxy Agent*, because of its role of providing SNMP access by proxy. (My justification for choosing SNMP as a management protocol was discussed in the previous chapter.)

   The second requirement can be met by a process that maps proprietary interactive access mechanisms (usually the same mechanisms mapped into SNMP by the proxy agent) into TCP, so that all INs can be accessed interactively via the TCP protocol. Some commonality is lost here. Most vendors who supply interactive access to their INs over TCP
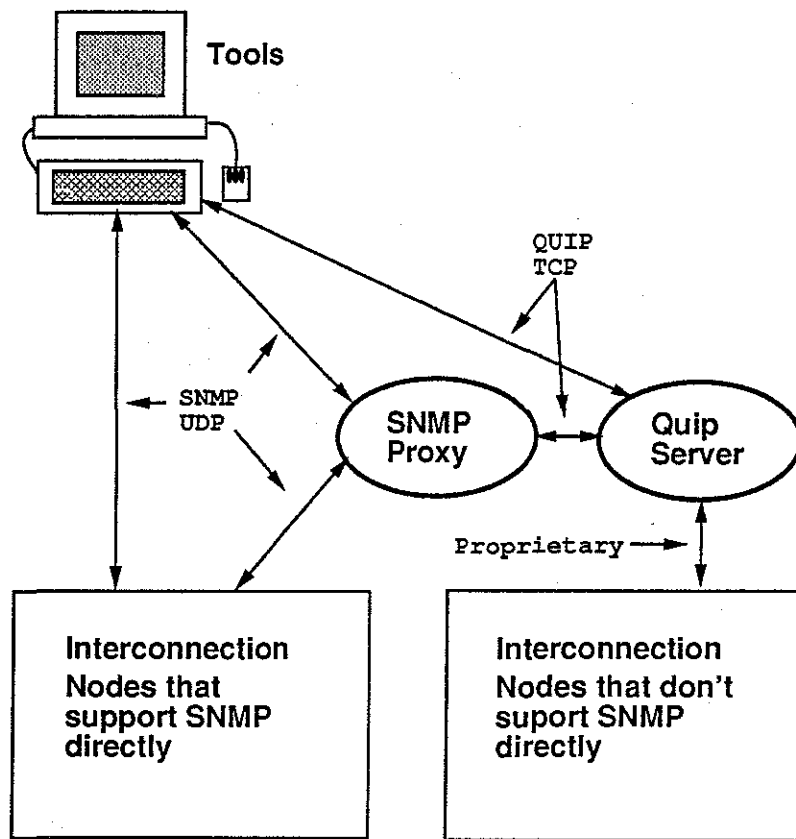
Figure 6.1: SNMP and QUIP Overview

do so via **telnet**, a TCP/IP virtual terminal service. Implementing full **telnet** service by proxy for INs not supporting **telnet** would have been more work than it was worth, since we hope to eventually make interactive access obsolete by providing the same capabilities via SNMP. Instead, I implemented a simpler mapping process for mapping between the various proprietary interactive mechanisms and TCP. This mapping is restricted to simple query/response interactions, and has thus been dubbed "Query Using Internet Protocol" (QUIP). Besides the **telnet**/QUIP difference, the interactive command language itself will vary among different vendors and IN types, but there is nothing we can do about that.

QUIP is implemented by two processes. One is a server named **quipd** (for *QUIP Daemon*). This process repeatedly accepts a query encoded in a *QUIP message* over a TCP connection, retrieves the requested information from the indicated IN, and returns the result over the TCP connection, in another QUIP message. The other process is a user interface, simply called **quip**. It reads a command from its standard input (frequently the keyboard), encodes it in a QUIP message, sends this message to the appropriate **quipd** server, waits for the response, and writes the result on its standard output (frequently a terminal screen or window).

6.4   Query Using Internet Protocol (QUIP)

There may of necessity, or for redundancy, be multiple **quipd** servers. For example, if a particular IN type can be accessed only via a serial line, and there are INs of this type scattered throughout the network with their serial lines attached to various host computers, there will need to be a **quipd** server running on each of these host computers to gain access to each of the INs. Even in the situation where INs of a given type can be accessed from any host in the network (using some protocol other than IP), redundant **quipd** servers may be desirable (e.g., one in each administrative domain) so that a network failure does not preclude access to INs on the near side of the failure.

Interactive **quip** programs may, of course, be run from anywhere on the network, and there can be as many **quip** programs running concurrently as operating system limits (such as the maximum number of TCP connections to a given QUIP server) allow. All events are interleaved within the **quipd** server, so that a delay in getting a response from one IN will not slow down a **quip** process requesting information from a different IN.

43

Quip needn't be run interactively. Since it uses the standard UNIX input/output streams, scripts can be written to run **quip** automatically. The scripts can read commands from a file or generate them itself, and the output can be stored in a file or piped to another process (e.g., for filtering or formatting of the results).

TCP (a connection oriented protocol), rather than UDP (a connectionless protocol), is used as the transport protocol between the **quip** process and the **quipd** server. I chose the connection oriented protocol because most of the proprietary protocols **quipd** uses to communicate with particular INs are connection oriented. It would be quite difficult to maintain the integrity of the connection between **quipd** and an IN, if the connection between **quip** and **quipd** were unreliable. Lost requests cannot simply be repeated, as the lost request may (or may not) have already changed the state of the IN connection. In other words, the **quip/quipd** connection would have to be made reliable, and we might just as well have started with a reliable transport service.

QUIP's usefulness is not limited to accessing INs. It is a simple way to make any command interface available via TCP. Often there are command interfaces to systems that are accessible only via serial-lines (such as an on-line library catalog system). One might want to be able to access this system from any host computer on the network. If the interface fits QUIP's query/response paradigm, a driver can easily be written for the QUIP server to solve this problem. I have used QUIP to provide an interface to the Intecom IBX telephone switch database. (We also access the LANmark bridges through this same interface.)

6.5   The SNMP Proxy Agent

As stated earlier, an SNMP Proxy Agent maps SNMP requests (normally received via a UDP/IP from a management tool) into the proprietary protocols, command languages, and physical access media provided by an IN vendor, obtains a response from the IN, and returns the result in an SNMP response message. My SNMP proxy agent is called **snmp_proxyd**, for *SNMP Proxy Daemon*. Much of the work that the **snmp_proxyd** needs to do is already done by the **quipd** process. Thus, the task of the **snmp_proxyd** can be reduced to that of performing the mapping between the SNMP protocol on one side and the proprietary command languages on the other. Like the interactive **quip** program, **snmp_proxyd** uses the QUIP protocol to communicate with INs via the **quipd** process. Thus, as far as the

snmp_proxyd process is concerned, *all* INs communicate via TCP, and it only needs to handle the varying command languages themselves. This has two major advantages. First, it simplifies the proxy agent by eliminating duplication between it and the QUIP server. Secondly, it makes it unnecessary to run a proxy server wherever direct IN access is needed (e.g., via serial line). The primary disadvantage of this approach is that it slows responses to SNMP queries, as the path from management tool to IN involves hops through two servers, or three network hops.

The proxy agent also serves as a caching agent. Each object instance value retrieved is given a *lifetime* attribute. If the same management tool requests the same object instance value twice in a row, the *lifetime* attribute is ignored, and the object instance's value is retrieved again, so that the same value is not returned twice without consulting the IN. On the other hand, if two different management tools request the value of the same object instance within the lifetime of that value, the proxy agent simply returns the cached value to the second requester. Thus, if many management tools are asking for the same information at nearly the same time, the INs being monitored will not be queried more often than if only one management tool is requesting the information. Traffic between the management tools and the proxy agent may be high, but not between the proxy agent and the INs.

Parsing IN responses to commands is, in general, a nontrivial task. The command languages provided by vendors are typically intended to be human interfaces, the results being prettily formatted so that they are easy to read by humans operators. It can be difficult, however, to write a computer program to parse such human-oriented displays and turn them back into a format easily manipulated by a computer, so that they can be mapped into SNMP response messages. Performing this rather tedious and error prone task for every type and model of IN (and every version of its command language) can make adding a new IN type rather unpleasant and time consuming. To ease this task, I have implemented a parser that allows a programmer to specify, using a simple grammar, what information in a human-oriented display is of interest. This specification is then passed to a library routine, along with the command response from an IN, and it returns a C structure containing the results in a form that is easily manipulated by a C program. This parser is called the Human Interface Parser (HIP).

Note that there is no particular reason that the proxy agent must use QUIP to communicate with all the INs it serves. Any protocol an IN supports that is written on top of TCP

or UDP would fit nicely into the design. Two obvious possibilities are **telnet** and SNMP itself. In the case of **telnet**, communicating with an IN directly would save the extra hop through **quipd**. This would probably be worth the effort, even though a **telnet** driver for the proxy agent would be a bit more difficult to write than a QUIP driver. An SNMP driver would be particularly useful, as it would enable a proxy agent to serve as a caching agent for *all* INs, not just those that don't support SNMP directly.

6.6   General Design Considerations

At this point it is helpful to identify a few general design considerations that are applicable to all of the above-mentioned components of the solution. Briefly, these considerations are:

1. Design for Speed. In order to meet our rather tight SNMP turnaround time requirements in the face of the obligatory multiple network hops and unfortunate conversions of information from computer-readable to human-readable and back again, we must perform our computational tasks as quickly as possible. We must also perform interleaving of actions wherever possible, to keep one request from delaying another unrelated request.

2. Design for Simple Drivers. **Quipd**, **quip**, and **snmp_proxyd** all have a certain amount of code that differs among the various IN types. This type-specific code for a particular IN type is called a *driver*. Whenever a new IN type is added to the system, a new set of drivers must be written. Our components must be designed in such a manner that drivers are as small and simple as possible.

3. Design for Portability. We require that our software be widely distributable. The proxy servers must run in each administrative domain. The QUIP server must run wherever access to an IN is available. The interactive QUIP program and other management tools must run nearly everywhere. The software must not make any more operating system or hardware specific assumptions than are strictly necessary.

I will keep these considerations in mind as I discuss each of the components in detail in the next four chapters.

6.7 Summary

In this chapter I have identified and justified the major design decisions made during the course of this project. I have also identified four major components of my design:

1. The QUIP server, **quipd**, which brings access to all INs up to the TCP level,

2. The QUIP interactive interface, **quip**, which provides an interactive interface over TCP to INs not providing such an interface,

3. The SNMP Proxy Agent, **snmp_proxyd**, which provides a mapping between SNMP and proprietary command languages, and

4. The Human Interface Parser (HIP), used by **snmp_proxyd** drivers to convert human-oriented displays into computer manipulable form.

In the following four chapters I will consider the design and implementation of each of these components in greater detail.

Chapter VII

The QUIP Server

In this chapter we will take a closer look at the design and implementation of the QUIP server, **quipd**. First I will describe QUIP messages and the QUIP library, a set of C routines used for sending these messages between the QUIP server and its clients, **quip** and **snmp_proxyd**. Then I will discuss the design and implementation of the main body of the QUIP server. Finally, I will describe the QUIP driver interface, and drivers for specific IN types.

7.1   QUIP Messages and the QUIP Library

QUIP messages are sent between Berkeley UNIX sockets using a *stream* protocol (over TCP). Since stream protocols do not have message boundaries, QUIP messages must contain some indication of how long each message is. For that reason, QUIP messages are sent in two portions. First a header is sent, containing, among other things, the length of the text of the message to be sent. The text itself is sent immediately after the header. See Figure 7.1 for a diagram of typical quip query and response messages.

The QUIP message header is a C structure containing the following members:

- **address** - A 32-byte field containing a null-terminated ASCII string, left-justified in the field, that identifies a particular IN. This address must be unique within a particular IN type. The Ethernet address of the IN is often used here.

- **id** - A 4-byte unique identifier for this message. On a query, this should be a unique bit pattern not used in any other message in this session (or at least not recently). On the response to a query, the same id is used as was in the query.

- **options** - A 32-bit flag word for setting various options. Currently the only option is a flag indicating whether the connection between **quipd** and the IN should be

Query	
address	0800200D13D\0
id	658001
options	(none)
nodetype	NT_IB
timeout	5
errors	(none)
datalen	10
data	show date\0

Response	
address	0800200D13D\0
id	658001
options	(none)
nodetype	NT_IB
timeout	5
errors	(none)
datalen	25
data	Sat Jul 28 05:34:21 1990\0

Figure 7.1: A Sample QUIP Query and Response

connection-oriented (i.e., left open between messages) or not (closed or reset to an initial state between queries).

- **nodetype** - A 4-byte IN type indicator. The current types are:

  - **NT_IB** - 3Com IB Bridge

  - **NT_IBX** - Intecom IBX Switch (and LANmark Bridges)

  - **NT_TRANSLAN** - VitaLink TransLAN Bridge

  These node types only need to distinguish between IN types well enough to allow the QUIP server to know what particular driver it needs to invoke to communicate with the IN. The NT_IB type, for instance, says to use the IB driver, which can be used for all types of IB bridges. The IB driver can communicate with any IB using an XNS protocol, even though the command languages used by the various IB models and revisions vary a great deal.

- **timeout** - A 4-byte integer specifying the number of seconds to wait for a response from an IN. If this timeout is exceeded, the QUIP server is to discard the request and report a timeout error in a response message.

- **errors** - A 32-bit flag word in which errors are flagged in a response message. All bits are zero if a query succeeded. The error flags are as follows:

  - **ER_BADQUERY** - Improperly formulated query.

  - **ER_UNKNOWN** - Unknown node type or address.

  - **ER_NORESPONSE** - No response from IN. Set when the timeout expires before receiving any response from the target IN.

  - **ER_INCOMPLETE** - Incomplete response from IN. Set when part of an IN response is lost or only a partial response is received before timeout expiration.

  - **ER_CONNECT** - Cannot connect to IN.

  - **ER_FAIL** - Query failed for some other reason (e.g., the QUIP server ran out of resources).

- **datalen** - A 4-byte integer specifying length of the message to follow, in bytes, including the terminating null byte.

All integers and flag words in the message header (**id**, **options**, **nodetype**, **timeout**, **errors**, and **datalen**) must be transmitted in network byte order.

The message portion itself is always a null-terminated ASCII string. All messages must have a length of at least one, even response messages with an error flag but no relevant text portion. There is normally a one-to-one matching between a *query* message to the QUIP server (containing a command), and a corresponding *response* message, containing either an error flag and an empty (null byte only) text string, or no error flag and the response to the command in the text portion.

The QUIP library contains three subroutines that facilitate communication between the QUIP server and its clients. Briefly, they are:

- **send_quip**(*sock, msg, header*) - Send message *msg* with the header specified by the *header* parameter, over socket *sock*.

- **send_quip_error**(*sock, error, header*) - Send an error response to the query specified by the *header* parameter, over socket *sock*. The *error* parameter specifies which error(s) to report.

- **recv_quip**(*sock, msg, header*) - Read a QUIP message from socket *sock*. The *header* parameter specifies where to put the header of the received message, and the *msg* parameter specifies where to put the message itself.

The *sock* parameters of all routines are integer socket file descriptors. These three library routines and their other parameters are described in more detail in the following paragraphs.

The **send_quip** routine is used to send both queries and responses. The *msg* parameter is a pointer to the null-terminated string to be sent. The *header* parameter is a pointer to an instance of the QUIP header structure defined above. All header fields except the **datalen** field must be filled in before **send_quip** is called. (In the case where send_quip is being used to reply to a query, a pointer to the header received with the query is passed to send_quip, as it already has all the fields except the **datalen** field filled in properly). If the **id** field is zero, a unique integer will be filled in automatically. This is the easiest way to get a unique id for sending a query. The **send_quip** subroutine automatically calculates the value of the **datalen** field, from the *msg* parameter length. After these changes are made to the structure pointed to by the *header* parameter, a copy of that structure is made, and all integers and flag words in this copy are converted to network byte order. Then this copy of

51

the header and the message are sent over the *sock* socket. If an error occurs while sending the message, **send_quip** returns a -1, otherwise it returns zero.

The **send_quip_error** routine is a shorthand way of sending an error response to a query. The *error* parameter is a flag word indicating which errors are to be reported. The *header* parameter is a pointer to the header structure from **recv_quip** associated with the query to which we are responding. The **send_quip_error** routine fills in the **error** field and calls **send_quip** with an empty string as its *msg* parameter. If errors occur while sending the message, **send_quip_error** returns -1, otherwise it returns zero.

The **recv_quip** routine is used to read both query and response messages. Its *msg* parameter is a pointer to a variable of type **char *** where a pointer to the received message string is to be placed. The *header* parameter is a pointer to the QUIP structure where the received message header is to be placed. All integers and flag words in the header are converted to host byte order before **recv_quip** returns. If the stream connection associated with *sock* is closed on the other end (e.g., the client process has terminated), the character pointer pointed to by the *msg* parameter is set to NULL. If an error occurs while reading the message, **recv_quip** returns -1, otherwise it returns zero.

7.2 The Main Body of the QUIP Server

After initialization (which includes opening a socket on which **quipd** listens for TCP connections), the QUIP server goes into an event loop where it waits for an event, processes pending events, does a bit of housekeeping, and then waits for the next event. There are four types of events that may occur:

1. Some timeout expires. This can happen either if the timeout specified in a query has expired without a response having been received from the target IN, or if some housekeeping chore needs to be done at this time.

2. A new connection with the QUIP server (from **quip** or **snmp_proxyd**) is requested. If so, the connection is accepted and initialized.

3. A QUIP *query* message has arrived for servicing. If so, the query is read and forwarded to the appropriate IN (or queued for forwarding later, if the IN already has a query pending).

4. Part of a response (or a complete response) has arrived from an IN. If so, the response fragment is read and appended to the fragments that arrived previously (if any). If the response is not yet complete (and no errors have been detected), no further action is taken. Otherwise, a response (or error response) is sent to the querying client process. Then, if there is another pending query for the same IN, it is initiated.

The above event types are listed in increasing order of priority. That is, all pending *IN input* arrivals are handled before all pending *quip message* arrivals, which are handled before all pending *connection requests*, which are in turn handled before all *timeouts expirations*. The purpose of this priority scheme is twofold. First, it ensures that queries already in progress are handled as soon as possible. Secondly (and more importantly) it keeps the QUIP server from getting bogged down accepting connection requests and new queries, should the server not be able to keep up with requests (e.g., if some client program goes haywire, sending queries in rapid-fire succession). This makes sure *some* queries are handled, at the expense of delaying (and possibly rejecting) new queries and connections.

For the sake of clarity, the above description of events and associated actions is oversimplified. Now let us take a more accurate and detailed look at each event type and how it is handled.

7.2.1 Timeouts and Housekeeping Chores

Timeouts are used to signal two different types of events in the QUIP server. First, each query has associated with it a timeout that specifies how long the server should wait for a response from an IN before it gives up and terminates the query. Secondly, there are certain per-IN resources (associated with open connections to INs) that it is most efficient to set up once and leave set up for a while, since more queries are expected to arrive soon for this particular IN. But there are not enough of these resources available for every IN in the network. Thus, after a certain amount of time passes with no queries for a particular IN, its resources are returned to the pool for use in handling requests to other INs (and the corresponding IN connection is closed). The **close** routine of the driver is used to close the connection itself.

Just before the server determines that it has no more to do at present and is getting ready to put itself to sleep until the next event occurs, it checks to see when the next timeout should occur. It then sets an alarm to make sure that it will wake up to handle the next

timeout promptly, in case no other events arrive to wake it up before that time. If an event or a set of events does occur before the timeout expires, these events are handled first, and then if any timeouts have expired by that time, they are handled before the server prepares to sleep again.

7.2.2 Connection Requests

Each time a client process wishes to begin communicating with the QUIP server, the client process requests a TCP connection to the server. When such a request is received, the server simply accepts the request (if it has enough resources to allow it to do so) and initializes data structures associated with the connection.

7.2.3 QUIP Query Arrivals

Whenever a QUIP query message arrives it is read in and timestamped. Then a check is made to see if a connection to the target IN is already open that can be used to answer the query. If the **QO_CONNECTION** flag is set in the message, a dedicated connection is required between the server and the IN for this client. In this case, we check to see if this connection has already been established for this client and IN. If so, this pre-established, dedicated connection is used to answer the new query. If the **QO_CONNECTION** flag is not set in the incoming message, we look instead to see if a connection to the target IN has been established for non-connection-oriented queries. In either case, if there is no appropriate connection already available, one is set established. This involves calling the **open** routine of the appropriate driver, determined by the **nodetype** field of the message.

Once a file descriptor and associated data structures have been found or established to answer the query, the query is put into a queue associated with the appropriate file descriptor. If there was not already a query in this queue, the first portion of the query (typically one command) is sent to the IN via the file descriptor. (A particular query may involve multiple IN commands, in which case the commands are sent to the IN one at a time, and the IN's responses to all of the commands are concatenated together and returned as if they were the response to a single command.) This command is sent using the **send** routine of the appropriate driver. At this point, we are finished handling the query arrival event.

A special case of the query arrival event occurs when a client closes the connection to

54

the QUIP server (or the operating system drops the connection due to some error). When this happens, the **recv_quip** routine sets the *msg* pointer to NULL to notify the server that the connection has been dropped. At this point, a search must be made through all open IN connections to ensure that any queries queued for this client are dequeued and discarded, and that any connection oriented IN sessions for this client are closed. The **close** routine of the driver is used to close such a connection. It may be that dequeuing a query will bring another query to the top of the queue. When that happens, the first command of the next query in the queue is sent to the IN at this time, using the driver's **send** routine.

7.2.4   IN Input Arrivals

When input arrives on a file descriptor associated with an IN connection, it is read by the QUIP server and then handed off to the **receive** routine of the appropriate driver. The block of data that has been received from the IN may be a simple ASCII response (as in the case where the IN is accessed via a serial line), or it may be a packet containing the response, perhaps in some encoded form. The **receive** routine of the driver extracts the response and does any post-processing that is necessary, returning the result. It also checks to see whether or not the response to the current command is complete (by checking some end-of-message indicator in the packet, or looking for a prompt), and whether or not there was some kind of error (such as a lost packet, determined by monitoring packet sequence numbers). It returns such status information in a flag word parameter.

If an error flag is returned, the remainder of this query is dequeued and an error message is sent back to the appropriate client. If the response to the current command is complete, the current query is checked to see if there are more commands left in the query. If so, the next one is sent on to the IN via the **send** routine of the driver. Otherwise, the query is dequeued and the complete response is sent to the client. In any case, if the current query is dequeued and there is another one in the queue, the first command of this next query is sent to the IN via the driver's **send** routine.

Normally, all IN connections will be closed by the QUIP server. If for some reason an IN connection is closed by the IN itself or (in the case of some error) by the operating system, the attempt to read data from the associated file descriptor will fail, and an indication will be given that the connection has been closed. In this case, error messages are sent to the client processes that initiated the queries associated with the dropped connection, the associated

55

query queue entries are removed, and the file descriptor and associated data structures are deallocated.

7.3 The QUIP Daemon Driver Interface

In the previous section, I made frequent references to the driver routines that handle low-level communication with particular IN types. There is a set of driver routines for each major type of IN supported. Each set has four routines, namely **open**, **send**, **receive**, and **close**. Let us look at the parameters and functions of each of these routines:

- **open**(*address*, *connection*) - Open a connection with the IN specified by the *address* parameter. The *address* parameter is a null-terminated character string, as received in the query message. The *connection* parameter is a boolean indicating whether the connection is to be dedicated to one client session (with the state of the IN connection maintained between commands) or whether it is sharable by various clients, the connection being reset to some initial state between queries. If successful, the **open** routine returns a the file descriptor associated with the connection it has established. Otherwise it returns -1.

- **send**(*fd*, *cmd*, *request*) - Send the command *cmd* to the IN connected to the file descriptor *fd*. The *cmd* parameter is a null-terminated character string representing a single IN command. The *request* parameter is a pointer to a structure containing all the information pertaining to the active query. Some elements of this structure must be used in the **send** and **receive** routines, e.g., to keep track of packet sequence numbers. The *request* structure is described below[1]. The **send** routine returns zero if successful, -1 otherwise.

- **receive**(*buffer*, *buflen*, *response*, *flags*, *request*) - Perform any post-processing on the data read from the IN, which is referenced by the character pointer *buffer*. The number of characters read is indicated by the integer *buflen* parameter. The *request* parameter points to the data structure associated with the current query. The resulting response is to be placed in a null-terminated string, and a pointer to this string is to be copied

---

[1]It is probably bad practice to make this whole structure available to the driver. Some information hiding would be appropriate here.

into the **char** * variable pointed to by the *response* parameter. The *flags* parameter is a pointer to a flag word where status indications are to be placed. Possible status indications are:

- **RC_COMPLETE** - The end of the response to the current command has been read. This can be determined by looking for some sort of end-of-message indicator in a packet, or by looking for a prompt from the IN.

- **RC_INCOMPLETE** - Some data has been lost, and the response cannot be completed. This might be flagged when a missing packet sequence number is detected.

- **RC_FAIL** - Some other error has been detected (such as a protocol error in a response packet).

The flag word is initialized to zero before the **receive** routine is called. If the flag word is zero upon the return of the **receive** routine, the response data has been processed but the end of the response has not yet been read. The **receive** routine returns zero if successful, -1 otherwise. Note that "successful" in this case means that no *system* error has occurred. Returning **RC_INCOMPLETE** or **RC_FAIL** in the flag word is not considered a failure of this routine, so a zero would be returned from the **receive** routine in these cases[2].

- **close**(*fd*) - Close the connection indicated by file descriptor *fd*. This routine usually just calls the UNIX **close** system call on the file descriptor, but it may be necessary in some cases (e.g., when accessing an IN via a serial line) to restore the connection to some initial state first.

The *request* structure contains all the information that the QUIP server maintains about a given query. It may be used by the **send** and **receive** driver routines. The members of this structure that may be useful to these driver routines are described briefly below. At present, these members are used only by driver routines that are managing datagram oriented IN connections.

- **id** - A 4-byte integer unique to this query, used to match response packets with the corresponding command packets.

---

[2]Also note the lack of symmetry between **send**, which actually writes to the file descriptor, and **receive**, which does not read directly from the file descriptor. This lack of symmetry should be eliminated.

- **cmdno** - The serial number of the current command in the sequence of commands associated with the current query. This is also used to match response packets with the corresponding command packets.

- **seq** - The sequence number expected in the next response packet. This is used to ensure that no packets of a response are lost.

7.4 IN Drivers

Now that I have discussed the driver interface in general, I will briefly review the three IN drivers that have been written to date. These are:

- **IB** - The 3Com IB driver.

- **TransLAN** - The VitaLink TransLAN driver.

- **IBX** - The Intecom IBX Man-Machine Interface (MMI) driver.

7.4.1 The IB Driver

Access to the 3Com IB bridges is available via two methods. One can either connect to an asynchronous serial port on the bridge, or one can use a datagram oriented protocol through the regular data interfaces of the bridge (in-band). Since the in-band approach makes it easier to get to all the bridges on the network, I wrote a driver for this access method. The datagram protocol is built on top of the XNS Internetwork Datagram Protocol (IDP), and is intended to be used by management programs running on a special management station provided by the vendor. I wrote a driver that uses the XNS IDP protocol on a VAX running 4.3bsd UNIX (the only UNIX version available to me that supports XNS) to communicate with the bridges.

Since no state information about queries is kept in the IB (queries are handled in their entirety each time a datagram is received), all communication with the IBs is non-connection-oriented. Theoretically, when one is using a datagram protocol, one UNIX socket should be able to be used to communicate with *all* IB bridges. In fact, this is not the case for sockets using the 4.3bsd XNS protocol implementation. Once a socket is used for communicating with one IB, it cannot be used to communicate with another IB. So each time information is needed from an IB for which we do not have a socket already set up, the IB **open**

routine is called. This routine obtains an XNS datagram socket, initializes several socket options, and associates the socket with the address of the particular IB with which we wish to communicate. The socket options initialized are the IDP packet header to be sent with every outgoing packet and the number of receive buffers to be allocated by the operating system for response packets from this particular IB (the default is too small, causing packets to be lost). Note that the **open** routine does not send any packets to the IB itself. It only sets up a socket for later use.

The **send** routine puts a unique integer, a concatenation of the **id** and **cmdno** fields of the *request* structure, into a field of the packet header intended for just such an identifying integer. It then sets up several other packet header fields, copies the command into the appropriate place in the packet, and sends the packet to the IB.

The **receive** routine checks packet header fields to ensure that the packet received corresponds to the current command, and that the sequence number in the received packet is correct. (Though each command will fit in one packet, a response may be long enough to fill many packets.) If a packet is lost, the **RC_INCOMPLETE** flag is set in the flag parameter word, and the **receive** routine returns. Otherwise, the response data is extracted from the packet. This response data contains an encoded ASCII string. The encoding compacts spaces by representing a string of spaces as a byte with the high-order bit on, and the low-order seven bits representing the number of spaces. It also represents newlines with a null byte. The **receive** routine converts this response into its full ASCII representation, and returns this null-terminated string via the *response* parameter. Finally, a bit in the packet header is checked to see if this is the last packet of the response corresponding to the current command. If so, the **RC_COMPLETE** flag is set in the flag word parameter.

The **close** routine is trivial. It simply closes the file descriptor passed as argument *fd*.

7.4.2 The TransLAN Driver

Like the IB, the VitaLink TransLAN provides access via a direct serial port or in-band via a data interface. I wrote a driver for the in-band interface. The protocol used is the XNS Sequential Packet Protocol (SPP). This is a reliable, connection oriented protocol.

The **open** routine sets up an XNS SPP socket. It then connects to the bridge using the UNIX **connect** system call. This actually causes the transfer of packets between the bridge and the UNIX operating system to establish a connection. Then several initial packets are

exchanged with the bridge (from the driver), completing the connection setup.

The **send** routine sends the characters of the command, one at a time (one per packet) to the bridge. It should be possible to send the whole command at once, but I could not discover how to make that work. Perhaps this feature is not supported, as the bridge expects to be communicating with a human typing at a keyboard, and the bridge echos each character as it is typed.

The **receive** routine examines the string read from the socket, looking for a prompt at the end. If one of several possible prompts is identified, the **RC_COMPLETE** flag is set. Then the routine returns. No packets have to be dealt with, as the SPP protocol returns only the ASCII response when a **read** operation is performed on the socket. No conversion of the response is necessary in this case.

The **close** routine is trivial, closing the file descriptor.

7.4.3   The IBX Driver

The IBX driver is a bit different from the other two, in two ways. First, we are accessing a telephone switch database interface (called the Man-Machine Interface, or MMI), through which access to a number of bridges is obtained. Secondly, the only available access is via one of several asynchronous serial ports.

We have connected several serial MMI ports from the IBX switch to serial ports on a UNIX host. One of these is used for non-connection-oriented requests (e.g., SNMP requests), and the others are used for connection oriented sessions (e.g., using the **quip** program for interactive access). The **open** routine opens one of these ports, depending on the value of the *connection* parameter. The address parameter is ignored, as the identifying address of a particular bridge (in this case, a telephone switch port number) must be sent along with commands to the MMI interface to indicate which bridge is to be accessed. After the serial port is opened, port options are set (e.g., parity) and the fact that this port is in use is recorded. Then the file descriptor is returned.

The **send** routine translates the command to upper case (as required by the MMI interface) and then sends the characters of the command to the serial port, one character at a time, with a small delay between characters. This delay is required because the MMI ports are not full duplex and tend to lose characters if they are sent too fast.

The **receive** routine removes null bytes from the characters read from the serial port

60

(which are sent as padding by the MMI interface), and checks for a prompt. If a prompt is found, the **RC_COMPLETE** flag is set. Then the routine returns.

The **close** routine closes the serial port and records the fact that it is free to be used again.

7.5 Summary

In this chapter we have looked at the detail of the QUIP Server design and implementation, including the QUIP message interface, the internal workings of the server, the driver interface, and the drivers that have been implemented thus far. In the next chapter, we will take a look at the interactive **quip** program in detail.

Chapter VIII

The QUIP Interactive Program

The QUIP interactive program, **quip**, is a rather simple program that provides an interactive query/response interface to INs that do not provide **telnet** or some other such interface. In this chapter we will look at the design and implementation of the **quip** program. But first, I will need to describe the database that **quip** and **snmp_proxyd** use to map IN names to their addresses, types, and other attributes.

8.1   The QUIP Database

The QUIP database, named **quiptab**, is a file containing one line of information per IN, in fields separated by white space (tabs or spaces). Access to information about a particular IN can be obtained via the **get_quiptab_entry** library routine, a routine in the **quip** library. (Other routines in this library were described in section **7.1**.)

The **get_quiptab_entry** routine takes two parameters. The first, *name*, is a null-terminated string containing the name of the IN for which we are requesting the database entry. The second parameter, *database*, also a null-terminated string, is the name of the database file to search. If the database file name is NULL, a default file name is tried, "/etc/quiptab." If the database entry for the specified name is not found, this routine returns a NULL pointer. Otherwise it returns a pointer to a C structure containing the information in the database corresponding to the *name* parameter. The fields in the database are described below, using the names in the C structure returned by **get_quiptab_entry**, and the order in which the fields are represented in the database file. (In the database file, anything between a '#' character and the end of the current line is considered a comment.)

1. **name** - The name of the IN, as an ASCII string.

2. **major_type** - The major type of the IN. In the database file, this is a ASCII string (currently **ib**, **translan**, or **ibx**). The **get_quiptab_entry** routine maps this string

to an integer corresponding to the **NT** node type C macros that identify a particular major node type and its associated driver (currently **NT_IB**, **NT_TRANSLAN**, or **NT_IBX**).

3. **minor_type** - An integer that differentiates between different command languages used by the various INs of the same major type.

4. **address** - An ASCII string representing the address of the IN (unique within major IN type). In most cases this ethernet address of the device. In the case of IBX bridges, this is the IBX switch port number associated with the bridge.

5. **server** - The host name of the quip server to be used to contact this IN, as an ASCII string.

Several example database entries are listed below:

uncch-cs	ib	0	08000200AF08	astro.mcnc.org
uncch-campus	ib	1	08000201C0DE	astro.mcnc.org
tucc	translan	0	08007C00010A	astro.mcnc.org
uncch-cs-sl1	ibx	0	304	mercury.cs.unc.edu

8.2   The QUIP Program

The **quip** program is invoked in one of the following two ways:

- **quip** *node_name*

- **quip -t** *database_file node_name*

In the first case the default database file is used, and in the second the database file is explicitly specified. In either case, the name of the IN to be queried is specified on the command line.

The **quip** program first obtains the database entry for the specified IN. Then it sets up a connection with the QUIP server indicated by the database entry. Next a trivial command is sent to the IN to ensure that it is reachable and up, and to obtain an initial prompt to display to the user. This trivial command is usually just a newline character of some sort. The response to this command is received and displayed on the standard output. Then the program goes into a loop accepting a command from the standard input, packaging it

up in a QUIP message, sending this message to the QUIP server, retrieving the response, and displaying it to the standard output. When an end-of-file indication is received on the standard input, the program closes its QUIP server connection and exits.

Since **quip** reads from the standard input and writes to the standard output, its input and output can be redirected from or to a file or process, as with most UNIX commands. This means that **quip** need not be run interactively. It can accept commands from a file or process, and write the responses to a file or process. It can also be run from a shell script. For example, one might put the following shell command in a shell script:

```
echo 'sh ss' | quip uncch-cs | awk -f formatter
```

This command executes the 'sh ss' command on the uncch-cs IB bridge, and formats the result using an *awk* script.

To facilitate running **quip** from a shell script, it is possible to combine multiple IN commands on a single line, separating them with a semicolon. For example, one might put the following command in a shell script:

```
echo 'sh ss; sh ns 0; sh ns 1' | quip uncch-cs
```

This shell command executes three commands on the specified IN. Semicolons are converted to newlines, and the entire string of commands is sent to the QUIP server in one message. The QUIP server will return the responses to all the commands, concatenated together into one response message.

The **quip** program has to be aware of the major type of the IN with which it is communicating. This is because the interactive access mechanisms provided by the various vendors have differences that show up at the user interface to **quip**. For example, some vendor supplied interfaces (connection oriented ones) provide prompts, while others (datagram oriented ones) do not. In the latter case, **quip** needs to provide a prompt of its own. In addition, some interfaces end certain commands with a character other than the newline character, such as an ESC (escape) character. When this is the case, **quip** must be prepared to terminate an input line with either a newline or an ESC[1].

---

[1] There is no driver interface to quip that enables these differences to be well isolated from the rest of the quip code. The major type of the IN is, however, only tested in two sections of the code, before and after the main loop. Variables set in the first of these two sections determine the decisions made in the loop without the need to consult the IN type again.

The **quip** program also tests to see if it is being run interactively or not. It uses this information to modify its behavior slightly depending on how it is being run. For example, it will not supply prompts if it is not being run interactively (and if the IN does not do so itself).

Timeouts in the query messages sent to the QUIP server are set as a function of the number of commands in the query, and whether the IN interface is connection or datagram oriented. Currently, the timeout is very high (one hour) for connection oriented sessions, as certain IN commands (to the IBX MMI interface, in particular) can take a very long time to complete. For datagram oriented interfaces, The timeout is currently five seconds times the number of IN commands in the query. The **quip** program is capable of re-sending queries to INs with datagram oriented interfaces, should a query time out or suffer lost packets on the response. If a timeout occurs with no response at all, the query is retried once. If only a partial response is received (a packet is lost), the query is retried nine times. This is because packets are often lost on queries with long responses from datagram oriented interfaces. (This packet loss is caused by bridges with inadequate buffering capacity, and is unrelated to the implementation of my software).

8.3 Conclusion

In this chapter I have described the implementation of the **quip** program. In the process, I have described the QUIP database and the retrieval procedure used to obtain information from this database (**get_quiptab_entry**). This database is used by both **quip** and **snmp_proxyd** to obtain information about particular INs.

In the next chapter, I will describe the Human Interface Parser (HIP), used by my SNMP Proxy Agent to pick relevant information out of an IN response. Then, in the following chapter, I will take a detailed look at the proxy agent itself.

Chapter IX

The Human Interface Parser (HIP)

The Human Interface Parser (HIP) was designed to ease the implementation of SNMP Proxy Agent drivers by facilitating the parsing and decoding of IN command responses. The HIP parser is capable of picking the specified information out of a response that is formatted to be read by a person, rather than a computer. It converts the information it has found into a format that is easily manipulated by a computer, storing the results in a C structure. In the following sections I will describe how to use the parser and, briefly, how the parser is implemented. We will then take a look at a few examples of parser use.

9.1 How to Use HIP

In this section I will describe how to use the HIP parser to select information from an IN response and store the selected information in a C structure.

9.1.1 The HIP Library Routines

Two library routines implement the HIP parser. I describe each of these briefly below:

- **hip_parse**(*specification*, *input*, *output*) - Parses and converts the input specified by the *input* parameter (a pointer to a pointer to a null-terminated ASCII string) according to the specification given by the *specification* parameter (a pointer to a null-terminated ASCII string). The **char *** pointer referenced by the *input* parameter will be updated to point to the character in the input stream just after the last object parsed, to allow the programmer to parse an input string using more than one call to **hip_parse**. Memory is allocated to store the resulting C structure, and a pointer to this structure is copied to the structure pointer variable referenced by the *output* parameter. The first member of this structure must be a **char **** pointer. After the call, this will point to a NULL-terminated array of pointers to memory blocks allocated during this

invocation of **hip_parse**. It is used by **hip_free** to free all memory allocated by this invocation. This routine returns zero if successful, -1 otherwise.

- **hip_free**(*structure*) - Frees the memory allocated by a previous invocation of the **hip_parse** routine. The *structure* parameter is a pointer to the C structure that was the result of a previous call to **hip_parse**. This routine returns no value.

9.1.2 The Parsing Specification

The *specification* parameter of **hip_parse** specifies what information is to be extracted from the input string, and how it is to be encoded in the resulting C structure. This specification string consists of a sequence of *object specifications*, separated by spaces. An object specification has the following three components:

1. **Count** - The number of instances of the specified object type to be found. This component can be one of the following four types:

   - Empty, which specifies a single instance.

   - A non-negative integer, which explicitly specifies the number of instances.

   - *, which specifies zero or more instances.

   - +, which specifies one or more instances.

2. **Object Type** - Either a mnemonic name that indicates the type of information to be extracted and how it is to be encoded, or a left (open) grouping symbol specifying one of several types of object grouping. The various object types are described below.

3. **Argument** - If the object type is not a grouping symbol an argument string may be specified, separated from the object type by a colon character. If there is no argument, the colon must be omitted. If the object type is a grouping symbol, all characters following the left (open) grouping symbol up to but not including a matching right (close) grouping symbol are considered the argument.

The non-grouping object types currently available are listed below. More types may be added rather easily, by following the example of how similar types are implemented[1]. No argument may be specified to these object types unless otherwise noted.

---

[1] No claim is made that the set of object types below is complete in any sense. This set of object types was sufficient to implement the existing drivers. It is likely that new object types will have to be defined to accomodate new drivers.

- **int** - The next integer in the input string is to be found and converted to a **long** integer representation and stored in the next word of the C structure. Negative integers are not recognized as such; the encoded integer is always the absolute value of the integer that was found[2].

- **inttok** - This is the same as **int**, except that an argument is specified that lists one or more non-digit characters, a sequence of which may be found instead of the sequence of digits comprising an integer. If an integer is found, it is stored in the C structure as with **int**. If instead a sequence of the argument characters is found, the first three of the characters in this sequence are used to compose a negative integer that is stored in the C structure in place of the positive integer normally saved there. The function producing this negative integer is accessible to the program calling the parser routine, and returns a number that is unique and deterministic for a sequence of any three characters. The purpose of **inttok** is to efficiently represent a field value that is nearly always an integer, but which has a small set of other potential values (such as "undefined" or "infinite").

- **str** - The next string of non-whitespace characters is to be found and converted to a null-terminated string. A pointer to this string is to be stored in the next member of the C structure. Whitespace is defined to be any sequence of characters composed of space, tab, carriage return, newline, vertical tab, or formfeed characters. If an argument string is specified, any character in that string can terminate the string being parsed, in addition to any whitespace character or the end of the input string.

- **hexint** - The next hexidecimal integer in the input string is to be found and converted to a **long** integer representation and stored in the next member of the C structure.

- **hexstr** - The next hexidecimal string in the input string is to be found and converted to a null-terminated string. A pointer to this string is to be stored in the next member of the C structure.

- **keystr** - This is the same as the **str** type, except that a table of string values matched by the **keystr** object type is maintained. If any given string value matched by **keystr**

---

[2]There should be some way to parse all integers, regardless of sign. Existing drivers did not need to parse negative integers, and the present behavior allows negative integers to be used by **inttok**.

68

is found more than once within the same invocation of **hip_parse**, all pointers to this string value will be to the same string instance. This can save a lot of memory space when parsing a large table where the values in some of the fields of the table are strings that occur frequently. For example, if we are parsing a large table that contains a field whose values can only be the strings "on" or "off," only two memory allocations will have to be made for these field values, instead of one per table entry.

- **ethadr** - The next ethernet address in the input string is to be found and converted to a six byte representation and stored in the next six bytes of the C structure.

Note that whenever the parser is searching for an object of a given type, the search begins immediately after the last object that was found. Any number of characters may be skipped over before the object in question is found. Thus, if we are looking for an integer and then an ethernet address, we will skip all non-digit characters, parse an integer, and then, beginning immediately after the last digit in the integer, will skip all characters until a sequence of six consecutive hexidecimal bytes (encoded in ASCII) are found.

There are four sets of grouping characters, with the following meanings:

- '' - Matching single quotes delimit a literal string to be found in the input string.

- () - Simple grouping characters, used to apply a repetition count to a sequence of object specifications.

- {} - Grouping characters that indicate that all objects within the group must be found within a single line.

- [] - Grouping characters that indicate that all objects within the group must be found within a single line, with the additional restriction that all the objects found corresponding to this specification (some number of instances of this sequence of objects, the number specified by the **count** component) must be found in a set of *consecutive* lines. This is useful for parsing tables.

9.2   Parser Implementation

The **hip_parse** parser is implemented as a loop that matches one object specification on each iteration. Each object specification is first broken down into its **count, object type,**

and **argument** components. Then a table lookup is done to determine which subroutine is associated with the given **object type**, and this subroutine is called **count** times, with the **argument** component as a parameter. Each of the **object type** subroutines returns an indication as to whether or not the object in question has been found. If the **count** component indicates a specific number of instances, the failure of such a subroutine to find what it is looking for means that the innermost grouping invocation has failed. If the **count** component is a *, such a failure terminates the parsing of the current object specification (always successfully). If the **count** component is a +, such a failure also terminates the object specification, unsuccessfully if no instances have been found, but successfully otherwise.

All grouping object types except the single quote (literal) type call the main parser routine recursively to parse the objects specified by their argument. The {} and [] grouping types, since they require all the objects within the group to be found within a single line, cause a boolean parameter called *inline* to be set to **true** on all sub-invocations of object parsing subroutines (this parameter is otherwise false). When the *inline* parameter is true, the parsing subroutines will not look beyond the current line for the object for which they are searching. The parsing routine for the {} grouping symbols will find the next line matching the contained object specifications, while the parsing routine for the [] grouping symbols will not look beyond the current and next lines. The parsing routine for the () grouping symbols simply calls the main parsing routine recursively. This pair of grouping symbols is useful only because it allows a count to be specified for a sequence of object specifications. The literal object type parser simply matches characters in the input string. It does not put anything into the resulting C structure.

A simple sequence of object types with definite instance counts will simply fill in successive members of the resultant C structure as indicated in the corresponding **object type** descriptions above. Indefinite counts (using * or +) must be handled differently. Since the number of instances is unknown by the routine invoking the parser, space cannot be allocated directly in the C structure for the instance values. Instead, a **long** integer indicating the number of instances found is inserted into the structure, followed by a pointer to an array of object instances, each of which is a structure filled in in the same manner as the main structure. Whenever the object type is not a (non-literal) grouping type, this structure is trivial, containing only one member.

70

```
SELECT COMMAND => dgn
DIAGNOSTIC NAME or ? or Return=END........... => lndg
TEST NUMBER or ?............................. => 3
SPECIFY PORT # or L or P or ?................ => 320
SPECIFY CURRENT STATISTICS TYPE  or R = RETRIEVE SAVED STAT or ? => 1
*** LAYER STATISTICS   DIU5                             08/25/89   09:28:40
   PORT  320     LN   0


                      transmitted    received
PACKETS:                 45515205    54556166
BROADCAST PACKETS:         931364    11256085
VIRTUAL CHN DATA:               0           0
VIRTUAL CHN PROTOCOL:           0           0
PACKETS DISCARDED:            161       51984
ELAPSED TIME:    022.00:22:34



CLEAR STATISTICS?  Y = YES, N = NO..........N => @

SELECT COMMAND =>
```

Figure 9.1: IBX LANmark Bridge Response for Example 1

Pointers to memory blocks allocated anywhere in the parser (excepting the allocation of the main structure) are put into an array of pointers so that these memory blocks can be freed by **hip_parse**. A pointer to this array is stored in the first member of the main structure that is returned by **hip_parse**. Memory blocks are allocated for strings and for the arrays of structures that hold instances corresponding to object specifications with indefinite counts.

9.3   Examples

A few examples will clarify the use of the **hip_parse** and **hip_free** routines.

9.3.1   Example 1: A Simple Example

Suppose we are retrieving data from an IBX LANmark bridge, and the response from the bridge, which is the input to the parser, is as shown in Figure 9.1. We wish to extract the port number and the ten counters in the table of transmitted and received packet counters. Since there are integers in this input string other than the ones we want, we will have to

71

```
struct lanmark_packets {
    char **memoryblocks;
    long port;
    long xmt_pkts, rcv_pkts;
    long xmt_bcast_pkts, rcv_bcast_pkts;
    long xmt_vchan_data, rcv_vchan_data;
    long xmt_vchan_proto, rcv_vchan_proto;
    long xmt_discards, rcv_discards;
};

struct lanmark_packets *lp;

char *lp_spec = "'LAYER STATISTICS' 'PORT' int 'PACKETS:' 10int";
char *lp_response = "SELECT COMMAND => dgn\nDIAGNOSTIC NAME or ? or ...";
char *lpr;

lpr = lp_response;
if (hip_parse(lp_spec, &lpr, &lp)) error("Can't parse lp_spec");
else {
    process_lp_data(lp);
    hip_free(lp);
}
```

Figure 9.2: C Code for Parsing Input Shown in Figure 9.1

use literal strings in the parsing specification to ensure that we get the right values. Our specification string for parsing this input is as follows:

```
"'LAYER STATISTICS' 'PORT' int 'PACKETS:' 10int"
```

This specification indicates that we are to find the string 'LAYER STATISTICS', and then the string 'PORT'. Next we are to parse the next integer (the port number). Finally, we are to find the string 'PACKETS:', and parse the next 10 integers.

The C structure into which the resulting integers are to be stored is shown in Figure 9.2 This figure also shows the declarations of the pointer to the C structure and the specification and input strings that are used in the invocation of **hip_parse** and **hip_free**. Finally, it demonstrates how these routines might be invoked. Note the required **memoryblocks** member, which is used by **hip_free** to free up all memory blocks allocated by **hip_parse**. In this case, since there are no strings or indefinite counts in the specification, there will be no such memory blocks to be freed, other than that allocated for the C structure itself (which is not listed in the **memoryblocks** list, but is freed by **hip_free**). Note also that

72

```
SELECT COMMAND => dgn
DIAGNOSTIC NAME or ? or Return=END............ => lndg
TEST NUMBER or ?.............................. => 3
SPECIFY PORT # or L or P or ?................. => 320

SPECIFY CURRENT STATISTICS TYPE  or R = RETRIEVE SAVED STAT or ? => 2
SPECIFY APPLICATION STATISTICS NUMBER.......1 => 2
**** DIU5 LANmark INTERFACE                           08/25/89   11:12:42
   PORT  320      LN   0


ELAPSED TIME  022.01:24:38

TRANSMITTED INTO LANmark     45606774
BROADCAST PACKETS TRANSMITTED  16083
                                     ETHERNET SRC         ETHERNET DEST
      DEST DOB ADDR UNKNOWN     33775  08 00 20 00 80 E8   08 00 20 06 47 32
      MULTICAST ETHERNET FRAMES 47829  08 00 20 01 28 79   FF FF FF FF FF FF
      NEW DEVICES ON COAX CABLE    15  08 00 2B 11 12 CA

RECEIVED FROM LANmark          46355
   BROADCAST PACKETS RECEIVED   48626
                                      LANmark SRC          ETHERNET DEST
    NONMULTICAST ENET FRAMES     1517  00 09 01 00 00 00   08 00 02 00 00 00
    MULTICAST ENET FRAMES       47109  00 09 01 00 00 00   01 80 C2 00 00 00
CLEAR STATISTICS?  Y = YES, N = NO..........N => @

SELECT COMMAND =>
```

Figure 9.3: LANmark Response for Example 2

the pointer **lpr** is initialized to the value of **lp_response**, and that a pointer to **lpr** is passed to **hip_parse**. The lpr pointer will be updated to the character following the last object successfully parsed in the input string (the integer 51984, in this case).

9.3.2 Example 2: A More Complex Example

My second example involves a two more object types and simple grouping. The LANmark response string for this example is shown in Figure 9.3. We wish to extract the elapsed time (as a string), and all counters and ethernet addresses in the response string. Our specification and C structure are shown in Figure 9.4.

```
struct lanmark_stats {
    char **memoryblocks;
    char *elapsed;
    long xmt, xmt_bcast;
    long addr_unknown;
    u_char au_src[6], au_dst[6];
    long xmt_mcast;
    u_char xmc_src[6], xmc_dst[6];
    long new_device;
    u_char nd_addr[6];
    long rcv, rcv_bcast;
    long rcv_nmcast;
    u_char rnmc_src[6], rnmc_dst[6];
    long rcv_mcast;
    u_char rmc_src[6], rmc_dst[6];
};


char *ls_spec =
"'ELAPSED TIME' str 2int 2(int 2ethadr) int ethadr 2int 2(int 2ethadr)";
```

Figure 9.4: C Code for Parsing Input Shown in Figure 9.3

9.3.3 Example 3: Parsing Large Tables

In this final example, I will demonstrate the use of a specification with an indefinite instance count, and the all-in-consecutive-lines grouping symbols ([]). I will also make use of a few more object types. An abbreviated display of the response string is shown in Figure 9.5. This response string contains two bridge learning tables, one for each of the two interfaces of an IB bridge. We wish to parse these tables separately. We do not need to extract the **No.** field, as this is implicitly obtained by the array index. We do have to extract the other fields, even the **Network** field, because of the infrequent non-integer values of this field (**Ib**, **Disable**, and **NetsAndIb**), even though without these special string values we could have deduced the network number from the order of the tables. Since the first three characters of each of these values are unique, we can use the **inttok** object type instead of the **str** type, and thus save space. We will also use the **keystr** object type to parse the **Age** field, since there are only four possible string values in this field (**User**, **Young**, **Middle**, and **Old**). We must use the all-in-consecutive-lines grouping symbols ([]) to enable us to determine where the first table ends. (Alternatively, we could have run both tables together and used the **Network** field to sort out which entry belonged to which interface, by using the all-

74

```
No.   Station Address   Network     Depth   Age
----  ----------------  ----------  -----   ------
  1   %08000200AF08     Ib            0     User
  2   %080089D01117     Disable       0     User
  3   %FFFFFFFFFFFF     NetsAndIb     0     User
  4   %080089A04115     0             0     Young
  5   %080089A04732     0             0     Young
  6   %080089A03792     0             1     Young
  7   %080089A03610     0             0     Young
  8   %08002B0EF925     0             0     Middle
  9   %08002000380F     0             1     Middle
 10   %08002000AA9F     0             0     Old
      . . .
      . . .
      . . .
----- total number of entries = 83


No.   Station Address   Network     Depth   Age
----  ----------------  ----------  -----   ------
  1   %08000200AF08     Ib            0     User
  2   %080089D01117     Disable       0     User
  3   %FFFFFFFFFFFF     NetsAndIb     0     User
  4   %080002010440     1             0     Young
  5   %08000200CE3B     1             1     Young
  6   %08000200E355     1             0     Young
  7   %080069020294     1             0     Middle
  8   %08002000952A     1             0     Middle
  9   %08002B020267     1             1     Middle
 10   %AA000400B09C     1             0     Young
      . . .
      . . .
      . . .
----- total number of entries = 268
```

Figure 9.5: IB Response for Example 3

```
struct ib_learntab_entry { u_char address[6] long network; long depth;
char *age; };

struct ib_learntab { long nentries; struct ib_learntab_entry *entry;
};

struct ib_learntabs { char **memoryblocks; struct ib_learntab
learntab[2]; };

struct ib_learntabs *ib_lt;

char *ib_spec = "2(*['%' ethadr inttok int keystr])"
```

Figure 9.6: C Code for Parsing Input Shown in Figure 9.5

in-a-line ({}) grouping symbols.) Our C structures and parsing specification are shown in
Figure 9.6. Note that the address of the ib_lt pointer variable (&ib_lt) is passed as the
output parameter to **hip_parse**.

9.4 Conclusion

In this chapter I have described the design, implementation, and use of the Human Interface
Parser (HIP). This parser was designed to facilitate the parsing and decoding of IN command
response strings intended to be read by humans. The HIP parser greatly simplifies coding of
the routines in the SNMP Proxy Agent that must extract data from the command responses
received from INs, thus making it much easier to write proxy agent drivers for new IN types.

Now that I have completed the description of the design of the QUIP server and QUIP
and HIP libraries (including the **get_quiptab_entry**, **send_quip**, **recv_quip**, **hip_parse**,
and **hip_free** routines used by the proxy agent), I am ready to describe the proxy agent
itself in detail. This I will do in the next chapter.

Chapter X

The SNMP Proxy Agent

I am now ready to describe the detailed design and implementation of the SNMP Proxy Agent, **snmp_proxyd**. This is the server process that implements the mapping between the SNMP protocol and the proprietary command languages of INs that do not support SNMP directly.

The proxy agent is composed of a common section of code and a set of drivers, one for each IN type. I will first describe the common code, and then the driver interface.

10.1 The Main Body of the Proxy Agent

The main body of the proxy agent begins by initializing global data structures and setting up a UDP socket on which it will listen for SNMP request packets. The data structures initialized include a tree structured mapping between the ASCII-mnemonic and numeric forms of SNMP object names, as shown in Figure 5.1, a list of structures containing information about each IN including its variable cache, an array for information about each QUIP server connection, a list if structures containing information about each outstanding QUIP query, and a list of all outstanding SNMP requests, which is used to facilitate the discarding of old unanswered requests.

The setup of the tree used to map between ASCII-mnemonic and numeric forms of object names requires a bit more explanation. A file called **/etc/snmp.variables** contains the mnemonic and numeric versions of each object name. This file is read, and a corresponding mapping tree constructed, during the initialization of the server. Each line of this file contains one object name in ASCII and its corresponding numeric form, as indicated by the following examples:

iso 1 org 3 dod 6 internet 1 mgmt 2 mib 1 system 1 sysDescr 1

iso 1 org 3 dod 6 internet 1 mgmt 2 mib 1 system 1 sysObjectID 2

iso 1 org 3 dod 6 internet 1 mgmt 2 mib 1 system 1 sysUpTime 3

This file must be updated, and the server re-started, whenever a new MIB is published or variables in the **private** portion of the object name tree are changed.

After initialization of these data structures and the UDP socket for SNMP requests is complete, the program enters an event loop where it accepts accepts SNMP requests and QUIP responses one at a time, performs the appropriate action for each event, deletes any old SNMP requests that have not been answered, and returns to the top of the loop. I will discuss each of the two event types in detail in the following sections.

10.1.1 SNMP Packet Arrival Events

When an SNMP packet is received, it is first parsed to extract such information as the community name, IN name, type of request (**set**, **get**, or **getnext**) and a list of object names (and values, if it is a **set** request). Note that, while the SNMP packet format contains provision for *proxy* information such as the specification of the IN to be accessed, most existing management tools do not support the *proxy* feature, and therefore cannot fill in this portion of the SNMP packet. Thus, an alternate mechanism for specifying the IN to be accessed is required. In order to allow existing tools to work with our system, we have adopted the convention of encoding the IN identification along with the community name. Thus, if a community named "public" is to be used on an access to the IN named "uncch-cs," one would specify a community name of "public.uncch-cs" to the tool. The proxy agent separates the real community name and the IN name portions of this "community name" when the packet is parsed. Unfortunately, some tools only allow for rather short community names (as few as 15 characters), so community names and IN names must be kept short. Since the "public" community is currently the only one commonly in use, I have allowed a null community portion to represent the "public" community. Thus, ".uncch-cs" means the same thing as "public.uncch-cs" to the current server implementation.

The parsing of the SNMP packet is done using public domain library routines written by Steve Waldbusser at Carnegie Mellon University (CMU). The CMU library has a routine that will parse an SNMP packet and return two data structures, one containing the variables and their values, and another containing other information about the request. A different routine

takes these same structures and builds an SNMP response packet. Of course, decoding the community ID as described above is done by our agent, not the CMU library.

Once the SNMP packet has been parsed, **get_quiptab_entry** is called to get information about the target IN. Then all the information about the request is put into a *request* data structure, which is added to two lists. One of these, a global request list, is used simply to gain quick access to "old" requests that have not been filled (e.g. if the IN is down) so that they can be discarded. (Note that, unlike the QUIP server, the proxy agent does not need to send a timeout response to the management tool. The management tool will time out the request itself.) The other list to which the request is added is part of the data structure containing IN-specific information about the target IN. (If this is the first SNMP request for this particular IN, this data structure is first created as described below.) The request is also timestamped as it is queued to enable the agent to determine when it is old.

Once the request has been queued, an attempt is made to fill the request. (The process of filling a request will be described in detail later.) If this attempt is successful (e.g., if it is a **get** or **getnext** request and the values of the the requested variables are all in the cache), the SNMP response packet is generated and sent to the appropriate management tool, and the request is dequeued. Otherwise, the process of trying to fill the request will initiate requests to the target IN to obtain the desired information or perform the desired functions, and the request will be left in the list to await the results of these actions.

If this was the first SNMP request for a particular IN, the process of queuing the request will fail to find a data structure for the target IN in the global list of such structures. When this happens, a structure for this IN is created, initialized, and added to the global list. The initialization process includes calling the **build_node** driver routine, which initializes the cache for this particular IN in an IN-type specific manner. (This cache initialization will be discussed in more detail momentarily.) In addition, if this is the first IN associated with a particular QUIP server, a connection is also set up with this server at this time.

The **build_node** driver routine is called to initialize the cache structure for a particular IN. The cache structure is a tree that takes the same form as the object name mapping tree described earlier, except that it contains only those branches relevant to the IN-type with which it is associated. It also will grow to include object instances and their values as such values are requested and obtained from an IN. The cache initialization driver routine builds a stump of this cache tree. It does this by specifying which major branches of the tree are to

be supported by this IN type, and associates a *retrieval procedure* and a *lifetime* with each of these branches. Each *retrieval procedure* is a driver routine that will be called to initiate queries of an IN to obtain the objects contained in the associated branch of the tree and their values. The *lifetime* associated with a branch of the tree indicates how long the cached *structure* of this branch of the tree will remain sufficiently current to be used as a basis for filling requests. (The *structure* of the branch determines which object instances exist in this branch at any given time.) Note that the structure of a branch can change, e.g., when table entries are added or deleted. There is also a separate *lifetime* associated with each object instance. This will be discussed further as I describe the filling of requests in the following paragraphs.

Filling an SNMP request is a rather complicated process. We will begin by assuming the request is a **get** request, and will later point out how a **getnext** request differs[1].

An SNMP **get** request contains a list of variables for which current values are sought. Filling such a request involves obtaining the value of each variable. If the value of every variable in the request can be obtained (all at the same time), the request is considered filled and a response packet can then be generated and sent. If it is determined that a particular requested object does not currently exist, an error response is generated indicating which object did not exist. If obtaining the value of some variable requires that the cache for some branch of the tree be updated, an IN query is initiated to obtain the necessary information to update the cache (unless the required query has already been initiated and a response is pending)[2]. If obtaining one variable's value requires a cache update, this does not stop the proxy agent from attempting to obtain the other requested variable values. This is because all necessary IN queries for this request should be initiated as soon as possible. However, any values obtained from the cache for this request will be freshly obtained when another attempt is made to fill the request. This ensures that all values returned are current as of the time the request is successfully filled.

Let us look in more detail at how one variable value is obtained. Suppose the ASCII representation of the variable name is

iso.org.dod.internet.mgmt.mib.interfaces.ifTable.ifEntry.ifInOctets.1

---

[1] I will not discuss the filling of set requests here, as they have not yet been implemented. Note also that community IDs and other protection mechanisms have also not been implemented and will therefore not be discussed here. (See Chapter 12 for suggested improvements.)

[2] Currently the only means of querying INs is via QUIP.

which has the numeric representation (1.3.6.1.2.1.2.2.1.10.1). The variable name is represented in numeric form in the SNMP request packet. A traversal of the cache tree associated with the target IN is made, using the node numbers to determine which of the current parent node's children is to be selected at each step. This traversal proceeds uneventfully through iso.org.dod.internet.mgmt.mib (1.3.6.1.2.1).

When the "interfaces (2)" node is reached, a pointer to a retrieval procedure is found in the node. This indicates that the full extent of this branch of the tree stump originally produced by the **build_node** driver routine has been reached. (Whether or not the tree extends further depends on whether or not the variables in this branch have been previously obtained.) Now the *expiration* attribute of this node is checked to see if the structure beyond this point in the branch exists and is sufficiently current to warrant further traversal without first requiring an update of this branch of the tree. (This *expiration* attribute is computed from the current time and the *lifetime* attribute of the node whenever the branch structure is updated.) If not, we next check the *retrieval_pending* flag in this node to see if the the retrieval procedure has already been called and we are waiting for the response. If this is the case, we are finished with this attempt to obtain the value of this variable. Otherwise the retrieval procedure at this node is called, specifying the target IN data structure as an argument, and we set the *retrieval_pending* flag. The retrieval procedure sends a QUIP query to the specified IN. This query includes all the IN commands necessary to retrieve all the variables in this branch of the tree. Note that it would be both inefficient and inadequate to obtain each variable individually. It would be inefficient because most single IN command responses contain information from which the values of multiple variables can be extracted or computed. There is no sense in throwing away the extra information once we have received it, since the major time delay is in obtaining a response, not extracting the information from that response. It would also be inadequate, as it is the *structure* of the branch (i.e., what variables are present in the branch) that needs updating at this point, not (yet) the value of the variable for which we are searching. (This distinction is not really important for **get** requests, but it is for **getnext** requests, discussed later.) In addition to sending a QUIP query to the target IN, the retrieval procedure adds an element to a global list that matches the unique id in the QUIP message to various information about this query (a pointer to the target IN data structure, the node in the cache tree that contained the retrieval procedure pointer, and a pointer to the driver procedure that will parse the response to the query and

update the cache). Since the retrieval procedure only *initiates* the query, and the response will take some time to arrive, we are now finished with this attempt to obtain the value of this variable. We will need to try again after the response has arrived. This will be discussed in the next section, on QUIP response events.

Now let us back up a minute and suppose that when we reach the "interfaces" node in our tree traversal, we find that the *expiration* attribute of this node indicates that the cached structure of this branch of the tree is sufficiently current. In this case, we continue our traversal of the tree through nodes "ifTable.ifEntry.ifInOctets.1" (2.1.10.1) to the end of the variable name. If at any point (before or after reaching the node with the retrieval procedure) we find that the next node in the object name sequence does not exist, our attempt to obtain this variable has failed and we will return an SNMP error packet flagging this variable. In this case, no attempt will be made to obtain the values of any other variables. If we reach the end of the variable name successfully (and this is a **get** request), we must also find that the last node encountered is a leaf of the cache tree, or we will handle this erroneous situation similarly. Next, we check the *expiration* attribute of the leaf node, to see if the value of this variable is current enough to be used. If not, we call the retrieval procedure as described above (which we have remembered since passing the node where we found its pointer), and we are done for now. If the *expiration* attribute indicates that the value is current, we must still check a list of tools that have received this value of the variable. There is such a list associated with each variable in the cache tree, and it represents a tool by the Internet number and port number of the client that sent the SNMP request. If we determine that the current value of this variable has already been sent to the tool that initiated the current request, we must call the retrieval procedure as described above to obtain a new value. Otherwise, we have found the variable we wanted, and there will be a pointer in the leaf node to the value of this variable. We copy this value into the request structure, add our current tool to the list of recipients of this value, and move on to the next variable.

Now let us suppose that the SNMP request is a **getnext** request. As you will recall from the description of SNMP in Chapter 5, the **get** request specifies a specific object instance, while the **getnext** request may specify either a specific object instance or a partial object name. When a **getnext** request is received, the tree is traversed exactly as with the **get** request, until the end of the specified object name is reached (potentially calling a retrieval

procedure or detecting an error along the way). When the end of the object name is reached, whether we are at a leaf of the cache tree or not, we switch to a different traversal method, wherein we perform a depth first traversal of the tree in increasing node number order (possibly backing up when we reach a dead end), until we find the next object instance in the tree. In this case, the name of the instance found as well as its value are copied into the request structure. Note that the only ways a **getnext** request can fail with a "nonexistent instance" error are if the first part of the specified object name matches the full name of an existing variable but has additional nodes specified, or if there is no variable in the tree with a name that comes after the specified name in the order in which we are traversing the tree. Note also that there may be a retrieval procedure called in the first portion of the traversal (the portion like **get**), and again in the second portion, where we are looking for the "next" instance. In this (rare) case, two IN queries are required, which will delay the response to the SNMP request.

We have now concluded our analysis of the handling of SNMP packet arrival events. In the next section I will discuss what happens when QUIP responses are received.

10.1.2 QUIP Response Arrival Events

Whenever we send a query to a QUIP server, we are guaranteed that we will eventually get some sort of response (unless the QUIP server hangs or has a bug). Possible responses are:

1. A QUIP message indicating failure to process the query (e.g., if the timeout expires or the server receives an incomplete response).

2. A QUIP message containing the desired response.

3. The QUIP connection is dropped. (This is not really a response, but it does cause a QUIP response arrival event.)

Let us consider each of these possible responses.

If we receive a QUIP message at all (cases 1 and 2), recall that there is a unique message ID in the response message that is identical to the message ID in the corresponding query message. We use this ID to find an entry containing information about this query in our list of pending QUIP requests, and remove this entry from the list. If the response is an error response (case 1), we simply clear the *retrieval_pending* flag in the node of the cache tree that contained the pointer to the retrieval procedure that initiated this query, so that

the next time that node is traversed we will call the retrieval procedure again. We do not retry the query, and we do not send an SNMP error to the management tool that sent the request. The tool will time out the request and may re-send it. We do not even dequeue the request, since there may be multiple requests waiting on this response. The proxy agent will remove old requests after a while to keep things tidy.

Before we describe what happens in the normal case when we receive a valid response from QUIP, let us consider what happens if the QUIP connection is dropped. Certainly we will wish to close the file descriptor and go through the list containing information about pending QUIP requests, removing all those that correspond to the dropped connection. Beyond that, we could simply set a flag in a table of QUIP servers to indicate that the connection must be reestablished next time an SNMP request comes in for an IN being served by this server. However, since the information cached for these INs will quickly become out of date anyway, I have taken a more drastic approach, preferring to remove the entries corresponding to all these INs from the list of IN data structures, discarding the cache trees, request lists and any other information we have about these INs. This has the advantage that the **quiptab** table will be re-consulted each time an SNMP request arrives for one of these INs, until a successful connection is made with the specified QUIP server. This allows a network operator to change the **quiptab** table to route requests for these INs to another server.

Now consider what happens when we receive a valid QUIP response. First, we use the unique ID in the message header to find the appropriate entry in the list of pending QUIP queries, and we remove this entry from the list. Then we call the **parser** driver routine indicated in the entry, passing as a parameter the pointer to the IN data structure associated with the query, also found in the entry. The **parser** routine will be described in detail in the next section. Briefly, it parses the IN response and calculates and inserts into the cache tree the values of all the variables determinable from the response string, along with a new *expiration* value for each. Note that the variables inserted do not all have to reside under the node where we originally found the retrieval procedure. It may be that certain variable values are found in the responses to different commands, and we may as well update them whenever we receive a new value, even if it has not been obtained using the retrieval procedure that would be used if the value of that variable had been requested directly. After the **parser** routine has completed its task, we calculate a new *expiration*

value for the node containing the retrieval procedure pointer and clear the *retrieval_pending* flag of the this node. Finally, we go through the list of pending requests associated with this IN and attempt to fill all the requests in the list. If any is filled, the SNMP response is generated and sent back to the tool, and the request is dequeued.

I have now finished describing the handling of QUIP response arrival events. In the next section I will describe the driver routines that must be implemented for each IN type.

10.2   The Proxy Agent Driver Interface

The driver for a particular IN type consists of a **build_node** routine and a pair of routines for each node of the cache tree with which the **build_node** routine has associated a retrieval procedure. This pair of routines consists of the retrieval procedure itself, which sends commands to an IN requesting the information required to fill in the cache tree below this node, and a corresponding parsing routine that parses the resulting IN response and inserts the new values of the variables determinable from this response into the cache tree. Note that this set of variables must at least include all variables under the node of the tree associated with the corresponding retrieval procedure, but is not limited to those variables. A brief overview of each of these routines and their parameters follows:

- **build_node(***cache_tree***)** - Build the stump of a cache tree by inserting branches that cover all supported variables in the tree. The *cache_tree* parameter is a pointer to an empty cache tree to which the branches are added. A branch is added by calling the routine

    cache_branch(*cache_tree, name, procedure, lifetime***)**

    where *name* is the ASCII representation of the object name prefix corresponding to the node, *procedure* is a pointer to the retrieval procedure, and *lifetime* is the number of seconds that the structure of this branch will remain current, which is used in computing the structure *expiration* value for the node at the root of the branch.

- **retrieve(***IN_struct, cache_node***)** - Retrieve information from an IN. The *IN_struct* parameter is a pointer to the data structure containing information about the target IN. It is used to obtain the IN type and address to put into the QUIP message. The *cache_node* parameter is a pointer to the node of the cache where the pointer to this

retrieval procedure was found. Once the QUIP message is built and sent, an entry for this message is added to the list of pending QUIP messages, using the routine

queue_quip(*message_id, parse_routine, IN_struct, cache_node*).

The parameters of this routine become the values of the members of the new list entry structure. The *message_id* parameter is the unique message ID in the QUIP message, which will be used to find this list entry when the response message arrives. The *parse_routine* parameter is a pointer to the **parse** driver routine (described below) that will parse the results and insert the new variable values into the cache tree.

- **parse**(*response, IN_struct*) - Parse an IN response and insert all the variables that can be computed from this response into the cache tree (part of the *IN_struct* parameter). The response is parsed using the **hip_parse** routine described in the previous chapter, and variables are inserted using the routine

variable_insert(*name, instance, type, value, size, IN_struct*).

The *name* parameter is the name of the variable to be inserted, excluding the instance specification. The *instance* parameter is the instance portion of the variable name. (It is a separate parameter to facilitate the insertion of variables for a number of instances using a loop.) The *type* parameter indicates the data type of the variable. The *value* parameter is a pointer to the new variable value, and the *size* parameter indicates its size in bytes. The *IN_struct* parameter a pointer to the IN data structure containing the cache tree into which the variables are to be inserted.

To date, drivers have been written for the IB, TransLAN, and LANmark bridges. These drivers currently implement the "system" and "interfaces" branches of the current MIB.

10.3   Conclusion

In this chapter I have given a detailed description of the SNMP Proxy Agent, which is the heart of this project. This agent, supported by the software described in detail in the preceding three chapters, makes it possible to access any IN in the network for which drivers have been written, using the standard SNMP network management protocol.

This chapter completes the detailed description of the design and implementation of the software for this project. In the remaining two chapters, I will evaluate the software with respect to the requirements presented in Chapter 4 (suggesting improvements to the existing system in the process), and will list projects that might be attempted in the future to build on the work accomplished here.

Chapter XI

An Evaluation of the Resulting System

In this chapter I will summarize the fruits of m;y efforts by comparing the resulting software system to the requirements specified in Chapter 4, suggesting improvements that will correct any deficiencies seen at this point in time.

11.1 Requirements vs. Results

In the final section of Chapter 4, I presented a requirements list for the system. I repeat that list here, along with an analysis of how successful the current system is in meeting each. Wherever the system falls short of a requirement, I have suggested improvements for correcting the situation.

Requirement 1 *Caching agents must be used to collect information from INs, cache it, and deliver it to tools.*

This requirement has been met by the SNMP Proxy Agent described in Chapter 10.

Requirement 2 *Caching agents must be fast enough to handle about one hundred requests per second.*

Experimental results show that the Proxy agent is able to handle about 115 simple SNMP requests per second, when running on a Sun 3/60. These results are for tests where the cache hit ratio is high, which is normally the case when there is a high rate of requests.

Requirement 3 *Caching agents must be distributable among administrative domains, and should share information with each other. It should also be possible for them to communicate with each other using a second, independent network.*

The proxy agent is widely distributable, as I have taken care to use a public domain SNMP library, and the code has also been placed in the public domain. It is also easily portable to any UNIX platform. However, agents are not yet able to communicate with each other. It would be a simple matter to modify the code to allow drivers using access mechanisms other than QUIP to get information from INs, and then to write an SNMP driver for the proxy agent. This would allow us to assign certain INs to certain proxy agents to decrease or eliminate the incidence of multiple proxy agents querying the same INs for the same information. From that point, it would be a fairly easy task to implement transmission of these SNMP packets over an independent network. Dialup lines would be inadequate for continual use, but would be useful during network failure diagnosis. A separate serial network from some central point to a point within each administrative domain might be advantageous.

Requirement 4 *Communication with caching agents must be done over standard transport protocols, to facilitate wide distribution of tools.*

The proxy agent uses UDP for transporting SNMP packets. This fully fulfills this requirement.

Requirement 5 *It should be possible to use standard management tools to perform any management task on an IN that can be performed via another means.*

The proxy agent has a number of deficiencies in this regard. There is still a lot of information that can be obtained only via interactive sessions. This problem can be solved by adding objects to the **private** branch of the MIB and upgrading the SNMP drivers to include these variables. I have obtained a subtree in the **private** branch for this purpose, which I call "mcnc" (48). Next, we need to implement the SNMP **set** request in the proxy agent and upgrade the **private** branch and drivers to include as many of the setting actions available on our INs as possible. I have found that some of the setting operations available via direct serial lines to some of our IN types do not work via the in-band mechanism (i.e., we cannot even perform these operations via QUIP). These operations are generally those that are most destructive, such as rebooting. We do not yet know if there are ways around these restrictions.

Requirement 6 *The management protocol should mask accidental differences in the way identical management functions are performed on different IN types, below the tool level.*

SNMP itself provides a common mechanism for performing common actions. Since our proxy agent brings all INs to the SNMP level, we only need to make sure that we implement the same variables to perform the corresponding actions for various IN types. This requirement has been met for all currently implemented variables.

Requirement 7 *The management protocol should have authorization mechanisms capable of preventing unauthorized IN access.*

Much work needs to be done in this area. We could easily integrate a server-based authentication system such as Kerberos[SNS88] with our software, which would enable us to reliably identify tool users (and servers) in a secure fashion. This authentication capability could then become the basis for restricting the set of IN commands that the QUIP server would issue, or the SNMP requests that the SNMP proxy server would honor, from a given user. We may also wish to further restrict certain classes of requests by IP source address.

Requirement 8 *The network management system should allow INs to be managed using imported tools that conform to current Internet standards.*

I have been largely successful in meeting this goal. I have been able to get all the applicable client programs that I have imported from other sites to work with the system, most without source code changes. This is primarily due to the convention of allowing the IN name to be specified along with the community name. Some client programs must be source code modified because they use what they consider the IN IP address (and we consider the IP address of the proxy agent) to uniquely identify a particular IN, whereas we must use this IP address in combination with the name or address of the IN itself (if this IN is being accessed by proxy).

Requirement 9 *The network management system must not use more than about 5% of any particular network resource (LAN or IN) under normal circumstances, and not more than about 15% while diagnosis is taking place. No IN should be queried more than a few times per second in any case.*

Network utilization is easy to calculate based on the sizes of network management packets, their frequency, and network bandwidth. IN utilization can only be calculated by the IN itself. Fortunately, the IB bridges give us such information, and we will assume the other

IN types experience similar loading for similar management requests. The last requirement, that an IN not be queried more than a few times a second, is met by the tools themselves, which normally will not query an IN more than once a second, and the cache of the proxy agent, which keeps multiple tools from making multiple redundant requests of an IN.

Experiments indicate that querying an IB bridge for a small amount of information once per second uses about 1.5% of the IB's CPU, while large queries at the same rate take about 16%.

The effective bandwidth of a 10Mbps Ethernet is about 400KBps, including all packet headers. A large SNMP query once per second to a proxy agent takes two packets of about 512 bytes each, using approximately 0.25% of the bandwidth of an Ethernet. Similar calculations show that refreshing the cache entries for a single IN (say, once per second maximum, regardless of the number of tools running), about 0.38% of the network bandwidth is used sending messages between the proxy daemon and the quip daemon, and 0.51% sending messages between the quip daemon and the IN. For small queries, the numbers are 0.06%, 0.16%, and 0.06%, respectively. Actual network (and IN) utilization will depend on the network hardware and configuration, the configuration of the network management system, and the number and type of tools active at any given moment. For example, if all components are on the same network, and a single tool is sending large SNMP requests to 2 INs once per second, we would expect the network utilization to be about (0.25 + 0.38 + 0.51) * 2) = 2.03%. Similarly, if 50 tools are sending small SNMP requests to 10 INs once every 10 seconds, utilization will be about (0.06 * 10 * 50 + (0.16 + 0.06) * 10)/10 = 3.22%. These numbers are well within our range of acceptability.

Requirement 10 *Round-trip time for a query of an IN must be less than one second.*

Very large SNMP queries (20 variables) have a round-trip time of about 0.6 seconds, with the tool and proxy daemon running on a Sun 3/60 and the quip daemon running on a VAX 11/750.

Requirement 11 *Operational status queries (queries for up or down status) must be particularly undemanding on INs and LANs, as they are executed frequently.*

I have already presented experimental results for large and small queries in the discussion of **Requirement 9**. Small queries are considerably more efficient than large ones, and they do not place a high demand on IN or network resources.

91

Requirement 12 *Interactive access to INs is required, and should use a common interface at the user level and standard transport protocols at the tool/network interface.*

This requirement is fully met by the **quip** program, which uses TCP to communicate with the QUIP server.

Requirement 13 *Our system must make it as easy as possible to add drivers that will give us access to more IN types.*

My experience indicates that it takes about one day for someone experienced with writing these drivers to write a new one for either the QUIP Server or the SNMP Proxy Agent. The existing QUIP Server drivers average 210 lines per driver, and this is by far the most difficult of the drivers to implement. The hardest part of this task is usually figuring out the manufacturer's proprietary protocol. The SNMP Proxy Agent drivers average 285 lines each, but most of this code is routine and repetitive. Much time can be saved in writing future drivers by following the examples set by existing ones. We feel this level of ease of driver implementation is entirely adequate. Modifications to the **quip** program are usually trivial.

The experience of having written a few drivers has, however suggested that the driver interfaces could use some improvement to increase their consistency and provide for better information hiding. These improvements were mentioned briefly in previous chapters.

11.2   Conclusion

In summary, I conclude that the system has met most of its requirements. The major shortcomings are listed below:

- QUIP needs an authentication system to keep IN configurations from being changed by unauthorized users.

- SNMP needs to be upgraded to handle SNMP **set** requests.

- When SNMP **set** requests are implemented, the proxy agent will need an authentication mechanism.

- The SNMP Proxy Agent needs an SNMP driver so that it can cache results from INs that support SNMP directly. This mechanism can also be used for communication of IN information between proxy agents.

- Existing SNMP drivers need to have more variables implemented, especially in the "private.mcnc" branch.

- QUIP Server and SNMP Proxy Agent drivers need to be implemented for more IN types.

I claim that the basic design of the system will make it easy to implement the above-mentioned improvements, as the system was designed with these capabilities in mind.

Chapter XII

Future Projects

The software described in this paper provides a foundation upon which a network management system can be built, as it provides the access to network management Interconnection Nodes that is required by a good network management system. Where do we go from here? What do we need to build on top of the foundation? Some good answers to these questions have already been discussed in Chapter 2, where I discussed network management tools pertaining to the management of INs. In this chapter we broaden our scope to include tools for managing local area networks as well, and give some specific suggestions for future projects.

Generally speaking, what we now need is *tools*. Tools may be either imported or written locally, and we will need to do both. Most of the tools we acquire or write will need to be public domain or cheap, as they will need to be widely distributed among the various institutions participating in the NCDN.

A few SNMP-based network management tools (from NYSERNet, CMU, MIT, the University of Tennessee, and a few others) are available free or for purchase. Our experience indicates that these tools are still rather immature. Some more mature LAN management tools are available, such as **etherfind**, **tcpdump**, and **NNStat**. These should be imported and distributed as widely as possible.

Below are listed some suggestions for SNMP-based tools that might be written locally.

1. A graphical network display tool for users. A simple tool might display a static, logical map of the entire network or a portion of it. A more ambitious project might use geographically oriented maps with zooming capabilities.

2. A graphical network display and control tool for network operators (using either logical or geographical maps). This tool should be extensible, enabling the network operator to set up his own key bindings and buttons, and associated actions based on SNMP variables and their corresponding values. Generalized plotting facilities should also be

94

available in such a tool, and it should also provide interactive access to INs.

3. A monitoring and alerting tool should be developed that uses a database specifying what is to be monitored (in terms of SNMP variables), threshold values for alerts, whom to notify of various detected anomalies, and the procedures used to notify these people. Various alerting mechanisms should be available, such as visual and audible signals, electronic mail, and telephone messages.

4. A tool for gathering statistics and placing them into a database. What statistics are to be gathered (in terms of SNMP variables), and how often, should be specifiable in the database itself. Report generation tools should be written to summarize the collected data.

Below are listed a couple of LAN management tools that might be written locally.

1. It would be quite useful to have a graphical network monitor (especially for Ethernet) similar to the Excelan LANalyzer or Network Systems Sniffer, that will work under X windows and run on a Sun using its Network Interface Tap (or some similar facility on another type of workstation). I am not aware of the existence of such a tool at present, though something similar is being worked on by Russ Taylor at the University of North Carolina Department of Computer Science.

2. A tool that would run on a host computer and analyze network traffic, looking for abnormal or interesting events. Such a tool could be integrated with the SNMP-based monitoring and alerting tool. It also might be able to detect new computers being added to the network and to generate tables of such new computers for administrative purposes. Substantial work has been done in this area by Randy Buckland at North Carolina State University.

In addition to tool writing itself, a very useful project would be to use the above-mentioned tools to do a thorough study of the NCDN, including the configuration of the network, protocol descriptions and characteristics, network utilization and traffic flow at various points in the network broken down by protocol type, and so forth.

In summary, I believe the NCDN is fertile ground for interesting and practical network management research and development efforts that will be of great benefit to the North

Carolina and Internet communities in the exciting years ahead. I hope my contribution will stimulate and facilitate these efforts.

BIBLIOGRAPHY

[Cer88]   V. Cerf. IAB Recommendations for the Development of Internet Network Management Standards. RFC 1052, Internet Activities Board, April 1988.

[CFSD88] J. Case, M. Fedor, M Schoffstall, and J. Davin. The Simple Network Management Protocol. RFC 1067, University of Tennessee at Knoxville, NYSERNet, Rensellaer Polytechnic, Proteon, January 1988.

[Com88]   Douglas Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture.* Prentice Hall, 1988.

[ISO90a]   Common Management Information Protocol Specification. International Standard 9596, International Organization for Standardization/International Electrotechnical Commission, 1990. Information Technology - Open Systems Interconnection.

[ISO90b]   Common Management Information Service Definition. International Standard 9595, International Organization for Standardization/International Electrotechnical Commission, 1990. Information Technology - Open Systems Interconnection.

[ISO90c]   Specification of Abstract Syntax Notation One (ASN.1). International Standard 8824, International Organization for Standardization/International Electrotechnical Commission, 1990. Information Processing - Open Systems Interconnection.

[MR88]   K. McCloghrie and M. Rose. Management Information Base for Network Management of TCP/IP-based internets. RFC 1066, Internet Activities Board, August 1988.

[RM88]   M. Rose and K. McCloghrie. Structure and Identification of Management Information for TCP/IP-based Internets. RFC 1065, Internet Activities Board, August 1988.

[SNS88]   Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems. MIT Project Athena, March 1988.

[Tan88]   Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, 1988.

[WB89]   U. Warrier and L. Besaw. Common Management Information Services and Protocol over TCP/IP (CMOT). RFC 1095, Unisys Corporation, Hewlett-Packard, April 1989.

Appendix A

Source Code

The following pages contain the source code for this project, in the following order:

- The QUIP interactive program,

- The QUIP daemon,

- The QUIP and HIP libraries,

- The SNMP Proxy Daemon, and

- Additional library routines used by multiple programs.

MCNC is the copyright holder for all of the code in this appendix.

```c
/* quip.c - QUIP interactive program. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#include <stdio.h>
#include <strings.h>
#include <sgtty.h>
#include <ctype.h>
#include <signal.h>

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

#include "verify.h"
#include "quiplib.h"

#define RETRY_INCOMPLETE  9
#define RETRY_NORESPONSE  1

#define QUIP_PORT 802

#define interactive (isatty(0) && isatty(1))

/* Return a socket connected to the specified service
 * on the specified host.
 */
int connect_tcp(host, service)
     char *host;
     char *service;
{
  vsn("connect_tcp");
  int sock;
  struct sockaddr_in sockaddr;
  struct hostent *hostent;
  struct servent *servent;
```

```
    vn(sock = socket(PF_INET, SOCK_STREAM, 0));
    sockaddr.sin_family = AF_INET;
    vt(hostent = gethostbyname(host));

    bcopy(hostent->h_addr, &(sockaddr.sin_addr.s_addr),
          hostent->h_length);
    servent = getservbyname(service, "tcp");
    if (servent) sockaddr.sin_port = servent->s_port;
    else sockaddr.sin_port = htons(QUIP_PORT);
    vn(connect(sock, &sockaddr, sizeof(sockaddr)));

    return(sock);
}

/* Inform user of usage of QUIP program. */
usage()
{
    fprintf(stderr, "usage: %s: [-t quiptab] node\n",
            command_name);
    exit(1);
}

/* QUIP interactive program main routine. */
int main(argc, argv)
      int argc;
      char *argv[];
{
    char *quiptab_file = NULL;
    char *node;
    char query[1024], *response, *p;
    char prompt[64];
    struct quipmsg msgout, msgin;
    int sock, i;
    int count_incomplete, count_noresponse;
    int connection = 1, node_echos = 0;
    int subcommands;
    struct quiptab_entry *entry;
    struct tchars tchars;
    char t_brkc_save;

    command_name = "quip";
    if (argc > 0) command_name = argv[0];

    for (i = 1; i < argc && argv[i][0] == '-'; i++)
      {
        switch (argv[i][1])
          {
          case 't':
            if (i > argc - 2) usage();
            quiptab_file = argv[++i];
            break;
          default:
            usage();
          }
      }
```

```c
if (i != argc - 1) usage();
node = argv[i];

if (!(entry = get_quiptab_entry(node, quiptab_file)))
  {
    fprintf(stderr, "Unknown node: (%s).\n", node);
    exit(1);
  }

if ((sock = connect_tcp(entry->server, "quip")) < 0)
  {
    fprintf(stderr, "Cannot connect to quip server (%s).\n",
            entry->server);
    exit(1);
  }

strcpy(prompt, "");
msgout.options = 0;
if (entry->major_type == NT_IB)
  {
    connection = 0;
    sprintf(prompt, "%s ib# ", node);
  }
else if (entry->major_type == NT_IBX)
  {
    if (interactive) node_echos = 1;
  }
else if (entry->major_type == NT_TRANSLAN)
  {
    if (interactive)
      {
        ioctl(0, TIOCGETC, &tchars);
        t_brkc_save = tchars.t_brkc;
        tchars.t_brkc = '\033';
        ioctl(0, TIOCSETC, &tchars);
      }
  }

if (connection) msgout.options |= QO_CONNECTION;
msgout.nodetype = entry->major_type;
strcpy(msgout.address, entry->address);

signal(SIGPIPE, SIG_IGN);
```

```
if (entry->major_type == NT_IBX) strcpy(query, "@\n");
else strcpy(query, "\n");
while (1)
   {
      for (p = query, subcommands = 0; *p; p++)
         {
            if (*p == ';' || *p == '\n')
               {
                  *p = '\n';
                  subcommands++;
               }
            if (entry->major_type == NT_TRANSLAN && *p == '\n')
               *p = '\r';
         }
      if (connection)
         msgout.timeout = 60 * 60;
      else
         msgout.timeout = 5 * subcommands;
```

```
count_incomplete = 0;
count_noresponse = 0;
do
    {
      msgout.id = 0;
      if ((send_quip(sock, query, &msgout))
            || recv_quip(sock, &response, &msgin)
            || !response)
        {
          fprintf(stderr,
                   "Connection with quip server (%s) \
dropped.\n",
                   entry->server);
          exit(1);
        }
      if (msgin.errors & ER_INCOMPLETE) count_incomplete++;
      else if (msgin.errors & ER_NORESPONSE)
        count_noresponse++;
    } while (!connection
            && (msgin.errors == ER_INCOMPLETE
                || msgin.errors == ER_NORESPONSE)
            && count_incomplete < RETRY_INCOMPLETE + 1
            && count_noresponse < RETRY_NORESPONSE + 1);
  if (msgin.errors & ER_INCOMPLETE)
    fprintf(stderr,
             "Error: Incomplete response received.\n");
  else if (msgin.errors & ER_NORESPONSE)
    fprintf(stderr, "Error: No response\n");
  else if (msgin.errors & ER_CONNECT)
    {
      fprintf(stderr, "Error: Cannot connect to node \
(%s).\n",
                node);
      goto done;
    }
  else if (msgin.errors)
    fprintf(stderr, "Error: Request failed.\n");
  if (msgin.errors & !interactive) exit(1);
  if (!msgin.errors)
    {
      if (node_echos && (p = index(response, '\n'))) p++;
      else p = response;
      printf("%s", p);
      free(response);
    }
  if (interactive) printf("%s", prompt);
  fflush(stdout);
```

```
      for (p = query;
           (int) (*p = fgetc(stdin)) != EOF
           && *p != '\n'
           && (tchars.t_brkc == (char) '\377'
               || *p != tchars.t_brkc);
           p++);
      *(++p) = '\0';
      if (feof(stdin) || ferror(stdin)) break;
   };

done:

   if (entry->major_type == NT_TRANSLAN)
      {
        if (interactive)
           {
             tchars.t_brkc = t_brkc_save;
             ioctl(0, TIOCSETC, &tchars);
           }
      }

   if (ferror(stdin)) exit(1);
   else exit(0);
}
```

```c
/* quipd.h - QUIP Daemon data structures and definitions */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#include "quiplib.h"
#include "bufferlib.h"
#include "drivertab.h"
#include <sys/time.h>
#include <netinet/in.h>

struct request
{
  int fd;
  struct quipmsg *msg;
  struct buffer *in, *out;
  u_short id;
  u_short cmdno;
  u_short seq;
  long timeout;
  struct request *prev, *next;
};

struct client
{
  struct sockaddr_in address;
};

struct proc
{
  u_char nodetype;
  u_char flags;
  int connect_fd;
  char address[MAX_ADDR_LEN];
  long lastuse;
  long timeout;
  struct request *requests;
};
```

```
struct fdtab
{
  u_char type;
  union
    {
      struct client client;
      struct proc proc;
    } state;
};

/* FD types */
#define FD_LISTEN        0
#define FD_CLIENT        1
#define FD_NODE          2
#define FD_AVAIL         255

/* Driver Routine Indexes */
#define DR_OPEN          0
#define DR_SEND          1
#define DR_RECV          2
#define DR_CLOSE         3

/* Proc flags */
#define PF_CONNECTION    (1 << 0)

/* Receive flags */
#define RC_COMPLETE      (1 << 0)
#define RC_INCOMPLETE    (1 << 1)
#define RC_FAIL          (1 << 2)

/* Global Data Structures */
extern int dumping;

/* Help routines for drivers. */
extern int dump();
```

```c
/* quipd.c - QUIP Daemon. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>

#include <stdio.h>
#include <netdb.h>
#include <strings.h>
#include <signal.h>
#include <errno.h>

#include "quipd.h"
#include "verify.h"

#define BLOCKSIZE 512
#define DEFAULT_TIMEOUT 5
#define CLOSE_TIMEOUT 60
#define DUMPFILE "quipdump"

/* Data Structures */

int nfds;
long unique_id;
fd_set fdset_in;
struct fdtab *fdtab = NULL;
FILE *dumpfile = NULL;

int listen_input(), client_input(), node_input();

int (*input[])() =
{
  listen_input,
  client_input,
  node_input,
};

/* Utilities */
```

```c
/* Get a socket connected to the named TCP service. */
static int tcp_service(name)
     char *name;
{
  vsn("tcp_service");
  int sock;
  struct servent *servent;
  struct sockaddr_in sockaddr_in;

  vn(sock = socket(AF_INET, SOCK_STREAM, 0));

  vt(servent = getservbyname(name, "tcp"));

  sockaddr_in.sin_family = AF_INET;
  sockaddr_in.sin_addr.s_addr = INADDR_ANY;
  sockaddr_in.sin_port = servent->s_port;
  vf(bind(sock, (struct sockaddr *) &sockaddr_in,
          sizeof(sockaddr_in)));
  listen(sock, 5);

  return(sock);
}

/* Get the next fd set in bitmask. */
static int nextfd(fdset) /* Inefficient */
     fd_set *fdset;
{
  int i;

  for (i = 0; i < nfds; i++)
    if (FD_ISSET(i, fdset))
      {
        FD_CLR(i, fdset);
        return(i);
      }
  return(-1);
}

/* Functions to be called for each major fd type */

/* Accept a connection. */
static int listen_input(listen_fd)
     int listen_fd;
{
  int fd;
  struct sockaddr_in from_addr;
  int addrlen = sizeof(from_addr);
```

```c
        fd = accept(listen_fd, &from_addr, &addrlen);
        if (fd < 0) perror("accept");
        else
            {
               fdtab[fd].type = FD_CLIENT;
               FD_SET(fd, &fdset_in);
            }
        return(0);
}

/* Get the next line from a command buffer. */
static char *next_line(buf)
        struct buffer *buf;
{
   vsp("next_line", char *);
   char *p, *q;
   int len;

   if ((p = index(buf->current, '\n'))) p++;
   else p = buf->end;

   len = p - buf->current;
   vt(q = (char *) malloc(len + 1));

   bcopy(buf->current, q, len);
   q[len] = '\0';
   buf->current = p;

   return(q);
}

/* Send a command to an IN. */
static int send_command(fd)
        int fd;
{
   vsn("send_command");
   char *p;
   struct proc *proc;
   struct request *request;

   proc = &(fdtab[fd].state.proc);
   request = proc->requests->next;

   vt(p = next_line(request->in));
   request->cmdno++;

   vf((*drivertab[proc->nodetype][DR_SEND])(fd, p, request));
   free(p);

   return(0);
}
```

```c
/* Close fd associated with an IN. */;
static int close_node_fd(node_fd)
     int node_fd;
{
  vsn("close_node_fd");
  struct proc *proc;

  proc = &(fdtab[node_fd].state.proc);
  vf((*drivertab[proc->nodetype][DR_CLOSE])(node_fd));
  while (proc->requests->next != proc->requests)
    request_dequeue(node_fd, proc->requests->next);

  free(proc->requests);
  FD_CLR(node_fd, &fdset_in);
  fdtab[node_fd].type = FD_AVAIL;

  return(0);
}

/* Close a connection to a client. */
static int close_client(client_fd)
     int client_fd;
{
  vsn("close_client");
  int i;
  struct request *requests, *first, *r;
  struct proc *proc;

  for (i = 0; i < nfds; i++)
    {
      if (fdtab[i].type == FD_NODE)
        {
          proc = &(fdtab[i].state.proc);
          if ((proc->flags & PF_CONNECTION)
              && proc->connect_fd == client_fd)
            vf(close_node_fd(i));
          else
            {
              requests = proc->requests;
              first = requests->next;
              for (r = first; r != requests; r = r->next)
                if (r->fd == client_fd)
                  {
                    r = r->prev;
                    request_dequeue(i, r->next);
                  }
              if (requests->next != first
                  && requests->next != requests)
                vf(send_command(i));
            }
        }
    }
  FD_CLR(client_fd, &fdset_in);
  close(client_fd);
```

```c
    return(0);
}

/* Return an initialized quip request. */
static struct request *new_request(client_fd, msg, query)
     int client_fd;
     struct quipmsg *msg;
     char *query;
{
  vsp("new_request", struct request *);
  struct request *request;
  struct timeval now;

  vt(request = (struct request *)
     malloc(sizeof(struct request)));
  request->fd = client_fd;
  request->msg = msg;
  request->in = buffer_new_string(query);
  request->in->end--;
  request->out = buffer_new(1);
  request->id = unique_id++;
  request->cmdno = 0;
  request->seq = 0;
  gettimeofday(&now, NULL);
  if (msg->timeout)
     request->timeout = now.tv_sec + msg->timeout;
  else request->timeout = now.tv_sec + DEFAULT_TIMEOUT;
  request->prev = NULL;
  request->next = NULL;

  return(request);
}

/* Initialize a new IN connection. */
static int node_fd_init(fd, request)
     int fd;
     struct request *request;
{
  vsn("node_fd_init");
  struct request *requests;
  struct proc *proc;
  struct quipmsg *msg;
```

```
      msg = request->msg;
      fdtab[fd].type = FD_NODE;
      proc = &(fdtab[fd].state.proc);
      proc->nodetype = msg->nodetype;
      strcpy(proc->address, msg->address);
      proc->flags = 0;
      if (msg->options & QO_CONNECTION)
        {
          proc->flags |= PF_CONNECTION;
          proc->connect_fd = request->fd;
        }
      else proc->connect_fd = -1;
      vt(requests = (struct request *)
         malloc(sizeof(struct request)));
      requests->prev = requests->next = requests;
      proc->requests = requests;
      proc->timeout = 0;
      FD_SET(fd, &fdset_in);

      return(0);
}

/* Find appropriate socket to IN for a new request. */
static int node_fd_find(request)
      struct request *request;
{
  vsn("node_fd_find");
  int fd;
  struct proc *proc;
  struct quipmsg *msg;

  msg = request->msg;
  if (msg->nodetype > NDRIVERS) return(-1);
  for (fd = 0; fd < nfds; fd++)
    if (fdtab[fd].type == FD_NODE)
      {
        proc = &(fdtab[fd].state.proc);
        if (proc->nodetype == msg->nodetype
            && !strcmp(proc->address, msg->address)
            && (!(msg->options & QO_CONNECTION)
                == !(proc->flags & PF_CONNECTION)))
          if (msg->options & QO_CONNECTION)
            if (proc->connect_fd == request->fd) break;
            else;
          else break;
      }

  if (fd >= nfds)
    {
      fd = (*drivertab[msg->nodetype][DR_OPEN])
        (msg->address, !!(msg->options & QO_CONNECTION));
      if (fd < 0) return(-1);
      vf(node_fd_init(fd, request));
    }
```

112

```c
      return(fd);
}

/* Enqueue a new request. */
static int request_enqueue(fd, request)
      int fd;
      struct request *request;
{
  struct request *requests;
  struct proc *proc;

  proc = &(fdtab[fd].state.proc);
  requests = proc->requests;
  request->prev = requests->prev;
  request->next = requests;
  requests->prev->next = request;
  requests->prev = request;
  if (!proc->timeout || proc->timeout > request->timeout)
    proc->timeout = request->timeout;

  return(0);
}

/* Discard a QUIP request. */
static void request_free(request)
      struct request *request;
{
  free(request->msg);
  buffer_free(request->in);
  buffer_free(request->out);
  free(request);
}

/* Handle a client QUIP request arrival event. */
static int client_input(client_fd)
      int client_fd;
{
  vsn("client_input");
  struct quipmsg *msg;
  struct request *request;
  char *query;
  int rc, node_fd;

  vt(msg = (struct quipmsg *) malloc(sizeof(struct quipmsg)));
  rc = recv_quip(client_fd, &query, msg);
  if (rc < 0 || !query)
    {
      free(msg);
      vf(close_client(client_fd));
      return(0);
    }
```

```c
vt(request = new_request(client_fd, msg, query));
if ((node_fd = node_fd_find(request)) < 0)
    {
      request_free(request);
      if (send_quip_error(request->fd, ER_CONNECT, msg))
        fprintf(stderr, "send_quip_error() failed\n");
    }
else
    {
      request_enqueue(node_fd, request);
      if (fdtab[node_fd].state.proc.requests->next == request)
        if (send_command(node_fd))
          fprintf(stderr, "send_command() failed");
    }

return(0);
}

/* Dequeue a QUIP request. */
int request_dequeue(fd, request)
      int fd;
      struct request *request;
{
  struct request *requests, *r;
  long rtimeout, timeout;
  struct timeval now;

  rtimeout = request->timeout;

  request->prev->next = request->next;
  request->next->prev = request->prev;
  request_free(request);
```

114

```c
        requests = fdtab[fd].state.proc.requests;
        timeout = fdtab[fd].state.proc.timeout;
        if (timeout == rtimeout)
            {
            if (requests->next == requests)
                {
                if (fdtab[fd].state.proc.flags & PF_CONNECTION)
                    timeout = 0;
                else
                    {
                    gettimeofday(&now, NULL);
                    timeout = now.tv_sec + CLOSE_TIMEOUT;
                    }
                }
            else
                {
                timeout = requests->next->timeout;
                for (r = requests->next->next;
                        r != requests;
                        r = r->next)
                    if (r->timeout < timeout) timeout = r->timeout;
                }
            fdtab[fd].state.proc.timeout = timeout;
            }

    return(0);
}

/* Handle an IN response arrival event. */
static int node_input(fd)
        int fd;
{
    vsn("node_input");
    struct proc *proc;
    struct request *requests, *request;
    char buffer[BLOCKSIZE];
    char *response;
    int flags;
    int rc;
    struct quipmsg msg;

    proc = &(fdtab[fd].state.proc);
    requests = proc->requests;
    request = requests->next;
```

```c
    if ((rc = read(fd, buffer, sizeof(buffer))) < 0)
       {
         perror("node_input read");
         vf(close_node_fd(fd));
         return(0);
       }
    if (rc == 0)
       {
         fprintf(stderr, "EOF on read, fd %d, address %s\n",
                 fd, proc->address);
         vf(close_node_fd(fd));
         return(0);
       }

    if (request == requests) return(0);

    if ((*drivertab[proc->nodetype][DR_RECV])
        (request, buffer, rc, &response, &flags))
       {
         fprintf(stderr, "Call to recv procedure %d failed.\n",
                 proc->nodetype);
         vf(close_node_fd(fd));
         return(0);
       }

    request->seq++;

    if (flags & RC_INCOMPLETE)
       {
         if (send_quip_error(request->fd,
                             ER_INCOMPLETE, request->msg))
           {
             vf(close_client(request->fd));
             fprintf(stderr, "send_quip_error() failed\n");
             free(response);
             return(0);
           }
         request_dequeue(fd, request);
       }
    else if (flags & RC_FAIL)
       {
         if (send_quip_error(request->fd, ER_FAIL, request->msg))
           {
             vf(close_client(request->fd));
             fprintf(stderr, "send_quip_error() failed\n");
             free(response);
             return(0);
           }
         request_dequeue(fd, request);
       }
    else
       {
         buffer_append_string(request->out, response);
         free(response);
       }
```

```
    if (flags & RC_COMPLETE
        && request->in->current >= request->in->end)
        {
          msg = *(request->msg);
          msg.options = msg.timeout = msg.errors = 0;
          if (send_quip(request->fd, request->out->start, &msg))
            {
              vf(close_client(request->fd));
              fprintf(stderr, "send_quip() failed\n");
              return(0);
            }
          request_dequeue(fd, request);
        }

    if (flags && requests->next != requests)
      if (send_command(fd))
        {
          fprintf(stderr, "send_command() failed\n");
          return(0);
        }

    return(0);
}

/* Major routines */

static int exitint()
{
  exit(1);
}

/* Initialize QUIP daemon. */
static int initialize()
{
  vsn("initialize");
  int fd;
  int i;
  struct timeval timeval;

  command_name = "quipd";
  signal(SIGPIPE, SIG_IGN);
  vn(fd = tcp_service("quip"));
  FD_ZERO(&fdset_in);
  nfds = getdtablesize();
  vt(fdtab = (struct fdtab *)
     malloc(sizeof(struct fdtab) * nfds));
  for (i = 0; i < nfds; i++) fdtab[i].type = FD_AVAIL;
  fdtab[fd].type = FD_LISTEN;
  FD_SET(fd, &fdset_in);
  vf(gettimeofday(&timeval, NULL));
  unique_id = timeval.tv_usec;

  signal(SIGHUP, exitint);

  return(0);
}
```

```c
/* Handle a timeout associated with specified fd. */
int timeout_fd(fd, now)
      int fd;
      long now;
{
  vsn("timeout_fd");
  struct request *requests, *first, *r;
  long rtimeout;

  requests = fdtab[fd].state.proc.requests;

  if (!(fdtab[fd].state.proc.flags & PF_CONNECTION)
      && requests == requests->next) vf(close_node_fd(fd));
  else
    {
      first = requests->next;
      for (r = first; r != requests; r = r->next)
        if (r->timeout <= now)
            {
              if (r->out->start == r->out->current)
                if (send_quip_error(r->fd, ER_NORESPONSE, r->msg))
                  fprintf(stderr, "send_quip_error() failed\n");
                else;
              else
                if (send_quip_error(r->fd, ER_INCOMPLETE, r->msg))
                  fprintf(stderr, "send_quip_error() failed\n");
              r = r->prev;
              request_dequeue(fd, r->next);
            }
      if (requests->next != first && requests->next != requests)
        vf(send_command(fd));
    }

  return(0);
}

/* Find the next pending timeout. */
static struct timeval *next_timeout()
{
  vsp("next_timeout", struct timeval *);
  int fd;
  struct timeval now;
  static struct timeval next;
  long timeout = 0;

  next.tv_usec = 0;
  next.tv_sec = 0;
  gettimeofday(&now, NULL);
```

```
    for (fd = 0; fd < nfds; fd++)
      {
        if (fdtab[fd].type == FD_NODE
            && fdtab[fd].state.proc.timeout)
          if (fdtab[fd].state.proc.timeout <= now.tv_sec)
            vf(timeout_fd(fd, now.tv_sec));
          else
            {
              timeout = fdtab[fd].state.proc.timeout - now.tv_sec;
              if (!next.tv_sec || timeout < next.tv_sec)
                next.tv_sec = timeout;
            }
      }

    if (!next.tv_sec) return(NULL);
    else return(&next);
}

/* QUIP daemon main loop. */
static int mainloop()
{
  vsn("mainloop");
  struct timeval *timeout;
  int fd, rc;
  fd_set readfds, readyfds;

  while (1)
    {
      timeout = next_timeout();
      FD_COPY(&readfds, &fdset_in);
      vt((rc = select(nfds, &readfds, NULL, NULL, timeout)) >= 0
          || errno == EINTR);
      if (!rc || (rc < 0 && errno == EINTR)) continue;
      FD_COPY(&readyfds, &readfds);
      while ((fd = nextfd(&readyfds)) >= 0)
        if (fdtab[fd].type == FD_NODE)
          vf((*input[fdtab[fd].type])(fd));
      FD_COPY(&readyfds, &readfds);
      while ((fd = nextfd(&readyfds)) >= 0)
        if (fdtab[fd].type == FD_CLIENT)
          vf((*input[fdtab[fd].type])(fd));
      FD_COPY(&readyfds, &readfds);
      while ((fd = nextfd(&readyfds)) >= 0)
        if (fdtab[fd].type == FD_LISTEN)
          vf((*input[fdtab[fd].type])(fd));
    }
}

/* QUIP daemon main routine. */
int main(argc, argv)
      int argc;
      char *argv[];
{
  vsn("main");
```

```
    vf(initialize());
    vf(mainloop());
    vm("Exited (this should never happen)\n");
}
```

```c
/* drivertab.h - Include file for QUIP daemon driver table. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#define NDRIVERS 3

extern int (*drivertab[NDRIVERS][4])();

int ib_open(), ib_send(), ib_recv(), ib_close();
int ibx_open(), ibx_send(), ibx_recv(), ibx_close();
int translan_open(), translan_send(), translan_recv(),
    translan_close();
```

```c
/* drivertab.c - QUIP daemon driver table. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#include <stdio.h>
#include "drivertab.h"

int (*(drivertab[][4]))() =
{
#ifdef IB
   {ib_open, ib_send, ib_recv, ib_close},
#else IB
   {NULL, NULL, NULL, NULL},
#endif IB
#ifdef IBX
   {ibx_open, ibx_send, ibx_recv, ibx_close},
#else IBX
   {NULL, NULL, NULL, NULL},
#endif IBX
#ifdef TRANSLAN
   {translan_open, translan_send, translan_recv, translan_close},
#else TRANSLAN
   {NULL, NULL, NULL, NULL},
#endif TRANSLAN
};
```

```c
/* ib_driver.c - QUIP daemon IB driver. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#ifndef vax
#undef IB
#endif

#ifdef IB

#include <sys/types.h>

#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/idp.h>

#include <stdio.h>
#include <ctype.h>
#include <strings.h>

#include "verify.h"
#include "quipd.h"

/* System buffers for response datagrams */
#define XNS_BUFS        100

/* Max datagram size */
#define MAX_PKT_SIZE    512

/* IB protocol number */
#define IB_PROTO        0x25

/* XNS net address used by IBs */
#define IB_NETWORK      "00.00.00.00"

/* XNS port address used by IBs */
#define IB_PORT         "00.4b"

#define HEADER_LENGTHS \
   (sizeof(struct idp) + sizeof(struct ib_header))
#define MAX_DATA_SIZE   (MAX_PKT_SIZE - HEADER_LENGTHS)
```

123

```c
#define SEQ(control)      (ntohs(control) >> 4)
#define MORE(control)     (ntohs(control) & 0x2)

struct ib_header
{
  u_short id;
  u_short cmdno;
  u_short control;
};

struct ib_packet
{
  struct ib_header header;
  u_char data[MAX_DATA_SIZE];
};

struct idp_packet
{
  struct idp header;
  struct ib_packet ib_packet;
};

/* Returns a pointer to a sockaddr_ns struct if bridge_addr \
 legal,
 * else NULL */
static struct sockaddr_ns *xns_address(ib_addr)
     char *ib_addr;
{
  vsp("xns_address", struct sockaddr_ns *);
  /* Matches bridge address.
   * %c makes sure there's nothing left over. */
  char *pat =
    "%2x%*[:. ]%2x%*[:. ]%2x%*[:. ]%2x%*[:. ]%2x%*[:. ]%2x%c";
  int nbytes;
  int b1, b2, b3, b4, b5, b6, c;
  /* 00.00.00.00#08.00.02.00.00.00#00.4b */
  static char address[12 + 18 + 6];
  struct ns_addr ns_addr();
  struct sockaddr_ns *sockaddr;

  vt((nbytes =
      sscanf(ib_addr, pat, &b1, &b2, &b3, &b4, &b5, &b6, &c)) \
== 6);

  sprintf(address, "%s#%02x.%02x.%02x.%02x.%02x.%02x#%s",
          IB_NETWORK, b1, b2, b3, b4, b5, b6, IB_PORT);

  vt(sockaddr = (struct sockaddr_ns *)
     malloc(sizeof(struct sockaddr_ns)));
  sockaddr->sns_family = AF_NS;
  sockaddr->sns_addr = ns_addr(address);

  return(sockaddr);
}
```

124

```
int ib_open(address, connection)
     char *address;
     int connection;
{
  vsn("ib_open");
  int sock;
  struct idp header;
  int rcvbuf = XNS_BUFS * MAX_PKT_SIZE;
  char on = 1, off = 0;
  struct sockaddr_ns local_sockaddr, *remote_sockaddr;

  vn(sock = socket(PF_NS, SOCK_DGRAM, 0));
  header.idp_pt = IB_PROTO;

  vf(setsockopt(sock, NSPROTO_PE, SO_DEFAULT_HEADERS,
                (char *) &header, sizeof(header)));
  vf(setsockopt(sock, NSPROTO_PE, SO_HEADERS_ON_INPUT,
                (char *) &on, sizeof(on)));
  vf(setsockopt(sock, SOL_SOCKET, SO_RCVBUF,
                (char *) &rcvbuf, sizeof(rcvbuf)));

  local_sockaddr.sns_family = AF_NS;
  bzero(&local_sockaddr.sns_addr, sizeof(struct ns_addr));
  vf(bind(sock, &local_sockaddr, sizeof(struct sockaddr_ns)));

  vt(remote_sockaddr = xns_address(address));
  vf(connect(sock,
             remote_sockaddr, sizeof(struct sockaddr_ns)));
  free(remote_sockaddr);

  return(sock);
}

int ib_send(fd, cmd, request)
     int fd;
     char *cmd;
     struct request *request;
{
  vsn("send_command");
  struct ib_packet to_pkt;
  int cmd_len, pkt_size;

  to_pkt.header.id = htons(request->id);
  to_pkt.header.cmdno = htons(request->cmdno);
  to_pkt.header.control = htons(0);
  request->seq = 0;

  cmd_len = strlen(cmd);
  if (cmd[cmd_len] == '\n') cmd_len--;
  bcopy(cmd, to_pkt.data, cmd_len);
  pkt_size = sizeof(struct ib_header) + cmd_len;

  vt(send(fd, (char *) &to_pkt, pkt_size, 0) == pkt_size);

  return(0);
}
```

```
static char *response_format(data, datalen)
     char *data;
     int datalen;
{
  vsp("response_format", char *);
  char *response, *p, *q, *r;
  int response_len, i;

  /* Remove extra trailing nulls
   * IB bug puts them there on long queries */
  for (p = data + datalen;
       p >= data + 2 && !*(p - 1) && !*(p - 2);
       p--);

  /* Allocate response string */
  for (q = data, response_len = 1; q < p; q++)
    if (!isascii(*q)) response_len += toascii(*q);
    else response_len++;
  vt(response = (char *) malloc(response_len));

  /* Convert response to string */
  for (q = data, r = response; q < p; q++)
    if (!isascii(*q))
      for (i = toascii(*q); i; i--) *(r++) = ' ';
    else if (!*q) *(r++) = '\n';
    else *(r++) = *q;
  *r = '\0';

  return(response);
}

int ib_recv(request, buffer, buflen, response, flags)
     struct request *request;
     char *buffer;
     int buflen;
     char **response;
     int *flags;
{
  vsn("ib_recv");
  struct idp_packet *idp_packet;
  struct ib_packet *ib_packet;
  int packet_id;

  idp_packet = (struct idp_packet *) buffer;
  vt(idp_packet->header.idp_pt == IB_PROTO);
  ib_packet = &idp_packet->ib_packet;

  vt(*response = (char *) malloc(1)); /* In case of error */
  strcpy(*response, "");
  *flags = 0;

  packet_id = ntohs(ib_packet->header.id);
  if (request->id != packet_id) return(0);

  vt(ntohs(ib_packet->header.cmdno) == request->cmdno);
```

126

```
    if (SEQ(ib_packet->header.control) != request->seq)
      {
        *flags |= RC_INCOMPLETE;
        return(0);
      }

  free(*response);
  vt(*response =
      response_format(ib_packet->data,
                          ntohs(idp_packet->header.idp_len)
                          - HEADER_LENGTHS));
  if (!MORE(ib_packet->header.control)) *flags |= RC_COMPLETE;

  return(0);
}

ib_close(fd)
      int fd;
{
  close(fd);
}

#endif IB
```

```
/* ibx_driver.c - QUIP daemon IBX driver. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#ifdef IBX

#include <sys/file.h>

#include <ctype.h>
#include <stdio.h>
#include <sgtty.h>
#include <strings.h>

#include "verify.h"
#include "quipd.h"

#define MMICONNECT "/dev/mmi?"
#define MAXMMIS       10

/* useconds to pause between chars sent */
#define IF_CHAR_DELAY         4

/* useconds to pause after prompt */
#define IF_PROMPT_PAUSE   125000

#define PROMPT " => "

/* MMI file name pattern for connections */
static char *mmi_connect = MMICONNECT;

/* fd associated with an open mmi */
static int mmi_fd[MAXMMIS];

/* Is mmi_fd initialized ? */
static int initialized = 0;
```

```
int ibx_open(address, connection)
     char *address;
     int connection;
{
  vsn("ibx_open");
  int fd;
  struct sgttyb sgttyb;
  int i;
  char *p;

#ifdef lint
  address = address;
#endif lint

  if (!initialized) {
    for (i = 0; i < MAXMMIS; i++) mmi_fd[i] = -1;
    initialized = 1;
  }
  p = &mmi_connect[strlen(mmi_connect) - 1];
  for (i = 0; i < MAXMMIS; i++)
    {
      if (mmi_fd[i] < 0)
        {
          *p = '0' + i;
          if ((fd = open(mmi_connect, O_RDWR)) >= 0)
            {
              mmi_fd[i] = fd;
              break;
            }
        }
    }
  if (i >= MAXMMIS) return(-1);
  vf(ioctl(fd, TIOCGETP, &sgttyb));
  sgttyb.sg_flags &= ~(ODDP|ECHO|CRMOD|RAW);
  sgttyb.sg_flags |= (EVENP|TANDEM|CBREAK);
  vf(ioctl(fd, TIOCSETP, &sgttyb));

  return(fd);
}

static char *cmdtranslate(string)
     char *string;
{
  vsp("cmdtranslate", char *);
  char *result, *p, *q;

  vt(result = (char *) malloc(strlen(string) + 1));

  for (p = string, q = result; *p; p++, q++)
    if (islower(*p)) *q = toupper(*p);
    else if (*p == '\n') *q = '\r';
    else *q = *p;
  *q = *p;

  return(result);
}
```

```
int ibx_send(fd, str, request)
     int fd;
     char *str;
     struct request *request;
{
  vsn("ibx_send");
  char *cmdstr, *p;

#ifdef lint
  request = request;
#endif lint

  vt(cmdstr = cmdtranslate(str));
  for (p = cmdstr; *p; p++)
    {
      vt(write(fd, p, 1) == 1);
      usleep(IF_CHAR_DELAY);
    }

  free(cmdstr);
  return(0);
}

/* get rid of nulls in string, and null terminate */
static char *denull(string, length)
     char *string;
     int length;
{
  vsp("denull", char *);
  char *result, *p, *q;

  vt(result = (char *) malloc(length + 1));

  for (p = string, q = result; length > 0; p++, length--)
    if (*p) *(q++) = *p;
  *q = '\0';

  return(result);
}

static int prompt(line)
     char *line;
{
  if (strlen(line) >= strlen(PROMPT)
      && !strcmp(line + strlen(line) - strlen(PROMPT), PROMPT))
    return(1);
  return(0);
}
```

```
  int ibx_recv(request, buffer, buflen, response, flags)
       struct request *request;
       char *buffer;
       int buflen;
       char **response;
       int *flags;
{
  vsn("ibx_recv");
  static struct buffer *lastline = NULL;
  char *newline;

#ifdef lint
  request = request;
#endif lint

  if (!lastline)
     {
       vt(lastline = (struct buffer *)
          malloc(sizeof(struct buffer)));
       vt(lastline->current = lastline->start =
          (char *) malloc(1));
       lastline->end = lastline->start + 1;
     }

  *flags = 0;

  vt(*response = denull(buffer, buflen));
  if (newline = rindex(*response, '\n'))
     {
       lastline->current = lastline->start;
       buffer_append_string(lastline, newline + 1);
     }
  else buffer_append_string(lastline, *response);

  if (prompt(lastline->start))
     {
       *flags |= RC_COMPLETE;
       usleep(IF_PROMPT_PAUSE);
     }

  return(0);
}

ibx_close(fd)
     int fd;
{
  int i;

  close(fd);
  for (i = 0; i < MAXMMIS; i++)
    if (mmi_fd[i] == fd) mmi_fd[i] = -1;

  return(0);
}

#endif IBX
```

```c
/* translan_driver.c - QUIP daemon Translan driver. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#ifdef TRANSLAN

#include "types.h"

#include <sys/socket.h>
#include <sys/time.h>
#include <netns/ns.h>
#include <netns/idp.h>
#include <netns/sp.h>
#include <netns/spidp.h>

#include <stdio.h>
#include <ctype.h>
#include <strings.h>
#include <signal.h>

#include "verify.h"
#include "quipd.h"

/* Maximum size of entire SPP datagram
 * including IDP and SPP headers */
#define MAX_DGRAM_SIZE   576

/* Maximum size of datagram without idp header
 * bytes read and written */
#define MAX_PKT_SIZE     (MAX_DGRAM_SIZE - sizeof(struct idp))

/* Maximum size of data portion of packet */
#define MAX_DATA_SIZE    (MAX_PKT_SIZE - sizeof(struct spidp))

/* Seconds to wait for open to finish */
#define OPEN_TIMEOUT     4

/* XNS net address used by translans */
#define XNS_NETWORK      "00.00.00.00"
```

```
/* Translan SPP service port */
#define XNS_PORT          "00.05"

struct packet
{
   struct sphdr sphdr;
   char buf[MAX_DATA_SIZE];
};

static int sigalrm()
{
   return(0);
}

/* Returns a pointer to a sockaddr_ns struct if bridge_addr \
 legal,
    else NULL */
static struct sockaddr_ns *xns_address(addr)
      char *addr;
{
   vsp("xns_address", struct sockaddr_ns *);
   /* Matches bridge address.
    * %c makes sure there's nothing left over. */
   char *pat =
     "%2x%*[:. ]%2x%*[:. ]%2x%*[:. ]%2x%*[:. ]%2x%*[:. ]%2x%c";
   int nbytes;
   int b1, b2, b3, b4, b5, b6, c;
   /* 00.00.00.00#08.00.02.00.00.00#00.00 */
   static char address[12 + 18 + 6];
   struct ns_addr ns_addr();
   struct sockaddr_ns *sockaddr;

   vt((nbytes = sscanf(addr, pat, &b1, &b2, &b3, &b4, &b5, \
&b6, &c))
      == 6);

   sprintf(address, "%s#%02x.%02x.%02x.%02x.%02x.%02x#%s",
           XNS_NETWORK, b1, b2, b3, b4, b5, b6, XNS_PORT);

   vt(sockaddr = (struct sockaddr_ns *)
      malloc(sizeof(struct sockaddr_ns)));
   sockaddr->sns_family = AF_NS;
   sockaddr->sns_addr = ns_addr(address);

   return(sockaddr);
}

static void hexcode(buf, str)
      char *buf;
      char *str;
{
   char hexbyte;
   int ls, lb, is, ib;
   int rc;
```

```
    ls = strlen(str);
    lb = ls/2;

    for (is = 0, ib = 0; is < ls; is++, ib++)
        {
           if (str[is] >= '0' &&  str[is] <= '9')
              buf[ib] = (str[is] - '0') * 16;
           else buf[ib] = (10 + str[is] - 'A') * 16;
           is++;
           if (str[is] >= '0' &&  str[is] <= '9') buf[ib] += \
  str[is] - '0';
           else buf[ib] += 10 + str[is] - 'A';
        }
}

#define MAGIC1 "00030003"
#define MAGIC2       "000000000000\
0003000200020000BFFFFFFFFFFFFFFFF\
0001000000000000FFFF000100010000\
00050000000000000010080FFFF0001\
0000"

int translan_open(address, connection)
     char *address;
     int connection;
{
  vsn("translan_open");
  int sock;
  struct sockaddr_ns local_sockaddr, *remote_sockaddr;
  struct packet packet;
  static struct itimerval open_timeout =
     {{0, 0}, {OPEN_TIMEOUT, 0}};
  static struct itimerval disable_timeout = {{0, 0}, {0, 0}};
  fd_set fdmask;

  vn(sock = socket(PF_NS, SOCK_SEQPACKET, 0));

  local_sockaddr.sns_family = AF_NS;
  bzero(&local_sockaddr.sns_addr, sizeof(struct ns_addr));
  vf(bind(sock, &local_sockaddr, sizeof(struct sockaddr_ns)));

  vt(remote_sockaddr = xns_address(address));

  vn(signal(SIGALRM, sigalrm));
  vf(setitimer(ITIMER_REAL, &open_timeout));
  if (connect(sock, remote_sockaddr, sizeof(struct sockaddr_ns))
       < 0)
     goto abort;
  vf(setitimer(ITIMER_REAL, &disable_timeout));

  packet.sphdr.sp_cc = 1;
  packet.sphdr.sp_dt = 0;
```

```
    hexcode(packet.buf, MAGIC1);
     if (write(sock, &packet, sizeof(packet.sphdr) + \
    strlen(MAGIC1)/2)
         < 0)
       goto abort;

     hexcode(packet.buf, MAGIC2);
      if (write(sock, &packet, sizeof(packet.sphdr) + \
    strlen(MAGIC2)/2)
         < 0)
       goto abort;

     FD_ZERO(&fdmask);
     FD_SET(sock, &fdmask);

      if (select(sock + 1, &fdmask, NULL, NULL, \
    &open_timeout.it_value)
         <= 0)
       goto abort;
      if (read(sock, &packet, MAX_PKT_SIZE) <= 0) goto abort;

      if (select(sock + 1, &fdmask, NULL, NULL, \
    &open_timeout.it_value)
         <= 0)
       goto abort;
      if (read(sock, &packet, MAX_PKT_SIZE) <= 0) goto abort;

      if (select(sock + 1, &fdmask, NULL, NULL, \
    &open_timeout.it_value)
         <= 0)
       goto abort;
      if (read(sock, &packet, MAX_PKT_SIZE) <= 0) goto abort;

     return(sock);

   abort:

     free(remote_sockaddr);
     close(sock);
     return(-1);
   }

   int translan_send(fd, str, request)
        int fd;
        char *str;
        struct request *request;
   {
     vsn("send_command");
     struct packet packet;
     char *p;

     packet.sphdr.sp_cc = 0;
     packet.sphdr.sp_dt = 192;
```

```
    for (p = str; *p; p++)
      {
        *(packet.buf) = *p;
        vn(write(fd, &packet, sizeof(packet.sphdr) + 1));
      }

  return(0);
}

int translan_recv(request, buffer, buflen, response, flags)
      struct request *request;
      char *buffer;
      int buflen;
      char **response;
      int *flags;
{
  vsn("translan_recv");
  struct packet *packet;
  char *p;
  int len;

  packet = (struct packet *) buffer;

  len = buflen - sizeof(struct sphdr);
  vt(*response = (char *) malloc(len + 1));
  strncpy(*response, packet->buf, len);
  (*response)[len] = '\0';

  p = *response + len - 2;
  if (!strcmp(p, "> ") || !strcmp(p, ": "))
    *flags = RC_COMPLETE;
  else *flags = 0;

  return(0);
}

translan_close(fd)
      int fd;
{
  close(fd);
}

#endif TRANSLAN
```

```c
/* quiplib.h - Data structures and definitions
 *    for QUIP library. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#include "types.h"

#define MAX_ADDR_LEN 32

/* Query Options */

/* Connection oriented, keep state */
#define QO_CONNECTION    (1 << 0)

/* Node types */
#define NT_IB          0
#define NT_IBX         1
#define NT_TRANSLAN    2

/* Error flags on response */

/* Improperly formulated query */
#define ER_BADQUERY    (1 << 0)

/* Unknown node type or address */
#define ER_UNKNOWN     (1 << 1)

/* No response from node */
#define ER_NORESPONSE  (1 << 2)

/* Incomplete response from node */
#define ER_INCOMPLETE  (1 << 3)

/* Can't connect to node */
#define ER_CONNECT     (1 << 4)

/* Failed for other reason */
#define ER_FAIL        (1 << 5)
```

```c
/* Structure for holding quip table entry */
struct quiptab_entry
{
  int major_type;
  int minor_type;
  char *name;
  char *address;
  char *server;
};

/* Structure for passing queries and responses
 * over a TCP/IP link */
struct quipmsg
{
  /* E.g., ethernet address or LANmark port # */
  char address[MAX_ADDR_LEN];

  /* Message identification */
  long id;

  /* E.g., save state for next query */
  long options;

  /* IB, Vitalink, LANmark, etc. */
  long nodetype;

  /* Seconds to wait for response */
  long timeout;

  /* Error flags, e.g., no response */
  long errors;

  /* Length of data string to follow */
  long datalen;

};

extern struct quiptab_entry *get_quiptab_entry();
extern int send_quip();
extern int recv_quip();
extern int send_quip_error();
extern int parse_quip();
```

```
/* quiplib.c - QUIP message library and HIP parser. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#include <sys/time.h>

#include <stdio.h>
#include <strings.h>
#include <ctype.h>

#include "quiplib.h"
#include "bufferlib.h"
#include "verify.h"

#include "memorylib.h"

#define DFLT_QUIP_TAB    "/etc/quiptab"
#define MAX_TAB_ENTRY    31
#define MAX_TAB_LINE     256

static long unique_id = 0;

struct node_type_table
{
  char *name;
  int number;
};

struct node_type_table node_type_table[] =
{
  {"ib",        NT_IB},
  {"ibx",       NT_IBX},
  {"translan",  NT_TRANSLAN}
};

/* Returns numeric node type encoding of specified type. */
static int get_node_type(type)
     char *type;
{
  int i;
```

```c
    for (i = 0;
         i < sizeof(node_type_table)/sizeof(*node_type_table);
         i++)
      if (!strcmp(type, node_type_table[i].name))
        return(node_type_table[i].number);
    return(-1);
}

/* Gets quiptab entry for specified IN. */
struct quiptab_entry *get_quiptab_entry(nameoraddr, filename)
      char *nameoraddr, *filename;
{
  vsp("get_quip_table", struct quiptab_entry *);
  FILE *quipfile = NULL, *fopen();
  char *p;
  char line[MAX_TAB_LINE + 1];
  char name[MAX_TAB_ENTRY + 1];
  char major_type[MAX_TAB_ENTRY + 1];
  char minor_type[MAX_TAB_ENTRY + 1];
  char address[MAX_TAB_ENTRY + 1];
  char server[MAX_TAB_ENTRY + 1];
  int rc;
  int nodetype;
  struct quiptab_entry *entry = NULL;
  int found = 0;

  if (!filename) filename = DFLT_QUIP_TAB;
  quipfile = fopen(filename, "r");
  if (!quipfile)
    {
      fprintf(stderr, "Can't open quip table (%s).\n",
              filename);
      goto abort;
    }

  while (!ferror(quipfile) && !feof(quipfile) && !found)
    {
      fgets(line, sizeof(line), quipfile);
      line[MAX_TAB_LINE] = '\0';
      if (p = index(line, '#')) *p = '\0';
      rc = sscanf(line, " %31s %31s %31s %31s %31s",
                  name, major_type, minor_type,
                  address, server);
      if (rc != 5) continue;
      if (!strcmp(nameoraddr, name) || !strcmp(nameoraddr, \
address))
        found = 1;
    }
```

140

```c
    if (!found)
        {
        if (ferror(quipfile)) perror(filename);
        goto abort;
        }
    if ((nodetype = get_node_type(major_type)) < 0)
        {
        fprintf(stderr, "Quip table unknown node type (%s):\n%s",
                major_type, line);
        goto abort;
        }

    vt(entry = (struct quiptab_entry *)
        malloc(sizeof(struct quiptab_entry)));
    vt(entry->name = (char *) malloc(strlen(name) + 1));
    strcpy(entry->name, name);
    entry->major_type = nodetype;
    entry->minor_type = atoi(minor_type);
    vt(entry->address = (char *) malloc(strlen(address) + 1));
    strcpy(entry->address, address);
    vt(entry->server = (char *) malloc(strlen(server) + 1));
    strcpy(entry->server, server);

  abort:

    if (quipfile) fclose(quipfile);
    return(entry);
}

/* Send a QUIP message. */
/* I copy the data to avoid two writes.
   This saves about 170ms of waiting time on the receive \
 side. */
int send_quip(sock, msg, header)
        int sock;
        char *msg;
        struct quipmsg *header;
{
  vsn("send_quip");
  struct timeval timeval;
  struct quipmsg *outmsg;
  char *buffer;
  int length;

  if (!unique_id)
     {
       vf(gettimeofday(&timeval, NULL));
       unique_id = timeval.tv_usec;
     }
```

141

```
        if (!header->id) header->id = unique_id++;
        header->datalen = strlen(msg) + 1;
        length = header->datalen + sizeof(struct quipmsg);
        vt(buffer = (char *) malloc(length));
        bcopy(header, buffer, sizeof(struct quipmsg));
        outmsg = (struct quipmsg *) buffer;
        outmsg->id = htonl(outmsg->id);
        outmsg->options = htonl(outmsg->options);
        outmsg->nodetype = htonl(outmsg->nodetype);
        outmsg->timeout = htonl(outmsg->timeout);
        outmsg->errors = htonl(outmsg->errors);
        outmsg->datalen = htonl(outmsg->datalen);
        bcopy(msg, buffer + sizeof(struct quipmsg), header->datalen);

        vt(write(sock, buffer, length));
        free(buffer);

        return(0);
}

/* Receive a QUIP message. */
int recv_quip(sock, msg, header)
        int sock;
        char **msg;
        struct quipmsg *header;
{
    vsn("recv_quip");
    char *p;
    int rc;
    int index;

    *msg = NULL;
    header->errors = 0;
    vt(!(rc = read(sock, header, sizeof(struct quipmsg)))
        || rc == sizeof(struct quipmsg));

    if (!rc) return(0);

    header->id = ntohl(header->id);
    header->options = ntohl(header->options);
    header->nodetype = ntohl(header->nodetype);
    header->timeout = ntohl(header->timeout);
    header->errors = ntohl(header->errors);
    header->datalen = ntohl(header->datalen);

    vt(*msg = (char *)
        malloc(header->datalen ? header->datalen : 1));
    if (!header->datalen) **msg = '\0';

    for (p = *msg; p - *msg < header->datalen; p += rc)
        vn(rc = read(sock, p, header->datalen - (p - *msg)));

    return(0);
}
```

142

```c
/* Send a QUIP error message. */
int send_quip_error(sock, error, header)
     int sock, error;
     struct quipmsg *header;
{
  vsn("send_quip_error");
  struct quipmsg outmsg;

  outmsg = *header;
  outmsg.options = outmsg.timeout = 0;
  outmsg.errors = error;
  vf(send_quip(sock, "", &outmsg));

  return(0);
}

/* Discard quiptab entry. */
void free_quiptab_entry(quiptab_entry)
     struct quiptab_entry *quiptab_entry;
{
  free(quiptab_entry->name);
  free(quiptab_entry->address);
  free(quiptab_entry->server);
  free(quiptab_entry);
}

/* HIP parser portion */

#define OPENPARENS      "(`{[<"
#define CLOSEPARENS     ")'}]>"
#define REPEATS         "*+"
#define ZEROPLUS        (-'*')
#define ONEPLUS         (-'+')
#define ETHADRSEP       " :."
#define BLOCKSIZE       2048

struct quip_structure
{
  char **pointers;
};

typedef int (*intfunct)();

struct parsetab
{
  char *name;
  intfunct funct;
};

static int parse_paren(), parse_line(), parse_line_consec();
static int parse_lit(), parse_int(), parse_str();
static int parse_inttok(), parse_hexint(), parse_hexstr();
static int parse_keystr(), parse_ethadr();
```

```
static struct parsetab parsetab[] =
{
  {"(",          parse_paren},
  {"{",          parse_line},
  {"[",          parse_line_consec},
  {"\",          parse_lit},
  {"int",        parse_int},
  {"str",        parse_str},
  {"inttok",     parse_inttok},
  {"hexint",     parse_hexint},
  {"hexstr",     parse_hexstr},
  {"keystr",     parse_keystr},
  {"ethadr",     parse_ethadr}
};

static char **keystr;
static int nkeystrs = 0, keystrsize = 0;

/* Find matching grouping character. */
static char *balance(p)
     char *p;
{
  int depth = 0;
  char *q = p;

  if (!index(OPENPARENS, *p)) return(NULL);
  do
    {
      if (*q == '\\')
        if (! *(++q)) return(NULL);
        else q++;
      if (! *q) return(NULL);
      if (index(OPENPARENS, *q)) depth++;
      else if (index(CLOSEPARENS, *q)) depth--;
      q++;
    } while (depth);
  if (index(OPENPARENS, *p) - OPENPARENS
       != index(CLOSEPARENS, *(q - 1)) - CLOSEPARENS)
    return(NULL);
  return(q);
}

/* Make a new copy of a string */
static char *strcopy(l, r)
     char *l, *r;
{
  char *s;

  s = (char *) malloc(r - l + 1);
  if (!s) return(NULL);
  strncpy(s, l, r - l);
  s[r - l] = '\0';
  return(s);
}
```

```c
/* Skip to a character from the character set. */
static char *skipto(s, charset)
     char *s, *charset;
{
  while (*s && !index(charset, *s)) s++;
  return(s);
}

/* Parse a parenthesized expression. */
static int parse_paren(arguments, in, out, inaline, pointers)
     char *arguments;
     struct buffer *in, *out;
     int inaline;
     struct buffer *pointers;
{
  return(parse2(arguments, in, out, inaline, pointers));
}

/* Parse stuff that must all be in one line. */
static int parse_line(arguments, in, out, inaline, pointers)
     char *arguments;
     struct buffer *in, *out;
     int inaline;
     struct buffer *pointers;
{
  int rc;
  struct buffer newin, newout;

  newin = *in;
  newout = *out;

  do
    {
      rc = parse2(arguments, &newin, &newout, 1, pointers);
      if (rc)
        {
          newin.current = skipto(newin.current, "\n");
          if (*newin.current) newin.current++;
        }
    } while (rc && *newin.current);

  *in = newin;
  *out = newout;
  return(rc);
}

/* Parse stuff that must all be in this line or next one. */
static int parse_line_consec(arguments, in, out,
                                inaline, pointers)
     char *arguments;
     struct buffer *in, *out;
     int inaline;
     struct buffer *pointers;
{
  int rc;
  struct buffer newin, newout;
```

```
      newin = *in;
      newout = *out;

      do
         {
           rc = parse2(arguments, &newin, &newout, 1, pointers);
           if (rc)
              {
                newin.current = skipto(newin.current, "\n");
                if (*newin.current) newin.current++;
                rc = parse2(arguments, &newin, &newout, 1, pointers);
                if (rc) return(-1);
              }
         } while (rc && *newin.current);

      *in = newin;
      *out = newout;
      return(rc);
}

/* Parse a literal string. */
static int parse_lit(arguments, in, out, inaline, pointers)
      char *arguments;
      struct buffer *in, *out;
      int inaline;
      struct buffer *pointers;
{
   struct buffer newin, newout;

   newin = *in;
   newout = *out;

   if (!strlen(arguments)) return(0);
   while (*newin.current)
      {
        for (;
              *newin.current
              && !(inaline && *newin.current == '\n')
              && *newin.current != arguments[0];
              newin.current++);
        if (*newin.current != arguments[0]) return(-1);
        if (!strncmp(newin.current, arguments, strlen(arguments)))
           {
             newin.current += strlen(arguments);
             *in = newin;
             *out = newout;
             return(0);
           }
        else newin.current++;
      }

   return(-1);
}
```

146

```
/* Parse a positive integer. */
static int parse_int(arguments, in, out, inaline, pointers)
     char *arguments;
     struct buffer *in, *out;
     int inaline;
     struct buffer *pointers;
{
  long i;
  struct buffer newin, newout;

  newin = *in;
  newout = *out;

  for (;
       *newin.current && !(inaline && *newin.current == '\n')
       && !isdigit(*newin.current);
       newin.current++);
  if (!isdigit(*newin.current)) return(-1);
  i = *newin.current - '0';
  for (newin.current++;
       isdigit(*newin.current);
       newin.current++)
    i = i * 10 + *newin.current - '0';

  buffer_append_long(&newout, i);

  *in = newin;
  *out = newout;
  return(0);
}

/* Parse a whitespace terminated string. */
static int parse_str(arguments, in, out, inaline, pointers)
     char *arguments;
     struct buffer *in, *out;
     int inaline;
     struct buffer *pointers;
{
  struct buffer newin, newout;
  char *p, *s;

  newin = *in;
  newout = *out;

  for (;
       *newin.current && isspace(*newin.current)
       && !(inaline && *newin.current == '\n');
       newin.current++);
  if (!*newin.current || isspace(*newin.current)) return(-1);

  for (p = newin.current;
       *p && !isspace(*p) && !index(arguments, *p);
       p++);
  s = strcopy(newin.current, p);
  if (!s) return(-1);
  newin.current = p;
```

```
    buffer_append_pointer(&newout, s);
    buffer_append_pointer(pointers, s);

    *in = newin;
    *out = newout;
    return(0);
}

/* Parse an integer or token. */
static int parse_inttok(arguments, in, out, inaline, pointers)
    char *arguments;
    struct buffer *in, *out;
    int inaline;
    struct buffer *pointers;
{
  long t;
  int i;
  struct buffer newin, newout;

  newin = *in;
  newout = *out;

  for (;
        *newin.current && !(inaline && *newin.current == '\n')
        && !(isalnum(*newin.current)
            || index(arguments, *newin.current));
        newin.current++);
  if (!*newin.current
      || !(isalnum(*newin.current)
          || index(arguments, *newin.current)))
    return(-1);

  if (isdigit(*newin.current))
      {
        *in = newin;
        return(parse_int(arguments, in, out, inaline));
      }

  for (i = 0, t = 0;
       i < sizeof(long) - 1 && *newin.current
       && (isalnum(*newin.current)
           || index(arguments, *newin.current));
       i++, newin.current++)
    t = (t << 8) | *newin.current;
  t = -t;

  for (;
        *newin.current
        && (isalnum(*newin.current)
            || index(arguments, *newin.current));
        newin.current++);

  buffer_append_long(&newout, t);
```

```c
        *in = newin;
        *out = newout;
        return(0);
}

/* Parse a hexidecimal digit. */
static int hexdigit(c)
        char c;
{
  return(isalpha(c) ?
          (isupper(c) ? tolower(c) : c) - 'a' + 10 : c - '0');
}

/* Parse a hexidecimal integer. */
static int parse_hexint(arguments, in, out,  inaline, pointers)
        char *arguments;
        struct buffer *in, *out;
        int inaline;
        struct buffer *pointers;
{
  long i;
  struct buffer newin, newout;

  newin = *in;
  newout = *out;

  for (;
        *newin.current && !(inaline && *newin.current == '\n')
        && !isxdigit(*newin.current);
        newin.current++);
  if (!isxdigit(*newin.current)) return(-1);
  i = hexdigit(*newin.current);

  for (newin.current++;
        isxdigit(*newin.current);
        newin.current++)
    i = i * 16 + hexdigit(*newin.current);

  buffer_append_long(&newout, i);

  *in = newin;
  *out = newout;
  return(0);
}

/* Parse a hexidecimal string. */
static int parse_hexstr(arguments, in, out, inaline, pointers)
        char *arguments;
        struct buffer *in, *out;
        int inaline;
        struct buffer *pointers;
{
  struct buffer newin, newout, subout;

  newin = *in;
  newout = *out;
```

```
        for (;
            *newin.current && *(newin.current + 1)
            && !(inaline && *newin.current == '\n')
            && !(isxdigit(*newin.current)
                && isxdigit(*(newin.current + 1)));
            newin.current++);
    if (!*newin.current
        || !(isxdigit(*newin.current)
            && isxdigit(*(newin.current + 1))))
        return(-1);

    subout.start = (char *) malloc(BLOCKSIZE);
    if (!subout.start) return(-1);
    subout.current = subout.start;
    subout.end = subout.start + BLOCKSIZE;

    while (*newin.current
            && isxdigit(*(newin.current))
            && isxdigit(*(newin.current + 1))
            && subout.current < subout.end)
        {
          *(subout.current) = hexdigit(*(newin.current++)) << 4;
          *(subout.current++) |= hexdigit(*(newin.current++));
        }
    if (subout.current >= subout.end) return(-1);
    buffer_append_long(&newout,
                        (long) (subout.current - subout.start));
    if (!(subout.start = (char *)
            realloc(subout.start, subout.current - subout.start)))
      return(-1);

    buffer_append_pointer(&newout, subout.start);

    *in = newin;
    *out = newout;
    return(0);
}

/* Parse a string, folding equal strings. */
static int parse_keystr(arguments, in, out, inaline, pointers)
      char *arguments;
      struct buffer *in, *out;
      int inaline;
      struct buffer *pointers;
{
  struct buffer newin, newout;
  char *p, *s;
  int i;

  newin = *in;
  newout = *out;
```

150

```
            for (;
                 *newin.current && isspace(*newin.current)
                 && !(inaline && *newin.current == '\n');
                 newin.current++);
            if (!*newin.current || isspace(*newin.current)) return(-1);

            for (p = newin.current; *p && !isspace(*p); p++);
            s = strcopy(newin.current, p);
            if (!s) return(-1);
            newin.current = p;

            for (i = 0; i < nkeystrs; i++)
              if (!strcmp(s, keystr[i]))
                {
                  free(s);
                  s = keystr[i];
                  break;
                };
            if (i >= nkeystrs)
              {
                if (nkeystrs == 0)
                  {
                    keystrsize = 128;
                    keystr = (char **)
                      malloc(keystrsize * sizeof(char *));
                    if (!keystr) return(-1);
                  }
                else if (nkeystrs >= keystrsize)
                  {
                    keystrsize *= 2;
                    keystr = (char **)
                      realloc((char *) keystr, keystrsize * sizeof(char \
    *));
                    if (!keystr) return(-1);
                  }
                keystr[nkeystrs++] = s;
              }

      buffer_append_pointer(&newout, s);
      buffer_append_pointer(pointers, s);

      *in = newin;
      *out = newout;
      return(0);
    }

/* Parse an Ethernet address. */
static int parse_ethadr(arguments, in, out,  inaline, pointers)
      char *arguments;
      struct buffer *in, *out;
      int inaline;
      struct buffer *pointers;
{
  struct buffer newin, newout;
  u_char address[6];
  int i;
```

```c
        newin = *in;
        newout = *out;

        do
            {
            for (;
                    *newin.current
                    && !(inaline && *newin.current == '\n')
                    && !isxdigit(*newin.current);
                    newin.current++);
            if (!isxdigit(*newin.current)) return(-1);

            for (i = 0; i < 6 && isxdigit(*newin.current); i++)
                {
                if (isxdigit(*(newin.current + 1)))
                    {
                    address[i] =
                        (hexdigit(*newin.current) << 4)
                            | hexdigit(*(newin.current + 1));
                    newin.current += 2;
                    }
                else address[i] = hexdigit(*(newin.current++));
                if (*newin.current
                    && index(ETHADRSEP, *newin.current))
                  newin.current++;
                }
            } while (i < 6);

        if (newout.end - newout.current < 6) return(-1);
        bcopy(address, newout.current, 6);
        newout.current += 6;

        *in = newin;
        *out = newout;
        return(0);
}

/* Find the parse routine associated with an operator. */
static intfunct parse_routine(type)
        char *type;
{
    int i;

    for (i = 0; i < sizeof(parsetab)/sizeof(*parsetab); i++)
        if (!strcmp(type, parsetab[i].name))
            return(parsetab[i].funct);
    return(NULL);
}

/* Break down the next operand specification. */
static int get_command(grammar, repeats, type, arguments)
        char **grammar, **type, **arguments;
        int *repeats;
{
    char *p;
```

152

```c
    if (index(REPEATS, **grammar)) *repeats = -(*((*grammar)++));
    else if (isdigit(**grammar))
       {
         *repeats = **grammar - '0';
         while (isdigit(*(++(*grammar))))
           *repeats = *repeats * 10 + **grammar - '0';
       }
    else *repeats = 1;

    if (!**grammar) *type = strcopy(*grammar, *grammar);
    else if (index(OPENPARENS, **grammar))
       {
         *type = strcopy(*grammar, *grammar + 1);
         if (!*type) return(-1);
       }
    else
       {
         p = skipto(*grammar, ": ");
         *type = strcopy(*grammar, p);
         if (!*type) return(-1);
         *grammar = p;
       }

    if (!**grammar) *arguments = strcopy(*grammar, *grammar);
    else if (index(OPENPARENS, **grammar))
       {
         p = balance(*grammar);
         if (!p) return(-1);
         *arguments = strcopy(*grammar + 1, p - 1);
         *grammar = p;
       }
    else
       {
         if (**grammar == ':') (*grammar)++;
         p = skipto(*grammar, " ");
         *arguments = strcopy(*grammar, p);
         *grammar = p;
       }

    while (**grammar && **grammar == ' ') (*grammar)++;

    return(0);
}

/* Low-level parse routine. */
static int parse2(grammar, in, out, inaline, pointers)
     char *grammar;
     struct buffer *in, *out;
     int inaline;
     struct buffer *pointers;
{
  int repeats, count, i;
  char *type, *arguments;
  intfunct pr;
  struct buffer newin, newout;
  struct buffer *subout;
```

153

```
      newin = *in;
      newout = *out;

      while (*grammar)
        {
          if (get_command(&grammar, &repeats, &type, &arguments))
            return(-1);

          pr = parse_routine(type);
          if (!pr) return(-1);

          if (repeats == ZEROPLUS || repeats == ONEPLUS)
            {
              subout = buffer_new(1);

              for (count = 0;
                   !(*pr)(arguments, &newin,
                          subout, inaline, pointers);
                   count++);

              if (repeats == ONEPLUS && count == 0)
                {
                  buffer_free(subout);
                  return(-1);
                }

              buffer_fit(subout);

              buffer_append_long(&newout, count);
              buffer_append_pointer(&newout, subout->start);
              buffer_append_pointer(pointers, subout->start);
              free(subout);
            }
          else
            for (i = 0; i < repeats; i++)
              if ((*pr)(arguments, &newin,
                        &newout, inaline, pointers)) return(-1);

          free(type);
          free(arguments);
        }

    *in = newin;
    *out = newout;
    return(0);
}

/* Free structure built by HIP parser. */
void hip_free(structure)
      struct quip_structure *structure;
{
  char **p;
```

154

```
    for (p = structure->pointers; *p; p++) free(*p);
    free(structure->pointers);
    free(structure);
}

/* Main HIP parser routine. */
int parse_quip(grammar, input, output)
    char *grammar, **input, **output;
{
    struct buffer in, out;
    int rc;
    struct buffer *pointers;

    in.start = *input;
    in.current = *input;
    in.end = *input + strlen(*input);

    pointers = buffer_new(1);

    out.start = (char *) malloc(BLOCKSIZE);
    if (!out.start) return(-1);
    out.current = out.start;
    out.end = out.start + BLOCKSIZE;

    buffer_append_pointer(&out, NULL);
    rc = parse2(grammar, &in, &out, 0, pointers);
    buffer_append_pointer(pointers, NULL);
    buffer_fit(pointers);
    ((struct quip_structure *) out.start)->pointers = (char **)
        pointers->start;
    if (rc)
        {
            hip_free(out.start);
            out.start = NULL;
            if (rc) return(-1);
        }
    else
        if (!(out.start = (char *)
                realloc(out.start, out.current - out.start)))
            return(-1);

    *output = out.start;
    *input = in.current;
    return(0);
}
```

```c
/* snmp_proxyd.h - Data structure definitions for SNMP
 *    proxy daemon.
 */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#define SECOND  1
#define MINUTE  60
#define HOUR    (MINUTE * 60)
#define DAY     (HOUR * 24)
#define MONTH   (DAY * 31)
#define YEAR    (MONTH * 12)

#define REQUEST_TIMEOUT 60 /* Forget un-filled requests after
                              this many seconds */

struct request
{
   struct snmp_info snmp_info;
   struct variable snmp_vars[NUMVARS];
   struct sockaddr_in from_addr;
   long timestamp;
   struct request *prev, *next;
   struct request *global_prev, *global_next;
};

struct varnode
{
   int num;
   int (*retrieve)();
   long lifetime;
   long deadline;
   int pending;
   struct variable *variable;
   struct varnode *child, *sibling;
   struct sockaddr_in *receivers;
   int nreceivers;
   int receiverlistsize;
};
```

```
struct node
{
  struct quiptab_entry *quiptab_entry;
  int quip_fd;
  struct request *requestlist;
  struct varnode *vartree;
  struct node *prev, *next;
};

struct quip_queue
{
  long id;
  int (*parse)();
  struct node *node;
  struct quip_queue *next, *prev;
};

extern struct node *nodelist;
extern struct quip_queue *quip_queue;

int cache_branch();
int queue_quip();
int variable_insert();
```

```c
/* snmp_proxyd.c - SNMP Proxy Daemon. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#include "types.h"

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

#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <ctype.h>
#include <signal.h>
#include <strings.h>

#include <snmp.h>
#include <snmp_impl.h>
#include <snmp_client.h>
#include <asn1.h>

#include "verify.h"
#include "quiplib.h"
#include "snmp_proxyd.h"

#define MAXPKTSIZE      2048
#define MAXLINESIZE     512
#define MIB             ".iso.org.dod.internet.mgmt.mib."
#define VARIABLE_FILE   "/etc/snmp.variables"

struct varmap
{
  char *sym;
  u_char num;
  struct varmap *child, *sibling;
};
```

```c
int dump_packet = 0;
int terminate_flag = 0;
int nfds, snmp_fd;
fd_set fdset_in;
struct node *nodelist;
struct varmap *varmap;
struct quip_queue *quip_queue;
struct request *global_requests;
int pid;

#ifdef IB
int build_var_tree_ib();
#endif IB
#ifdef IBX
int build_var_tree_ibx();
#endif IBX
#ifdef TRANSLAN
int build_var_tree_translan();
#endif TRANSLAN

int (*build_var_tree[])() =
{
#ifdef IB
  build_var_tree_ib,
#else IB
  NULL,
#endif IB
#ifdef IBX
  build_var_tree_ibx,
#else IBX
  NULL,
#endif IBX
#ifdef TRANSLAN
  build_var_tree_translan
#else TRANSLAN
  NULL
#endif TRANSLAN
};

/* Get next fd set in bitmask. */
static int nextfd(fdset) /* Inefficient */
     fd_set *fdset;
{
  int i;

  for (i = 0; i < nfds; i++)
    if (FD_ISSET(i, fdset))
      {
        FD_CLR(i, fdset);
        return(i);
      }
  return(-1);
}
```

```
/* Get socket "connected" to named UDP service. */
int udp_service(name)
     char *name;
{
  vsn("udp-service");

  int sock;
  struct servent *servent;
  struct sockaddr_in sockaddr_in;

  vn(sock = socket(AF_INET, SOCK_DGRAM, 0));
  vt(servent = getservbyname(name, "udp"));

  sockaddr_in.sin_family = AF_INET;
  sockaddr_in.sin_addr.s_addr = INADDR_ANY;
  sockaddr_in.sin_port = servent->s_port;
  vf(bind(sock,
          (struct sockaddr *) &sockaddr_in, \
 sizeof(sockaddr_in)));

  return(sock);
}

/* Get socket connected to named TCP service. */
int connect_tcp(host, service)
     char *host;
     char *service;
{
  vsn("connect_tcp");
  int sock;
  struct sockaddr_in sockaddr;
  struct hostent *hostent;
  struct servent *servent;

  vn(sock = socket(PF_INET, SOCK_STREAM, 0));
  sockaddr.sin_family = AF_INET;
  vt(hostent = gethostbyname(host));

  bcopy(hostent->h_addr, &(sockaddr.sin_addr.s_addr),
        hostent->h_length);
  vt(servent = getservbyname(service, "tcp"));
  sockaddr.sin_port = servent->s_port;
  if (connect(sock, &sockaddr, sizeof(sockaddr)) < 0)
    {
      close(sock);
      return(-1);
    }
  else return(sock);
}
```

```
/* Queue information about a pending quip request. */
int queue_quip(msgid, parse, node)
        long msgid;
        int (*parse)();
        struct node *node;
{
  vsn("queue_quip");
  struct quip_queue *quipnode;

  vt(quipnode = (struct quip_queue *)
      malloc(sizeof(struct quip_queue)));
  quipnode->id = msgid;
  quipnode->parse = parse;
  quipnode->node = node;
  quipnode->next = quip_queue;
  quipnode->prev = quip_queue->prev;
  quip_queue->prev->next = quipnode;
  quip_queue->prev = quipnode;

  return(0);
}

/* Insert a variable in the variable translation map. */
int varmap_insert(varmap, def)
        struct varmap *varmap;
        char *def;
{
  vsn("varmap_insert");

  struct varmap *v, *newv;
  char *p;
  u_char num;
  char *newsym;

  if (! *def) return(0);

  vt(p = index(def, ' '));
  *p = '\0';
  num = atoi(p + 1);

  for (v = varmap->child; v; v = v->sibling)
    if (v->num == num) break;

  if (!v)
    {
      vt(newv = (struct varmap *)
          malloc(sizeof(struct varmap)));
      vt(newsym = (char *) malloc(strlen(def) + 1));
      strcpy(newsym, def);
      newv->sym = newsym;
      newv->num = num;
      newv->child = NULL;
      newv->sibling = NULL;
```

```c
        if (varmap->child)
            {
                for (v = varmap->child; v->sibling; v = v->sibling);
                v->sibling = newv;
            }
        else varmap->child = newv;
        v = newv;
    }

    vt(p = index(p + 1, ' '));
    while (*(p + 1) && *(p + 1) == ' ') p++;

    return(varmap_insert(v, p + 1));
}

/* Get variables from MIB definition file. */
int get_variables(variable_file)
{
    vsn("get_variables");
    FILE *file, *fopen();
    char buffer[MAXLINESIZE];

    vt(varmap = (struct varmap *) malloc(sizeof(struct varmap)));
    varmap->sym = NULL;
    varmap->num = 0;
    varmap->child = NULL;
    varmap->sibling = NULL;

    vt(file = fopen(variable_file, "r"));
    while (fgets(buffer, sizeof(buffer), file))
        {
            buffer[strlen(buffer) - 1] = ' ';
            vf(varmap_insert(varmap, buffer));
        }
    fclose(variable_file);

    return(0);
}

/* Initialize proxy daemon */
static int initialize()
{
    vsn("initialize");

    command_name = "snmp_proxyd";

    pid = getpid();

    nfds = getdtablesize();
    FD_ZERO(&fdset_in);

    vn(snmp_fd = udp_service("snmp"));
    FD_SET(snmp_fd, &fdset_in);

    vt(nodelist = (struct node *) malloc(sizeof(struct node)));
    nodelist->prev = nodelist->next = nodelist;
```

```
      vf(get_variables(VARIABLE_FILE));

      vt(quip_queue = (struct quip_queue *)
         malloc(sizeof(struct quip_queue)));
      quip_queue->id = 0;
      quip_queue->parse = NULL;
      quip_queue->next = quip_queue->prev = quip_queue;

      global_requests = (struct request *)
        Malloc(sizeof(struct request));
      global_requests->global_next = global_requests;
      global_requests->global_prev = global_requests;

      return(0);
}

/* Split community name into community and node name \
 portions. */
int split_community_node(request, community, node)
      struct snmp_info *request;
      char **community, **node;
{
  vsn("split_community_node");
  int i;

  for (i = 0;
       i < request->clen && request->community[i] != '.';
       i++);
  vt(*community = (char *) malloc(i + 1));
  strncpy(*community, request->community, i);
  (*community)[i] = '\0';
  if (i == request->clen)
    {
      vt(*node = (char *) malloc(1));
      strcpy(*node, "");
    }
  else
    {
      vt(*node = (char *) malloc(request->clen - i));
      strncpy(*node, request->community + i + 1,
              request->clen - i - 1);
      (*node)[request->clen - i - 1] = '\0';
    }
  return(0);
}

/* Convert symbolic variable name to numeric form. */
int symtonum(varmap, varname, varnum, varnumlen)
      struct varmap *varmap;
      char *varname;
      u_char *varnum;
      int varnumlen;
{
  vsn("symtonum");
  struct varmap *v;
  char *p, *q;
```

163

```
      if (varnumlen == 0) return(0);

      p = index(varname + 1, '.');
      if (!p) p = varname + strlen(varname);

      for (v = varmap; v; v = v->sibling)
        if (!strncmp(varname + 1, v->sym, p - varname - 1)
            && !v->sym[p - varname - 1])
          break;
      if (!v)
        {
          for (; varnumlen; varnumlen--)
            {
              for (q = varname + 1; q < p; q++)
                if (!isdigit(*q)) break;
              vt(q == p);
              *varnum  = atoi(varname + 1);
              varnum++;
              varname = p;
              p = index(varname + 1, '.');
              if (!p) p = varname + strlen(varname);
            }
          vf(varnumlen);
          return(0);
        }
      else
        {
          *varnum = v->num;
          return(symtonum(v->child, p, varnum + 1, varnumlen - 1));
        }
}

/* Insert a variable into the cache (low-level routine). */
struct variable *variable_inserter(vartree, varnum, varnumlen,
                                   asntype, value, length, \
  lifetime)
      struct varnode *vartree;
      u_char *varnum;
      int varnumlen;
      int asntype;
      char *value;
      int length;
      int lifetime;
{
  vsp("variable_inserter", struct variable *);
  struct varnode *v, *newv;
  struct timeval timeval;

  if (vartree->retrieve)
    {
      gettimeofday(&timeval, NULL);
      vartree->deadline = timeval.tv_sec + vartree->lifetime;
      vartree->pending = 0;
    }
```

```
    if (varnumlen == 0)
        {
        if (vartree->variable) free(vartree->variable);
        vt(vartree->variable =
            (struct variable *) malloc(sizeof(struct variable)));
        vartree->variable->type = asntype;
        vartree->variable->val_len = length;
        bcopy(value, &(vartree->variable->val), length);
        gettimeofday(&timeval, NULL);
        vartree->deadline = timeval.tv_sec + lifetime;
        vartree->pending = 0;
        vartree->nreceivers = 0;
        return(vartree->variable);
        }

    for (v = vartree->child; v; v = v->sibling)
        if (v->num == *varnum) break;

    if (!v)
        {
        vt(newv = (struct varnode *)
            calloc(1, sizeof(struct varnode)));
        newv->num = *varnum;

        if (vartree->child)
            {
            for (v = vartree->child; v->sibling; v = v->sibling);
            v->sibling = newv;
            }
        else vartree->child = newv;
        v = newv;
        }

    return(variable_inserter(v, varnum + 1, varnumlen - 1,
                             asntype, value, length, lifetime));
}

/* Insert a variable into the cache (high-level routine). */
int variable_insert(varname, instance, asntype,
                    value, length, lifetime, node)
        char *varname;
        int instance;
        int asntype;
        char *value;
        int length;
        int lifetime;
        struct node *node;
{
  vsn("variable_insert");
  u_char *varnum;
  int varnumlen;
  char *p;
  char *fullvarname;
  char cinstance[4];
  struct variable *v;
```

165

```c
      if (instance) sprintf(cinstance, ".%d", instance);
      else strcpy(cinstance, "");
      vt(fullvarname =
          (char *) malloc(strlen(MIB) + strlen(varname)
                          + strlen(cinstance) + 3));
      strcpy(fullvarname, MIB);
      strcat(fullvarname, varname);
      strcat(fullvarname, cinstance);
      strcat(fullvarname, ".0");

      for (p = fullvarname, varnumlen = 0; *p; p++)
        if (*p == '.') varnumlen++;
      vt(varnum = (u_char *) malloc(varnumlen));

      vf(symtonum(varmap->child, fullvarname, varnum, varnumlen));
      vt(v = variable_inserter(node->vartree, varnum, varnumlen,
                                    asntype, value, length, lifetime));
      bcopy(varnum, v->name, varnumlen);
      v->length = varnumlen;
      free(fullvarname);
      free(varnum);
      return(0);
}

/* Insert a branch into a variable tree stump (low-level \
 routine). */
int varnode_inserter(vartree, varnum,
                        varnumlen, retrieve, lifetime)
      struct varnode *vartree;
      u_char *varnum;
      int varnumlen;
      int (*retrieve)();
      long lifetime;
{
  vsn("varnode_inserter");
  struct varnode *v, *newv;

  if (varnumlen == 0)
    {
      vartree->retrieve = retrieve;
      vartree->lifetime = lifetime;
      return(0);
    }

  for (v = vartree->child; v; v = v->sibling)
    if (v->num == *varnum) break;

  if (!v)
    {
      vt(newv = (struct varnode *)
          calloc(1, sizeof(struct varnode)));
      newv->num = *varnum;
```

166

```
        if (vartree->child)
           {
               for (v = vartree->child; v->sibling; v = v->sibling);
               v->sibling = newv;
           }
        else vartree->child = newv;
        v = newv;
     }

   return(varnode_inserter(v, varnum + 1, varnumlen - 1, \
  retrieve,
                                lifetime));
}

/* Insert a branch into a variable tree stump (high-level \
  routine). */
int cache_branch(vartree, varname, retrieve, lifetime)
     struct varnode *vartree;
     char *varname;
     int (*retrieve)();
     long lifetime;
{
  vsn("cache_branch");
  u_char *varnum;
  int varnumlen;
  char *p;
  char *fullvarname;

  vt(fullvarname = (char *)
     malloc(strlen(MIB) + strlen(varname) + 1));
  strcpy(fullvarname, MIB);
  strcat(fullvarname, varname);

  for (p = fullvarname, varnumlen = 0; *p; p++)
    if (*p == '.') varnumlen++;
  vt(varnum = (u_char *) malloc(varnumlen));

  vf(symtonum(varmap->child, fullvarname, varnum, varnumlen));
  free(fullvarname);
  vf(varnode_inserter(vartree,
                       varnum, varnumlen, retrieve, lifetime));
  free(varnum);

  return(0);
}

/* Set up data structure for a new IN. */
struct node *build_node_struct(quiptab_entry)
     struct quiptab_entry *quiptab_entry;
{
  vsp("build_node_struct", struct node *);
  struct node *n, *n2;

  vt(n = (struct node *) malloc(sizeof(struct node)));
  n->quiptab_entry = quiptab_entry;
```

```c
    vt(n->requestlist = (struct request *)
       malloc(sizeof(struct request)));
    n->requestlist->prev = n->requestlist->next = n->requestlist;

    vt(n->vartree = (struct varnode *)
       calloc(1, sizeof(struct varnode)));

    for (n2 = nodelist->next; n2 != nodelist; n2 = n2->next)
      if (!strcmp(quiptab_entry->server, \
  n2->quiptab_entry->server))
        break;
    if (n2 != nodelist) n->quip_fd = n2->quip_fd;
    else
      {
        if ((n->quip_fd = connect_tcp(quiptab_entry->server, \
  "quip"))
            < 0)
        /* free things here */
          return(NULL);
        FD_SET(n->quip_fd, &fdset_in);
      }

    n->next = nodelist;
    n->prev = nodelist->prev;
    nodelist->prev->next = n;
    nodelist->prev = n;

    if (build_var_tree[quiptab_entry->major_type])
      vf((*(build_var_tree[quiptab_entry->major_type]))
         (n->vartree));

    return(n);
}

/* Throw away an SNMP request. */
int dequeue_request(request)
      struct request *request;
{
  request->prev->next = request->next;
  request->next->prev = request->prev;
  request->global_prev->global_next = request->global_next;
  request->global_next->global_prev = request->global_prev;
  free(request);

  return(0);
}

/* See if receiver has received specified variable value. */
static int find_receiver(receiver, varnode)
      struct sockaddr_in *receiver;
      struct varnode *varnode;
{
  int i;
```

```
      return 0;
      for (i = 0; i < varnode->nreceivers; i++)
        if (!bcmp(receiver, &varnode->receivers[i],
                    sizeof(struct sockaddr_in)))
          return 1;
      return 0;
}

/* Record the fact that receiver has received
   current value of this variable. */
static int put_receiver(receiver, varnode)
      struct sockaddr_in *receiver;
      struct varnode *varnode;
{
  vsn("put_receiver");

  if (varnode->nreceivers == varnode->receiverlistsize)
    if (varnode->receiverlistsize)
      vp(varnode->receivers = (struct sockaddr_in *)
          realloc(varnode->receivers,
                    (varnode->receiverlistsize *= 2)
                  * sizeof(struct sockaddr_in)));
      else vp(varnode->receivers = (struct sockaddr_in *)
              malloc((varnode->receiverlistsize = 1)
                      * sizeof(struct sockaddr_in)));
  varnode->receivers[varnode->nreceivers++] = *receiver;
  return(0);
}

/* Find variable in cache (low-level routine). */
int get_var2(variable, receiver, depth,
                vartree, status, retrieve)
      struct variable *variable;
      struct sockaddr_in *receiver;
      int depth;
      struct varnode *vartree;
      int *status;
      int (**retrieve)();
{
  vsn("get_var2");
  struct varnode *v;
  struct timeval timeval;
  int i;
```

```
        if (depth == variable->length)
          {
            if (vartree->variable == NULL) *status = 2;
            else
              {
                gettimeofday(&timeval, NULL);
                if (timeval.tv_sec > vartree->deadline
                    || find_receiver(receiver, vartree)) *status = 1;
                else
                  {
                    *status = 0;
                    variable->type = vartree->variable->type;
                    bcopy(&vartree->variable->val, &(variable->val),
                          vartree->variable->val_len);
                    variable->val_len = vartree->variable->val_len;
                    vf(put_receiver(receiver, vartree));
                  }
              }

            return(0);
          }

        for (v = vartree->child; v; v = v->sibling)
          if (v->num == variable->name[depth]) break;

        if (v)
          {
            if (v->retrieve)
              {
                *retrieve = v->retrieve;
                gettimeofday(&timeval, NULL);
                if (timeval.tv_sec > v->deadline)
                  {
                    if (v->pending) *retrieve = NULL;
                    *status = 1;
                  }
                else return(get_var2(variable, receiver,
                                     depth + 1, v, status, retrieve));
              }
            else return(get_var2(variable, receiver, depth + 1, v,
                                 status, retrieve));
          }
        else *status = 2;
        return(0);
}

/* Find variable in cache (high-level routine). */
int get_var(variable, receiver, node, status)
      struct variable *variable;
      struct sockaddr_in *receiver;
      struct node *node;
      int *status;
{
  vsn("get_var");
  int (*retrieve)() = NULL;
```

```
      vf(get_var2(variable, receiver, 0,
                  node->vartree, status, &retrieve));
      if (*status == 1 && *retrieve) (*retrieve)(node);
      return(0);
}

/* Find next variable in cache (low-level routine). */
int getnext_var2(variable, receiver, depth,
                    vartree, status, retrieve, next)
      struct variable *variable;
      struct sockaddr_in *receiver;
      int depth;
      struct varnode *vartree;
      int *status;
      int (**retrieve)();
      int *next;
{
  vsn("getnext_var2");
  struct varnode *v;
  struct timeval timeval;
```

```
  if (*next && vartree->variable)
     {
       gettimeofday(&timeval, NULL);
       if (timeval.tv_sec > vartree->deadline
           || find_receiver(receiver, vartree)) *status = 1;
       else
          {
            *status = 0;
            variable->type = vartree->variable->type;
            bcopy(vartree->variable->name, variable->name,
                  vartree->variable->length);
            variable->length = vartree->variable->length;
            bcopy(&vartree->variable->val, &(variable->val),
                  vartree->variable->val_len);
            variable->val_len = vartree->variable->val_len;
            put_receiver(receiver, vartree);
            return(0);
          }
     }
  if (depth == variable->length) *next = 1;
  if (*next) v = vartree->child;
  else
     {
       for (v = vartree->child; v; v = v->sibling)
         if (v->num >= variable->name[depth]) break;
     }
  for (; v; v = v->sibling)
     {
       if (v->retrieve)
          {
            *retrieve = v->retrieve;
            gettimeofday(&timeval, NULL);
            if (timeval.tv_sec > v->deadline)
               {
                 if (v->pending) *retrieve = NULL;
                 *status = 1;
                 return(0);
               }
          }
       vf(getnext_var2(variable, receiver, depth + 1,
                       v, status, retrieve, next));
       if (*status == 0 || *status == 1) break;
     }
  if (!v)
     {
       *status = 2;
       *next = 1;
     }
  return(0);
}
```

172

```c
/* Find next variable in cache (high-level routine). */
int getnext_var(variable, receiver, node, status)
      struct variable *variable;
      struct sockaddr_in *receiver;
      struct node *node;
      int *status;
{
  vsn("getnext_var");
  int (*retrieve)() = NULL;
  int next = 0;

  vf(getnext_var2(variable, receiver, 0,
                  node->vartree, status, &retrieve, &next));
  if (*status == 1 && *retrieve) (*retrieve)(node);
  return(0);
}

/* Fill a request from the cache if possible. */
int fill_request(request, node)
      struct request *request;
      struct node *node;
{
  vsn("fill_request");

  int status;
  u_char packet[MAXPKTSIZE];
  int pktlen = sizeof(packet);
  int fromlen = sizeof(struct sockaddr_in);
  int i, j;
  struct variable var;
  struct timeval now;
```

```c
      gettimeofday(&now, NULL);
      if (!(request->snmp_info.command == GET_REQ_MSG
            || request->snmp_info.command == GETNEXT_REQ_MSG)
          ||  request->timestamp <= now.tv_sec - REQUEST_TIMEOUT)
        {
          vf(dequeue_request(request));
          return(0);
        }
    status = 0;
    for (i = 0; i < request->snmp_info.vars; i++)
        {
          var = request->snmp_vars[i];
          if (var.name[var.length - 1])
            {
              var.name[var.length] = 0;
              var.length++;
            }
          if (request->snmp_info.command == GET_REQ_MSG)
            vf(get_var(&var, &(request->from_addr), node, &status));
          else vf(getnext_var(&var, &(request->from_addr),
                              node, &status));
          if (status == 1) return(0); /* wait for some variable */
          if (status == 2) {           /* non-existent variable */
            request->snmp_info.errstat = SNMP_ERR_NOSUCHNAME;
            request->snmp_info.errindex = i + 1;
            break;
          }

          if (request->snmp_info.command == GET_REQ_MSG)
            {
              bcopy(&(var.val), &(request->snmp_vars[i].val),
                    sizeof(var.val));
              request->snmp_vars[i].val_len = var.val_len;
              request->snmp_vars[i].type = var.type;
            }
          else
            bcopy(&var, &(request->snmp_vars[i]), sizeof(var));
        }

    request->snmp_info.command = GET_RSP_MSG;
    vf(snmp_build(packet, &pktlen,
                  &(request->snmp_info), request->snmp_vars));
    vt(sendto(snmp_fd, packet, pktlen,
              0, &(request->from_addr), fromlen));

    vf(dequeue_request(request));

    return(0);
}
```

174

```c
/* Enqueue a new SNMP request. */
int queue_request(request, node)
     struct request *request;
     char *node;
{
  vsn("queue_request");
  struct node *n;
  struct timeval now;
  struct request *r;
  struct quiptab_entry *quiptab_entry;

  for (n = nodelist->next; n != nodelist; n = n->next)
    if (!strcmp(n->quiptab_entry->name, node))
      break;
  if (n == nodelist)
    {
      quiptab_entry = get_quiptab_entry(node, NULL);
      if (!quiptab_entry) return(0);
      if (quiptab_entry->major_type
          >= sizeof(build_var_tree)/sizeof(build_var_tree[0])
          || !build_var_tree[quiptab_entry->major_type])
        return(0);

      if (!(n = build_node_struct(quiptab_entry))) return(0);
    }

  gettimeofday(&now, NULL);

  request->timestamp = now.tv_sec;

  request->next = n->requestlist;
  request->prev = n->requestlist->prev;
  n->requestlist->prev->next = request;
  n->requestlist->prev = request;

  request->global_next = global_requests;
  request->global_prev = global_requests->global_prev;
  global_requests->global_prev->global_next = request;
  global_requests->global_prev = request;

  vf(fill_request(request, n));

  return(0);
}

/* Handle SNMP request event. */
int snmp_input()
{
  vsn("snmp_input");
```

```
        struct sockaddr_in from_addr;
        int from_len = sizeof(struct sockaddr_in);
        int inbytes;
        u_char inbuffer[MAXPKTSIZE];
        struct snmp_info snmp_info;
        struct variable snmp_vars[NUMVARS];
        char *community, *node;
        struct request *request;
        int i;

        vn(inbytes = recvfrom(snmp_fd, inbuffer, MAXPKTSIZE,
                              0, &from_addr, &from_len));
        snmp_info.maxvars = NUMVARS;
        snmp_info.clen = SID_MAX_LEN;
        vf(snmp_parse(inbuffer, inbytes, &snmp_info, snmp_vars));
        vf(split_community_node(&snmp_info, &community, &node));
        free(community);
        if (!*node)
          {
            free(node);
            return(0);
          }

        vt(request = (struct request *)
           malloc(sizeof(struct request)));
        request->snmp_info = snmp_info;

        bcopy((char *) snmp_vars, (char *) request->snmp_vars,
              sizeof(snmp_vars));
        request->from_addr = from_addr;

        vf(queue_request(request, node));
        free(node);

        return(0);
}

/* Free a variable tree */
static free_vartree(varnode)
      struct varnode *varnode;
{
    struct varnode *v, *vnext;

    for (v = varnode->child; v; v = vnext)
      {
        vnext = v->sibling;
        free_vartree(v);
      }
    if (varnode->variable) free(varnode->variable);
    free(varnode);
}
```

```c
/* Free data structure associated with an IN. */
static void free_node(node)
     struct node *node;
{
   struct request *request;
   struct quip_queue *q;

   free_quiptab_entry(node->quiptab_entry);

   for (request = node->requestlist; request->next != request; )
      dequeue_request(request->next);
   free(request);

   free_vartree(node->vartree);

   for (q = quip_queue->next; q != quip_queue; q = q->next)
      if (q->node == node)
         {
            q->prev->next = q->next;
            q->next->prev = q->prev;
            q = q->prev;
            free(q);
         }

   free(node);
}

/* Close connection to a QUIP daemon. */
static int close_quip_fd(fd)
     int fd;
{
   vsn("close_quip_fd");
   struct node *n, *n_old;

   close(fd);
   FD_CLR(fd, &fdset_in);
   for (n = nodelist->next; n != nodelist; n = n->next)
      if (n->quip_fd == fd)
         {
            n->prev->next = n->next;
            n->next->prev = n->prev;
            n_old = n;
            n = n->prev;
            free_node(n_old);
         }

   return(0);
}
```

```c
/* Handle a QUIP message arrival event. */
static int quip_input(fd)
        int fd;
{
  vsn("quip_input");
  char *response;
  struct quipmsg msgin;
  struct quip_queue *q;

  vf(recv_quip(fd, &response, &msgin));
  if (!response) return(-1);

  for (q = quip_queue->next; q != quip_queue; q = q->next)
    if (q->id == msgin.id) break;
  if (q != quip_queue)
    {
      q->prev->next = q->next;
      q->next->prev = q->prev;
      if (!msgin.errors) vf((*(q->parse))(response, q->node));
      free(q);
    }
  free(response);

  return(0);
}

/* Delete unanswered SNMP requests. */
static void delete_old_requests()
{
  struct timeval deadline;

  gettimeofday(&deadline, NULL);
  deadline.tv_sec -= REQUEST_TIMEOUT;

  while (global_requests->global_next != global_requests
         && global_requests->global_next->timestamp
         < deadline.tv_sec)
    dequeue_request(global_requests->global_next);
}

/* Main event loop of SNMP proxy daemon */
static int mainloop()
{
  vsn("mainloop");
  int fd, afd, rc;
  fd_set readfds, readyfds;
  struct sockaddr_in sockaddr;
  int sockaddrsize = sizeof(struct sockaddr_in);

  while (1)
    {
      FD_COPY(&readfds, &fdset_in);
      vt((rc = select(nfds, &readfds, NULL, NULL, NULL)) >= 0
         || errno == EINTR);
      if (terminate_flag) exit(1);
      if (rc < 0 && errno == EINTR) continue;
```

```
        FD_COPY(&readyfds, &readfds);
        while ((fd = nextfd(&readyfds)) >= 0)
          if (fd != snmp_fd)
            {
              if (quip_input(fd))
                {
                  fprintf(stderr, "quip_input() failed\n");
                  close_quip_fd(fd);
                }
            }

        if (FD_ISSET(snmp_fd, &readfds)) vf(snmp_input());

        delete_old_requests();
      }
}

/* Set terminate flag so proxy daemon will quit when safe. */
int terminate()
{
  terminate_flag = 1;
}

/* Main routine for SNMP proxy daemon. */
int main(argc, argv)
      int argc;
      char *argv[];
{
  vsn("main");

  signal(SIGTERM, terminate);
  vf(initialize(argc, argv));
  vf(mainloop());
  vm("Exited (this should never happen)\n");
}
```

```c
/* ib_driver.c - SNMP Proxy Daemon IB driver. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#ifdef IB

#include <sys/types.h>
#include <stdio.h>
#include <strings.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <snmp.h>
#include <snmp_impl.h>
#include <snmp_client.h>
#include <asn1.h>

#include "verify.h"
#include "quiplib.h"
#include "snmp_proxyd.h"

/* Parse system part of MIB. */
int parse_ib_system(response, node)
     char *response;
     struct node *node;
{
  vsn("parse_ib_system");
  char *headgrammar =
    "[str:, `, booted'] [`PROMS:' str:- `-' *str]";
  struct
    {
      char **pointers;
      char *sysname;
      char *boardname;
      int nrevtoks;
      char **revtok;
    } *headstruct;
  char *resp;
  char *sysdescr;
  int i, l;
  struct request *r;
```

180

```c
resp = response;
if (parse_quip(headgrammar, &resp, (char **) &headstruct))
    return(0);

l = strlen("Bridge Communications " )
    + strlen(headstruct->boardname) + 1;
for (i = 0; i < headstruct->nrevtoks; i++)
    l += strlen(headstruct->revtok[i]) + 1;
vt(sysdescr = (char *) malloc(l));
strcpy(sysdescr, "Bridge Communications ");
strcat(sysdescr, headstruct->boardname);
for (i = 0; i < headstruct->nrevtoks; i++)
    {
    strcat(sysdescr, " ");
    strcat(sysdescr, headstruct->revtok[i]);
    }
hip_free(headstruct);

vf(variable_insert("system.sysDescr", 0,
                    ASN_OCTET_STR, sysdescr,
                    strlen(sysdescr), MINUTE, node));

for (r = node->requestlist->next;
     r != node->requestlist;
     r = r->next)
    vf(fill_request(r, node));

return(0);
}
```

```c
/* Parse interfaces part of MIB. */
int parse_ib_interfaces(response, node)
      char *response;
      struct node *node;
{
  vsn("parse_ib_interfaces");
  char *headgrammar =
    "ethadr '\n ' [str:, ', booted'] 'PROMS:' *[str:- '- ' \
 *str]";
  char *statgrammar;
  struct
     {
       char **pointers;
       u_char address[6];
       char *sysname;
       int nboards;
       struct
          {
            char *boardname;
            int nrevtoks;
            char **revtok;
          } *board;
     } *headstruct;
  struct
     {
       char **pointers;
       int rcvpkts, rcvbytes, rcvmcastpkts, rcvbcastpkts;
       int crc, frameerr, toolong, lostpkts;
       int xmtpkts, xmtbytes, xmtmcastpkts, xmtbcastpkts;
       int deferred, singlecoll, multicoll;
       int latecoll, excesscoll;
       union {
         struct
            {
              int neterr;
            } type0;
         struct
            {
              int carrierlost, babble;
            } type1;
       } un;
     } *statstruct[8];
  char **boarddescr;
  int i, j, l;
  int intvalue;
  struct request *r;
  char varname[1024];
  char *resp;
  int minor_type;
  int ninterfaces, ifno;
  int one = 1, zero = 0, intsz = sizeof(int);
  int iftype[8];

  minor_type = node->quiptab_entry->minor_type;
  if (minor_type < 0 || minor_type > 1) return(0);
```

```
resp = response;
if (parse_quip(headgrammar, &resp, (char **) &headstruct))
  return(0);

vt(boarddescr = (char **)
   malloc(sizeof(char *) * headstruct->nboards));
for (i = 0; i < headstruct->nboards; i++)
  {
    l = strlen("Bridge Communications " )
      + strlen(headstruct->board[i].boardname) + 1;
    for (j = 0; j < headstruct->board[i].nrevtoks; j++)
      l += strlen(headstruct->board[i].revtok[j]) + 1;
    vt(boarddescr[i] = (char *) malloc(l));
    strcpy(boarddescr[i], "Bridge Communications ");
    strcat(boarddescr[i], headstruct->board[i].boardname);
    for (j = 0; j < headstruct->board[i].nrevtoks; j++)
      {
        strcat(boarddescr[i], " ");
        strcat(boarddescr[i], headstruct->board[i].revtok[j]);
        if (!strncmp(headstruct->board[i].revtok[j],
                   "IECM", 4))
          iftype[i - 1] = 6;
        else if (!strncmp(headstruct->board[i].revtok[j],
                        "IBCM", 4))
          iftype[i - 1] = 1;
        else if (!strncmp(headstruct->board[i].revtok[j],
                        "ITCM", 4))
          iftype[i - 1] = 22;
      }
  }

free(boarddescr[0]);
ninterfaces = headstruct->nboards - 1;
for (i = 0; i < ninterfaces; i++)
  {
    if (minor_type == 0)
      statgrammar = "'Network' 'Statistics' 8int '-' 10int";
    else if (iftype[i] == 22)
      statgrammar = "'Path' 'Statistics' 12int";
    else
      statgrammar = "'Path' 'Statistics' 19int";
    if (parse_quip(statgrammar, &resp,
                 (char **) &(statstruct[i])))
      return(0);
  }

vf(variable_insert("interfaces.ifNumber", 0,
                 ASN_INTEGER, &ninterfaces,
                 intsz, SECOND * 5, node));

/* Kludge to make snmpxperfmon work.
 * Needs something after ip subtree */
intvalue = 0;
vf(variable_insert("icmp.icmpInMsgs", 1,
                 ASN_INTEGER, &intvalue, sizeof(int),
                 SECOND * 5, node));
```

```
for (i = 0; i < ninterfaces; i++)
    {
    ifno = i + 1;
    vf(variable_insert("ip.ipAddrTable.ipAddrEntry.ipAdEntI \
fIndex.1.1.1",
                                ifno,
                                ASN_INTEGER, &ifno, sizeof(int),
                                SECOND * 5, node));

    vf(variable_insert("interfaces.ifTable.ifEntry.ifIndex",
                                ifno, ASN_INTEGER, &ifno,
                                intsz, SECOND * 5, node));

    vf(variable_insert("interfaces.ifTable.ifEntry.ifDescr",
                                ifno, ASN_OCTET_STR, boarddescr[ifno],
                                strlen(boarddescr[ifno]), SECOND * \
5, node));

    vf(variable_insert("interfaces.ifTable.ifEntry.ifType",
                                ifno, ASN_INTEGER, &(iftype[i]),
                                intsz, SECOND * 5, node));

    intvalue = 1500;
    vf(variable_insert("interfaces.ifTable.ifEntry.ifMtu",
                                ifno, ASN_INTEGER, &intvalue,
                                intsz, SECOND * 5, node));

    if (iftype[i] == 6)
      intvalue = 10000000;
    else if (iftype[i] == 1)
      intvalue = 5000000;
    else intvalue = 0;
    vf(variable_insert("interfaces.ifTable.ifEntry.ifSpeed",
                                ifno, GAUGE, &intvalue,
                                intsz, SECOND * 5, node));

    vf(variable_insert("interfaces.ifTable.ifEntry.ifPhysAd \
dress",
                                ifno, ASN_OCTET_STR, \
 headstruct->address,
                                6, SECOND * 5, node));

    vf(variable_insert("interfaces.ifTable.ifEntry.ifAdminS \
tatus",
                                ifno, ASN_INTEGER, &one,
                                intsz, SECOND * 5, node));
    vf(variable_insert("interfaces.ifTable.ifEntry.ifOperSt \
atus",
                                ifno, ASN_INTEGER, &one,
                                intsz, SECOND * 5, node));

    vf(variable_insert("interfaces.ifTable.ifEntry.ifInOcte \
ts",
                                ifno, COUNTER, \
 &(statstruct[i]->rcvbytes),
                                intsz, SECOND * 5, node));
```

184

```
        intvalue = statstruct[i]->rcvpkts
          - (statstruct[i]->rcvmcastpkts
            + statstruct[i]->rcvbcastpkts);
        vf(variable_insert("interfaces.ifTable.ifEntry.ifInUcas \
tPkts",
                          ifno, COUNTER, &intvalue,
                          intsz, SECOND * 5, node));


        intvalue = statstruct[i]->rcvmcastpkts
          + statstruct[i]->rcvbcastpkts;
        vf(variable_insert("interfaces.ifTable.ifEntry.ifInNUca \
stPkts",
                          ifno, COUNTER, &intvalue,
                          intsz, SECOND * 5, node));


        vf(variable_insert("interfaces.ifTable.ifEntry.ifInDisc \
ards",
                          ifno, COUNTER, &zero,
                          intsz, SECOND * 5, node));


        intvalue = statstruct[i]->crc + statstruct[i]->frameerr
          + statstruct[i]->toolong + statstruct[i]->lostpkts;
        vf(variable_insert("interfaces.ifTable.ifEntry.ifInErro \
rs",
                          ifno, COUNTER, &intvalue,
                          intsz, SECOND * 5, node));


        vf(variable_insert("interfaces.ifTable.ifEntry.ifInUnkn \
ownProtos",
                          ifno, COUNTER, &zero,
                          intsz, SECOND * 5, node));


        intvalue = statstruct[i]->xmtbytes;
        vf(variable_insert("interfaces.ifTable.ifEntry.ifOutOct \
ets",
                          ifno, COUNTER, &intvalue,
                          intsz, SECOND * 5, node));


        intvalue = statstruct[i]->xmtpkts
          - (statstruct[i]->xmtmcastpkts
            + statstruct[i]->xmtbcastpkts);
        vf(variable_insert("interfaces.ifTable.ifEntry.ifOutUca \
stPkts",
                          ifno, COUNTER, &intvalue,
                          intsz, SECOND * 5, node));


        intvalue = statstruct[i]->xmtmcastpkts
          + statstruct[i]->xmtbcastpkts;
        vf(variable_insert("interfaces.ifTable.ifEntry.ifOutNUc \
astPkts",
                          ifno, COUNTER, &intvalue,
                          intsz, SECOND * 5, node));
```

```
            vf(variable_insert("interfaces.ifTable.ifEntry.ifOutDis \
cards",
                                ifno, COUNTER, &zero,
                                intsz, SECOND * 5, node));

        if (node->quiptab_entry->minor_type == 0)
          intvalue = statstruct[i]->un.type0.neterr;
        else
          if (iftype[i] == 22) intvalue = 0;
          else
            intvalue = statstruct[i]->un.type1.carrierlost
                + statstruct[i]->un.type1.babble;
        vf(variable_insert("interfaces.ifTable.ifEntry.ifOutErr \
ors",
                                ifno, COUNTER, &intvalue,
                                intsz, SECOND * 5, node));

        vf(variable_insert("interfaces.ifTable.ifEntry.ifOutQLen",
                                ifno, GAUGE, &zero,
                                intsz, SECOND * 5, node));
        hip_free(statstruct[i]);
        free(boarddescr[ifno]);
    }

  hip_free(headstruct);
  free(boarddescr);
  for (r = node->requestlist->next;
       r != node->requestlist;
       r = r->next)
    vf(fill_request(r, node));

  return(0);
}

/* Parse system part of MIB. */
int retrieve_ib_system(node)
      struct node *node;
{
  vsn("retrieve_ib_system");
  struct quipmsg msgout;
  char *query;

  msgout.options = 0;
  msgout.nodetype = node->quiptab_entry->major_type;
  msgout.id = 0;
  strcpy(msgout.address, node->quiptab_entry->address);
  query = "sh vers";
  msgout.timeout = 20;
  vf(send_quip(node->quip_fd, query, &msgout));
  vf(queue_quip(msgout.id, parse_ib_system, node));

  return(0);
}
```

```c
/* Parse interfaces part of MIB. */
int retrieve_ib_interfaces(node)
     struct node *node;
{
  vsn("retrieve_ib_interfaces");
  struct quipmsg msgout;
  char *query;

  msgout.options = 0;
  msgout.nodetype = node->quiptab_entry->major_type;
  msgout.id = 0;
  strcpy(msgout.address, node->quiptab_entry->address);
  if (node->quiptab_entry->minor_type == 0)
    query =
      "sh addr\n sh vers\nsh ns 0\n sh ns 1\n sh ns 2\n sh ns \
 3";
  else
    query =
      "sh addr\n sh vers\nsh ps 0\n sh ps 1\n sh ps 2\n sh ps \
 3";
  msgout.timeout = 10;
  vf(send_quip(node->quip_fd, query, &msgout));
  vf(queue_quip(msgout.id, parse_ib_interfaces, node));

  return(0);
}

/* Build variable tree stump. */
int build_var_tree_ib(vartree)
     struct varnode *vartree;
{
  vsn("build_var_tree_ib");

  vf(cache_branch(vartree, "system.sysDescr",
                  retrieve_ib_system, MINUTE));
  vf(cache_branch(vartree, "interfaces",
                  retrieve_ib_interfaces, SECOND * 5));
  vf(cache_branch(vartree, "ip",
                  retrieve_ib_interfaces, DAY));

  return(0);
}

#endif IB
```

```
/* ibx_driver.c - SNMP Proxy Daemon IBX driver. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution'
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#ifdef IBX

#include <sys/types.h>
#include <stdio.h>
#include <strings.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <snmp.h>
#include <snmp_impl.h>
#include <snmp_client.h>
#include <asn1.h>

#include "verify.h"
#include "quiplib.h"
#include "snmp_proxyd.h"

/* Parse system part of MIB. */
int parse_ibx_system(response, node)
     char *response;
     struct node *node;
{
  vsn("parse_ibx_system");
  char *headgrammar =
    "'*** CONFIGURATION' str 'PORT' str 'RELEASE:' str";
  struct
    {
      char *type, *port, *release;
    } *headstruct;
  char *resp;
  char *sysdescr;
  struct request *r;
  int l;
```

188

```
resp = response;
if (parse_quip(headgrammar, &resp, (char **) &headstruct))
    return(0);
free(response);

l = strlen("Intecom IBX LANmark Port Release ") + 1;
l += strlen(headstruct->type) + strlen(headstruct->port)
    + strlen(headstruct->release);
vt(sysdescr = (char *) malloc(l));
sprintf(sysdescr, "Intecom IBX LANmark %s Port %s Release %s",
        headstruct->type, headstruct->port,
        headstruct->release);

vf(variable_insert("system.sysDescr", 0,
                    ASN_OCTET_STR, sysdescr, strlen(sysdescr),
                    MINUTE, node));

for (r = node->requestlist->next;
     r != node->requestlist;
     r = r->next)
    vf(fill_request(r, node));

return(0);
}
```

```
/* Parse interfaces part of MIB. */
int parse_ibx_interfaces(response, node)
      char *response;
      struct node *node;
{
  vsn("parse_ibx_interfaces");
  char *headgrammar =
    "ethadr '\n ' [str:, ', booted'] 'PROMS:' *[str:- '-' \
 *str]";
  char *statgrammar;
  struct
     {
       u_char address[6];
       char *sysname;
       int nboards;
       struct
          {
            char *boardname;
            int nrevtoks;
            char **revtok;
          } *board;
     } *headstruct;
  struct
     {
       int rcvpkts, rcvbytes, rcvmcastpkts, rcvbcastpkts;
       int crc, frameerr, toolong, lostpkts;
       int xmtpkts, xmtbytes, xmtmcastpkts, xmtbcastpkts;
       int deferred, singlecoll, multicoll;
       int latecoll, excesscoll;
       union {
         struct
            {
              int neterr;
            } type0;
         struct
            {
              int carrierlost, babble;
            } type1;
       } un;
     } *statstruct[8];
  char **boarddescr;
  int i, j, l;
  int intvalue;
  struct request *r;
  char varname[1024];
  char *resp;
  int minor_type;
  int ninterfaces, ifno;
  int one = 1, zero = 0, intsz = sizeof(int);
  int iftype[8];

  minor_type = node->quiptab_entry->minor_type;
  if (minor_type < 0 || minor_type > 1) return(0);
```

```
    resp = response;
    if (parse_quip(headgrammar, &resp, (char **) &headstruct))
        return(0);

    vt(boarddescr = (char **)
        malloc(sizeof(char *) * headstruct->nboards));
    for (i = 0; i < headstruct->nboards; i++)
    {
        l = strlen("Bridge Communications " )
            + strlen(headstruct->board[i].boardname) + 1;
        for (j = 0; j < headstruct->board[i].nrevtoks; j++)
            l += strlen(headstruct->board[i].revtok[j]) + 1;
        vt(boarddescr[i] = (char *) malloc(l));
        strcpy(boarddescr[i], "Bridge Communications ");
        strcat(boarddescr[i], headstruct->board[i].boardname);
        for (j = 0; j < headstruct->board[i].nrevtoks; j++)
        {
            strcat(boarddescr[i], " ");
            strcat(boarddescr[i], headstruct->board[i].revtok[j]);
            if (!strncmp(headstruct->board[i].revtok[j],
                        "IECM", 4))
                iftype[i - 1] = 6;
            else if (!strncmp(headstruct->board[i].revtok[j],
                            "IBXCM", 4))
                iftype[i - 1] = 1;
            else if (!strncmp(headstruct->board[i].revtok[j],
                            "ITCM", 4))
                iftype[i - 1] = 22;
        }
    }

    ninterfaces = headstruct->nboards - 1;
    for (i = 0; i < ninterfaces; i++)
    {
        if (minor_type == 0)
            statgrammar = "'Network' 'Statistics' 8int '-' 10int";
        else if (iftype[i] == 22)
            statgrammar = "'Path' 'Statistics' 12int";
        else
            statgrammar = "'Path' 'Statistics' 19int";
        if (parse_quip(statgrammar, &resp,
                    (char **) &(statstruct[i])))
            return(0);
    }
    free(response);

    vf(variable_insert("interfaces.ifNumber", 0,
                    ASN_INTEGER, &ninterfaces,
                    intsz, SECOND * 5, node));

    for (i = 0; i < ninterfaces; i++)
    {
        ifno = i + 1;
        vf(variable_insert("interfaces.ifTable.ifEntry.ifIndex",
                        ifno, ASN_INTEGER, &ifno,
                        intsz, SECOND * 5, node));
```

```
        vf(variable_insert("interfaces.ifTable.ifEntry.ifDescr",
                        ifno, ASN_OCTET_STR, boarddescr[ifno],
                        strlen(boarddescr[ifno]), SECOND * \
  5, node));

        vf(variable_insert("interfaces.ifTable.ifEntry.ifType",
                        ifno, ASN_INTEGER, &(iftype[i]),
                        intsz, SECOND * 5, node));

        intvalue = 1500;
        vf(variable_insert("interfaces.ifTable.ifEntry.ifMtu",
                        ifno, ASN_INTEGER, &intvalue,
                        intsz, SECOND * 5, node));
                        ,

        if (iftype[i] == 6)
          intvalue = 10000000;
        else if (iftype[i] == 1)
          intvalue = 5000000;
        else intvalue = 0;
        vf(variable_insert("interfaces.ifTable.ifEntry.ifSpeed",
                        ifno, GAUGE, &intvalue,
                        intsz, SECOND * 5, node));

        vf(variable_insert("interfaces.ifTable.ifEntry.ifPhysAd \
dress",
                        ifno, ASN_OCTET_STR, \
 headstruct->address,
                        6, SECOND * 5, node));

        vf(variable_insert("interfaces.ifTable.ifEntry.ifAdminS \
tatus",
                        ifno, ASN_INTEGER, &one,
                        intsz, SECOND * 5, node));
        vf(variable_insert("interfaces.ifTable.ifEntry.ifOperSt \
atus",
                        ifno, ASN_INTEGER, &one,
                        intsz, SECOND * 5, node));

        vf(variable_insert("interfaces.ifTable.ifEntry.ifInOcte \
ts",
                        ifno, COUNTER, \
 &(statstruct[i]->rcvbytes),
                        intsz, SECOND * 5, node));

        intvalue = statstruct[i]->rcvpkts
          - (statstruct[i]->rcvmcastpkts
            + statstruct[i]->rcvbcastpkts);
        vf(variable_insert("interfaces.ifTable.ifEntry.ifInUcas \
tPkts",
                        ifno, COUNTER, &intvalue,
                        intsz, SECOND * 5, node));
```

```
        intvalue = statstruct[i]->rcvmcastpkts
          + statstruct[i]->rcvbcastpkts;
      vf(variable_insert("interfaces.ifTable.ifEntry.ifInNUca \
stPkts",
                          ifno, COUNTER, &intvalue,
                          intsz, SECOND * 5, node));


      vf(variable_insert("interfaces.ifTable.ifEntry.ifInDisc \
ards",
                          ifno, COUNTER, &zero,
                          intsz, SECOND * 5, node));


      intvalue = statstruct[i]->crc + statstruct[i]->frameerr
          + statstruct[i]->toolong + statstruct[i]->lostpkts;
      vf(variable_insert("interfaces.ifTable.ifEntry.ifInErro \
rs",
                          ifno, COUNTER, &intvalue,
                          intsz, SECOND * 5, node));


      vf(variable_insert("interfaces.ifTable.ifEntry.ifInUnkn \
ownProtos",
                          ifno, COUNTER, &zero,
                          intsz, SECOND * 5, node));


      intvalue = statstruct[i]->xmtbytes;
      vf(variable_insert("interfaces.ifTable.ifEntry.ifOutOct \
ets",
                          ifno, COUNTER, &intvalue,
                          intsz, SECOND * 5, node));


      intvalue = statstruct[i]->xmtpkts
        - (statstruct[i]->xmtmcastpkts
          + statstruct[i]->xmtbcastpkts);
      vf(variable_insert("interfaces.ifTable.ifEntry.ifOutUca \
stPkts",
                          ifno, COUNTER, &intvalue,
                          intsz, SECOND * 5, node));


      intvalue = statstruct[i]->xmtmcastpkts
          + statstruct[i]->xmtbcastpkts;
      vf(variable_insert("interfaces.ifTable.ifEntry.ifOutNUc \
astPkts",
                          ifno, COUNTER, &intvalue,
                          intsz, SECOND * 5, node));


      vf(variable_insert("interfaces.ifTable.ifEntry.ifOutDis \
cards",
                          ifno, COUNTER, &zero,
                          intsz, SECOND * 5, node));
```

```c
        if (node->quiptab_entry->minor_type == 0)
          intvalue = statstruct[i]->un.type0.neterr;
        else
          if (iftype[i] == 22) intvalue = 0;
          else
            intvalue = statstruct[i]->un.type1.carrierlost
              + statstruct[i]->un.type1.babble;
        vf(variable_insert("interfaces.ifTable.ifEntry.ifOutErr \
ors",
                           ifno, COUNTER, &intvalue,
                           intsz, SECOND * 5, node));

      vf(variable_insert("interfaces.ifTable.ifEntry.ifOutQLen",
                         ifno, GAUGE, &zero,
                         intsz, SECOND * 5, node));
    }

  for (r = node->requestlist->next;
       r != node->requestlist;
       r = r->next)
    vf(fill_request(r, node));

  return(0);
}

/* Retrieve system part of MIB. */
int retrieve_ibx_system(node)
     struct node *node;
{
  vsn("retrieve_ibx_system");
  struct quipmsg msgout;
  char *queryfmt, *query;

  msgout.options = 0;
  msgout.nodetype = node->quiptab_entry->major_type;
  msgout.id = 0;
  strcpy(msgout.address, "0");

  queryfmt = "@\n@\ndgnlndg\ndgn\nlndg\n3\n%s\n7\n@\n@\n";
  vt(query = (char *) malloc(strlen(queryfmt) - strlen("%s")
                             + \
 strlen(node->quiptab_entry->address)
                             + 1));
  sprintf(query, queryfmt, node->quiptab_entry->address);
  msgout.timeout = 20;

  vf(send_quip(node->quip_fd, query, &msgout));
  vf(queue_quip(msgout.id, parse_ibx_system, node));

  return(0);
}
```

```
/* Parse ip part of MIB */
int parse_ibx_ip(response, node)
     char *response;
     struct node *node;
{
  vsn("parse_ibx_ip");
  struct request *r;
  int intval;

  intval = 1;
  vf(variable_insert("ip.ipAddrTable.ipAddrEntry.ipAdEntIfInd \
ex.1.1.1",
                       1, ASN_INTEGER, &intval,
                       sizeof(int), DAY, node));
  intval = 2;
  vf(variable_insert("ip.ipAddrTable.ipAddrEntry.ipAdEntIfInd \
ex.1.1.1",
                       2, ASN_INTEGER, &intval,
                       sizeof(int), DAY, node));

  intval = 3;
  vf(variable_insert("ip.ipAddrTable.ipAddrEntry.ipAdEntIfInd \
ex.1.1.1",
                       3, ASN_INTEGER, &intval,
                       sizeof(int), DAY, node));

  intval = 4;
  vf(variable_insert("ip.ipAddrTable.ipAddrEntry.ipAdEntIfInd \
ex.1.1.1",
                       4, ASN_INTEGER, &intval,
                       sizeof(int), DAY, node));

  for (r = node->requestlist->next;
       r != node->requestlist;
       r = r->next)
    vf(fill_request(r, node));

  return(0);
}

/* Retrieve interfaces part of MIB. */
int retrieve_ibx_interfaces(node)
     struct node *node;
{
  vsn("retrieve_ibx_interfaces");
  struct quipmsg msgout;
  char *query;
```

195

```c
    msgout.options = 0;
    msgout.nodetype = node->quiptab_entry->major_type;
    msgout.id = 0;
    strcpy(msgout.address, node->quiptab_entry->address);
    if (node->quiptab_entry->minor_type == 0)
      query =
        "sh addr\n sh vers\nsh ns 0\n sh ns 1\n sh ns 2\n sh ns \
  3";
    else
      query =
        "sh addr\n sh vers\nsh ps 0\n sh ps 1\n sh ps 2\n sh ps \
  3";
    msgout.timeout = 10;
    vf(send_quip(node->quip_fd, query, &msgout));
    vf(queue_quip(msgout.id, parse_ibx_interfaces, node));

    return(0);
}

/* Retrieve ip part of MIB. */
int retrieve_ibx_ip(node)
        struct node *node;
{
    vsn("retrieve_ibx_ip");
    struct quipmsg msgout;
    char *query;

    msgout.options = 0;
    msgout.nodetype = node->quiptab_entry->major_type;
    msgout.id = 0;
    strcpy(msgout.address, node->quiptab_entry->address);
    query = "sh da";
    msgout.timeout = 10;
    vf(send_quip(node->quip_fd, query, &msgout));
    vf(queue_quip(msgout.id, parse_ibx_ip, node));

    return(0);
}

/* Build variable tree stump. */
int build_var_tree_ibx(vartree)
        struct varnode *vartree;
{
    vsn("build_var_tree_ibx");

    vf(cache_branch(vartree, "system.sysDescr",
                    retrieve_ibx_system, MINUTE));
    vf(cache_branch(vartree, "interfaces",
                    retrieve_ibx_interfaces, SECOND * 5));
    vf(cache_branch(vartree, "ip",
                    retrieve_ibx_ip, DAY));

    return(0);
}

#endif IBX
```

196

```c
/* translan_driver.c - SNMP Proxy Daemon Translan driver. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#ifdef TRANSLAN

#include <sys/types.h>
#include <stdio.h>
#include <strings.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <snmp.h>
#include <snmp_impl.h>
#include <snmp_client.h>
#include <asn1.h>

#include "verify.h"
#include "quiplib.h"
#include "snmp_proxyd.h"

/* Parse interfaces part of MIB */
int parse_translan_interfaces(response, node)
     char *response;
     struct node *node;
{
  vsn("parse_translan_interfaces");
  int i;
  int one = 1, zero = 0, intsz = sizeof(int);
  int intval;
  struct request *r;

  intval = 5;
  vf(variable_insert("interfaces.ifNumber", 0, ASN_INTEGER,
                     &intval, intsz, SECOND * 5, node));

  for (i = 1; i <= 5; i++)
    {
      vf(variable_insert("interfaces.ifTable.ifEntry.ifIndex",
                         i, ASN_INTEGER, &i,
                         intsz, DAY, SECOND * 5, node));
```

197

```
        vf(variable_insert("interfaces.ifTable.ifEntry.ifAdminS \
tatus",
                            i, ASN_INTEGER, &one,
                            intsz, SECOND * 5, node));
        vf(variable_insert("interfaces.ifTable.ifEntry.ifOperSt \
atus",
                            i, ASN_INTEGER, &one,
                            intsz, SECOND * 5, node));

    }

  for (r = node->requestlist->next;
       r != node->requestlist;
       r = r->next)
    vf(fill_request(r, node));

  return(0);
}

/* Retrieve system part of MIB. */
int retrieve_translan_system(node)
      struct node *node;
{
  vsn("retrieve_translan_system");
  char *sysdescr = "Vitalink Translan NP-III";
  struct request *r;

  vf(variable_insert("system.sysDescr", 0,
                      ASN_OCTET_STR, sysdescr,
                      strlen(sysdescr), MINUTE, node));

  for (r = node->requestlist->next;
       r != node->requestlist;
       r = r->next)
    vf(fill_request(r, node));

  return(0);
}

/* Retrieve interfaces part of MIB. */
int retrieve_translan_interfaces(node)
      struct node *node;
{
  vsn("retrieve_translan_interfaces");
  struct quipmsg msgout;
  char *query;

  msgout.options = 0;
  msgout.nodetype = node->quiptab_entry->major_type;
  msgout.id = 0;
  strcpy(msgout.address, node->quiptab_entry->address);
  query = "\r";
  msgout.timeout = 10;
  vf(send_quip(node->quip_fd, query, &msgout));
  vf(queue_quip(msgout.id, parse_translan_interfaces, node));
```

```
      return(0);
  }

  /* Retrieve ip part of MIB */
  int retrieve_translan_ip(node)
        struct node *node;
  {
    vsn("retrieve_translan_ip");
    struct request *r;
    int intval;
    int i;

    for (i = 1; i <= 5; i++)
      vf(variable_insert("ip.ipAddrTable.ipAddrEntry.ipAdEntIfI \
  ndex.1.1.1",
                         i, ASN_INTEGER, &i,
                         sizeof(int), DAY, node));

    for (r = node->requestlist->next;
         r != node->requestlist;
         r = r->next)
      vf(fill_request(r, node));

    return(0);
  }

  /* Build variable tree stump. */
  int build_var_tree_translan(vartree)
        struct varnode *vartree;
  {
    vsn("build_var_tree_translan");

    vf(cache_branch(vartree, "system.sysDescr",
                        retrieve_translan_system, MINUTE));
    vf(cache_branch(vartree, "interfaces",
                        retrieve_translan_interfaces, SECOND * 5));
    vf(cache_branch(vartree, "ip",
                        retrieve_translan_ip, DAY));

    return(0);
  }

  #endif TRANSLAN
```

```c
/* bufferlib.h - Support for buffer library. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

struct buffer
{
  char *start, *current, *end;
};

struct buffer *buffer_new(), *buffer_new_string();
void buffer_free(), buffer_append(), buffer_fit();
void buffer_append_string(), buffer_append_pointer(),
  buffer_append_long();
```

```c
/* bufferlib.c - Buffer library - for character buffers. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#include "bufferlib.h"
#define ML2
#include "memorylib.h"

/* Get a new buffer of given size. */
#ifdef MEMORYLOSS
struct buffer *MLbuffer_new(nbytes, file, line)
      int nbytes;
      char *file;
      int line;
#else MEMORYLOSS
struct buffer *buffer_new(nbytes)
      int nbytes;
#endif MEMORYLOSS
{
  struct buffer *b;

  b = (struct buffer *) Malloc(sizeof(struct buffer));
  b->start = (char *) Malloc(nbytes);
  b->current = b->start;
  b->end = b->start + nbytes;
  return(b);
}

/* Get a new buffer initialized to string. */
#ifdef MEMORYLOSS
struct buffer *MLbuffer_new_string(string, file, line)
      char *string;
      char *file;
      int line;
#else MEMORYLOSS
struct buffer *buffer_new_string(string)
      char *string;
#endif MEMORYLOSS
{
  struct buffer *b;
  int length = strlen(string) + 1;
```

```c
    b = (struct buffer *) Malloc(sizeof(struct buffer));
    b->start = string;
    b->current = b->start;
    b->end = b->start + length;
    return(b);
}

/* Free a buffer. */
void buffer_free(buffer)
     struct buffer *buffer;
{
  free(buffer->start);
  free(buffer);
}

/* Append to a buffer. */
#ifdef MEMORYLOSS
void MLbuffer_append(buffer, data, nbytes, file, line)
     struct buffer *buffer;
     char *data;
     int nbytes;
     char *file;
     int line;
#else MEMORYLOSS
void buffer_append(buffer, data, nbytes)
     struct buffer *buffer;
     char *data;
     int nbytes;
#endif MEMORYLOSS
{
  int current;
  int size;

  if (buffer->end - buffer->current < nbytes)
    {
      current = buffer->current - buffer->start;
      size = (current + nbytes) * 2;
      buffer->start = Realloc(buffer->start, size);
      buffer->end = buffer->start + size;
      buffer->current = buffer->start + current;
    }
  bcopy(data, buffer->current, nbytes);
  buffer->current += nbytes;
}
```

```c
/* Append a string to a buffer. */
#ifdef MEMORYLOSS
void MLbuffer_append_string(buffer, string, file, line)
     struct buffer *buffer;
     char *string;
     char *file;
     int line;
#else MEMORYLOSS
void buffer_append_string(buffer, string)
     struct buffer *buffer;
     char *string;
#endif MEMORYLOSS
{
  buffer_append(buffer, string, strlen(string) + 1);
  buffer->current--;
}

/* Used for aligning data types. */
static struct alignstruct
{
  long l1;
  char c1;
  long l2;
  char *p1;
  char c2;
  char *p2
} alignstruct;

#define ptr_align \
  ((char *) &alignstruct.p2 - (char *) &alignstruct.p1 - \
 sizeof(char *))

/* Append a pointer to a buffer. */
#ifdef MEMORYLOSS
void MLbuffer_append_pointer(buffer, pointer, file, line)
     struct buffer *buffer;
     char *pointer;
     char *file;
     int line;
#else MEMORYLOSS
void buffer_append_pointer(buffer, pointer)
     struct buffer *buffer;
     char *pointer;
#endif MEMORYLOSS
{
  int current_align, adjust;

  current_align =
    (buffer->current - (char *) &alignstruct.p1) % ptr_align;
  adjust = (ptr_align - current_align) % ptr_align;

  buffer->current += adjust;
  buffer_append(buffer, &pointer, sizeof(char *));
}
```

```c
#define long_align \
  ((char *) &alignstruct.l2 \
   - (char *) &alignstruct.l1 - sizeof(long))

/* Append a long integer to a buffer. */
#ifdef MEMORYLOSS
void MLbuffer_append_long(buffer, lng, file, line)
     struct buffer *buffer;
     char *lng;
     char *file;
     int line;
#else MEMORYLOSS
void buffer_append_long(buffer, lng)
     struct buffer *buffer;
     char *lng;
#endif MEMORYLOSS
{
  int current_align, adjust;

  current_align =
    (buffer->current - (char *) &alignstruct.l1) % long_align;
  adjust = (long_align - current_align) % long_align;

  buffer->current += adjust;
  buffer_append(buffer, &lng, sizeof(long));
}

/* Reduce buffer to minimum size. */
#ifdef MEMORYLOSS
void MLbuffer_fit(buffer, file, line)
     struct buffer *buffer;
     char *file;
     int line;
#else MEMORYLOSS
void buffer_fit(buffer)
     struct buffer *buffer;
#endif MEMORYLOSS
{
  int current;

  current = buffer->current - buffer->start;
  buffer->start = Realloc(buffer->start, current ? current : 1);
  buffer->current = buffer->start + current;
  buffer->end = buffer->current;
}
```

```
/* memorylib.h - Memory library definitions. */

#ifndef MEMORYLOSS
char *Malloc(), *Realloc();
#endif MEMORYLOSS
```

```c
/* memorylib.c - Versions of memory allocation routines that \
   core dump
 *    on error.
 */

#include <signal.h>

char *Malloc(nbytes)
    int nbytes;
{
  char *malloc(), *p;

  if (!(p = malloc(nbytes)))
     {
       perror("malloc");
       kill(getpid(), SIGQUIT);
     }
  return(p);
}

char *Realloc(pointer, nbytes)
    char *pointer;
    int nbytes;
{
  char *realloc(), *p;

  if (!(p = realloc(pointer, nbytes)))
     {
       perror("realloc");
       kill(getpid(), SIGQUIT);
     }
  return(p);
}
```

```
/* verify.h - Error handling setup macros.*/

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

/* verify setup */
#define vs(funct_name, error_funct, rtn_val_type, rtn_val) \
  char *_v_funct_name = (funct_name);\
  _V_ERROR_FUNCT _v_error_funct = error_funct;\
  rtn_val_type _v_rtn_val = (rtn_val)\

/* verify setup, specified error function,
 * standard natural number return */
#define vsfn(funct_name, error_funct) \
  vs(funct_name, error_funct, int, -1)

/* verify setup, specified error function,
 * standard non-null pointer return */
#define vsfp(funct_name, error_funct, rtn_val_type) \
  vs(funct_name, error_funct, rtn_val_type, NULL)

/* verify setup, standard error function,
 * standard natural number return */
#define vsn(funct_name) vsfn(funct_name, _v_stderr)

/* verify setup, standard error function,
 *standard non-null pointer return */
#define vsp(funct_name, rtn_val_type) \
  vsfp(funct_name, _v_stderr, rtn_val_type)

/* standard error function(s) defined in verify-lib.c */

extern int _v_stderr();

/* global variable(s) defined in verify-lib.c */

extern char* command_name;

/* global variable(s) defined elsewhere */

extern int errno;
```

```
/* typedef needed to get this to work with C++ */
typedef int (*_V_ERROR_FUNCT)();
```

```c
/* verify.c - Error handling macro support. */

/*
 * Copyright (c) 1989, 1990 by
 * Microelectronics Center of North Carolina (MCNC)
 *
 * Permission to use, copy, modify, and distribute this
 * software for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting documentation,
 * the name of MCNC not be used in advertising or publicity
 * pertaining to distribution of the program without
 * specific prior permission, and notice be given in
 * supporting documentation that copying and distribution
 * is by permission of MCNC.  MCNC makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
 */

#include <stdio.h>

extern int errno, sys_nerr;
extern char *sys_errlist[];

char *command_name = "";

/* Prints file and function name, line number, and failing \
 expression. */
void _v_stderr(file_name,
               line_number,
               funct_name,
               expression)
{
  int en = errno;

  errno = 0;
  fprintf(stderr, "\n%s: %s():", command_name, funct_name);
  if (en)
    {
      if (en < sys_nerr)
        fprintf(stderr, " \"%s\"", sys_errlist[en]);
      else
        fprintf(stderr, " \"Errno %d\"", en);
    }
  else fprintf(stderr, " \"Verification failure\"");
  fprintf(stderr, ", %s, line %d:\n",
          file_name, line_number, funct_name);
  fprintf(stderr, "   [ %s ]\n", expression);
  fflush(stderr);
}
```

```
define(vt,`if(errno=0,($1));else \
  return((*_v_error_funct)(__FILE__,__LINE__,_v_funct_name,"tr \
anslit($1,"
,| )"),_v_rtn_val)')dnl
define(vm,`return((*_v_error_funct)(__FILE__,__LINE__,_v_func \
t_name,$1),_v_rtn_val)')dnl
define(vf,`vt(!($1))')dnl
define(vn,`vt(($1) >= 0)')dnl
define(vp,`vt(($1) > 0)')dnl
undefine(`define')dnl
undefine(`eval')dnl
undefine(`incr')dnl
undefine(`ifelse')dnl
undefine(`unix')dnl
undefine(`maketemp')dnl
undefine(`divert')dnl
undefine(`sinclude')dnl
undefine(`divnum')dnl
undefine(`syscmd')dnl
undefine(`substr')dnl
undefine(`undivert')dnl
undefine(`errprint')dnl
undefine(`ifdef')dnl
undefine(`len')dnl
undefine(`index')dnl
undefine(`include')dnl
undefine(`dumpdef')dnl
undefine(`shift')dnl
changequote( , )dnl
undefine( changequote )dnl
```