NETWORK MANAGEMENT OF ASYNCHRONOUS TRANSFER MODE SWITCHING SYSTEMS

BY
ANDREW G. SCHMIDT
B.S., Purdue University, 1987

THESIS
Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois
# Table of Contents

Chapter

1 Introduction ................................................................. 1

2 Overview Of Xunet 2 .................................................... 3
   2.1 Network Architecture .............................................. 4
   2.2 Xunet 2 Switch Architecture ................................... 5
      2.2.1 Queue Module Architecture ................................. 7
      2.2.2 Line Interface Architecture ................................. 9
      2.2.3 Switch Control Architecture ............................... 9
   2.3 Xunet 2 Router Architecture ................................. 10
   2.4 Routing and Management Architecture ....................... 12
   2.5 Summary .......................................................... 12

3 SMI ................................................................. 13
   3.1 Names ............................................................ 15
   3.2 Structure and Identification of Management Information Syntax ......................................................... 18
      3.2.1 Application-Wide Defined Types .......................... 19
      3.2.2 NetworkAddress .............................................. 19
      3.2.3 IpAddress .................................................. 19
      3.2.4 Counter .................................................... 19
      3.2.5 Gauge ..................................................... 20
      3.2.6 TimeTicks ............................................... 20
      3.2.7 Opaque .................................................. 20
   3.3 Managed Objects .................................................. 20
      3.3.1 Managed Object Lists ...................................... 21
      3.3.2 Managed Object Tables ................................... 21
   3.4 Extensions to the MIB .......................................... 24
   3.5 Summary .......................................................... 24

4 The Management Information Base ................................. 25
   4.1 Introduction ..................................................... 25
   4.2 Objects .......................................................... 26
      4.2.1 The System Group ......................................... 27
      4.2.2 The Interfaces Group ..................................... 28
      4.2.3 The Internet Protocol Address Table Group ........... 28
      4.2.4 The Internet Protocol Routing Table Group ........... 28
4.2.5 The Internet Control Message Protocol Group ........................................... 29
4.2.6 The Transmission Control Protocol Group ................................................. 29
4.2.7 The User Datagram Protocol Group .......................................................... 30
4.2.8 The Exterior Gateway Protocol Group ......................................................... 30
4.3 MIB-II Modifications and Objects ................................................................. 31
4.4 Summary ........................................................................................................ 33

5 The Simple Network Management Protocol ...................................................... 34
  5.1 Elements of the SNMP Architecture ............................................................ 36
     5.1.1 Operations Supported on Management Information ............................... 37
     5.1.2 Administrative Framework .................................................................. 37
     5.1.3 MIB References ............................................................................... 39
  5.2 Protocol Specification ..................................................................................... 39
     5.2.1 Elements of Procedure ....................................................................... 41
     5.2.2 Common Constructs .......................................................................... 42
     5.2.3 GetRequest Protocol Data Unit ......................................................... 42
     5.2.4 GetNextRequest Protocol Data Unit ................................................. 45
     5.2.5 GetResponse Protocol Data Unit ...................................................... 48
     5.2.6 SetRequest Protocol Data Unit ......................................................... 48
     5.2.7 Trap Protocol Data Unit .................................................................... 50
  5.3 Summary ........................................................................................................ 52

6 Xunet Agent ...................................................................................................... 54
  6.1 Introduction .................................................................................................. 54
  6.2 Possible Agent Implementations ................................................................. 55
     6.2.1 SNMP MUX Protocol ....................................................................... 55
     6.2.2 Proxy Agent ..................................................................................... 58
     6.2.3 SNMP Library Agent ....................................................................... 59
     6.2.4 Xunet Agent Organization ............................................................... 60
  6.3 Summary ........................................................................................................ 63

7 Xunet MIB ........................................................................................................ 64
  7.1 Introduction .................................................................................................. 64
  7.2 Managed Objects ........................................................................................ 65
     7.2.1 The Top Level Switch Objects ......................................................... 66
     7.2.2 The Second Level Line Card Objects .............................................. 66
     7.2.3 Third Level Line Card Objects ....................................................... 67
     7.2.4 Forth Level Virtual Path Group ....................................................... 69
     7.2.5 The Fifth Level Virtual Channel Group ......................................... 70
  7.3 Summary ........................................................................................................ 71

8 Conclusion ......................................................................................................... 72

Appendix ............................................................................................................ 76

A Xunet MIB ........................................................................................................ 76
B  SMI Definitions ................................................................. 94

Bibliography ................................................................. 97
List of Figures

2.1 Xunet 2 Network Architecture. ........................................... 5
2.2 Xunet 2 Network Topology. .............................................. 6
2.3 Xunet 2 Switch. .......................................................... 7
2.4 Xunet 2 Router. .......................................................... 10

3.1 Naming Hierarchy. ........................................................ 16
3.2 Example of Managed Object List. .................................... 22
3.3 Example of Managed Object List Addition. ....................... 23
3.4 Example Table Definition. ............................................. 23

5.1 Manager/Agent Relationship. ......................................... 35
5.2 SNMP Protocol Data Units. ........................................... 40
5.3 SNMP Common Constructs ............................................. 43
5.4 SNMP GetRequest-PDU. ............................................... 45
5.5 SNMP GetNextRequest-PDU. ......................................... 47
5.6 SNMP GetResponse-PDU. ............................................. 48
5.7 SNMP SetRequest-PDU. ............................................... 49
5.8 SNMP Trap-PDU. ...................................................... 53

6.1 SNMP Proxy Agent Model. ........................................... 58

8.1 NMS Xunet View. .................................................... 73
8.2 UIUC Xunet View. .................................................... 74
8.3 ATM Cells Transmitted View. ...................................... 74
Chapter 1

Introduction

In this thesis, I discuss the design and implementation of a Network Management System (NMS) [26, 51] for an Asynchronous Transfer Mode (ATM) [31] switching system based on the switching fabric of the Experimental University Network (Xunet) II [23] switch. The Xunet II switch is a collaborative effort in developing an ATM platform that facilitates experimentation with high speed Gigabit communications. The Network Management System developed utilizes the Simple Network Management Protocol (SNMP) [4].

The SNMP agent provides a portable and modular system without sacrificing efficiency. This thesis presents the results of the design and implementation of the SNMP agent daemon on the Xunet II system of control computers and routers. The implementation of an SNMP agent on an ATM switch is significantly different from the other platforms to which the protocol has been used in the past [51] due to the differences in line speeds and media types.

Results from this thesis reveal the structure of the ATM Managed Information Base (MIB) [42], and the various techniques that can be used to retrieve and store data from the managed switch.
This thesis is divided into three main parts. Chapter 2 provides an overview of the Xunet II switching fabric. Chapters 3, 4, and 5 contain an overview of the management protocol and managed information. Chapters 6, 7, and 8 discuss the implementation details of the Xunet SNMP agent and Network Manager communication structure.
Chapter 2

Overview Of Xunet 2

The Xunet 2 is an experimental wide-area network that serves as a testbed for research on high speed ATM data communications techniques. The initial focus is data communications between FDDI [48] local area networks and a Broadband-ISDN [25] style backbone network. Xunet 2 is part of the Gigabit Network Project, known as the BLANCA Testbed [7], sponsored by the Corporation for National Research Initiatives [8]. As a member, AT&T intends to upgrade the transmission facilities to a rate of 600 Megabit/sec in sections of the network.

The network connects AT&T Bell Laboratories in Murray Hill, Lawrence Livermore National Laboratories, Sandia National Laboratories and four universities (The University of California at Berkeley, the University of Illinois at Urbana-Champaign, the University of Wisconsin at Madison, and Rutgers University). The BLANCA testbed also includes students from the University of Pennsylvania and Columbia University. The program also includes Ameritech, Bell Atlantic, and Pacific Bell.

Xunet 2 has three principal components:

1. **Routers** - that are used as protocol gateways;
2. **Switches** - to act as the actual network fabric,


Long-haul transmission among sites use DS3 [2] facilities. Backbone performance objectives cover a wide range of transmission speeds. At each switch, the backbone should have less than a 1 second queueing delay for short messages and should allow endpoints to transfer files at 75% of the trunk bandwidth for trunks that are otherwise idle.

Each router attaches a local area network to the backbone network. The backbone network provides virtual circuit connectivity between the routers. Circuit setup and tear-down are done by a control computer associated with each switch, and the network, as a whole, is managed by a network management station.

### 2.1 Network Architecture

The goal of Xunet 2 is to provide a platform for demonstration and study of techniques for high-speed wide-area communication between local area networks that use the Internet TCP/IP [9] protocol suite. However, the backbone is capable of carrying traffic of other types, including multimedia and real-time [19] traffic. In order for the network to be practical, it must be managed and administered by software systems that support diverse and changing network components and multiple administrative domains.

The switching fabric is based on standards adopted for Broadband-ISDN [49]. The most important points are:

1. **Asynchronous Transfer Mode**, in which all information is carried in fixed size cells to improve switching and multiplexing performance.
2. **Virtual Circuit Switching**, where the call establishment semantics of circuit switching are combined with packet switching to achieve transmission efficiency.

Figure 2.1 shows the general architecture of the experimental network. Switches are linked by long-haul transmission lines. Each switch is managed by a control computer that is also responsible for circuit setup and tear-down. The four initial endpoints on the network, as shown in Figure 2.2, have both a switch and a local router.

### 2.2 Xunet 2 Switch Architecture

The Xunet 2 switch is the basic element of the network fabric. The switch is an descendent of the Datakit VCS [22] and is comprised of the following three main components: the address translation module, the trunk interface modules, and the queue modules. The cabinet is capable
of housing up to eight pairs of queue/interface modules that are all controlled by a single translation module.

A system architecture diagram is shown in Figure 2.3. In this Figure, two trunk interface/queue modules are shown along with the address translation module. The address translation module is capable of communication with the control computer via the trunk interfaces or an on board Ethernet [25] interface.

The interconnection among the various components is made on three main system buses. The contention bus carries cells that come into the switch via the trunk interface cards to the address translation module. The broadcast bus distributes cells to the trunk interface modules for output from the switch. The arbitration bus is used with an arbitration protocol to determine which module will next have permission to transmit on the contention bus. In addition, a forth bus, the maintenance bus transfers commands and status reports between the various circuit cards in a single equipment shelf.
Access to the contention bus occurs in cycles that are one cell long. As shown in Figure 2.3, cells from trunk A enter the line interface $L_A$ and are immediately transferred to the queue handler $Q_A$, where they are stored before transmission over the backplane bus. Module $Q_A$ competes with the other modules for access to the contention bus, and when given access, transmits a cell over the backplane to the address translation module $T$. The address translation module uses a 5 bit source address and the 16 bit Virtual Circuit Identifier (VCI) [31] in the incoming cell to index into the translation memory. The memory contains the 5 bit module address, for the destination module, and a 16 bit VCI that replaces the VCI in the new cell header. The cell with its new VCI is transmitted over the broadcast bus. From there, it is picked up by a queue handler with a slot address that matches the destination. $Q_B$ queues the cell for later transmission to the line module $L_B$ and trunk $B$.

### 2.2.1 Queue Module Architecture

Arbitration by the queue handlers is performed via a distributed group protocol [55]. During each arbitration cycle, each queue handler with data to transmit places its slot address onto
the arbitration bus. The protocol insures that all but one module withdraws from contention during the arbitration cycle. The address of the selected module remains on the bus. The priority bit insures that isochronous traffic has priority over other traffic types. The group bit insures that the modules share the backplane bandwidth fairly within each priority class.

The queue handler module supports a dynamic buffer allocation in which the buffer size for a conversation changes during the call. There are two queues (incoming and outgoing) for each of 32 K virtual circuits. The module contains 32 Mbytes of memory that stores up to 512 K cells. Dynamic buffer allocation is supported by allocating memory for cell storage on the fly when cells arrive and releasing memory after they leave. The cells are chained together to form a separate first-in-first-out queue for each virtual circuit. There is a per virtual circuit limit on the maximum number of cells that can be stored on a queue.

Each queue is capable of supporting its own service discipline. The queue handler has a hardware state machine that implements the queue service discipline. The queueing policy is table-driven so that different policies can be downloaded. The present hardware can handle up to 16 service classes. For each service class, there is a list of the VCIIs with a data queued awaiting service. When a queue first becomes non-empty, its VCI is put on the end of a service list for the particular class of service required by that virtual circuit. The server determines which service list to process next and removes the VCI from the head of the list. The module then serves the queue for that VCI, sending cells to the line interface module or backplane.

The congestion control information in the cell headers is also processed by the queue handler. There is a microprocessor on the module that monitors per virtual circuit congestion information stored in the hardware. The microprocessor can modify the outgoing information to reflect the local congestion conditions.
2.2.2  Line Interface Architecture

Line interface modules act as the conduit for arriving and departing cells from the switch. A common interface is used between all line interface cards and queue modules. This allows different line cards to be developed which reuse existing queue modules.

Two main line interfaces are currently utilized in the Xunet 2, a 200 Megabit/sec fiber trunk and a DS3 trunk. In addition, design and development are underway for the support of a HIPPI [35, 3] line interface to the switch. The line interface interface will support connections to a FORE Systems Inc. switch [10].

2.2.3  Switch Control Architecture

The control computer is responsible for call setup and tear-down, as well as maintenance and basic administration of the switch. In Xunet 2, one endpoint communicates with another endpoint using a virtual circuit. A user originates a connection by sending a call setup request to a switch. The switch then forwards the request to its control computer that manages the switch. Assuming multiple switches are in between hosts, the call setup message is forwarded from each of the control computers along the path.

Communication with the control computers can occur in one of two ways. First, the computer can be connected to the network like an ordinary ATM host, communicating with other machines on the network over virtual circuits. Second, it can communicate directly with the address translation module over an Ethernet interface.

The address translation module supports a set of commands for interaction with the control computer. Call setup involves cooperation between the two sides of a virtual circuit: the origi-
Figure 2.4: Xunet 2 Router.

inactor and the destination. The abstract model of the address translation module allows each side of the call to proceed asynchronously. The call is established when both sides rendezvous.

The maintenance subsystem in the control computer is responsible for monitoring the integrity of the switch hardware and transmission facilities. Every circuit card in the switch, including the address translation module, contains a maintenance processor. These processors are interconnected by a serial maintenance bus on the backplane. The maintenance processor on the address translation module is master of this bus. Periodically, the control computer instructs the master to poll the cards for status. If a card is installed, its maintenance processor responds with status information, such as the card type and serial number. Commands from the control computer are relayed over the maintenance bus to enable, disable, or reset a card. The control computer is then remotely managed by SNMP, over virtual circuits, from the network management system (see section 6.2.4).

2.3 Xunet 2 Router Architecture

The Xunet 2 router serves as the main interface between the local area networks and the ATM backbone. The device that functions as a router is constructed of a high performance workstation and is responsible for routing over two interfaces. The first interface in the router
is the FDDI interface which is connected to a local ring. This ring is intended to serve as a location for hosts who wish to use the Xunet 2 ATM backbone without being directly attached to a switch via an ATM interface. The second interface in the router is a 200 Mbit/sec line card used to make the connection to the switch. Both of these interfaces have on-board RISC [50] processors and local memory. The configuration of a router and its connection to a switch is shown in Figure 2.4.

When a packet arrives from the FDDI ring, it is DMA’d into a buffer. The FDDI interface board appends a receiver buffer descriptor at the tail of a ring buffer. Periodically, the host processor checks the ring buffer to see if packets have arrived. Each receiver buffer descriptor contains a pointer to a packet. The host extracts the destination address from the packet and translates it into a virtual circuit identifier to be used on the backbone network. The host appends a transmit buffer descriptor containing the VCI to the tail of a ring buffer scanned by the ATM trunk interface. Finally, the ATM interface fetches the packet from memory, fragments it into cells, and transmits the cells to the backbone network.

When an appropriate virtual circuit does not yet exist, the router sends a call setup request to its local Xunet 2 switch. The packet is queued pending completion of the call setup. Experimentation is taking place within the Xunet group on improved call setup schemes [32].

When a packet arrives from a backbone router destined for the FDDI ring, it must be reassembled by the VME interface board. Once the packet is assembled, it is routed to the FDDI ring using the same procedure described above. In some cases, a packet from either the local area network or the backbone will be destined for a control process in the router. In addition, the router will generate packets to be sent over the local network or the backbone. If the router does contain a route entry for the FDDI destination address, it will use the Address Resolution
Protocol [34] to determine that address. Routers can be managed by SNMP through virtual circuit connections or via the FDDI interface (see section 6.2.4).

2.4 Routing and Management Architecture

Routers will forward all routing packets from the local area network to a process in the management system that acts as a routing supervisor. This process uses the received information to learn which networks are attached to the backbone and sends routing packets to be forwarded to the local network. The topology, traffic levels, and alarm data from the backbone network are monitored continually and are used to create an overall picture of the network state. From this database, routing information is distributed to the routers attached to the backbone.

2.5 Summary

The Xunet SNMP management system solves three main problems: realtime monitoring, system configuration, and statistics gathering. Management of the network elements is accomplished in conjunction with a centralized network management system and the distributed SNMP agents. An agent is a software program housed within a managed network device that stores management data and responds to the manager’s requests for the data. The agents accumulate a set of information about the state of their local environment. This information can be obtained by the network management system either through polling or by directly communicating with the network management system. The management system also serves as a tool for the human administrator. From the management system, an administrator can display and edit a map of the network, monitor network load, and receive alarm notification.
Chapter 3

SMI

The following chapter review common terminology and definitions for the structure and identification of management information on TCP/IP-based internets. The management protocols provide a simple, workable architecture and a system for managing TCP/IP-based internets like the Xunet.

The description of the following protocols are sufficient background information for the understanding of the Xunet implementation of network management. It should be noted that the focus of this project is to provide network management functionality for the Xunet by the end of spring of 1993. Certain sacrifices have to be made in the implementation of a management system in regard to the protocols of choice. It is the author’s belief that future management systems may not utilize SNMP, but will be based on OSI[40, 45] and the Common Management Internet Protocol (CMIP)[20, 21]. To achieve this goal, research is taking place within the Xunet community to add support for these emerging standards[54]. In addition, this research effort will, as much as possible, support the SNMP protocol implementation over the OSI protocol suite.
The current internet management framework is governed by the Structure and Identification of Management Information (SMI). The SMI components discusses how managed objects contained in the MIB are defined, and how the MIB for network management describes the managed objects. And finally, the SMI describes the management protocol, SNMP, which defines the protocol used to communicate with managed devices.

The Internet Activities Board recommends that all IP and TCP implementations be network manageable. This implies implementation of the Internet MIB and, at least, one of the two recommended management protocols SNMP or CMOT[40]. It should be noted that at this time SNMP is a full Internet standard and CMOT is a draft standard.

The philosophy driving the creation and implementation of the Xunet network management systems is simplicity of both design and implementation. Furthermore, it should not overburden the managed device. This philosophy coincides with the Internet standards community. Within the Internet standards community, the two-prong approach for network management of TCP/IP-based internets is being implemented. In the short-term, with simplicity in mind, the Simple Network Management Protocol will be used to manage devices in the community. The long-term goal, as with the Xunet project, is to utilize the OSI network management framework.

Two documents from the Internet standards community have been produced to explain management information, RFC 1065 and RFC 1066. They define the Structure of Management Information and the Management Information Base, respectively (see section 4.1). These documents are designed to be compatible with both the SNMP and the OSI network management framework.

This chapter reviews common definitions associated with the structure and identification of management information for TCP/IP-based internets. The Structure and Identification of
Management Information (SMI)[42] is concerned with organizational and administrative policy. This chapter does not address information discussing the Management Information Base or the Simple Network Management Protocol. The MIB and SNMP details are addressed in chapters 4 and 5, respectively.

Managed objects are accessed via a virtual information store or MIB. Objects in the MIB are defined in Abstract Syntax Notation One[20] and each contains a Name, Syntax, and Encoding. Every name is a unique identifier and is referred to as the **OBJECT IDENTIFIER**. The syntax for an object defines the ASN.1 data structure instances of the object. For simplicity, there is a restriction on using the entire set of ASN.1 constructs as the syntax for an object. When defining an object instance, only the following ASN.1 primitive types are utilized:

- **INTEGER** - a potentially enumerated type taking a cardinal number as its value.
- **OCTET STRING** - a type made up of a string of octets.
- **OBJECT IDENTIFIER** - a type which refers to an authoritative designation.
- **NULL** - a empty or null valued place holding data type.

These ASN.1 types are called **Object Syntax**. Finally, the encoding of an object type defines how instances of that object are to be represented.

### 3.1 Names

Names are the primary method for object identification within the SMI. All name generation and identification is done in a hierarchical structure. The **OBJECT IDENTIFIER** serves as the name for the managed object.
A name, or **OBJECT IDENTIFIER**, is the sequence of integers that makes up the tree managed objects. This tree provides unique names to all objects. There is no restriction on the length of the values. Hence, the tree is unbounded. A naming policy is that a name never be reused.

The tree consists of the root and contains a number of labeled nodes via edges. Each node may also contain subtrees with the subtrees being uniquely labeled. The root node has, at least, three subtrees. The first node, labeled iso (1), is administered by the International Organization for Standardization. The second subtree, labeled ccitt (0), is controlled by the International Telegraph and Telephone Consultative Committee. The third subtree, joint-isoccitt(2), is jointly administered by the ISO[20] and the CCITT. The hierarchy from the root node is depicted in Figure 3.1.
Under the iso(1) node, a node is set aside, org(3), by the International Standards Organization for exclusive use by other national organizations. The nodes allocated under this subtree are assigned to the U.S. National Institutes of Standards and Technology. Recently, one node was transferred to the U.S. Department of Defense, dod(6). Under the dod(6) subtree falls the internet(1) **OBJECT IDENTIFIER**.

Given the above description of the naming structure, it is possible to commence constructing standard prefixes for a MIB **OBJECT IDENTIFIER**. The primary prefix is constructed by following the tree from the root to the identified leaf. For example, the prefix identifying the internet(1) subtree would be constructed as follows:

```
internet OBJECT IDENTIFIER ::= { iso org(3) dod(6) 1 }.
```

This naming convention may also be expressed as:

```
1.3.6.1.
```

Currently, the IAB[9] has specified the policy and allocation of the subtree under the internet(1) subtree. This subtree contains the four nodes: directory(1), mgmt(2), experimental(3), and private(4). The directory(1) is reserved for OSI directory services.

The mgmt(2) is reserved for the identification of objects that directly relate to the process of management. Within this subtree, organizations register their approved MIBs. Organizations are assigned a unique **OBJECT IDENTIFIER** that corresponds to the root of their tree and a leaf of the the (iso(1),org(3),dod(6),internet(1),mgmt(2)) tree.

New MIB experimentation and testing is conducted within the subtree labeled experimental(3). Like the other trees, this object group is managed by the Internet
Assigned Numbers Authority. For example, if a company is testing an experimental MIB for an uninterruptable power supply, it may be assigned the subtree starting at 67. Therefore, the full path name of their \textbf{OBJECT IDENTIFIER} would be (iso(1),org(3),dod(6),internet(1),experimental(3),ups(67)). The Internet Assigned Numbers Authority reserves the ability to dictate how the experimental subtree will be numbered beyond the point of (iso(1),org(3),dod(6),internet(1),experimental(3)). A list of Reassignment of Experimental MIBs to Standard MIBs can be found in [38]. As of this writing, some of the current experimental codes are being used for: Token Ring-like Objects[30], DS3 Interface Type Objects[13], and T1 Carrier Objects[11].

The final subtree allocated by the Internet Assigned Numbers Authority for the Internet is the private(4) subtree. This tree contains an entry called enterprises(1) which is used as the root for organizational subtrees. Typically, vendors providing MIBs for their products obtain a node within this subtree. For example, the internetworking vendor Proteon[24] has been allocated the subtree (iso(1),org(3),dod(6),internet(1),private(4), enterprises(1),proteon(1)). From this point, Proteon has constructed a group of \textbf{OBJECT IDENTIFIERS} that are only valid and relevant to its products.

3.2 Structure and Identification of Management Information

Syntax

As previously stated, managed objects have a syntax defined by the ASN.1 data type Object-Syntax. In the ASN.1 language this is called a CHOICE that can either be one of the four primitive ASN.1 types, an application-wide type defined within the SMI, or a aggregate of the
simple types. When an instance of an object is identified, it may be transmitted by applying
the basic encoding rules[21] of ASN.1 for the appropriate object type syntax.

3.2.1 Application-Wide Defined Types

Within the SMI, several name data types are defined. This section outlines and describes these
data types. The addition of data types is permitted, provided that the types resolve into the
types defined in ASN.1.

3.2.2 NetworkAddress

This subtype represents a network address relating to a specific protocol. In the first revision
of the SMI, the only protocol supported was the Internet protocol (IP). Subsequent work has
lead to the proposed addition of other protocols, such as OSI and IPX[15].

3.2.3 IpAddress

This data type represents an Internet protocol address. The length is a 32-bit value that is
comprised of an OCTET STRING of length 4, in network byte-order.

3.2.4 Counter

This data type represents a positive valued integer. The Counter type will monotonically
increase until it reaches its maximum value, \(2^{32} - 1\), then it wraps around and starts increasing
again from zero.
3.2.5 Gauge

This data type is employed to set a high water mark. It has the ability to increase or decrease, but will retain and report its maximum value when queried. A Gauge is a non-negative integer with a maximum value of $2^{32} - 1$.

3.2.6 TimeTicks

This data type takes a non-negative integer value and counts the time in hundredths of a second since some epoch. When object types are defined from this ASN.1 type, they must identify the reference epoch.

3.2.7 Opaque

This application-wide type allows the user to get around restrictions of the ASN.1 syntax. With the Opaque type, a value is encoded into a string of octets using ASN.1’s basic rules. It is possible that other encodings can use opaquely-encoded data with the ASN.1 EXTERNAL type.

3.3 Managed Objects

This section details the procedure for adding objects to the Management Information Base. For details on the objects added for support of the Xunet MIB, see Chapter 7.

The format for adding an object involves a collection of five fields.

1. **OBJECT**: a textual name, called the **OBJECT DESCRIPTOR**. This field also contains the **OBJECT IDENTIFIER** described in section 3.1.

2. **Syntax**: The corresponding ASN.1 syntax used to instantiate the object. See section 3.2.
3. **Definition:** A textual field that describes the representation of an object. For example, “Number of Input ATM cells dropped due to buffer overflow”.

4. **Access:** Specified type of access rights permitted for an object. This field may be one of read-only, read-write, write-only, or not-accessible.

5. **Status:** Specified the implementation state of an object. This field is used to determine the priority of an object, or its value in network management. This field may be one of mandatory, optional, obsolete or de-valued.

### 3.3.1 Managed Object Lists

Objects can be, and typically are, defined in the MIB by their relationship in the naming hierarchy. This type of naming is referred to as a list. For example, see [42]. Figure 3.2 depicts object types defined in a MIB. In the Figure, each instance of the object type comprises information represented by instances of the former three object types. Then, a fourth object type might also be defined in the MIB as shown in Figure 3.3.

### 3.3.2 Managed Object Tables

In much the same way that lists are formed, two dimensional tables can be constructed. These tables are instantiated to consist of zero or more rows, each with the same number of columns. Tables are typically used to store information, (e.g., routing tables). As illustrated in [42], a fifth object type might also be defined in the MIB in the manner illustrated in Figure 3.4.
OBJECT:

atIndex \{ atEntry 1 \}
Syntax:
   INTEGER
Definition:
   The interface number for the physical address.
Access:
   read-write.
Status:
   mandatory.

OBJECT:

atPhysAddress \{ atEntry 2 \}
Syntax:
   OCTET STRING
Definition:
   The media-dependent physical address.
Access:
   read-write.
Status:
   mandatory.

OBJECT:

atNetAddress \{ atEntry 3 \}
Syntax:
   NetworkAddress
Definition:
   The network address corresponding to the
   media-dependent physical address.
Access:
   read-write.
Status:
   mandatory.

**Figure 3.2:** Example of Managed Object List.
OBJECT:
-------
   atEntry { atTable 1 }
Syntax:
   AtEntry ::= SEQUENCE {
      atIndex
      INTEGER,
      atPhysAddress
      OCTET STRING,
      atNetAddress
      NetworkAddress
   }
Definition:
   An entry in the address translation table.
Access:
   read-write.
Status:
   mandatory.

Figure 3.3: Example of Managed Object List Addition.

OBJECT:
-------
   atTable { at 1 }
Syntax:
   SEQUENCE OF AtEntry
Definition:
   The address translation table.
Access:
   read-write.
Status:
   mandatory.

Figure 3.4: Example Table Definition.
3.4 Extensions to the MIB

The SMI provides the ability to extend the MIB in three ways. As previously discussed (see section 3.1), the SMI has provisions for extension via the experimental(3) and private(4) sub-trees. In addition, the capability exists for the current standard MIB to be changed, thereby providing extension by default.

In order to facilitate MIB extensions, several ground rules have been developed. These rules are summarized as follows:

1. Declare old object types obsolete, but do not delete their names;

2. Augment the object type definition corresponding to a list by appending non-aggregate object types to the object types in the list.

3. Define entirely new object types.

In addition, new versions may not change the semantics of any previously defined object without changing the name of that object.

3.5 Summary

This chapter has reviewed common definitions associated with the structure and identification of management information for TCP/IP-based internets. The Structure and Identification of Management Information is concerned with organizational and administrative policy. This chapter does not address information discussing the Management Information Base or the Simple Network Management Protocol. The MIB and SNMP details are addressed in chapters 4 and 5, respectively.
Chapter 4

The Management Information Base

4.1 Introduction

The MIB within the Internet is defined with TCP/IP based systems in mind. Several fundamental design axioms, such as simplicity and functionality, have gone into the first version of the MIB-I [43]. These axioms are applied to subsequent MIBs and the Xunet MIB. For further examples see [6, 30, 39].

From its inception, the IAB has requested working groups, chartered to defined SNMP to keep it and their respective MIBs as simple as possible. The list of objects defined in MIB-I include only those elements that are considered absolutely essential. The starting point for the MIB’s definition is taken from the High-Level Entity Management Systems (HEMS) [33], the initial SNMP specification [17], and the CMIS/CMIP memos [27, 28]. Each defined object has a status of required (see section 3.3). It should be kept in mind that the fundamental axioms are not overly restrictive, since the designers of the SMI include provisions for growth and expansion, see section 3.4.
In order for an object to be included in the MIB, it has to meet the criterion of being essential. Furthermore, it must be included in all above mentioned MIB definitions. The criteria can be summarized as follows [42]:

- An object needs to be essential for either fault or configuration detection and management.
- The initial SNMP protocol implements very weak security, therefore, all operations allow limited control properties.
- The object must have a history of use and show utility.
- Only a limited number of objects would be supported to ease the task of implementation, this limit is set at approximately 100.
- Objects have to be the lowest common denominators in that they should not be derived from each other.
- Objects that were closely tied to one vendor or system are not permitted.
- In order to reduce the potential CPU load on the managed device, only one counter per critical section is suggested.

### 4.2 Objects

This section outlines the objects included in the standard MIB-I and MIB-II [44] definition. All objects included in the MIB-I are defined with the Abstract Syntax Notation One. The actual representation of the MIB objects is defined in the Structure and Identification of Management Information covered in section 3.
<table>
<thead>
<tr>
<th>group</th>
<th>No. of Objects</th>
<th>Intended Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>3</td>
<td>Describes Managed Node</td>
</tr>
<tr>
<td>interfaces</td>
<td>22</td>
<td>Describes Physical Interfaces</td>
</tr>
<tr>
<td>at</td>
<td>3</td>
<td>Address Translation</td>
</tr>
<tr>
<td>ip</td>
<td>33</td>
<td>Internet Protocol Specific</td>
</tr>
<tr>
<td>icmp</td>
<td>26</td>
<td>Internet Control Message Protocol</td>
</tr>
<tr>
<td>tcp</td>
<td>17</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>udp</td>
<td>4</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>egp</td>
<td>6</td>
<td>Exterior Gateway Protocol</td>
</tr>
<tr>
<td>Total</td>
<td>114</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.1: MIB-I Groups and Object Counts.**

The MIB-I contains eight object groups. This number was expanded to 10 in the MIB-II. The groups are divided into the following sections: System, Interfaces, Address Translation, Internet Protocol, Internet Control Message Protocol, Transmission Control Protocol, User Datagram Protocol, Exterior Gateway Protocol, Transmission (*MIB-II only*), and SNMP (*MIB-II only*).

This subdivision facilitates the naming convention and allows the agent to implement only the groups that are relevant to the device. For example, a workstation or host not involved with the EGP [47] protocol is not required to implement that group. However, if it is desirable to have a device implement an object in one of the groups, it must then implement all of the objects from that group. Table 4.1 contains a listing of the various groups defined in MIB-I along with the number of object within the group.

### 4.2.1 The System Group

The System Group is responsible for maintaining general information about the managed device. This information is relatively static and contains entries, such as device name, and the vendor’s authoritative identification. Also, the system uptime is maintained in hundredths of a second, from when the device was last re-initialized. Several additions have been made to this group in the transition to MIB-II, these additions are highlighted in section 4.3.
4.2.2 The Interfaces Group

Managed devices maintain information about the physical network connection in the Interfaces Group. The types of information contained in a MIB-I implementation are restricted to physical layer type data. For example, it might include the number of physical interfaces, the maximum transfer rate, and status information. The Interfaces Group is also the location for network/device loading information. In this group, the numbers of total packets and octets that have arrived or been discarded on an interface, are tallied. Several additions have been made in the transition to MIB-II, and are highlighted in section 4.3.

4.2.3 The Internet Protocol Address Table Group

This group maintains the device’s IP addressing information. Its use is heavily tied to the initial constraint that the MIB be intended for the support of Internet based devices.

The type of information contained in the IP Address Table Group is related to the actual values for the devices interfaces, and an index that uniquely identifies the interface.

4.2.4 The Internet Protocol Routing Table Group

The Internet Protocol Routing Table Group is closely related to routers and gateways, and is most useful for their management. The IP Routing Table contains an entry for each route presently known to this entity.

For each route, a table maintains the destination along with the interface index, metric, nexthop, route type, protocol, and age. The value for nexthop is the IP address of the next IP router on that interface. The protocol corresponds to the protocol used to determine the rout-
ing information, typically the Internet Control Message Protocol (ICMP) [18] or the Exterior Gateway Protocol.

This list of objects is extremely diverse, for the complete listing see [43].

4.2.5 The Internet Control Message Protocol Group

The ICMP protocol involves the efficient distribution for information regarding error and control messages on IP based networks. It allows ICMP software on one machine to communicate errors to ICMP software on another machine. ICMP supports thirteen allocated message types for errors [9].

The data maintained within the MIB relate to the number of ICMP messages received (this counter does not distinguish different ICMP message types), followed by counters for all possible ICMP error packets received and transmitted.

4.2.6 The Transmission Control Protocol Group

The Transmission Control Protocol Group maintains information about specific TCP connections present in the device that is being managed. Objects within this group also maintain transient information about particular TCP connections.

For each TCP connection, a large number of table entries are maintained. Among these are:

1. The number of times TCP connections have made a direct transition to the SYN-SENT state from the CLOSED state.

2. The number of times TCP connections have made a direct transition to the CLOSED state from either the ESTABLISHED state or the CLOSE-WAIT state.
3. The number of TCP connections for which the current state is either ESTABLISHED or CLOSE-WAIT.

4. The total number of TCP segments received and transmitted.

5. And finally, a table containing information about a particular state of a current TCP connection.

This list of objects is extremely diverse, for the complete listing see [43].

4.2.7 The User Datagram Protocol Group

The UDP [9] protocol provides services above the IP protocol layer with the additional property that communicating entities can distinguish among multiple destinations within a given host. UDP services are connection-less in nature, and consequently are unreliable [9].

For each UDP message, a large number of table entries are maintained. For example, the total number of UDP datagrams delivered to UDP users, the total number of received UDP datagrams for which there was no application at the destination port, and the number of received UDP datagrams that could not be delivered for reasons other than the lack of an application at the destination port.

4.2.8 The Exterior Gateway Protocol Group

The Exterior Gateway Protocol is responsible for the transmission of routing update messages to convey information about reachable networks to EGP neighbors. EGP gateways can transmit two types of information to their neighbors. The first involves destination networks that are reachable entirely within the gateway's autonomous systems. The second involves destination networks that the gateway has learned about, which are outside of the autonomous system.
Within the MIB, information maintained concerns:

- The number of EGP messages received without error.
- The number of EGP messages received that proved to be in error.
- The total number of locally generated EGP messages.
- The number of locally generated EGP messages not sent due to resource limitations within an EGP entity.
- The EGP neighbor table and the state of the communication with the neighbor.

### 4.3 MIB-II Modifications and Objects

The changes and improvements made for the MIB-II included:

- **Incremental additions to reflect new operational requirements.**
- **Backward compatibility with MIB-I/SMI and the SNMP.**
- **Improved clarity and readability.**
- **Improved support for multi-protocol entities.**

In addition, when describing certain objects the term *deprecated* was added. This term is intended to signify that the object(s) will eventually be removed from the standard MIB. Thus, any object labeled *deprecated* in MIB-II has the potential of being removed in subsequent standard MIB definitions (e.g., MIB-III). MIB-II only marks one object as *deprecated*, the address translation group.
The MIB-II has also made the modification of adding a type **DisplayString** which is intended to represent data read by humans. The **DisplayString** is a string of octets restricted to the NVT ASCII character set [36]. Currently only two objects are defined using the **DisplayString** datatype in the MIB-II, **sysDescr** and **ifDescr**.

The system group was augmented to contain four additional members, **sysContact**, **sysName**, **sysLocation**, and **sysServices**. The purpose of these new objects is to improve record keeping in regards to the actual location of a managed device and to those who may be personally responsible for the device.

The Interfaces Group has received a large number of MIB-II additions. The first addition involved devices (e.g., MAC-layer bridges), in that they no longer need to support the IP. Also, several new values have been added to the possibilities for the **ifType** object. These include: ppp(23), softwareLoopback(24), ethernet-3Mbit(26), nsip(27), slip(28), ultra(29), ds3(30), sip(31), and frame-relay(32) [44].

The Address Translation Group has been downgraded to the **deprecated** status due to the need to support nodes and devices in a multi-protocol environment (e.g., both Internet and OSI protocol suites). As a solution to the **deprecated** status of the Address Translation Group, network protocol groups will be responsible for maintaining two address translation tables, one for each direction of the mapping. For example, going from IP addresses to physical addresses, and then back to IP addresses.

Two new groups are added in the transition to MIB-II. They are the Transmission Group and the SNMP Group. The Transmission Group has been added to improve MIB-I’s ability to distinguish between different types of physical layers. The SNMP Group is intended to
serve as a location for defining of variables specific to an application. For example, statistical information should be kept in the SNMP subgroup of MIB-II.

4.4 Summary

This chapter has provided an overview of the virtual information store called the Management Information Base. A MIB is a simplistic and streamlined collection of data that provides basic elements of information to the network manager. The Internet MIB has been developed to facilitate the management of system utilizing the TCP/IP protocol suite. However, the Internet MIB is very flexible in design and has consequently been employed for managing the Xunet system.
Chapter 5

The Simple Network Management Protocol

A major advantage to network managers for using SNMP is that it is a specified standard protocol for the Internet community. The majority of network products utilizing TCP/IP and are network manageable are expected to adopt and implement this specification. Therefore, SNMP was chosen as the foundation for management within the current Xunet II system.

The SNMP protocol is derived from previous work done in the area of network management. The main thrust of this research was to derive an efficient means for network managers to maintain an ever growing internet. The product of this early work was the Simple Gateway Monitoring Protocol (SGMP) [16]. Considerable changes have been made in the transition to the SNMP, therefore, the resulting protocol is not backward-compatible with SGMP. However, the original philosophy, design decisions, and architecture remain intact. In order to avoid confusion, new UDP ports were allocated for use by the SNMP.
A basic principle of SGMP and SNMP are simplicity and efficiency. It is a simple protocol which requires few resources. This philosophy is seen in the number of operations that are exported, that is, the exterior view of a functional agent. The operation available are:

- **get** - retrieves specific entry of management information.

- **get-next** - retrieves several entries of management information.

- **set** - modifies the value of an entry of management information.

- **trap** - reports exceptions or extraordinary events.

The managed network architectural model consists of one or more management station responsible for a number of network elements. This relationship is illustrated in Figure 5.1. The network management stations execute applications that monitor and control network elements. The network elements are not restricted by device type and may be any of the following: hosts, gateways, terminal servers, etc. These devices have a process responsible for maintaining the virtual store of the MIB.
Within this architecture, SNMP communicates management information between the network management stations and the agents. The goals for the SNMP working group [4] are:

1. Minimize development cost for management agent software necessary to support the protocol;

2. Increase the degree of management function that is remotely supported, thereby admitting fullest use of internet resources in the management task,

3. Increase the degree of management function that is remotely supported, thereby imposing the fewest possible restrictions on the form and sophistication of management tools,

4. Simplify the sets of management functions to ensure comprehension and utilization by developers of network management tools.

5.1 Elements of the SNMP Architecture

In the communication between a network manager and agent, the messages are encoded in a restricted subset of non-aggregate data types of ASN.1. The restrictions are related to the concept that the agent provides simple services and is easily implemented. The use of ASN.1 is, in part, due to its successful use in SNMP’s predecessor SGMP.

An additional impetus for the use of ASN.1 encoding is the eventual transition to OSI based management protocols. The set of ASN.1 types incorporated into SNMP is slightly larger than that of SGMP. For a complete listing, see section 3.

A final simplification to the ASN.1 encoding involves a set of limitations on the basic encode rules (BER) [21]. These restrictions state that all encoding use the definite-length form and non-constructor encodings, rather than constructor encodings.
5.1.1 Operations Supported on Management Information

Interaction between an SNMP manager and agent can be subdivided into one of two possible categories. First, the interaction is restriction to only the inspection of managed information in which the manager asynchronously polls the agent. Second, the modification of managed information where the manager is allowed to request that a object’s value be changed.

This strategy places the burden of monitoring the network state with the manager’s polling frequency and type of data polled. However, to provide the agent with the ability to communicate extraordinary (traps) events in a timely fashion, a limited number of un-requested messages have been defined within the SNMP protocol. These are to act as a guide to the manager so that it can better focus the polled objects. The limitation on the number of un-requested messages is an attempt to reduce the burden on agent implementors. For more information on the implementation of SNMP traps see [41].

5.1.2 Administrative Framework

The administrative framework concerns policies, such as authentication and data accessibility, used between the communicating SNMP applications. It should be noted that the most recent version of SNMP has a very simplistic administrative framework. Currently work is being conducted to improve deficiencies [15, 29].

Within SNMP, a group of managed device is considered to belong to, or be encapsulated by a community which is typically restricted to a geographical area or domain. The concept of a community is extended within the agent since the agent will utilize a community name in its authentication process. A manager wishing to query an agent will construct a message for that agent which contains the community name with which the manager wishes to be associated.
It is the agent’s responsibility to authenticate the manager based on the *community* provided. Since the *community* is not being encrypted, there is serious potential for compromised security.

Once the agent has determined in which *community* the manager is a member, it can then allow the manager to have either a complete view of its information or allow access to only a subsection. The names of the object types represented in a SNMP MIB view need not belong to a single sub-tree of the object type name space. Furthermore, the agent uses the *community* information to restrict the ability of the manager to modify the agents database.

The SNMP agent also supports the concept of an access mode, which may be either READ-ONLY, or READ-WRITE. The combination of the access mode with the view is called the community profile. The profile is then used by the agent to specify access privileges to objects in the various MIB views. The relationship of MIB views, access modes and access privileges as stated in RFC-1157, are as follows:

- If the variable is defined in the MIB with “Access” of “none”, it is unavailable as an operand for any operator;

- If the variable is defined in the MIB with “Access” of “read-write” or “write-only” and the access mode of the given profile is READ-WRITE, then that variable is available as an operand for the `get`, `set`, and `trap` operations;

- Otherwise, the variable is available as an operand for the `get` and `trap` operations.

- In those cases where a “write-only” variable is an operand used for the get or trap operations, the value given for the variable is implementation-specific.
5.1.3 MIB References

MIB references from the manager to the agent are always restricted in scope to a single agent. And within that agent, the references to objects are implemented with unique variable names. Naming of objects within the agent is done in one of two ways. It can either be accomplished in one of the Internet-standard MIB documents, or any SMI compliant definition document. When a manager exchanges information with an agent, the object or group of objects is explicitly referred by the variable name. These variable names are of the dot notation OBJECT IDENTIFIER introduced in section 3.

5.2 Protocol Specification

In order to minimize the complexity of the management agents, the exchange of SNMP messages requires only an unreliable datagram service. Primarily, this service is provided by the User Datagram Protocol and every message is entirely and independently represented by a single UDP datagram. The SNMP protocol has been standardized to use UDP port number 161 for manager message exchanges and UDP port number 162 for messages of type trap. Correctly constructed messages can never exceed 484 octets. However, it is not required, but the SMI recommends, that agents be able to process larger datagrams.

Each SNMP message is made up of three parts: version number, community name, and protocol data unit (PDU). The PDU portion of the UDP pay-load can be one of five types: GetRequest-PDU, GetNextRequest-PDU, GetResponse-PDU, SetRequest-PDU, and Trap-PDU. For example, Figure 5.2 shows the ASN.1 definitions for these PDUs.
RFC1157-SNMP DEFINITIONS ::= BEGIN
IMPORTS
    ObjectName, ObjectSyntax, NetworkAddress, IpAddress, TimeTicks
    FROM RFC1155-SMI;
-- top-level message
Message ::= 
    SEQUENCE {
        version       -- version-1 for this RFC
            INTEGER {
                version-1(0)
            },
        community    -- community name
            OCTET STRING,
        data         -- e.g., PDUs if trivial
            ANY       -- authentication is being used
    }
-- protocol data units
PDUs ::= 
    CHOICE {
        get-request
            GetRequest-PDU,
        get-next-request
            GetNextRequest-PDU,
        get-response
            GetResponse-PDU,
        set-request
            SetRequest-PDU,
        trap
            Trap-PDU
    }
END

Figure 5.2: SNMP Protocol Data Units.
5.2.1 Elements of Procedure

As stated in section 5.1.2, the process of network management in the SNMP environment is typically driven by manager initiated polling. On the manager side of the exchange, this process takes place in four steps:

1. Construct a ASN.1 PDU signaling the request (e.g., GetRequest-PDU).

2. Deliver the PDU along with its community name, transport address, and target address to the authentication service.

3. Construct an ASN.1 message object.

4. Encode the message with the BER of ASN.1, then serialize and send the message to the managed device.

Upon reception of the SNMP message from a device (agent), the following steps are performed. If any of the tests fail, the message is discarded:

1. Parse the datagram to re-construct the ASN.1 message object.

2. Verify the version number.

3. The agent then attempts to validate the authentication of the received message using the community name, user data, source and destination addresses. The agent's authentication scheme should either return an ASN.1 object or signal an authentication failure. In the case of a failure, the agent can generate an authentication failure trap PDU and transmit it to a preassigned manager.
4. If the authentication check passes, the agent parses the returned ASN.1 object to construct the requested object. Should the process of determining if the agent contains the object fail, then it discards the datagram and performs no further action.

5.2.2 Common Constructs

This section covers protocol data units that may be utilized within the SNMP protocol. Figure 5.3 illustrates the common construct and the following is an explanation of these construct.

Since the SNMP is specified to utilize a connection-less transport service, the inclusion of RequestIDs is necessary so that communicating entities can differentiate messages. The RequestIDs also provide the entities with the ability to determine if the message received is a duplicate. It also determines the order of messages.

The ErrorStatus determines if an exception occurred while processing a request. A non-zero ErrorStatus is then used as an index to one of five possibilities. The VarBind pairs a variable name with its value. Each VarBind entry contains two fields, the object name and its syntax or data type. A list of VarBind entries is then constructed via a VarBindList. The VarBindList is a simple list of names and their corresponding values.

5.2.3 GetRequest Protocol Data Unit

Figure 5.4 illustrates a message type of GetRequest-PDU. This message type is generated by a manager interested in querying the value of an agent object. The agents’ response to the GetRequest-PDU depends on a number of standardized rules:

- If the agent determines that the value of the VarBind field does not match a value maintained in its virtual store, then it returns the identical GetRequest-PDU with the
-- request/response information
RequestID ::= 
   INTEGER
ErrorStatus ::= 
   INTEGER {
      noError(0),
      tooBig(1),
      noSuchName(2),
      badValue(3),
      readOnly(4)
      genErr(5)
   }
ErrorIndex ::= 
   INTEGER
-- variable bindings
VarBind ::= 
   SEQUENCE {
      name
      ObjectName,
      value
      ObjectSyntax
   }
VarBindList ::= 
   SEQUENCE 
   OF 
   VarBind

**Figure 5.3:** SNMP Common Constructs.
value of the **ErrorStatus** set to *NoSuch.Name*. The value of the error-index field is the index of the object name component in the received message.

- If the agent determines that the value of the **VarBind** field is an aggregate type (one that is not permitted), then it returns the identical **GetRequest-PDU** with the value of the **ErrorStatus** set to *NoSuch.Name*. The value of the error-index field is the index of the object name component in the received message.

- If, in response to the **GetRequest-PDU**, the agent determines the size of the **GetResponse-PDU** exceeds the limitations placed on messages, then it returns the identical **GetRequest-PDU** with the value of the **ErrorStatus** set to *tooBig*. The value of the error-index field is zero.

- If the agent determines that the value of the **VarBind** can not be retrieved for some reason that is not covered by one of the above rules, then it returns the identical **GetRequest-PDU** with the value of the **ErrorStatus** set to *genErr*. The value of the error-index field is the index of the object name component in the received message.

If after processing the **GetRequest-PDU** messages, and passing the tests above, then the agent responds to the originator of the message with a **GetResponse-PDU**. Within the **GetResponse-PDU**, the agent places the name and value of the objects named in the **VarBind** field. The value of the **ErrorStatus** field of the **GetResponse-PDU** is set to *noError*. The value of the error-index field is zero. The value of the request-id field of the **GetResponse-PDU** is that of the received message.
The form of the GetRequest-PDU is:
GetRequest-PDU ::= [0]
   IMPLICIT SEQUENCE {
       request-id
       RequestID,
       error-status -- always 0
       ErrorStatus,
       error-index -- always 0
       ErrorIndex,
       variable-bindings
       VarBindList
   }
( ipRouteMetric1.9.1.2.3 = 3 ))

If the management station is interested in continuing with the table retrieval, it would then issue the following PDU, to which the agent would supply its next entry.

```c
GetNextRequest ( ipRouteDest.9.1.2.3,
ipRouteNextHop.9.1.2.3,
ipRouteMetric1.9.1.2.3 )
```

This stop and go exchange will continue until the table has been exhausted. The manager can determine if the routing table has been exhausted because the OBJECT IDENTIFIER prefix will change.

In Figure 5.5, the ASN.1 definition of the `GetNextRequest` is illustrated. This PDU is identical to the `GetRequest-PDU` except for the type, therefore, many of the same rules apply.

When an `GetNextRequest-PDU` is received the agent performs the following tests:

1. If the agent determines that the value of the `VarBind` in the field does not lexicographically precede the name of some object available for the get operations in the current view of the virtual store, then the receiving entity returns to the originator a message of type `GetResponse-PDU` with the value of the `ErrorStatus` set to `noSuchName`. The value of the error-index field is the index of the object name component in the received message.

2. If in response to the `GetRequest-PDU`, the agent determines that the size of the `GetResponse-PDU` exceeds the limitations placed on messages, then it returns the
GetNextRequest-PDU ::= [1]

IMPLICIT SEQUENCE {
    request-id
        RequestID,
    error-status -- always 0
        ErrorStatus,
    error-index -- always 0
        ErrorIndex,
    variable-bindings
        VarBindList
}

Figure 5.5: SNMP GetNextRequest-PDU.

identical GetRequest-PDU with the value of the ErrorStatus set to tooBig. The value of the error-index field is zero.

3. If the agent determines that the value named in the VarBind fields lexicographical successor cannot be retrieved for some reason that is not covered above, then the agent returns to the originator a message of type GetResponse-PDU with the value of the ErrorStatus set to genErr. The value of the error-index field is the index of the object name component in the received message.

After processing the GetRequest-PDU messages, and passing the tests above, the agent responds to the originator of the received message with a GetRequest-PDU. Within the GetRequest-PDU, the agent places the name and value of the object named in the VarBind field and the lexicographical ordering of the names of all objects available for get operations in the relevant MIB view. The value of the ErrorStatus field of the GetRequest-PDU is set to noError. The value of the error-index field is zero. The value of the request-id field of the GetRequest-PDU is that of the received message.
GetResponse-PDU ::=  
[2]
IMPLICIT SEQUENCE {
    request-id
    RequestID,
    error-status
    ErrorStatus,
    error-index
    ErrorIndex,
    variable-bindings
    VarBindList
}

**Figure 5.6:** SNMP GetResponse-PDU.

### 5.2.5 GetResponse Protocol Data Unit

The **GetResponse-PDU**, as described above, is the PDU that returns from the agent in response to either of the **GetRequest-PDU**, **GetNextRequest-PDU**, or **SetRequest-PDU**. The ASN.1 syntax for the **GetResponse-PDU** is shown in Figure 5.6. As can be seen, this PDU is similar to the **GetRequest-PDU**. This type of PDU is almost exclusively processed by the network manager application.

### 5.2.6 SetRequest Protocol Data Unit

Within the SNMP environment a limited set of operations are allowed for the modification of the virtual store. These MIB modifications are requested via the SNMP **SetRequest-PDU**. The ASN.1 language definition of the message in shown in Figure 5.7.

When an **SetRequest-PDU** is received the agent performs the following tests:

1. If the agent determines that the value of the **VarBind** in the field is not available for set operations in the MIB view, then the receiving entity sends to the originator of the
SetRequest-PDU ::= 
[3] IMPLICIT SEQUENCE {
  request-id
    RequestID,
  error-status  -- always 0
    ErrorStatus,
  error-index  -- always 0
    ErrorIndex,
  variable-bindings
    VarBindList
}

Figure 5.7: SNMP SetRequest-PDU.

message GetResponse-PDU of the same form as the request with the exception that the ErrorStatus set to noSuchName. The value of the error-index field is the index of the object name component in the received message.

2. If in response to the SetRequest-PDU the agent determines that the value the manager is attempting to place into the MIB does not conform to the objects type, length, or value that is consistent with that required for the variable, then it returns the identical GetRequest-PDU with the value of the ErrorStatus set to badValue.

3. If in response to the SetRequest-PDU, the agent determines that the size of the GetResponse-PDU would exceed the limitations placed on messages, then it returns the identical GetRequest-PDU with the value of the ErrorStatus set to tooBig. The value of the error-index field is zero.

4. If the agent determines that the value named in the VarBind field cannot be modified for some reason that is not cover above, then the agent returns to the originator a message of
type GetResponse-PDU with the value of the ErrorStatus set to genErr. The value of the error-index field is the index of the object name component in the received message.

After processing the SetRequest-PDU messages, and passing the tests above, the agent responds to the originator of the received message with a GetRequest-PDU. Each of the objects named in the VarBind field of the received message will have its value modified to that of the PDUs. The modification to the MIB should be implemented in an atomic fashion and it is typically implemented in two phases. In the first phase, all the values are copied. In the second phase, when the agent has determined that all of the values can be safely modified, the MIB is updated.

The agent then responds to the sending manager with a GetResponse-PDU. The value of the ErrorStatus field of the GetResponse-PDU is set to noError. The value of the error-index field is zero. And, the value of the request-id field of the GetResponse-PDU is that of the received message.

5.2.7 Trap Protocol Data Unit

In the SNMP model of network management, the responsibility for acquiring data from the managed devices is placed with the managers in their polling frequency and type data polled. To provide the agent with the ability to communicate extraordinary (traps) events in a timely fashion a limited number of unsolicited messages have been defined. These messages act as a guide to the manager so that it may better focus the polled objects. There is a limitation on the number of these unsolicited messages due to the attempt to reduce the burden on agent implementors and keep the protocol simple.

The ASN.1 language for a trap is shown in Figure 5.8. The fields within the PDU are:
• enterprise - The value of the agent’s sysObjectID.

• agent-addr - The value of the agent’s NetworkAddress.

• generic-trap - One of the possible trap causing events (describe below).

Within this language seven different types of generic-traps are defined to be:

- The coldStart Trap - This trap type is sent by the agent to signify that it is (re)initializing itself.

- The warmStart Trap - This trap type is sent by the agent to signify that it is (re)initializing itself, but the objects within the view are not to be altered.

- The linkDown Trap - This trap type is sent by the agent when it determines that an interface has changed to a down status (communication failure). The PDU will contain the first element of the VarBind, and the name and value of the changed ifIndex interface.

- The linkUp Trap - This trap type is sent by the agent when it determines that an interface has changed to an up status (communication re-established). The PDU will contain the first element of the VarBind, and the name and value of the changed ifIndex interface.

- The authenticationFailure Trap - This trap type is sent by the agent when it determines that an SNMP message has arrived and has failed the authentication check.

- The egpNeighborLoss Trap - This trap type is sent by the agent when it determines that an EGP peer has been marked down and the peer relationship no longer
remains. The PDU will contain the first element of the **VarBind**, and the name and value of the changed ifIndex interface.

- **The enterpriseSpecific Trap** - This trap type is sent by the agent when it determines that one of its **enterpriseSpecific** events has occurred. The *specific-trap* field identifies the particular trap which occurred.

  - **specific-trap** - The value of the agent’s **enterpriseSpecific**, else zero.
  
  - **time-stamp** - The value of the agent’s **sysUpTime** object at the time the event occurred.
  
  - **variable-bindings** - The variable list containing information about the trap.

### 5.3 Summary

This chapter provide an overview of the Simple Network Management Protocol (SNMP), its interaction with a network manager, and various message types. The SNMP is the basis for current implementation of network management system on todays TCP/IP internets. The majority of manageable network products utilizing TCP/IP and are expected to adopt and implement the SNMP specification. Therefore, SNMP was chosen as the foundation for management within the current Xunet II system.
Trap-PDU ::= [4]
IMPLICIT SEQUENCE {
  enterprise -- type of object generating
    -- trap, see sysObjectID
    OBJECT IDENTIFIER,
  agent-addr -- address of object generating
    NetworkAddress, -- trap
  generic-trap -- generic trap type
    INTEGER {
      coldStart(0),
      warmStart(1),
      linkDown(2),
      linkUp(3),
      authenticationFailure(4),
      egpNeighborLoss(5),
      enterpriseSpecific(6)
    },
  specific-trap -- specific code, present even
    INTEGER, -- if generic-trap is not
    -- enterpriseSpecific
  time-stamp -- time elapsed between the last
    TimeTicks, -- (re)initialization of the network
    -- entity and the generation of the
    trap
  variable-bindings -- ‘interesting’ information
    VarBindList
}

Figure 5.8: SNMP Trap-PDU.
Chapter 6

Xunet Agent

6.1 Introduction

An agent is a software program housed within a managed network device that stores management data and responds to the manager’s requests for the data. An agent process provides the manager with MIB data constrained that it tax the managed device minimally.

Most current SNMP agents are down loaded to the managed device at system start up time. These agents typically require at least 50 KB of program code and memory [52]. Actual memory requirements depend on the functionality of the agent with the following guidelines [5]: 32 KB is required for an agent without a MIB or support for timers; extra protocols require 48 KB, plus an additional 0.25 KB for each MIB variable supported by the agent.

Implementing the agents can take several forms, depending on the constraints of the system. Over the course of developing the Xunet agent different constraints were examined and possible design tradeoffs were weighed. The following sections detail various techniques for implementing
agents. Their strengths and weaknesses are illustrated with respect to their applicability to the Xunet project.

6.2 Possible Agent Implementations

6.2.1 SNMP MUX Protocol

On typical systems, an agent speaking SNMP is often implemented as a user-process that reads kernel variables in order to realize the MIB. This approach works if the information needed resides the kernel or in stable storage. However, it is difficult to have user-processes implement network services communication between the SNMP agent and other processes.

One technique used to solve this problem uses a protocol called the SNMP multiplexing (SMUX) [46] protocol. When a user-process (SMUX peer) wishes to export a MIB, it initiates a SMUX association to an SNMP agent, registers itself, and fields management operations for objects within its MIB.

It is possible to generalize the SNMP agent to know only about the SNMP group of MIB-I. All other portions of the MIB can be implemented by other processes. This allows a computer manufacturer to provide SNMP access for its operating system in binary form.

There are two approaches defined in [46] that can be used when trying to integrate MIBs with the SNMP agent: request-response and cache-ahead.

The request-response model propagates the SNMP requests received by the SNMP agent to the process that exports the MIB. The SMUX peer then performs the required operation(s) to retrieve the MIB object and returns a response to the agent. The SNMP agent propagates this response to the NMS. This request-response model is agent-driven since the SNMP agent initiates all transactions.
The cache-ahead model requires the SMUX peer to periodically update the SNMP agent with its MIB subtree objects. The SNMP agent will respond to NMS requests with locally maintained information. SNMP set requests should be passed immediately to the SMUX peer. This cache-ahead model is peer-driven in that the SMUX peer initiates all transactions.

Both the request-response and cache-ahead models have advantages and disadvantages. Therefore, the SMUX protocol supports both models via the following technique. The protocol between the SNMP agent and the SMUX peer is based on the request-response model and the SMUX peer. But, SMUX peer may also employ the cache-ahead model with other user processes.

In keeping with the fundamental design decisions of SNMP, the implementation of the SMUX protocol has been kept very simple. An SNMP agent should listen for incoming connections and, upon establishing a connection, the SMUX peer issues an OpenPDU to initialize the SMUX association. The SNMP agent can either accept or reject the association. If the agent rejects the connection, it will issue a closePDU and close the TCP connection. If the agent is willing to accept the association, then no response is sent to the SMUX peer.

After an association has been established between the SNMP agent and the SMUX peer each subtree defined in the MIB must be register with the SNMP agent. The registration procedure begins with the SMUX peer issuing a RReqPDU. When the SNMP agent receives a RReqPDU, it issues a reply RReqPDU. The SNMP agent then returns a RRspPDU in the same order that the RReqPDU’s were received.

The SMUX peer has the added capability of notifying the NMS of exceptions (traps). When the SMUX peer issues a trap, it sends a SNMP Trap-PDU to the SNMP agent. When the
SNMP agent receives a **Trap-PDU**, it propagates the message to the NMS that has been configured to accept exception messages.

Once the SMUX peer has completed MIB registration, the communication between the NMS and SMUX peer is completely governed by the SNMP agent. The SNMP agent may receive SNMP **GetRequest-PDU**, **GetNextRequest-PDU**, or **SetRequest-PDU** which includes one or more variable-bindings within a subtree registered by a SMUX peer. It is the SNMP agent’s responsibility to send an equivalent SNMP PDU containing only those variables within the subtree registered by that SMUX peer to the appropriate SMUX peer. When the SMUX peer receives the forwarded PDU, it performs the necessary local operations and replies with a corresponding SNMP **GetResponse-PDU**. The SNMP agent then correlates this result and propagates the resulting **GetResponse-PDU** to the NMS.

When either the SNMP agent or the SMUX peer wishes to release the SMUX association, the **ClosePDU** is transmitted. The connection is then closed and all subtree registrations for that SMUX peer associations are released.

The SMUX protocol provides a very robust system for development of MIB extensions. The ability provides a *firewall* between existing agents and the experimental extension and allows vendors to distribute agents in binary form. However, the SMUX approach for managing the Xunet project has been abandoned for the simple lack of widespread acceptance for the protocol. The target platform, a Silicon Graphics workstation, did not provide any support for the protocol. In addition, sampling a variety of local agents turned up no support for SMUX. To date, only one environment supports the protocol, the ISODE [40]. Consequently, another technique was needed for the Xunet agent.
6.2.2 Proxy Agent

Proxy agent software permits an SNMP manager to monitor, or control agents that are not directly addressable via the transport protocol. Several examples are possible. A vendor may already have a proprietary management scheme implemented and may want to provide support for SNMP. In this case, an intermediate host would be used to convert the proprietary protocol into the SNMP format, see Figure 6.1. Similarly, SNMP proxy agents may protect other SNMP agents from redundant network management requests through the use of caches. SNMP proxy agents may also implement elaborate MIB access policies.

The proxy agent daemon:

- Listens for SNMP queries and commands from logically remote network management stations;
- translates and retransmits those as appropriate network management queries or cache lookups,
- listens for and parses the responses,
- translates the responses into SNMP responses,
• returns those responses as SNMP messages to the network management station that are the originators of the transaction.

The proxy agent daemon can also emit SNMP traps to identified trap receivers. Finally, proxy agents are used when the managed device is incapable of implementing or understanding the SNMP datagram protocol stack. In this case, a proxy agent would act as the middleman between the device and the network management station. Requests could be sent to the device over a serial link and be held at the proxy agent until polled. This type of management is used for devices like printers, uninterruptable power supplies, and climate control equipment.

Proxy agents are provided by several vendors [37]. The proxy agent daemon is designed to make the addition of vendor specific variables a straight forward task. Proxy applications typically come with source code, including a set of portable libraries for generating and parsing SNMP messages, and a set of command line utilities.

Initially, the proxy agent approach to network management was utilized for the Xunet switch control hardware. Communication with the switching fabric was implemented via an intermediary host. However, this approach was rendered obsolete due to improvement in the switch control computer. The modifications to the control computer allowed direct communication via a datagram protocol with the NMS. This allowed the technique of a ASN.1 Embedded Agent, described below, to be used for the final version of the Xunet SNMP agent.

### 6.2.3 SNMP Library Agent

The final approach examined, and ultimately used for the Xunet SNMP agent, is the technique of utilizing an extensible system of ASN.1/SNMP libraries. Extensible libraries allow the agent builder to concentrate on the tasks of retrieving and storing MIB objects. The details of ASN.1
encapsulation, decapsulation, and trap generation are left to the library routines. The agent libraries examined provide a full, RFC-compliant, unit-tested body of source code [1, 37]. These agent libraries are independent of the transport layer rules and impose no requirements to the underlying communications medium. They also make no assumptions about the operating system, and are designed to be installed under a variety of operating systems.

By utilizing an application programming interface (API), the development of the agent allows for the use of a well defined and easy to use interface. Testing of management submodules can be accomplished independently from the development of the master agent and other sub-agent modules. This allows a managed device to have a single SNMP agent that handles the entire set of managed objects.

6.2.4 Xunet Agent Organization

In an SNMP API agent, the core of the code is a set of functions called by local transport layer modules. The agent is responsible for calling the functions developed by the MIB designer for reading and writing instances of object variables. Consequently, any number of MIBs can be implemented simultaneously, MIBs can be dynamically added or deleted for the NMS’s view, and all added MIBs will appear as a single MIB.

The interactions between the MIB-access code and the agent libraries are as follows:

1. Initialize local environment. This includes opening MIB data files, opening sockets [9] for interprocess/NMS communication, and configuring trap handling information;

2. Listen for SNMP packet from the NMS,

3. Use library code to parse and decode ASN.1 packets. An authorization check is performed, and, if the check fails, a trap message is sent to the appropriate NMS,
4. Find MIB variable in virtual store and check the access rights. If permitted, read or write instance of MIB variable,

5. Call library code to construct ASN.1 response packet,

6. Send response packet to NMS that issued the request packet,


The library code utilized in the agent is a product developed by Paul Freeman Associates, Inc [1]. The core code consists of several modules with few interdependencies. The modules of major concern perform input packet decomposition, analysis, and processes packet variables using the locally provided services of the MIB variable interface module. A trap module constructs an authenticationFailure packet when the access verification module is unable to map an authentication datum (see section 3). Modules also provide ASN.1 encoding and decoding.

MIB access is accomplished via a set of agent callable functions to read and write object instances. A MIB compiler is used to generate data structure from SMI definitions. The agent developer supplies the MIB variables access functions along with view configuration information (see section 5.1.2).

Three functions are required for entry points for each MIB supported. The first module initializes the data area used by the MIB objects. This function is called once for each configured MIB upon agent startup.

The second MIB function is a state indication function. This function is used to signify the beginning and ending of the process of receiving a SNMP packet and to allocate resources in a way that is synchronous with packet processing. Examples of these resources are temporary
buffers and kernel access locks. The function is called before variable is accessed from any MIB module and after all processing is complete.

Finally, a function is provided to retrieve or set a MIB variable. This function takes as arguments the objects identifier, name length and access requested. The return value is that of the found variable or an error code. This function allows the agent to process GetNextRequests and maintains a pointer to the retrieved variable value so that subsequent requests start at the correct entry in the MIB. All sets are performed in a two phase commit, as specified in section 5.2.6.

Heavyweight multi-threading is implemented via the error code returned by a MIB module. If a MIB module is required to process and return a value for an object that could potentially take a long time, it can mark the variable as being processed and then fork to field further requests. The child process completes the processing of the current packet. Upon completion, the parent is informed and the child terminates. Finally, the parent process releases the marked variable for further access and updates the SNMP MIB received statistics.

Compliance with the various traps defined in section 5.2.7 has been implemented in the Xunet agent. Traps other than authenticationFailure can be generated by the agent through a set of trap building functions. The agent generates traps for coldStart and warmStart events and traps for controller warnings, like high temperature. When the agent determines that a trap should be sent, it can call the snmp_build_trap() function. The snmp_build_trap() function takes as arguments the trap type (e.g. coldStart, warmStart, linkDown, ...), the output data, enterprise name, and local IP address. The function returns a pointer to the beginning of the trap PDU. The NMS trap recipient is specified in the startup configuration data.
6.3 Summary

As shown by the above description, the implementation of low level details is left to the SNMP libraries. The core code provides a clean and easy way to use a set of functions for ASN.1 encoding/decoding, authentication, view management, and trap processing. The agent developer is only concerned with the details of the MIB to be supported, and the functions necessary to retrieve and set MIB objects. For these reason and the robustness of the library, this system was chosen for the Xunet SNMP agent.
Chapter 7

Xunet MIB

7.1 Introduction

This chapter describes the collection of managed object that make up the Xunet MIB. The goal of the Xunet MIB is to define a concise set of managed objects used to instrument the Xunet switch.

A first priority of the MIB is managing quickly accessible objects. In addition, the design of the MIB was always constrained by the fundamental SNMP axiom of a simple and small collection of objects. The managed physical media is restricted to the connection between the router and the switching hardware. However, the implementation of the agent code has made allowances for support of a DS3 or SONET MIB. It should also be noted that the final version of the Xunet MIB is closely related to, and influenced by, the MIBs developed in various standards bodies [14, 12, 39, 53].

The actual implementation of the Xunet agent includes MIB-II support with the added extension of the MIB describe below. The current implementation supports SNMP version 1.
As shown below, by subdividing the task the problem of developing the MIB was addressed in several stages.

7.2 Managed Objects

For reasons of clarity, the MIB objects contained in the SMI compliant definition are ordered into two hierarchies, hardware and software. Each of these groups are then further subdivided to achieve fine grain granularity. The NMS can obtain information on the entire switch and if desired focus on the part of the switch that contributes to the aggregate information. Before describing each section in detail, the following descriptions provide a brief overview. The complete Xunet MIB is included in Appendix A.

The top level of the hierarchy addresses objects that relate to the entire switch and system faults. Further subdivisions are made below the top level for a board group and a board module group. At this level items of interest are the size of translation memory, hardware version numbers, number of line modules installed and error on these modules. The lowest levels contain line card statistics on buffer status/utilization, and cells dropped due to overflow.

The MIB hierarchy allows the NMS to view the routing and VPI/VCI translation information in several levels of granularity. The highest level will contain general information like the software version and aggregate information for all the the VPIs that the switch in currently processing. Further subdivisions are then made on a board and VCI basis. For example, all of the cells transmitted on a single VCI can be polled by the NMS.

The follows sections detail each group within the Xunet MIB.
7.2.1 The Top Level Switch Objects

The switch group is the concentration point of general switch information on a line card level. The group contains six entries:

- **XunetHardwareVersion** - The version number of the Xunet hardware.
- **XunetSoftwareVersion** - The version number of the Xunet switch controller software.
- **XunetMaxPaths** - The maximum number of paths the translation memory can support.
- **XunetMaxChannels** - The maximum number of channels the translation memory can support.
- **XunetAtmAddress** - The network address of the switch.
- **XunetUptime** - The length of time the switch has been operational.

Currently, five entries are read from a configuration file. The last entry is determined from the control computers uptime. Subsequent releases of the agent could determine its address via a bootp [9] type protocol and maximum path and channel number can be gained by directly examining the translation memory.

7.2.2 The Second Level Line Card Objects

Each line card within the switch is responsible for maintaining its own set of objects. The ability to maintain the objects has been provided at every step in the switch design and SNMP only facilitates access to this information.

Line cards are made up of two components, a physical connection to the network and a queuing engine (see section 2.2.2). In this group MIB support is primarily concerned with buffer information.
The entries in the line card group are tables for each line cards. Each table contains the following information:

- **obufIntType** - interface type of line card interface to network.
- **obufType** - specifies the queuing discipline used for servicing the buffer.
- **obufOperStatus** - operational status designates if the buffer is enabled or disabled.
- **obufBufferSize** - buffer size in cells.
- **obufQueueLength** - number of cells currently queued in buffer.
- **obufOverflows** - number of cells dropped due to buffer overflow.

The only physical layer information is stored in the interface type. As previously stated, additional MIB support for the physical layer can be provided via other MIBs.

### 7.2.3 Third Level Line Card Objects

In the interest of generality, the MIB is design with as few architectural specific entries as possible. The current Xunet switching hardware supports only one input/output port per line card. However, the MIB does not contain this restriction, each line card can contain several ports. All ports are then indexed under their respective line cards and each index contains a table with the following entries:

- **portNumber** - the number of this ports on the line card.
- **portAdminStatus** - the current desired status of this port, up or down.
- **portOperStatus** - the current status of this port, up or down.
- **portStatusTime** - the time since this port has changed state.

- **portRemoteAtmAddress** - the address of the device connected to this port.

- **portRemoteIpAddress** - the device IP address connected to this port.

- **portMaxPathsIn** - the maximum number of input VPIs on this port.

- **portNumPathsIn** - the current number of VPIs active on this port.

- **portMaxBandwidthIn** - the maximum cells per second incoming on this port.

- **portAllocBandwidthIn** - the allocated bandwidth in cells per second for this port.

- **portUsedBandwidthIn** - the cell per second bandwidth actually being used.

- **portReceivedCells** - number of cells received counter.

Due to the simplex nature of the traffic on the switch, each input object has a corresponding output entry.

- **portMaxPathsOut** - the maximum number of output VPIs on this port.

- **portNumPathsOut** - the current number of VPIs active on this port.

- **portMaxBandwidthOut** - the maximum outgoing cells per second on this port.

- **portAllocBandwidthOut** - the allocated bandwidth in cells outgoing per second for this port.

- **portUsedBandwidthOut** - the cell per second bandwidth actually being used.

- **portTransmittedCells** - the number of cells transmitted.
7.2.4 Forth Level Virtual Path Group

In keeping with the hierarchy of managed objects, the next set of objects drops one level lower in the utility for a NMS’s view. A virtual path is defined as a conceptual path through the network and a single virtual path may contain several channels. The virtual path entries are composed of a table of eleven items as follows:

- **pathPort** - the input line card of this path.
- **pathVPI** - the input VPI of this path.
- **pathStatus** - the operational status of the path.
- **pathNumOutputs** - the number of destination ports (multicast)
- **pathMaxChannels** - the maximum number of VC’s allowed on path.
- **pathNumChannels** - the current number of VC’s allocated.
- **pathMaxBandwidth** - the maximum allowable bandwidth of the path.
- **pathAllocBandwidth** - the current allocated bandwidth of the path.
- **pathUsedBandwidth** - the amount of bandwidth actually being used.
- **pathCells** - the number of cells transmitted on this path.
- **pathUptime** - the amount of time the path has existed.

As previously stated, calls are by default simplex, hence, different parameters may exist for the data flow in the opposite direction. The objects for the reverse direction are:

- **opathPort** - the output line card of this path.
- **opathVP1** - the output VPI of this path.

- **opathStatus** - the operational status of the path.

- **opathMaxChannels** - the maximum number of VC’s allowed on path.

- **opathNumOutputs** - the number of destination ports (multicast)

- **opathNumChannels** - the current number of VC’s allocated.

- **opathMaxBandwidth** - the maximum allowable bandwidth of the path.

- **opathAllocBandwidth** - the current allocated bandwidth of the path.

- **opathUsedBandwidth** - the amount of bandwidth actually being used.

- **opathCells** - the number of cells transmitted on this path.

- **opathUptime** - the amount of time the path has existed.

### 7.2.5 The Fifth Level Virtual Channel Group

The final level of the MIB hierarchy is the virtual channel group. Virtual channels are the last level of abstraction for the virtual circuits traversing the network. The channel group is made up of a table containing an entry for each channel allocated.

Each entry in the table contains 9 items as follows:

- **chanPort** - the input line card of this path.

- **chanVPI** - the input VCI of this channel.

- **chanStatus** - the operational status of the channel.

- **chanNumOutputs** - the number of destination ports (multicast).
• **chanAllocBandwidth** - the current allocated bandwidth of the channel.

• **chanUsedBandwidth** - the amount of bandwidth actually being used.

• **chanCells** - the number of cells transmitted on this channel.

• **chanUptime** - the amount of time the channel has existed.

### 7.3 Summary

It should be noted that the current version of the MIB has areas for growth and improvement. There are no provisions made for the support of end to end configuration and status information. This type of information would help the NMS to determine if there was a high degree of integrity to the virtual circuits.

There is also no support for ATM adaptation layer statistics. The decision to exclude this information was due to the fact that the adaptation layers have not be standardized upon, and any MIB support would soon be obsolete. Finally, there are no pointers to information that could be maintained by other MIBs (e.g. SONET, DS3). This omission was allowed for the first pass of the agent, as the project matures and these network interface are supported, these MIBs should be supported as well.
Chapter 8

Conclusion

This thesis has discussed the design and implementation of a system for managing an Asynchronous Transfer Mode switch. The protocol selected was the Simple Network Management Protocol due to its portable and modular structure.

The process of managing the Xunet switch has been subdivided into three areas:

- **Configuration** - This group concerns viewing the steady state of the network, modifying routing information, versions of control software, maximum virtual paths, maximum virtual circuits, and available ingress and egress bandwidth.

- **Realtime Monitoring** - This group concerns trouble-shooting running networks and finding faults quickly. The objects are used to monitor VPI/VCI values, operational status, ingress and egress quality of service and traffic parameters.

- **Statistics Gathering** - This group addresses the load and utilization of the interfaces in the switch. The objects contain information on ATM cells transmitted/received, and cells in error.
The implementation of these three areas has been achieved by this project. For example, configuration information in now available to the NMS via a view of the MIB-II information. The manager has the ability to make changes to the system via SNMP set requests. To facilitate monitoring the network, a map has been constructed for the entire Xunet system. The top level of this map is shown in Figure 8.1.

The task of realtime monitoring in now possible by allowing the NMS to poll a variety of objects in the extended Xunet MIB. A lower level view of just the University of Illinois site is shown in Figure 8.2. As the devices make transitions from running to disabled, their glyph states are modified on the NMS map view.

Finally, statistical information is readily available to the network manager via queries to either the Xunet MIB and the MIB-II objects. The manager has the ability to poll this information to view the flow of data through the network and spot bottlenecks. An example graph of ATM cells transmitted and received by the Illinois Xunet switch during a Telnet session is shown in Figure 8.3.
Figure 8.2: UIUC Xunet View.

Figure 8.3: ATM Cells Transmitted View.
In conclusion, the results from this project are the structure of an ATM Managed Information Base. And the various techniques that can be used to retrieve and store data from the managed device. This work has achieved its goal of providing control over this new technology and helping move the Xunet system closer towards mainstream networking.
Appendix A

Xunet MIB

=====================================================================
-- @(#)xunet-switch-mib 1.0 93/03/15

Xunet-Switch-MIB DEFINITIONS ::= BEGIN

IMPORTS
Counter, Gauge, IpAddress, TimeTicks
FROM RFC1155-SMI
OBJECT-TYPE
FROM RFC1212
EntryStatus
FROM RFC1271-MIB
atmSwitch, AtmAddress
FROM Xunet-Extensions-MIB;

=====================================================================
-- MIB Groups

hardware OBJECT IDENTIFIER ::= { atmSwitch 1 }
xunet OBJECT IDENTIFIER ::= { hardware 1 }
line-cardGroup OBJECT IDENTIFIER ::= { xunet 1 }
moduleGroup OBJECT IDENTIFIER ::= { xunet 2 }

software OBJECT IDENTIFIER ::= { atmSwitch 2 }
AnsaInterface OBJECT IDENTIFIER ::= { software 1 }
switchGroup OBJECT IDENTIFIER ::= { AnsaInterface 1 }
portGroup OBJECT IDENTIFIER ::= { AnsaInterface 2 }
pathGroup OBJECT IDENTIFIER ::= { AnsaInterface 3 }

-- The XunetSwitch Group

XunetSwitchTable OBJECT-TYPE
SYNTAX SEQUENCE OF LineCardEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A table of Xunet switch line-card information."
 ::= { line-cardGroup 1 }

XunetSwitchEntry OBJECT-TYPE
SYNTAX LineCardEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A table entry."
INDEX { line-cardIndex }
 ::= { line-cardTable 1 }

XunetSwitchEntry ::= SEQUENCE {
 line-cardIndex INTEGER,
 line-cardVersion INTEGER,
 line-cardModel INTEGER,
 line-cardSerialNumber INTEGER,
 numberOfLineCards INTEGER,
 vpiLookupErrors Counter,
 vciLookupErrors Counter
}

XunetSwitchIndex OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The index of this line-card within the Xunet switch."
 ::= { line-cardEntry 1 }

XunetSwitchVersion OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The Xunet switch line-card version number."
::= { line-cardEntry 2 }

XunetSwitchModel OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The Xunet switch line-card model number."
::= { line-cardEntry 3 }

XunetSwitchSerialNumber OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The Xunet switch line-card serial number."
::= { line-cardEntry 4 }

XunetSwitchNumberOfModules OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The total number of line-cards in the switch."
::= { line-cardEntry 5 }

XunetSwitchVpiLookupErrors OBJECT-TYPE
SYNTAX Counter
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The number of VPI lookup errors from translation memory."
::= { line-cardEntry 6 }

XunetSwitchVciLookupErrors OBJECT-TYPE
SYNTAX Counter
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The number of VCI lookup errors from translation memory."
::= { line-cardEntry 7 }

-------------------------------------------------------------------------------------------------
-- The Line Card Group

outputBufferTable OBJECT-TYPE
SYNTAX SEQUENCE OF OutputBufferEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A table of output buffer information."
::= { moduleGroup 1 }

OutputBufferEntry ::= 
SEQUENCE {
  obufType INTEGER,
  obufOperStatus INTEGER,
  obufBufferSize INTEGER,
  obufQueueLength Gauge,
  obufOverflows Counter
}

obufType OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The type of this output buffer."
::= { outputBufferEntry 2 }

obufOperStatus OBJECT-TYPE
SYNTAX INTEGER { 
other(1),
enabled(2),
disabled(3)
}
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The operational status of the output buffer."
::= { outputBufferEntry 3 }

obufBufferSize OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The logical size of this output buffer, in cells."
::= { outputBufferEntry 4 }
obufQueueLength OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The number of cells in this output buffer."
::= { outputBufferEntry 5 }

obufOverflows OBJECT-TYPE
SYNTAX Counter
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The number of cells dropped when the output buffer was full."
::= { outputBufferEntry 6 }

----------------------------------------
-- The Switch Group

XunetHardwareVersion OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The version of the Xunet switch hardware."
::= { switchGroup 1 }

XunetSoftwareVersion OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The version of the Xunet switch software."
::= { switchGroup 2 }

XunetMaxPaths OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The maximum number of VPIs the trans. memory can support."
::= { switchGroup 3 }

XunetMaxChannels OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The maximum number of VCIs the trans. memory can support.
 ::= { switchGroup 4 }

xunetAddress OBJECT-TYPE
SYNTAX AtmAddress
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The network address of this Xunet switch."
 ::= { switchGroup 5 }

xunetUptime OBJECT-TYPE
SYNTAX TimeTicks
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The length of time the switch has been operational."
 ::= { switchGroup 6 }

-- The Second Level Line Card Group

numberOfPorts OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The number of ports on this line card."
 ::= { portGroup 1 }

portTable OBJECT-TYPE
SYNTAX SEQUENCE OF PortEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A table of info about the ports on this line card."
 ::= { portGroup 2 }

portEntry OBJECT-TYPE
SYNTAX PortEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A table entry containing port information."
INDEX { portNumber }
::= { portTable 1 }

PortEntry ::= SEQUENCE {
portNumber INTEGER,
portAdminStatus INTEGER,
portOperStatus INTEGER,
portStatusTime TimeTicks,
portRemoteAtmAddress AtmAddress,
portRemoteIpAddress IpAddress,

portMaxPathsIn INTEGER,
portNumPathsIn Gauge,
portMaxBandwidthIn INTEGER,
portAllocBandwidthIn Gauge,
portUsedBandwidthIn Gauge,
portReceivedCells Counter,

portMaxPathsOut INTEGER,
portNumPathsOut Gauge,
portMaxBandwidthOut INTEGER,
portAllocBandwidthOut Gauge,
portUsedBandwidthOut Gauge,
portTransmittedCells Counter
}

portNumber OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The value of this object identifies the port."
::= { portEntry 1 }

portAdminStatus OBJECT-TYPE
SYNTAX INTEGER {
up(1),
down(2),
}
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the current desired status of this port, up or down."
::= { portEntry 2 }

portOperStatus OBJECT-TYPE
SYNTAX INTEGER {
up(1),
down(2)
}
ACCESS read-only
STATUS mandatory
DESCRIPTION
" the current status of this port, up or down."
::= { portEntry 3 }

portStatusTime OBJECT-TYPE
SYNTAX TimeTicks
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the time since this port has chanaged state."
::= { portEntry 4 }

portRemoteAtmAddress OBJECT-TYPE
SYNTAX AtmAddress
ACCESS read-only
STATUS mandatory
DESCRIPTION
" the address of the device connected to this port."
::= { portEntry 5 }

portRemoteIpAddress OBJECT-TYPE
SYNTAX IpAddress
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the IP address of the device connected to this port."
::= { portEntry 6 }

portMaxPathsIn OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the maximum number of input VPIs on this port."
::= { portEntry 7 }

portNumPathsIn OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the current number of VPIs active on this port."
::= { portEntry 8 }

portMaxBandwidthIn OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The maximum incoming bandwidth of this port, in cells per second."
::= { portEntry 9 }

portAllocBandwidthIn OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the allocated bandwitch in cells per second for this port."
::= { portEntry 10 }

portUsedBandwidthIn OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the cell per second bandwidth actually being used."
::= { portEntry 11 }

portReceivedCells OBJECT-TYPE
SYNTAX Counter
ACCESS read-only
STATUS mandatory
DESCRIPTION
"number of cells recieved counter."
::= { portEntry 12 }

portMaxPathsOut OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the maximum number of output VPIs on this port."
::= { portEntry 13 }

portNumPathsOut  OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the current number of VPIs active on this port."
::= { portEntry 14 }

portMaxBandwidthOut OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the maximum cells per second outgoing on this port".
::= { portEntry 15 }

portAllocBandwidthOut OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the allocated bandwidth in cells per second outgoing for this port."
::= { portEntry 16 }

portUsedBandwidthOut OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the output cell per second bandwidth actually being used."
::= { portEntry 17 }

portTransmittedCells OBJECT-TYPE
SYNTAX Counter
ACCESS read-only
STATUS mandatory
DESCRIPTION
"number of cells transmitted counter."
::= { portEntry 18 }

------------------------------------------------------------------------
-- The Virtual Path Group
pathTable OBJECT-TYPE
SYNTAX SEQUENCE OF PathEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A table of information about the virtual paths
on the Xunet switch."
::= { pathGroup 1 }

pathEntry OBJECT-TYPE
SYNTAX PathEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A table containing virtual path information."
INDEX { pathPort, pathVPI }
::= { pathTable 1 }

PathEntry ::= SEQUENCE {
pathPort INTEGER,  
pathVPI INTEGER,  
pathStatus EntryStatus,  
pathNumOutputs Gauge,  
pathMaxChannels INTEGER,  
pathNumChannels Gauge,  
pathMaxBandwidth INTEGER,  
pathAllocBandwidth Gauge,  
pathUsedBandwidth Gauge,  
pathCells Counter,  
pathUptime TimeTicks
}

pathPort OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the input line card of this path."
::= { pathEntry 1 }

pathVPI OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the input VPI of this path."
 ::= { pathEntry 2 }

pathStatus OBJECT-TYPE
SYNTAX EntryStatus
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the operational status of the path."
 ::= { pathEntry 3 }

pathNumOutputs OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the number of output ports for multicast purposes."
 ::= { pathEntry 4 }

pathMaxChannels OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the maximum number of VC’s allowed on path."
 ::= { pathEntry 5 }

pathNumChannels OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the current number of VC’s allocated."
 ::= { pathEntry 6 }

pathMaxBandwidth OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the maximum allowable bandwidth of the path."
 ::= { pathEntry 7 }

pathAllocBandwidth OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the current allocated bandwidth of the path."
::= { pathEntry 8 }

pathUsedBandwidth OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the amount of bandwidth actually being used."
::= { pathEntry 9 }

pathCells OBJECT-TYPE
SYNTAX Counter
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the number of cells transmitted on this path."
::= { pathEntry 10 }

pathUptime OBJECT-TYPE
SYNTAX TimeTicks
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The time since this path was created,
in hundredths of a second."
::= { pathEntry 11 }

outputPathTable OBJECT-TYPE
SYNTAX SEQUENCE OF OutputPathEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A table of virtual paths originating at this Xunet switch."
::= { pathGroup 3 }

outputPathEntry OBJECT-TYPE
SYNTAX OutputPathEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A table of output path information."
INDEX { opathPort, opathVPI }
 ::= { outputPathTable 1 }

OutputPathEntry ::= 
SEQUENCE {
opathPort INTEGER, 
opathVPI INTEGER, 
opathStatus EntryStatus, 
opathMaxChannels INTEGER, 
opathNumChannels Gauge, 
opathMaxBandwidth INTEGER, 
opathAllocBandwidth Gauge, 
opathUsedBandwidth Gauge, 
opathCells Counter, 
opathUptime TimeTicks }

opathPort OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the line card of this path."
 ::= { outputPathEntry 1 }

opathVPI OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the VPI of this path."
 ::= { outputPathEntry 2 }

opathStatus OBJECT-TYPE
SYNTAX EntryStatus
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the status of this path entry."
 ::= { outputPathEntry 3 }

opathMaxChannels OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the maximum number of VC’s allowed on path."
::= { outputPathEntry 4 }

opathNumChannels OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the current number of VC’s allocated."
on this path.
::= { outputPathEntry 5 }

opathMaxBandwidth OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the maximum allowable bandwidth of the path."
::= { outputPathEntry 6 }

opathAllocBandwidth OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the current allocated bandwidth of the path."
::= { outputPathEntry 7 }

opathUsedBandwidth OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the amount of bandwidth actually being used."
::= { outputPathEntry 8 }

opathCells OBJECT-TYPE
SYNTAX Counter
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the number of cells transmitted on this path."
::= { outputPathEntry 9 }

opathUptime OBJECT-TYPE
SYNTAX TimeTicks
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the amount of time the path has existed."
::= { outputPathEntry 10 }

-- The Virtual Channel Group

channelTable OBJECT-TYPE
SYNTAX SEQUENCE OF ChannelEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A table of virtual channels on this Xunet switch."
::= { channelGroup 1 }

channelEntry OBJECT-TYPE
SYNTAX ChannelEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"A table entry containing channel information."
INDEX { chanPort, chanVPI, chanVCI }
::= { channelTable 1 }

ChannelEntry ::= 
SEQUENCE {
  chanPort INTEGER,
  chanVPI INTEGER,
  chanVCI INTEGER,
  chanStatus EntryStatus,
  chanNumOutputs Gauge,
  chanAllocBandwidth Gauge,
  chanUsedBandwidth Gauge,
  chanCells Counter,
  chanUptime TimeTicks
}

chanPort OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-write
STATUS mandatory
DESCRIPTION
"The input line card of this channel."
::= { channelEntry 1 }

chanVPI OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-write
STATUS mandatory
DESCRIPTION
"The input VPI of this channel."
::= { channelEntry 2 }

chanVCI OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the input VCI of this channel."
::= { channelEntry 3 }

chanStatus OBJECT-TYPE
SYNTAX EntryStatus
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the operational status of this channel."
::= { channelEntry 4 }

chanNumOutputs OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the number of destination ports (multicast)."
::= { channelEntry 5 }

chanAllocBandwidth OBJECT-TYPE
SYNTAX Gauge
ACCESS read-write
STATUS mandatory
DESCRIPTION
"the maximum allowable bandwidth of the channel."
::= { channelEntry 6 }

chanUsedBandwidth OBJECT-TYPE
SYNTAX Gauge
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the amount of bandwidth actually being used."
::= { channelEntry 7 }

chanCells OBJECT-TYPE
SYNTAX Counter
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the number of cells transmitted on this channel."
::= { channelEntry 8 }

chanUptime OBJECT-TYPE
SYNTAX TimeTicks
ACCESS read-only
STATUS mandatory
DESCRIPTION
"the amount of time the channel has existed."
::= { channelEntry 9 }

END

Xunet-Extensions-MIB DEFINITIONS ::= BEGIN

IMPORTS
enterprises FROM RFC1155-SMI;

---------------------------------------------

xunet OBJECT IDENTIFIER ::= { enterprises 326 }
admin OBJECT IDENTIFIER ::= { xunet 1 }
systems OBJECT IDENTIFIER ::= { xunet 2 }
atmAdapter OBJECT IDENTIFIER ::= { systems 1 }
atmSwitch OBJECT IDENTIFIER ::= { systems 2 }

---------------------------------------------
-- Textual Conventions

AtmAddress ::= OCTET STRING (SIZE (8))

END
Appendix B

SMI Definitions

This appendix contains the Definition of RFC-1155 Structure of Management Information[42]. The listing for the SMI is shown below.

```
RFC1155-SMI DEFINITIONS ::= BEGIN
EXPORTS -- EVERYTHING
   internet, directory, mgmt,
   experimental, private, enterprises,
   OBJECT-TYPE, ObjectName, ObjectSyntax, SimpleSyntax,
   ApplicationSyntax, NetworkAddress, IpAddress,
   Counter, Gauge, TimeTicks, Opaque;
   -- the path to the root
internet OBJECT IDENTIFIER ::= { iso org(3) dod(6) 1 }
directory OBJECT IDENTIFIER ::= { internet 1 }
mgmt OBJECT IDENTIFIER ::= { internet 2 }
experimental OBJECT IDENTIFIER ::= { internet 3 }
private OBJECT IDENTIFIER ::= { internet 4 }
enterprises OBJECT IDENTIFIER ::= { private 1 }
   -- definition of object types
OBJECT-TYPE MACRO ::= BEGIN
   TYPE NOTATION ::= "SYNTAX" type (TYPE ObjectSyntax)
      "ACCESS" Access
      "STATUS" Status
   VALUE NOTATION ::= value (VALUE ObjectName)
   Access ::= "read-only"
      | "read-write"
      | "write-only"
      | "not-accessible"
   Status ::= "mandatory"
      | "optional"
      | "obsolete"
```
END
   -- names of objects in the MIB
ObjectName ::= 
   OBJECT IDENTIFIER
-- syntax of objects in the MIB
ObjectSyntax ::= 
   CHOICE {
      simple
         SimpleSyntax,
   -- note that simple SEQUENCES are not directly
   -- mentioned here to keep things simple (i.e.,
   -- prevent mis-use). However, application-wide
   -- types which are IMPLICITly encoded simple
   -- SEQUENCES may appear in the following CHOICE

      application-wide
         ApplicationSyntax
   }

SimpleSyntax ::= 
   CHOICE {
      number
         INTEGER,
      string
         OCTET STRING,
      object
         OCTET STRING,
      empty
         NULL
   }

ApplicationSyntax ::= 
   CHOICE {
      address
         NetworkAddress,
      counter
         Counter,
      gauge
         Gauge,
      ticks
         TimeTicks,
      arbitrary
         Opaque
   }
-- other application-wide types, as they are
-- defined, will be added here
}

-- application-wide types
NetworkAddress ::=  
  CHOICE {
    internet
    IpAddress
  }

IpAddress ::= 
  [APPLICATION 0]  -- in network-byte order 
    IMPLICIT OCTET STRING (SIZE (4))

Counter ::=  
  [APPLICATION 1]  
    IMPLICIT INTEGER (0..4294967295)

Gauge ::=  
  [APPLICATION 2]  
    IMPLICIT INTEGER (0..4294967295)

TimeTicks ::=  
  [APPLICATION 3]  
    IMPLICIT INTEGER (0..4294967295)

Opaque ::=  
  [APPLICATION 4]  -- arbitrary ASN.1 value, 
    IMPLICIT OCTET STRING  -- "double-wrapped"

END
Bibliography


