To the Graduate Council:

I am submitting herewith a thesis written by Kenneth W. Key entitled "The Implementation of the Simple Gateway Monitoring Protocol." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.

Dr. Jeffrey D. Case, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Provost
and Dean of The Graduate School

## STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Kenneth M. Ky

Date July 13th, 1990

# THE IMPLEMENTATION OF THE SIMPLE GATEWAY MONITORING PROTOCOL

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Kenneth W. Key

August, 1990

## DEDICATION

This thesis is dedicated to all of those who work so that information can flow free.

## ACKNOWLEDGEMENTS

I thank Chuck Davin, Marty Schoffstall, Mark Fedor, Wengyik Yeong, Keith McCloghrie, and Marshall Rose for their efforts in developing the SGMP and the SNMP into today's network management standard.

I thank Chris Jepeway, Steve Kilgore, Keith Moore, Sam Nicholson, and the other denizens of the CS Graduate lab who I've harassed with questions when I hit stumbling blocks in coding.

I thank Dr. Tom Dunigan for helping a novice get started with UNIX and TCP/IP and Dr. David Straight for his data structures class and support.

I thank my family for their support over the many years of my life as a professional student. I also thank my "adopted family", Ron Tipton, Gail Meglitsch, and Katy Eggering, for all their help in getting through life as a CS graduate student. "What a long strange trip it's been."

I thank my friends in the CS department and Team budnet for putting up with weird hours and ramblings ("zero-one zero-four zero-one zero-two zero-four").

Most of all, I thank Dr. Jeffrey D. Case for his ideas, his energy, his support, and his work in making the SGMP and the SNMP networking standards. I am grateful to him for the opportunity to contribute to this project.

"That dog can hunt"

## ABSTRACT

The Simple Gateway Monitoring Protocol is one of several network management protocols proposed to address the needs of managing today's heterogeneous computer networks. The structure of several proposed network management protocols were examined with attention to the potential strengths and weaknesses of each. The Simple Gateway Monitoring Protocol (SGMP) was implemented to examine its practicability as a network management protocol. Questions about using the SGMP included whether the protocol's use of the Abstract Syntax Notation One (ASN.1) language introduce complexity that would counter the advantage of the protocol's simple design and whether connectionless transport be sufficient for a network management protocol. Many programs were written to communicate via the SGMP. Central to all of the programs was a library of routines that created and parsed the SGMP messages. This library was written in the C programming language and designed to be portable to many different computer environments.

It was found that the SGMP could be used to manage networks. The use of ASN.1 in the SGMP was limited to a small subset that was easily implemented. The use of connectionless transport allowed the SGMP to function when connection-oriented protocols failed. The SGMP has evolved into a standard for the management of Internet Protocol-based networks.

# TABLE OF CONTENTS

# LIST OF FIGURES

# CHAPTER 1

## Introduction

The purpose of this thesis is to report on the research and results of a study in implementing the Simple Gateway Monitoring Protocol (SGMP) [1]. The goal of this study was to produce practicable, portable, applications that communicate via the SGMP. A secondary goal was to develop a deeper understanding of the issues present in developing network management protocols and applications. Lastly, it was hoped that a broad overview would be ascertained of how different components of networks operate and could be managed.

The SGMP was proposed by Case, Fedor, Schoffstall, and Davin as a method to extract information from network gateways for the purposes of network management [1]. Many questions arise when considering any new protocol. Can the protocol, as specified, function? Can it do anything useful? What is its impact on the systems that implement it? How extensible is it?

For the SGMP, several more specific questions come to mind. A hotly debated topic of network management is the choice of transport for network management protocols. Would the SGMP's use of connectionless transport allow it to function in a reasonable manner? What problems accompany the use of connectionless transport? The SGMP departed from many previous networking protocols by using the Abstract Syntax Notation One (ASN.1) for the encoding of messages. This brings up the questions of whether the subset of ASN.1 that the SGMP

specifies is itself implementable and what the impact on systems would be.

This thesis is divided into four parts. Chapter 2 is a review of Internet network management protocols and procedures that were proposed prior to or during the course of the work on the thesis. Chapter 3 details the specific work that went into the implementation of the SGMP applications that were the result of this thesis. The problems encountered and resolved in developing the applications are described in Chapter 4. Finally, Chapter 5 summarizes what was learned by implementing the SGMP and provides recommendations for further study in network management protocols and applications.

# CHAPTER 2

# Network Management

## 2.1 Introduction

The routers that connect an internet hold a wealth of information that can help network managers in the performance of their management duties. Among other things, network managers are interested in how a network is being used, whether a problem exists, and if so, identifying the source of the problem. By accessing the data a router has collected, network managers can gain some insight to answer these questions. The difficulty is in how to extract this information from the router and present it to the network manager in a form that is usable.

### 2.1.1 What Is The Internet?

An internet is a network of networks, i.e., a collection of computer networks that have been interconnected to make a larger network composed of the constituent networks. The Internet is the term commonly used to refer to the world-wide collection of networks that use the Internet Protocol (IP) and those protocols that are based on IP which grew out of the original Advanced Research Projects Agency network (ARPANET). The ARPANET is based on the Catenet Model for Internetworking proposed by Vint Cerf [2]. The Internet is currently composed of national wide area networks, regional networks, and local area networks that are

seamlessly interconnected by IP routers. The fundamental quantum of information that is transferred via this Internet is a *packet*, which is a collection of octets that consists the data being transferred and a header that contains information on the packets origination and destination. The routers at the network interconnection points transfer packets from one network to another.

### 2.1.2 What Is A Gateway?

Gateway is an old Internet term for a packet switch that transfers packets from one network to one of possibly several others to which the gateway is attached. As the Internet Protocol is an Internation Standards Organization (ISO) Reference Model Layer 3 specification, IP *gateways* are now referred to as IP *routers*. At each point where the networks of an internet are connected there is a device that acts as an IP router. This could be a dedicated device or a time-sharing computer system with multiple network interfaces. In either case, the router is responsible for selecting which network to forward a packet on in order for the packet to reach its destination. It may be that the network that the router has forwarded the packet on is only one step (or *hop*) into the packet's journey to its final destination. In this case, the packet relies on the next router connected to that network to perform the same forwarding function to send the packet closer yet to its final destination. It is by these actions that packets may pass through multiple routers and travel across multiple networks to reach their target destinations.

Routers make their decisions based on the information they have at the time the packet arrives. The Catenet model was designed to take advantage of rich interconnectivity and to use whatever alternative routes are available when a pre-

viously available route becomes unavailable. As a result, the router needs to maintain a routing information database that includes information about which hosts/networks are reachable via which network interface. It also needs to be able to sense when there are problems with its networks, such as a communications line being down or an interface unconnected so that it can update this database. Lastly, a router needs to inform other routers of the state of information in its database so they can update their own databases. This routing information interchange is accomplished via routing protocols. There are currently several routing protocols used in the Internet including the Exterior Gateway Protocol (EGP) [3], Open Shortest Path First protocol (OSPF) [4], and the Routing Information Protocol (RIP) [5].

These routers collect other information besides routing information. Many network interface cards are instrumented to detect error conditions on Physical and Data Link layers of the network. They can detect conditions such as runt or over-sized packets, bad checksums, collisions, loss of line synchronization, loss of token, etc. Some routers have implemented simple software counters that are incremented when a network interface detects one of these conditions with minimal impact on the router's performance. This allows the router to gather information that is valuable to network managers without affecting the router's primary job: routing packets.

### 2.1.3 What Does A Network Manager Do?

There are organizations that plan, implement, maintain, and improve the networks that compose the internets. Since the internets are hierarchical in nature,

so are these management organizations. The manager of an extended local area network is primarily concerned with whether the hosts on the local network can efficiently communicate with each other and if they can reach sites outside of the local network. Likewise, the manager(s) of a regional autonomous network are concerned with the ability to communicate between the connected member networks, with a heightened concern about reaching remote networks. In both situations, the network managers are primarily concerned with the operational status of the network in question as well as problems originating at the entry points (hosts or member networks).

A network manager's job can be broken into two major functions: fire-fighting and administration. A network manager in *fire-fighting* mode is trying to address an immediate problem that is affecting the usability of the network. When network managers are *administrating* a network they must plan for the future growth of the network and account for its current use.

The extraction of information from routers for the purpose of managing the internet was part of the basic Catenet model outlined in Internet Engineering Note 48 [2]. Router information is often important in determining the cause of a network's problems. Checking the status of a router's network interface can give quick confirmation that the link associated with that interface is down. One critical internetwork problem is the formation of *black holes*, where invalid routing information has been propagated to routers such that they route packets for a particular host or network to a router that either drops the packets or sends them in a routing loop. By tracing the routing tables of the routers involved in the paths, such *black holes* can be detected and the source of the false routing information determined.

The information routers collect is also useful in the administrative duties of a network manager. A measure of utilization of the network's resources can be established by periodic polling of the counters maintained by the routers. The manager can check the packet counts on a router's interfaces to determine if the router or any of its links are approaching saturation and thus need to be upgraded to maintain acceptable performance.

### 2.1.4 How Can A Manager Extract The Information?

Until recently, network managers were required to use a variety of methods to extract this information from the routers. A IP router's connectivity information can be inferred, although not directly extracted, via the PING and TRACEROUTE programs. The Routing Information Protocol (RIP) has a facility to transfer its entire routing table to an entity when it is queried. Some routers let network managers access a command line interface via Telnet to view router data. These methods are limited in that they only present data of a limited scope relevant to the protocol being used to extract the information, such as routing information from RIP. They are also limited in that they are not consistent across the wide variety of routers manufactured. For example, the command line interface of routers from Proteon, Incorporated, is different from Wellfleet, Incorporated, and Network Systems Corporation.

While the Telnet command line interface access is better than nothing, managers want other information that is not available in such a fashion. The Host Monitoring Protocol (HMP) [6] was used to monitor and control the ARPANET network entities. This, however, does not solve the problem of dealing with hetero-

geneous equipment. In the past few years, many network management protocols have been proposed to address this, such as the High-Level Entity Monitoring System/Protocol (HEMS/HEMP) [7], the Simple Gateway Monitoring Protocol (SGMP) [1], the Simple Network Management Protocol (SNMP) [8], and Common Management Information Services/Common Management Information Protocol over TCP/IP (CMOT) [9].

## 2.2 Simple Network Testing Tools

Network managers have developed simple tools that perform very specific functions to assist them in finding anomalies they frequently encounter in the routine management of their networks.

### 2.2.1 PING

One tool that many IP network managers find useful is PING [10]. PING is a tool that sends an Internet Control Message Protocol (ICMP) Echo Request Packet to a specified IP address that causes the remote host to send back an ICMP Echo Response packet. It can send a stream of these packets (usually at one second intervals) and, by using sequence numbers, keep track of round-trip delay and packet loss. This last aspect is important since long delay times suggest that the network is heavily loaded but still functional. If packets are being dropped, it can be indicative of a severely overloaded network performing selective preemption, route flapping, or other problems. Of course, not getting any response is indicative of a connectivity problem between the source and the target site. This tool is useful for a wide variety of targets since it is required that all IP entities

support the ICMP Echo Request packet type and almost all do. Another nice aspect of using PING is that ICMP Echo Reply is usually supported in the heart of an IP implementation so if a machine is nonfunctional at a higher level such as Telnet or the Transmission Control Protocol (TCP) transport, it will still respond to the more rudimentary ICMP echo requests.

### 2.2.2 QUERY

QUERY is a tool for extracting routing tables from hosts and routers that implement the Routing Information Protocol (RIP) which is included in Berkeley UNIX[1] (BSD) and is in wide-spread use on local area networks [11]. The program displays the entire contents of a remote machine's routing table, allowing the network manager to look for anomalies. These anomalies include the absence of routes that indicate a network router failure and routes with higher than expected metrics which indicate possible communications link failures.

### 2.2.3 TRACEROUTE

Another useful tool is TRACEROUTE [12]. TRACEROUTE sends a Unreliable Datagram Protocol (UDP) packet to a remote host with an IP Time-to-live field that is deliberately too small to reach its destination. When a router receives a packet that has expired, it sends an ICMP Time-Exceeded packet back to TRACEROUTE. When a response is received, TRACEROUTE sends out another packet with the TTL one unit larger to get a response from the next hop. TRACEROUTE displays the addresses of the entities that the Time-Exceeded

[1]UNIX is a trademark of AT&T.

messages came from and the round-trip delay for the packets sent. This tells the network manager the route being taken for these packets, which can indicate failed lines or other routing anomalies when routes are other than expected. The round-trip delay can be used to indicate what links in the route are introducing the most delay.

## 2.3 The Host Management Protocol

One of the first internetwork management protocols was the Host Management Protocol (HMP) [6]. The HMP was designed as a network management transport protocol that could be used to communicate with internet entities that can send or receive IP messages. It describes a general header that specifies the type of host, type of message being sent, authentication, and transport information to allow the monitoring host to detect lost messages.

### 2.3.1 HMP Messages

Information is gathered by the HMP through asynchronous events and polling for information. Events, also known as traps, are unsolicited asynchronous messages which are sent by the monitored entity when unusual events occur, such as machine crashes. Polling gathers both status and statistical information and allows for the controlling of the monitored entity.

### 2.3.2 HMP Polling

When an entity receives a poll, its response is controlled by the type field of the HMP header. If the poll is a request for status information the entity creates a new message based on configuration and operational status and sends it back to the monitoring station. If the poll is a request for statistical information, the entity sends back the latest statistical report of the type requested. Note that the system for gathering the statistical information is in the monitored entity itself. The entity gathers the information for some defined time period, saves the message in a buffer to await a poll, and continues to gathers information for the next report. Control requests can also be contained in the poll message. This allows the monitoring entity to set parameters on the monitored entity by sending a control message in a poll.

### 2.3.3 Analysis Of HMP

Many of the concepts introduced in the HMP can be found in the more modern network management protocols. One idea central to its design is that the monitoring host should take most of the load of management and keep the monitored entity's burden small. HMP uses a connectionless datagram transport that lets the monitoring station control retransmissions rather than sharing the responsibility with the monitored entity. HMP allows the monitored entity to notify asynchronously the monitoring host of unusual events. It also allows control messages be sent to modify the monitored entity's characteristics.

The protocol does have some severe limitations. While the protocol is ex-

pandable to include new messages and new machine types, the messages must be predefined. This means that a monitoring station must have *a priori* knowledge of the specific machine being polled, as well as the type and format of each individual message available from the monitored machine. Also, in the absence of further evidence, it must be assumed that a report must fit within a single IP datagram. There is a *More-bit* in the control flag data field that is commented as indicating that there is more information to poll for, but no description on how this is done. There are serious implications if this is done by simply issuing a second poll of the same type as this would cause synchronization problems if two or more monitoring entities were polling hosts.

## 2.4 The High-level Entity Management System

The High-level Entity Management System (HEMS) [7] is a much more sophisticated management system than the HMP. It was proposed as a management protocol that all IP hosts could run to standardize the management of IP entities. Like the HMP, it uses a query-response paradigm to extract information from the monitored entity and asynchronous trap generation to notify monitoring stations of events. Where it differs greatly from HMP is in how the information being extracted is organized and retrieved.

One of the main ideas driving the design of the HEMS is that managers should not have to know the individual entity that they are trying to manage but only the general parameters for the class of entity being addressed [13]. For this to happen, the management system had to define not only the management protocol, but the data representation and the methodology for extracting the information.

### 2.4.1 High-level Entity Management Protocol (HEMP)

The protocol for accessing the managed entities is the High-level Entity Management Protocol (HEMP) [7]. It was designed to be a standard method of encapsulation, authentication, and encryption for all messages being passed by the HEMS system. The HEMP breaks messages into five (5) types: request, event, reply, protocol error, and application error. The last three of these are the responses to a request.

HEMP messages are used in both the query-response and the event modes of communication. When an application wants to extract information, it formulates a request and sends it to the entity. The entity then performs whatever processing is necessary and sends a reply message if no errors have been detected. If errors are found in the HEMP encapsulation (i.e., data encoding errors, invalid version number, etc.) a protocol error message is returned. If there has been an actual procedural error (i.e., failed authentication, invalid request), an application error message is sent back. When used to send event messages, the protocol uses the same encapsulation, authentication, and encryption envelope to simplify the processing of protocol messages.

The transport protocol for the HEMP is both TCP and UDP. The HEMS activity was seen to fall into three categories: status monitoring, fire-fighting, and event reporting. Status monitoring primarily takes place while the network was healthy and a connectionless transport could be used since the information being retrieved is mostly of administrative significance. Fire-fighting, however, takes place while the network is in trouble. A reliable transport protocol is considered essential in this situation. In the case of event reporting, it is thought that if the

problem is of sufficient severity, it will probably recur and another event report generated, thus allowing a connectionless transport could be used. As such, the HEMP calls for TCP to be used for the query-reply traffic and UDP to be used for the event monitoring.

### 2.4.2 HEMS Data Organization And Representation

The HEMS system organizes the management information into a tree hierarchy. The information is grouped together into seven general sub-hierarchies: System Variables, Event Controls, Interfaces, IP Network Layer, IP Routing Table, IP Transport, and IP Applications Layer. Each of these is further subdivided as necessary, such as by protocol, until the individual variable is named. A data node is named by the path from the root of the tree to the data node. A node in the tree is referred to as a *dictionary* and represents all the variables contained below it. Once down to the individual variable level, there is still more subdivision. Each variable has with it an attributes structure that contains the ASN.1 tag of the variable, its value format, textual descriptions, the unit of measurement, precision, and a properties bit-string. Additionally, each node in the tree can have a sub-node named *VendorSpecific* which can contain any information a vendor may wish to represent in the tree.

The information in the HEMS is encoded using Abstract Syntax Notation one (ASN.1) [14, 15]. ASN.1 is an International Standard method to represent data in a machine independent form. This eliminates many of the classic problems of how many bits are in an integer, what order bytes are in, how are characters represented, and what arithmetic type the data is. Each data element in ASN.1 is

a tuple of type, length, and data. The type can be a simple type, such as integer or octet string, or a complex type composed of sets, sequences, or other aggregate operations. Each node of the HEMS data tree is represented uniquely as a complex type that depends on the context of the types preceding it down to the individual element level. In this way, the ASN.1 tags for the nodes in the tree represent the name of the node in the same hierarchical fashion, following the unique path from the root to the element.

### 2.4.3 HEMS Query Language

The key to a network management protocol is the ability to extract the information. To do this, the HEMS implements a powerful but simple query language that is used to communicate with a query language processor that resides on the IP entity. The queries and responses are encoded in ASN.1. The query processor deals with the ASN.1 objects that make up a query in a stack fashion to determine the actual data to return. The reply is a sequence of ASN.1 objects representing the subset of the variable tree that was requested.

The ASN.1 objects that make up a request can be either an opcode, a template, a path, a value, or a filter. The opcodes consist of Begin, End, Get, Get-attributes, Get-range, Set, Create, and Delete and are the actions that are performed against the data tree in the query language. A template is an object that names one or more variables in the data tree. This can be a discrete variable or the name of a node in the data tree which implies all of the variables underneath that node in the tree. A path is a special degenerative case of a template as it identifies only one node in the data tree. A value is a template representation where the data

values have been filled in with the desired values. A filter is a boolean expression that is executed against a collection of variables in a node to select specific entries.

The use of the filter is of particular interest. While the data in the HEMS are represented in a hierarchical tree fashion, in actuality most of the IP variables are logically organized in table or array forms, such as routing tables. Unfortunately, there is no way inherent with a tree hierarchy to tie together variable values in separate nodes that correspond to the same table entry, with the possible exception of introducing an arbitrary table entry index. Through the use of a filter, an expression is evaluated on the basis of one variable (such as a route's network address) and variable values in other nodes that are tied to this table entry are returned by accessing that node. Note that this association is not carried out by the tree topology, but is a supplemental mechanism added specifically to facilitate the extraction of table entries and the filter mechanism.

As previously stated, a query is a series of ASN.1 objects presented to the query language parser. As this is a stack-based language, the objects consist of arguments and operators in postfix order. The BEGIN operator takes a "dictionary" and a path as its arguments and establishes that the path (the sub-node in the data tree) is the context for all future operations. An END removes the context imposed by the BEGIN and terminates the ASN.1 object being formulated as a reply for the query. GET retrieves the data described by the object currently on the stack and puts the value filled object on the stack. GET-ATTRIBUTES is a form of GET that retrieves the attributes data objects for the objects described on the stack rather than the values of the objects themselves. GET-RANGE is another special operand that is used to specify memory to be read (which is represented as an octet string ASN.1 data type). SET sets the values in the data tree based on the

values in the objects described on the stack. The CREATE operand is used to add a table entry into the data tree and the DELETE operand removes an entry, via the table entry specification mechanism described earlier in the filter data types.

### 2.4.4 Analysis Of HEMS/HEMP

The HEMS/HEMP brings many new ideas to the network management arena. The use of a standardized data tree helps make management of new network entities easier since the controls for managing the box are in a standardized location. The creation of a vendor-specific area under each node area allows each vendor to put the particular variables specific to their product in an accessible area. The attributes sub-tree associated with each node lets an application retrieve information about variables that it may not have *a priori* knowledge to use in displaying or calculating values based on the variables. The use of ASN.1 in the protocol brings in a relatively well understood architecture independent method of complex data representation that has already been agreed upon in the world arena.

There are a few concerns about the HEMS/HEMP, however. There was much disagreement about what the transport protocol should be in a network management protocol. The use of a reliable transport protocol means that the network management application loses control of the retransmission frequency when it makes its request. In a *fire-fighting* situation, the traffic generated by the retransmissions of network management requests may just further contribute to the network problem that the manager is trying to address. Another concern is the load an implementation of the protocol would present on an IP entity. While the query processor may be relatively simple, the stack of limited size, and using

pointers may conserve memory and processor resources, the time the IP entity spends performing these tasks is time not doing its designated function. While the protocol could be implemented on multitasking hosts and major routers, there is some concern that it would be too resource intensive for a personal computer sized IP implementation, and the protocol calls for all IP entities to support the HEMS/HEMP.

## 2.5 The Simple Gateway Monitoring Protocol

The Simple Gateway Monitoring Protocol (SGMP) [1] was proposed shortly after the HEMS/HEMP proposal came out. A stated SGMP design strategy was to "minimize the number and complexity of management functions realized by the gateway itself." [1] It was thought that implementing a complex protocol such as the HEMS/HEMP on an IP router would severely impact the router's performance.

Another stated design decision was to require "only an unreliable datagram transport, and, furthermore, that every protocol message is entirely and independently representable by a single transport datagram." [1] Since SGMP was designed to address IP routers, this meant that the protocol used UDP as its sole transport, unlike the HEMS/HEMP which used both TCP and UDP.

The SGMP uses a query/response paradigm for the extraction of the router information. It also has an event-driven message system to allow routers to send unsolicited messages to the monitoring stations when a router itself detects problems. The router is monitored and controlled by extracting or setting defined variables. These variables are organized by a hierarchical naming convention so that the name space can be extended to contain new variables as need arises in

the evolution of the protocol. An authentication protocol is also defined to help address the security needs of limiting the control and monitoring of routers to only those authorized.

### 2.5.1 Message Types

The SGMP specifies only four message types: GET REQUEST, SET REQUEST, TRAP, and GET RESPONSE. The GET REQUEST message is used to request the current values of the variables specified in the message. The SET REQUEST message is used to change the values of variables to that specified in the message. The TRAP message is sent, unsolicited, to the monitoring stations and contain variables that pertain to the type of event that caused the TRAP. The GET RESPONSE message is sent in response to GET REQUEST and SET REQUEST messages and returns the current values for the variables the request specified or error messages if errors were detected in the requests.

The structure of these messages is defined using a limited subset of the ASN.1 data representation conventions. It should be noted that at the time the SGMP was being formulated, ASN.1 did not exist and the representations were actually defined using ASN.1's predecessor, X.409 [16]. However, as ASN.1 is a superset of X.409, it is proper to refer to the current International Standard for data representation, ASN.1, to prevent confusion. The GET REQUEST, SET REQUEST, and GET RESPONSE messages are identical in structure, varying only by the application type number that differentiates the three messages. This greatly simplifies the packet parser/generator implementations by allowing the different messages to share common code. It also simplifies the message handling sections of agents

and monitoring implementations by providing a common structure. The messages consist of a request ID number, an error status indicator, an error index, and a sequence of variable name/variable value tuples. The TRAP message has its own unique structure that requires separate message handling and parser/generator functions. This is not a major problem as TRAP messages represent a totally different type of information than the request/response variables and would have to be treated differently in any case. A trap message consists of a trap type and a list of values that are specific to the type of trap that has been generated.

### 2.5.2 Processing The Different Message Types

When an application generates a GET REQUEST message, it sets the request ID to a unique value so that it can identify the response associated with this request. The error status and error index are set to zero. Lastly, it adds the sequence of variable names and an associated variable value. The actual value of the variable values are unimportant in a GET REQUEST message as they are unused and are there only to maintain symmetry of structure.

The SET REQUEST message is similar to the GET REQUEST message except that the variable values are now important. The values are the actual values that the router should set the variables to upon receipt of this message.

The GET RESPONSE message is created in response to the GET/SET REQUESTS. If the GET REQUEST contains a variable name that does not lexicographically precede a variable in the router's variable name space, a GET RESPONSE is generated that is identical to the GET REQUEST except that the error status is set to *gmp_err_nix_name* and the error index is set to the position

of the offending variable in the list of variables. Otherwise, a GET RESPONSE is prepared with the actual variable values. If this response is larger than the maximum allowed by the protocol (specified as 484 octets), that response is discarded and a GET RESPONSE identical to the GET REQUEST is generated with the error status set to *gmp_err_too_big*. If none of these conditions has occurred, then response is sent back with the request ID set to the value of the original GET REQUEST's Request ID.

The GET RESPONSE generated from a SET REQUEST is similar in logic. If a SET REQUEST contains a fully specified variable name that does not exist in the router's variable name space, a GET RESPONSE is generated that is identical to the SET REQUEST except that the error status is set to *gmp_err_nix_name*. If the generated response for this message is too big, a GET RESPONSE is sent back with the *gmp_err_too_big* error status. If the SET REQUEST contains a variable value that is inappropriate for the variable being set, a GET RESPONSE is generated with the error status set to *gmp_err_bad_value*. If none of these apply, then a GET RESPONSE identical to the SET REQUEST is sent back. Note that the protocol does not require the application send back the final value of the variables after the set function has been performed. It is up to the application to send a follow up GET REQUEST to confirm the values.

TRAP messages are generated when the router senses some error condition within itself. It generates a message with the type set appropriate to the event it has monitored and sends a copy to every monitoring station it has been preconfigured to notify via trap messages. The trap types that are predefined by the protocol are Cold Start (0, a full reboot), Warm Start (1, a reinitialization that does not affect configuration), Link Failure (2, an interface has failed), Authenti-

cation Failure (3, a message was received that failed in the authentication layer), and Exterior Gateway Protocol (EGP) Neighbor Loss (4, an EGP peer has been marked down). Negative trap type values are reserved for implementation specific variables.

### 2.5.3 The Variable Name Space

Each variable in the router's variable space is named by the value of an octet string. Since the GET REQUEST message extracts the variable that is the lexicographical antecedent of the variable named, it is important that related variables be grouped together in lexicographical ordering. The variable name space is meant to be extensible, so a hierarchical naming scheme is also needed. This is done by treating each octet in the octet string name as if it was a sub-identifier in an ASN.1 Object Identifier. The first octet determines that the variable is for a router's value by using the value of *1*. The second octet is used to differentiate between version, configuration, network, and protocol variables. This convention is followed so that the variables are automatically grouped with other variables that are similar in nature as the variable space gets defined. It also allows for new variable sub-classes to be added to the variable tree by defining a new octet value at the appropriate level.

As it turns out, the variable name space for an operating router is dynamic. Some variables are defined by a series of octets that specify the general class of the variable and the actual instance of the variable is finalized based on existing routing table entries. Therefore, as routing table entries are created and deleted by the normal operation of the router, variable names appear and disappear. The

fact that a monitoring station does not have total *a priori* information about the variable name space is not a problem due to the fact that the GET REQUEST function operates by retrieving the variable that is the lexicographical antecedent of the one named. What this means is that a GET REQUEST can send a variable name that is only specified up to a point that specifies the class of variable it wants to extract. It will then get back the first variable that exists in that class since the first fully qualified name is the lexicographical antecedent of the partial specification. By using this variable name in the next GET REQUEST that is sent, the next variable after this one is extracted. This procedure can then be repeated as necessary to extract the entire contents of a variable class. In any case, a variable's existence does not have to be known beforehand by the monitoring station as it will be extracted when the GET REQUESTS span across the variable class.

While the lexicographical *greater than* function of the GET REQUEST brings with it the ability to extract dynamic variables and recurse across classes to retrieve tables of information, it makes extracting a single specific variable more difficult. The protocol addresses this by specifying that each variable will end in "an implementation specific octet string of non-zero length." [1] What this means is that after the octet string that names a specific instance of a variable, there will be at least one more octet after that name. Thus a GET REQUEST that had a variable named down to the point of being unique, but without the implementation specific trailer, would extract that particular variable because it is the lexicographical antecedent of the variable the request specifies.

### 2.5.4 Authentication Protocol

The authentication protocol is specified as a session-layer protocol that provides a binding between the monitored router and the monitoring station. The authentication protocol appends a header to the SGMP message that consists of a message length, a session ID length, a session ID, and the user data. The message length is a two octet count in network byte order of the length of the entire authentication protocol packet. The session ID length is a one octet count of the length of the session ID. The session ID is a sequence of octets that identifies the session between the monitoring station and the router. The user data is the SGMP message that is being sent between the two entities.

### 2.5.5 Analysis Of The SGMP

The SGMP is quite simple when compared to HEMS/HEMP. It has only one variable extraction operator that performs a lexicographical *greater than* function to determine which variable to extract. The ASN.1 that it uses is much more limited in scope than the HEMS/HEMP requires, reducing the complexity of the parser/generators needed for the data interchange. The use of UDP means that the monitoring application, not the transport layer or the monitored router, controls the sending and retransmission of the SGMP packets and time outs. This avoids the situation where the management transport is contributing to the network problems by uncontrolled retransmissions and effectively blocking the attempts to access the offending router. The variables are accessible in a hierarchical fashion so that tables can be extracted by repeated GET REQUESTS. Further data filtering can then be performed by the monitoring station, relieving the router of this

burden.

The protocol does have its limitations. While it can extract multiple variables out in one request, it is limited to only that number of variables that will fit into one message of less than 484 octets. Also, in trying to extract a dynamic table such as the routing table, the synchronization of the data is questionable. This can be argued to be inconsequential as it is a dynamic table and the data are extracted over a relatively short period of time. Therefore, the addition or deletion of routes since a variable was extracted is the same as the addition or deletion of routes since a routing table was atomically extracted and is merely a best attempt snapshot of the table. In both cases, attempts to act upon the data may be out of date depending on what happened in the router since the extraction.

## 2.6 The Simple Network Management Protocol

The Simple Network Management Protocol (SNMP) [8] is the product of the same authors of the Simple Gateway Monitoring Protocol. Early protocol experiments for the SNMP, specified in IDEA 11 [17], are backwards compatible with the SGMP, but the final protocol is not. The protocol document does state that "the original philosophy, design decisions, and architecture remain intact." [8]

At the time that the IDEA 11 document was produced, there were three proposals competing to be the standard for the management of TCP/IP-based internets: the HEMS/HEMP, the SGMP/SNMP, and CMOT (which will be discussed later). The Internet Activities Board (IAB), the "coordinating committee for Internet design, engineering, and management," [18] acted to organize the network management efforts. The IAB recommended that the SNMP be the basis for network

management in the short-term, with CMOT the long-term solution [19]. The result of this was three separate working groups which developed a new SNMP based on the IDEA 11 SNMP, finished the CMOT specifications, and created a Management Information Base (MIB) that the two protocols would share to ease the transition to ISO Open Systems Interconnect (OSI) networking.

The primary changes between the SGMP and the SNMP are the replacement of the variable space, the modification of the messages, the addition of a new message type, and the replacement of the authentication layer. The variable space that the SGMP addressed is replaced with a system that is international in scope and sharable with the CMOT protocol. This new variable space is named the Management Information Base (MIB) [20]. The new message type is named GET NEXT REQUEST, which performs the lexicographical *greater than* operation of the SGMP GET REQUEST. The GET REQUEST definition was modified to perform an *equal to* operation. The authentication protocol is combined with the management protocol and defined as an ASN.1 wrapper around the SNMP messages.

### 2.6.1 The MIB And The SNMP

The MIB is the result of the efforts of the MIB working group following the recommendations of the Internet Activities Board outlined in Request For Comments (RFC) 1052. It is intended to define network management variables for use both by the SNMP and CMOT protocols, "so as to ensure compatibility of monitored items for both network management frameworks." [19] The variables contained in the MIB are defined using ASN.1 and uniquely named by Object

Identifiers. This provides a very similar variable naming and accessing scheme as used in the SGMP while having the advantage of being registered with an International Standard. The structure of the variables contained in the MIB is defined in the Structure and Identification of Management Information for TCP/IP-based internets (SMI) [21]. The SMI lists the conventions used for naming the MIB objects and the syntax of ASN.1 used to represent the values of the MIB objects. This syntax is a restricted subset of ASN.1 that is more complex than the original SGMP syntax.

The MIB specifies the framework of the variable space to be used by both the SNMP and CMOT. The MIB includes definitions of aggregate objects which the SNMP cannot directly address as it specificly disallows aggregate types. The SNMP deals with this by defining an instance identifier for every MIB object. The generic form of the MIB identifier is the object ID with an appended .0 subidentifier. In the cases of the aggregate types, the SNMP defines the instance identifiers on a case-by-case basis. These instance identifiers are generally things such as interface number for referencing into interface tables or network addresses for referencing into routing tables.

### 2.6.2 The SNMP Messages

Every SNMP message consists of a version number, a community, and a data portion. The version number is an integer, with the current version defined as zero. The biggest difference between the SNMP messages and the SGMP messages is the SNMP Community string that is in every SNMP message. This is essentially the equivalent of the session ID that was used for the SGMP but with different

duties. The community string defines the access that the SNMP message has to the MIB. Each community string defines the administrative relationship between the agent and the management entity. The community string controls what portions of the MIB the monitoring station can view, what kind of access to the MIB objects the monitoring station has, and even what entity other than the one that received the message should the message be applied against. The latter is known as proxy SNMP and allows a monitoring station to query a proxy SNMP agent and have its requests dealt with by the rules pertaining to its access to the proxy agent. The data portion of an SNMP message is referred to as a protocol data unit (PDU). SNMP supports five PDU's: GetRequest, GetNextRequest, GetResponse, SetRequest, and Trap.

The GetRequest, GetNextRequest, GetResponse, and SetRequest PDUs are essentially identical in structure to the SGMP GET/SET REQUEST/RESPONSE message types. They consist of a request ID, an error status, an error index, and a list of variables. These variables are defined as a sequence of object identifier and object value pairs, which are similar in concept to the SGMP variable lists. One difference is that the SGMP required the sending of placeholder values in the GET REQUEST message for symmetry. The SNMP allows the ASN.1 NULL type as a valid value, which eliminates this need in GetRequest and GetNextRequest PDU's and reduces packet size as well.

The Trap PDU definition is quite different from its SGMP counterpart. It consists of an enterprise variable, agent address, a generic trap variable, a specific trap variable, a time stamp, and an optional list of variables. The enterprise variable is the enterprise Object Identifier that uniquely identifies the vendor and type of device that generated the trap. The agent address is the actual address of the

source of the trap. The generic trap value can be set to one of seven pre-defined values: cold start (system reboot, value of 0), warm start (reinitialization, value of 1), link down (communications link has failed, value of 2), link up (a communications link has been enabled, value of 3), authentication failure (an SNMP message was received that failed authentication, value of 4), EGP neighbor loss (an EGP peer has been marked down, value of 5), and enterprise specific (this is a vendor's implementation specific trap, value of 6). If this value is set to *enterprise specific*, the application needs to check the specific trap variable in the context of the value of the enterprise variable to determine what type of trap this is. The time stamp shows the number of ticks (defined as 1/100ths of a second) since the network entity was initialized. Lastly, the optional list of variables is defined the same as the Get/Set Request/Responses. The *linkUp* and *linkDown* define the first element of the variable list as the name and value of the *ifIndex* instance for the affected link. The *egpNeighborLoss* trap defines the first element of the variable list the name and value of the egpNeighAddr instance for the effected neighbor. The rest of the traps have contents defined as implementation specific.

### 2.6.3 Processing The Different PDUs

As expected, the processing of SNMP PDUs is very similar to that of the SGMP messages. The SNMP GetRequest is used to perform an *equal to* variable extraction. This means that each object in the PDU's variable list must contain a fully qualified variable name, including the specific instance identifier. This PDU is used to extract the variables that the management station has *a priori* knowledge about or is interested in the variable's actual existence, such as a route.

The GetNextRequest is essentially the SGMP GET REQUEST, where a lexicographical *greater than* function is applied from the variable list specified Object Identifier to the MIB variable space. Note that the use of instance identifiers, specifically the user of the generic instance identifiers for variables where there is only one instance, exactly parallels the SGMP system of putting an implementation specific octet string at the end of variable names. This allows the GetNextRequest PDU to be used exactly like the SGMP GET REQUEST in algorithms for table traversal or extracting specific variables without having to use the GetRequest PDU.

The SetRequest PDU is a little more refined than the SGMP SET REQUEST. It requires the application processing the request to determine if all the variables in the set request can be set to the requested value. This is primarily configuration checking beyond the normal type checking that the SGMP performed. Also, it requires that each variable assignment take place as if simultaneous with respect to all other variable assignments in the same PDU. This allows the SNMP to create or modify table entries that by their nature require several variables to be present at the time of entry creation.

The GetResponse is sent in response to GetRequest, GetNextRequest, and SetRequests. For all requests, if the response creates a PDU that is too large due to local limitations, it sends back a GetResponse that looks like the request except that the error status is set to *tooBig*. If an object cannot be retrieved or set for reasons that are not covered in the error messages, a GetResponse identical to the request is sent back with the error status set to *genError* and the error index pointing to the offending object. When a GetRequest or SetRequest has been processed that has a variable Object Identifier that does not exactly match one

in the MIB or is an aggregate object, a GetResponse identical to the request is sent back with an error status of *noSuchName* and the error index pointing to the offending object. In the case of the SetRequest, if the community does not have write permission for that object, the *noSuchName* error is also generated. When a GetNextRequest that contains an object that is not lexicographically less than any of the objects in the variable space has been processed, a GetResponse is sent back with the error status of *noSuchName.*

### 2.6.4 Authentication In The SNMP

The SNMP has provisions for authentication mechanisms, but defines only a trivial scheme in the protocol document. The SNMP message header has the PDU in an area referred to as *user data.* The design is such that the authentication systems can do whatever transformations they need to convert PDUs to user data and vice versa. This allows the addition of encryption information, fields that are authentication protocol specific, or whatever is necessary to satisfy the implementors' and users' security requirements. However, the goal of these management protocols is open management across administrative boundaries, so the SNMP document only refers to a trivial authentication protocol where the PDU is put into the user data with no transformations.

### 2.6.5 Analysis Of The SNMP

Compared with other network protocols, the SGMP and the SNMP are almost identical. The MIB variable space is quite a bit larger than the SGMP's variable space, but this is understandable since the SNMP's scope is more than just IP

routers. Several of the differences between the SGMP and the SNMP can be seen as moves to conform more closely to international standards, such as the use of Object Identifiers and the reworking of the ASN.1 PDU specifications. Additionally, the SNMP has benefited from the development and operational lessons learned from wide-spread use of the SGMP [22]. For example, the addition of the proxy network management concept outlined in the SNMP document was not present in the SGMP, but rather the direct result of experimentation with the SGMP. As a result of this experience, the SNMP has all of the advantages of the SGMP: small, easy to implement, and low overhead on the monitored entities. Lastly, as the name change implies, the protocol was expanded to manage networks, not just monitor routers.

The SNMP still has some limitations. While it does have provisions for authentication services, nothing has been fully accepted by the SNMP community other than the trivial authentication protocol. Work is in progress to address this shortcoming, however [23]. Also, the *as if simultaneous* aspect of the SetRequest operation may be difficult to implement. If a PDU contained two requests to set the same variable's value, this is a race condition and therefore the request clearly must be rejected. However, the success of a SET operation is dependent on more than value of other variables in the PDU; it depends on the state of the entity being set. This state includes variables that not only have dynamic values, but their very existence is dynamic. For example, an entity may have a route deleted via the normal operation of the routing protocol. This deletes an instance of variables in the routing table. If the network management entity was in the middle of performing operations against that table entry when it disappeared, the *as if simultaneous* aspect is lost. The only way to prevent this from occurring is to lock

down the managed entity whenever a network management operation takes place, a very undesirable alternative. The SNMP authors never used the term *atomic* in their description of the SetRequest, opting for a *best effort* paradigm instead.

## 2.7 CMIS/CMIP Over TCP/IP

During the time that the HEMS/HEMP and the SGMP were being defined, a global networking standard was being developed named the Open Systems Interconnect (OSI). One of the items addressed by this effort was network management. The proposed OSI network management architecture is the Common Management Information Services/Common Management Information Protocol (CMIS/CMIP). A group of vendors formed the Network Management working group (NETMAN) and proposed CMIS/CMIP Over TCP/IP (CMOT) as a network management protocol [9]. This protocol uses the same Management Information Base (MIB) [20] and Structure and Identification of Management Information (SMI) [21] as the Simple Network Management Protocol (SNMP). CMOT, as defined in RFC 1095, is based on the Draft International Standards CMIS/CMIP DIS 9595 and DIS 9596 [24, 25].

### 2.7.1 CMIS/CMIP

The CMIS/CMIP is an application level network management protocol that allows the transfer of network management information between peer systems. CMIS/CMIP consists of two parts. The CMIS defines the Application Service Element (ASE) for sending management information and commands [24]. An ASE is the module responsible for presenting a strictly specified common interface

to the applications that build upon the CMIS/CMIP. The CMIS provides a set of command primitives and semantics that are to be used by managers and managed entities alike. The CMIP is the protocol specifications needed to allow CMIS ASE's to intercommunicate [25]. It describes the actual Protocol Definition Units and supporting ASN.1 constructs needed to transfer the information between the CMIS ASE's.

### CMIS

The CMIS ASE defines six services: M-EVENT-REPORT, M-GET, M-SET, M-ACTION, M-CREATE, and M-DELETE. These services are grouped in the kernel Functional Unit. The CMIS specifies additional Functional Units that can be invoked to support options in these kernel services, such as allowing multiple objects to be specified, multiple responses, and filtering. The multiple objects Functional Unit must be specified when scoping or synchronization options are used in a service. The multiple responses Functional Unit must be specified to allow multiple objects to be transferred in response to a single object request. The filter Functional Unit has to be specified when the filter options are used in a service. When an application initializes the peer to peer communication, known as an Association, option negotiation between the ASEs determine what additional Functional Units can be supported by both ends.

The services break down into two categories: notification and operations. The M-EVENT-REPORT service sends notifications of events to manager entities. The other services perform operations on management information. The management information is a collection of objects similar to the MIB discussed in the SNMP section. These objects are grouped in classes that can also be grouped in a hierar-

chical fashion. Unlike the SNMP representation of data, objects can be aggregate objects such as table entries or even tables themselves and not just the discrete leaf nodes in the Management Information Tree (MIT).

The M-EVENT-REPORT service is analogous to traps in the SGMP/SNMP. This service is used by a managed entity to notify a peer CMIS ASE application that some event of mutual interest has occurred. This can be used to inform a manager that an interface has failed. It can be sent in a non-confirmed mode, where no acknowledgment of the message is expected (such as SNMP traps), or in confirmed mode, where retransmissions will occur until an acknowledgment is received. An M-EVENT-REPORT request consists of an invoke identifier, which is a unique identifier for the message, the mode of operation, the class of the object that had the event, the object's instance, and the type of event. The time of the event and the general information for the event fields are optional. If confirmation is requested, the M-EVENT-REPORT reply consists of an invoke identifier, optionally the managed object's class and instance and the current time of the response, and the event type and event reply if this is a successful event notification or the error codes if the event notification is unsuccessful. The M-EVENT-REPORT service does not use any of the options that require additional Functional Units.

The M-GET service is used to retrieve information from a peer CMIS ASE application. This service only operates in confirmed mode, with the information requested being sent in the confirmation. An M-GET request consists of a unique invoke identifier, the base object class, and the base object instance. Optionally, it may contain scoping, filter, synchronization, an attribute identifier list, and access control information. Scoping and filtering are started from the point in the

Management Information Tree named by the base object class and the base object instance. When scoping and filtering are not used, this names the object to be retrieved. Scoping indicates how many levels deep into the object variables need to be extracted: the base object, the objects *n* levels down from the base, the base object and all those down to and include the *nth* level objects, or the base object and all those below. This could be used to specify a request for just the routing table. A filter is a logical expression that is applied to the values of the objects in the scope. The objects that evaluate to true are selected. This could be used in conjunction with scoping to select the routing table entries that are for network 128.169.0.0. Synchronization comes in two forms: atomic and best effort. If atomic is specified, all of the extractions in the request are checked to make sure they can be performed. Otherwise, only those that can be are completed and their partial results returned. Note that this only affects extractions when multiple managed objects have been specified. The attribute identifier list specifies those attributes of the object for which values should be returned. The lack of an attribute identifier list is an indication to return all attributes for that object. The access control data consists of data in whatever form is needed by the access control system used by this implementation of the ASE for security. An M-GET response consists of the invoke identifier and, if the extraction was a success, the attribute list of identifiers and values. Additionally, if the response is one of several objects being returned and not the last object, it may contain a linked identifier. If more than just the base object was identified in the request, the managed object class and instance identifier that corresponds to the managed object's attributes will be returned as well. The response can also contain the time the response was generated. Lastly, it will contain any error notifications if the response is a failure confirmation.

The M-SET service is used to modify the value of an object's attribute. The M-SET consists of almost the same components as the M-GET. The M-SET request consists of an invoke identifier, mode, base object class and instance, and attribute list with optional scope, filtering, access control and synchronization. The difference is the mode and the use of an attribute list instead of the attribute identifier list. The mode indicates whether or not the request is to be confirmed. The attribute list is the list of attribute identifiers and the values to which they are to be set. The rules on scoping, filtering, and synchronization are the same as with the M-GET. The M-SET response consists of the invoke identifier, a linked identifier, the managed object class and instance identifier, an attribute list, current time, or error notifications based on the same conditions as the M-GET response.

The M-ACTION service causes the CMIS ASE to perform some predefined action on the manager's behalf. For example, this could be the disabling of a network interface. The M-ACTION request consists of an invoke identifier, mode, base object class and instance, and action type. The action type specifies the particular action to be performed while the other mandatory items are the same as in the M-GET and M-SET. Additionally, the M-ACTION request may contain scope, filtering, access control information, synchronization, and action information. Again, most of these items are also optional in the M-GET and M-SET and are treated the same as in those requests. The action information specifies whatever information might be needed as arguments to the action type specified. The M-ACTION response contains the invoke identifier, a linked identifier, the managed object class and managed object instance identifier under the same conditions as the M-GET and M-SET. Additionally, it may contain the action type that was specified in the request, the action reply information, if any, that is appropriate

for that action, the current time that the response was generated, and, if this is a failure confirmation, any error notifications.

The M-CREATE service is used to create a new instance of a managed object. This could be used to create a new entry in a routing table. An M-CREATE request consists of an invoke identifier and the managed object class. Optionally, the request can contain the managed object instance, the superior object instance, access control information, a reference object instance, and an attribute list. The superior object instance indicates what object will be the superior of the new object. If the superior object instance is not specified, then the managed object instance indicates the instance of the new object. If that is not specified, then a new instance number is assigned to the new object by the ASE. When the new object is created, the values listed in the attributes list are assigned to those attributes in the new objects. If the reference object instance is specified, then those attributes not listed in the attributes list are set to the values of the reference object. If a reference object instance is not specified, then default values are assigned by the ASE. The M-CREATE response consists of the invoke identifier and, if the create was successful, an attribute list of all of the instance's attributes, the managed object class and the managed object instance. The instance will be sent if it was set by the ASE rather than via an instance identifier or the superior instance identifier. If the create was not successful, the response will contain error notification. The response can contain the current time that the response was generated regardless of the success or failure. Note that the M-CREATE service does not use any of the options that require additional Functional Units.

The M-DELETE service is used to delete an instance of a managed object. This could be used to delete several entries from a routing table. The M-DELETE

request consists of an invoke identifier, a base object class, and a base object instance. It may optionally have scoping, filtering, synchronization, and access control all similar to those of the M-GET and M-SET. The M-DELETE response consists of the invoke identifier, a linked identifier if this is a linked response, the managed object class and instance if more than the base object was specified, or error notification if the deletion failed. Optionally, it may also have the current time that the response was generated.

The logical connection between the two CMIS ASEs is established and released through the use of another OSI protocol, the Association Control Service Element (ACSE). The CMIS uses the ACSE service A-ASSOCIATE to allow a CMIS ASE to request a connection to another CMIS ASE. Through the use of this service, applications are able to exchange initialization information. The CMIS also uses the ACSE services A-ABORT and A-RELEASE to terminate the associations.

### CMIP

The CMIP is the actual protocol definition that describes how the CMIS information is exchanged [25]. It is built atop the OSI remote procedures protocol named the Remote Operations. Application Service Elements such as the CMIS/CMIP communicate with a remote node's CMIS ASE by constructing an Application Protocol Data Unit (APDU) that contains the message to be exchanged. CMIP uses the services of the Remote Operations Service Element (ROSE) to actually send the APDU to the remote CMIS ASE.

The application is responsible for establishing the association between the two CMIS ASEs by using the ACSE A-ASSOCIATE service. When an application invokes a CMIS service, it communicates with the CMIS ASE as described in the

CMIS section above. The CMIS ASE sends the requests to the CMIP Machine, which is responsible for forming Application Protocol Data Units and transferring them via the use of services of the ROSE. The form of these APDUs for the various CMIS service requests are detailed in the CMIP specification in the Abstract Syntax Notation One (ASN.1) language so that it is rigidly defined. The CMIP Machine sends the APDU to the associated remote CMIP Machine by use of the ROSE service RO-INVOKE. The remote CMIP Machine checks the APDU it has received and sends it to its CMIS ASE if the APDU has not been corrupted. The CMIS ASE then delivers the message to the application using the CMIS services. If the APDU has been corrupted, the CMIP Machine constructs an APDU relevant to the error it has detected and sends it back to the invoking machine via the RO-REJECT-P service.

At this point, the message has been delivered to the remote CMIS ASE which passes it on to the remote application using the message. The application may generate a response, such as a confirmation to the message. This response is sent back through the CMIS ASE to the CMIP Machine. The CMIP Machine builds the APDU corresponding to the response and sends it back to the originating node. If the response is a notification that the message has been accepted, the response is sent back via the RO-RESULT service. If it is notification that the message was rejected, it sends it back via the RO-ERROR service. When the originating node's CMIP Machine receives the APDU, it checks to see if its been corrupted. If it has, it sends back an APDU containing error information using the RO-REJECT-P service. Otherwise, the message is delivered to the CMIS ASE which delivers it to the application using its services.

### 2.7.2 CMOT

The goal of CMOT is to use OSI network management on TCP/IP networks to manage TCP/IP networks. The International Standards specify the CMIS/CMIP network management and the existing RFC's specify the TCP/IP networks. The primary task in specifying the CMOT protocol is in defining the interfaces between the two different sets of standards. These interfaces take place primarily in two areas: the actual protocol stack and the organization of management information.

When looking at network protocol sets, it is common to place the components in areas that correspond to the duties defined in the seven layers of the ISO reference model [26]. For example, the CMIS/CMIP and the SNMP belong in the application layer, TCP and TP0 belong in the transport layer, and IP and OSI IP belong in the network layer. There are usually well defined interfaces between the individual layers of the protocol stack. It is obvious that in order to put OSI network management services on top of TCP/IP, the result is a protocol stack TCP/IP layers on the bottom and OSI layers on the top. The problem is deciding where to make the transition from the TCP/IP protocols to the OSI protocols.

Other research was being conducted on the Internet into ISO OSI transition and a protocol for running ISO transport services over TCP had already been published. One proposal the NETMAN group entertained was to implement the CMIS/CMIP, ROSE, ACSE, an ISO presentation layer, and an ISO session layer on top of this ISO transport service that would run over TCP [27]. It was felt that it would be too expensive to implement "because of the vendor-specific nature of each implementation and the overhead required to implement complete OSI layers." [28] CMOT instead adopted a protocol for ISO presentation services on

top of TCP/IP. This protocol was limited in that it could only support applications that only needed ACSE, ROSE, and a limited directory service. This protocol was derived explicitly for use with the CMIP and supports exactly what is required by the CMIP [29].

The CMIS/CMIP was defined in an object oriented manner. As a result, it is designed to manage *objects*. "A 'managed object' is an abstraction (or logical view) for the purposes of network management of a 'manageable' physical or logical resource of the network." [8] Each object belongs to an object class which contains objects that are related by some common properties. Each object has attributes that define the various properties of a particular object. Thus, a specific routing table entry would be a managed object with the attributes of network, metric, next hop, etc. It would also be a member of the object class Routing Table Entry, along with the other routing table entries.

CMOT deals with management information through two main relationship hierarchies. The Internet MIB defines a registration hierarchy where managed objects and attributes are given a unique Object Identifier. Objects also have the relationship of belonging into common object classes. The object classes themselves can be grouped together with other object classes to form a containment hierarchy. The containment hierarchy is primarily what the CMIS/CMIP uses to access network management objects and was not defined in the Internet MIB and SMI. The CMOT specification deals with this by supplying a definition of the object classes and their containment relationships with other object classes in an appendix. By applying this class mapping, the CMIS/CMIP functions can manipulate the TCP/IP information tree defined by the MIB with the full functionality of scoping, filtering, and linked replies.

The one glaring difference between CMOT and the CMIS/CMIP as defined in the International Standards is the CMOT M-INITIALIZE, M-TERMINATE, and M-ABORT services that are absent in the CMIS. These services are defined as pass through services to ACSE services and it is assumed that these were dropped from the CMIS Draft International Standard when it became an International Standard.

### 2.7.3 Analysis Of CMOT

The CMIS/CMIP is a very powerful network management protocol. It allows access to individual variables or can extract full tables with equal ease. It provides means of manipulating data, invoking action, and reporting events in a logical manner. It is an accepted Internation Standard that will be see increased use as OSI network is implemented. CMOT is an attempt to utilize that power to manage existing networks and to be a transition tool used in managing these TCP/IP networks as they migrate to OSI. It provides vendors with a common network management interface that has the power to support nearly any network management applications.

The protocol suffers the same complexity problems as the HEMS/HEMP. It has controlled some of this by making negotiation of the Functional Units possible, so that a limited resource machine theoretically only has to support the kernel functions. There is a problem with this concept in that if a manager wants to extract just one variable instance from an entity that does not support the filter and scope extensions, it has to extract an entire object. Most of the power of the protocol comes from its optional components. As a result, the minimum required system is a very inflexible and inefficient protocol, and network management implementations

will have to be quite complex to deal with these minimal systems.

CMOT made provisions to run under UDP with the idea that it could be used in a fire-fighting mode like the SNMP. Unlike the SNMP, however, there is no mention as to whether the UDP datagram must fit in a single IP datagram. This is critical if the UDP datagram is fragmented across several IP datagrams, since the loss of a single IP datagram would require a total retransmission of all the IP datagrams. In fire-fighting mode, the loss of a single datagram may be highly likely, depending on the problem being fought. If it was assumed that the UDP message would fit in a single IP datagram, the overhead required by all of the various layers it calls and the encoding for class and instance make the amount of data that could be extracted in the datagram quite small. Also, the instance naming conventions for accessing many of the table objects requires *a priori* knowledge to extract the full table an entry at a time. As a result, the UDP transport is essentially useless for data extraction. However, it should be noted that using UDP for issuing actions should still be practicable, provided the user data for the action is not too large.

# CHAPTER 3

# Implementing The Simple Gateway Monitoring Protocol

## 3.1 Introduction

The Internet tradition of implementing a protocol before committing it to paper was followed in the development the SGMP. The four authors of the SGMP Request For Comments split into four development groups. Chuck Davin, then working for the router manufacturer Proteon, Inc., was to implement the agent in that company's P4200 product. Marty Schoffstall, then of Rensselaer Polytechnic Institute (RPI) was to implement clients that would be the basis of a Network Operations Center. Mark Fedor, then of Cornell University, was implement an agent using the RPI SGMP libraries in his GATED implementation, a daemon that allowed Berkeley UNIX based machines to communicate with several routing protocols simultaneously and act as routers. Jeff Case, at the University of Tennessee, Knoxville, was to lead an effort to produce clients that could be used in trouble shooting networks. Together, this provided two independent implementations of clients, two independent implementations of agents, and three independent SGMP parser/builder implementations.

After deciding to implement the SGMP clients, several design decisions had to be made about the implementation. The first design decision was that a core library to parse and build the SGMP packets should be implemented and used as the basis of all the SGMP applications. Second, it was decided that the library and

the applications be as portable as possible. Third, that the particular problems that should be addressed by the fire-fighting tools had to be determined.

While in the process of implementing these clients and agents, the protocol was open to evolution based on the problems and experiences of the implementors. The primary purpose of this implementation effort was to be certain that the protocol would be implementable and performed the desired functions before it was committed to publication. Because of this flexibility and openness to new concepts, several ideas that were not strictly client tools surfaced and were investigated. Some of the more interesting ones of these include SGMPQUERY, a program that simulated a router SGMP agent using real router data; proxy SGMP, which would allow a hierarchical control of access to the managed agents; and HOSTSGMPD, which was an experiment in developing an agent to run on hosts rather than routers using the SGMP and extending on the variable naming concepts.

## 3.2 Portable Library

The portable library is the core of all of the SGMP applications that this thesis produced. It provides a common programming interface that is used to create the SGMP requests and to parse the responses that the clients receive from the agents. As the core, it is called repeatedly during the running of an SGMP application, so robustness and speed were considered important in its design. The desire of portability to a wide variety of platforms, some of which are limited in their resources, often ran counter to this specific goal and forced the sacrifice of speed for smaller memory utilization. Lastly, as the specification was still not finalized, a certain amount of flexibility to add new data types or structures needed

to be present.

Three main methods were considered in building the parser/builder library. The SGMP language is expressed in X.409 [16] which is now a subset of Abstract Syntax Notation One (ASN.1). There exists an ASN.1 code generator named PEPY in the ISO Development Environment [30] package created by Marshall Rose, then of the Wollongong Group. A cursory examination of PEPY showed it to be intimidating in its complexity. The UNIX operating system includes language tools named YACC and LEX [31, 32]. These tools produce large amounts of code as output and was considered inappropriate from the standpoints of portability and size. The method chosen was to write a parser/builder system in the popular C programming language that would implement the subset of ASN.1 used by the SGMP. This would allow both close control and tuning of the code and portability to a wide variety of platforms.

### 3.2.1 ASN.1 Considerations For The SGMP

The ASN.1 language defines each data element as a tuple of Identifier, Length, and Contents. It states: "Two FORMS of data elements are distinguished by means of the Identifier: primitive and constructor. A PRIMITIVE element is one whose contents is atomic, that is, has no further internal structure of data elements. A CONSTRUCTOR element is one whose Contents itself is a data element, or a series of data elements. Constructor elements are thus recursively defined." [16] The four protocol messages defined in RFC 1028 makes heavy use of this recursive ability to define sequences of variable-name value tuples to address the actual data values that are being extracted from or set in the routers. When looking at

a recursively encoded data element, it appears as the sequence of octets denoting the type Identifier, the encoded octets representing Length of the Contents, and the Contents themselves which consist of a series of octets representing an ASN.1 encoded data element. It becomes obvious that to build a packet, one must first recurse down to the lowest level, encode the primitive data element as Type, Length, Contents, and then pop up a level to encode the next lowest data element using the lower levels element's encoded octets as the Content for this element in a recursive fashion. This meant that the data stream would have to be encoded in a right-to-left fashion with the SGMP message fully defined in some internal representation before encoding began.

The ASN.1 elements used in the SGMP specification were limited to INTEGER, OCTET STRING, SEQUENCE, SEQUENCE OF, and application types that were IMPLICIT SEQUENCE. In the early phase of implementations, an agreement was made to limit OCTET STRING elements to primitive form and not allow CONSTRUCTOR form. Also, lengths were limited to the Short and Long forms, eliminating the Indefinite form. This greatly simplified the implementation of the early libraries. These restrictions were eliminated as the work on the libraries progressed.

### 3.2.2 Packet Builder Internals

It should be noted that in designing an SGMP packet builder, it is the builder's choice as to whether or not to use constructor octets and indeterminate lengths. No obvious benefit was seen from using constructor octet strings in an SGMP packet compared to the complexity that implementing it brought to the code.

Also, using the recursive packet building procedure described above, the length of the contents is always known so there is no need to use indeterminate lengths. Since the elimination of both items makes a packet that is easier to parse, neither aspect was used in designing the packet builder.

The packet builder builds a tree of linked lists that is recursively traversed to build up the ASN.1 packet. Each linked list consists of the elements that have to be encoded to make up the Contents of the data element that points to the list. The builder works in two phases. First, the structure representing the packet being created is established by calling functions defined for each individual data type added for the packet. Second, the *build_packet()* function is called to recursively traverse the tree, building up the ASN.1 encoded Components until the final encoded packet is produced.

Each node of the tree is an identical structure named ELEMENT as shown in Figure 1. The Type variable consists of the ASN.1 value that corresponds to the type of element that this particular element is supposed to address. The length variable is the length, in octets, of the octet string referenced to by the *string_ptr* variable. The *string_ptr* variable points to the encoded data component of the element. It is filled in during the *build_packet()* function call or, if the element is a primitive type, when the element is created.

Since the constructor octet string Type is eliminated, the only constructor form data elements are the SEQUENCES: sequence, sequence of, and implicit sequence. An SGMP packet is defined using an implicit sequence, so all elements are going to be a member of a sequence. The *start_ptr* variable of the ELEMENT structure points to the first element that composes the sequence of data elements that makes up this element's data Component. The *end_ptr* variable is a pointer to the last

```
/* value element - used in the linked list of values */
struct ELEMENT {
  int type;      /* ASN.1 type code */
  long length;  /* of string - if any */
  unsigned char *string_ptr; /* the corresponding encoded octet string */
  struct ELEMENT *start_ptr; /* ptr to first element of new sequence */
  struct ELEMENT *end_ptr;   /* ptr to last element of new sequence */
  struct ELEMENT *next;      /* ptr to next element in curr. sequence */
  struct ELEMENT *seq_ptr;   /* ptr to current active sequence */
};
```

Figure 1: The ELEMENT structure.

element in the sequence and is used primarily to make additions to the end of the sequence more rapidly. The elements that make up a sequence are connected via the next variable, starting with the variable pointed to by the sequence element's *start_ptr*. The *seq_ptr* variable is only used in the root data element. It is a place holder for the sequence that elements are currently being added to. It is present in this structure for symmetry to simplify the implementing of the memory allocation and freeing routines.

A data structure that is defined in the recursive nature of the SGMP packet is the *var_op_type*. This is a sequence consisting of an octet string that lists the variable name and either an integer or an octet string that contains the data value associated with the named variable. These are allocated and manipulated both internally to the library and externally by the programmer using the library. To aid in manipulating these data constructs, the *VAR_VAL* structure is defined as shown in Figure 2.

The *var_name_ptr* variable points to the OCTET structure, shown in Figure 3, which contains the octet string that defines which variable is being addressed per the conventions established in RFC 1028. The type of a variable is the ASN.1 type value, either a primitive octet string or primitive integer. If it is of type octet string, *value_ptr* points to the OCTET structure that contains the octet string representing the value. If the Type is integer, the value is stored in the variable *long_value*.

The variable *length* is the number of bytes used to hold the octet string. The variable *string_ptr* points to the memory that holds the actual octet string. Note that the length is necessary as the octet string can contain nulls in the middle and the C language does not directly support length encoded strings.

```
struct VAR_VAL {
    struct OCTET *var_name_ptr; /* variable name */
    int type;                   /* OCTET or INTEGER */
    struct OCTET *value_ptr;    /* used if type octet string */
    long long_value;            /* used if type integer */
                                /* (32-bit max integer allowed) */
};
```

Figure 2: The *VAR_VAL* structure.

```
struct OCTET {
    long length;                /* length of octet string */
    unsigned char *string_ptr;  /* actual encoded octet string */
};
```

Figure 3: The OCTET structure.

### 3.2.3 Using The Packet Builder

An SGMP structure consists of two main parts, despite the recursive nature of the definition. The first part is the header that contains the application Identification type, the packet identification number, the error status, and the error index. An additional component of the header in proxy SGMP is the IP address, to be addressed in a later section about the proxy SGMP extensions. These elements are part of an initial sequence (implicit from the application type) and end with a *sequence of* data element that starts the second part of the structure. This second part is a sequence of *var_op_type* elements which consist of a sequence of a variable name and a variable value as described in the *VAR_VAL* structure.

Only a few library function calls have to be used when writing an SGMP program using the Packet Builder. First, there are the *make* functions that create the *VAR_VAL* combinations of the SGMP variables in question. Second, there is the *start_packet()* function that establishes the initial structure of the SGMP packet tree. Third, there is the *add_var_val()* call to add the variables in question to the tree. Fourth, there is the *build_packet()* call that traverses the tree to produce the final ASN.1 encoded packet. Fifth, the authentication information is added to the packet via the *add_auth()* function. Last, there are the *free* functions that recover the memory allocated for the packet tree and for the *VAR_VAL* structures.

There are two *make* functions: *make_var_val_octet()* and *make_var_val_integer()*. The *make_var_val_octet()* function is for creating a *VAR_VAL* structure with an octet string value from textual representations of octet strings for the variable name and for the value. The *make_var_val_integer()* function creates a *VAR_VAL* with an integer value from a textual representation of an octet string for the variable

name and a long value as the integer. The textual representation of the octet string is a character string consisting of pairs of hexadecimal numbers separated by a blank. Each pair represents a single octet in the string. Both routines return a pointer to the *VAR_VAL* structure created to accommodate the variable.

The *start_packet()* function is called with the data that makes up the header part of the of the SGMP packet. That is, the application type, packet ID number, error status, and error index. The function creates the necessary ELEMENT structures and strings them together so that all the programmer needs to do is to add the *VAR_VAL* structures for the variables. The function returns the root ELEMENT structure of the tree.

If the application is type TRAP, one can add integer or octet strings to the end of the trap. This is trap-specific data and is ignored for all the traps defined in RFC 1028. Otherwise, the *add_var_val()* is called with the root ELEMENT structure's *seq_ptr* variable and pointer to the *VAR_VAL* structure allocated for the variable to be addressed by the packet. The routine is called repeatedly, once for reach variable to be added to the sequence.

Once all the variables have been added to the message, the *build_packet()* function is called with the root pointer and the pointer to a long word to hold the length of the encoded ASN.1 packet. It returns a character string that is the encoded packet.

The *add_auth()* function is called with the pointer returned by *build_packet()*, the length of the packet returned by *build_packet()*, a character string that represents the authentication *session*, the length of this session string, and a pointer to a long word to hold the total length of the packet returned from *add_auth()*.

The function returns a pointer to a packet it has created that includes length of the SGMP packet, the session ID length, the session string, and the ASN.1 encoded SGMP packet. This authentication is considered a separate layer from the SGMP packet and could include some encryption of the SGMP packet. The SGMP specification does not call for a particular authentication function, and this implementation initially uses the identity function on the SGMP packet.

### 3.2.4 Packet Parser

The packet parser performs the inverse function of the packet builder. That is, it takes a string of octets and attempts to extract the SGMP information contained inside the string. As mentioned earlier, the parser originally was built on the assumption that constructor octet strings and indefinite lengths would not be used. It was later decided that these features should be added for the sake of compliance to ASN.1. It is true that the SGMP only uses a subset of ASN.1, and it was argued that making these restrictions would only be specifying a further subset. However, it was noted that an implementor using an ASN.1 implementation tool might not have control over whether the code used constructor octet strings and indefinite lengths and not be able to interoperate because of non-compliance to this restriction. Because of this, all good implementations of the SGMP should be able to handle indefinite lengths and constructor octets.

Since the SGMP packet is built from tuples of Identification Type, Length, and Content, where Content could recursively contain more tuples, it is easy to parse the packet from the start of the packet towards the end. To parse a tuple, one first needs to determine the Type of the data tuple being addressed, then the

length, and then the Contents. By calling the Type-specific parsing routine based on the Type and passing the length associated with the Type, the routine knows the Type of data to expect in the Contents element. This routine can then call a type-specific parsing routine based on the Type of the contents and thus recursively parses the packet. Through all of this, the pointer to the position in the packet being parsed progresses from the beginning of the packet to the end. The lengths can be checked against the end of the packet to determine if they were corrupted. The types expected can be checked against the variable types in the Content to see if they have been corrupted. Much of the code in the parser is the detection and handling of these potential parser error conditions.

The parsing of an SGMP packet is simpler than the building of the packet. Since an SGMP packet consists of two main parts, a header structure was defined to hold the pointers needed to parse a packet and the information contained in the SGMP header. The header structure was name HEADER and is as shown in Figure 4.

The *working_ptr* variable points to the location in the actual encoded packet that is currently being parsed. The *start_ptr* points to the beginning of the packet being parsed. The *end_ptr* is the estimated end of the packet calculated from the *start_ptr's* address plus the length. This is used quite often to keep the parser from going past the end of a packet should an error occur while trying to parse its contents. The *appl_type* variable holds the application type of the packet received, that is, GET_ REQUEST, GET_RESPONSE, SET_REQUEST, TRAP, PROXY_GET, or PROXY_SET. The *id_num* variable holds the SGMP packet id number. The *error_status* variable holds the error status code value contained in the packet. The *error_index* variable holds the index into the packet where the

```
struct HEADER {
  unsigned char  *working_ptr; /* ptr to current position in packet */
  unsigned char  *start_ptr;   /* ptr to beginning of packet */
  unsigned char  *end_ptr;     /* calculated end of packet
  int appl_type;               /* Message type */
  long id_num;                 /* identifier number */
  long error_status;           /* Error status code */
  long error_index;            /* index into message of error */
  unsigned long ip_addr;       /* IP address (used for proxy) */
  long eoc_counter;            /* count of End-Of-Contents expected */
};
```

Figure 4: The HEADER structure.

error, if any, occurred. The *ip_addr* holds, in network byte order, the IP address of the remote router to query. This is required for proxy SGMP and is discussed later. The *eoc_counter* is used while parsing the packet to keep count of how many End-Of-Content packet data types that need to be seen to match the number of indefinite length encodings that the parser has encountered.

### 3.2.5 Using The Packet Parser

Using the packet parser consists of four main steps. First, the authentication layer header must be extracted to get to the ASN.1 encoded SGMP packet. Second, the SGMP *header* part needs to be parsed out. Third, the variable name-value information that was requested needs to be parsed out. Last, the memory allocated needs to be freed for further use.

The *del_auth()* function is called with a pointer into the raw packet received, a pointer to a *long* type variable to hold the length contained in the authentication header, a pointer to receive the address of the session id in the packet, and a pointer to a long variable to hold the session ID length. The routine returns a pointer to the ASN.1 encoded SGMP packet. If there was an authentication protocol being used other than *trivial*, the packet would then possibly be decrypted.

The check length is checked against the actual packet length received by the application. The application then calls the *parse_header()* function with the pointer to the SGMP packet returned and the length of the SGMP packet. The *parse_header()* function returns a pointer to a HEADER structure that has had everything up to the variable-value sequence parsed out of the packet and checked. The HEADER structure is filled with the values found in the SGMP header. If the values indi-

cate that there were no errors, the packet is further processed. If an error status is returned, then the implementation may want to process the packet to examine the variable pointed to by *error_index* or just drop the packet, depending on the implementation.

If the application type is a trap and the trap is one of those defined in RFC 1028, then no further parsing is done as they do not use the *var_list* component to send any data. If it is an implementation specific trap, *parse_int()* or *parse_octet()* is called depending on the type of data the implementation expects. If the application type is not a trap, the routine *parse_var_val()* is repeatedly called with the *header_ptr* to return *VAR_VAL* structures containing the information on each variable returned.

Lastly, *free_var_val()* is called for each variable returned after processing the variable is done. Also, the pointer to the HEADER structure is passed to *free()* to free memory used by the structure.

### 3.2.6 Proxy SGMP And The Portable Library.

While in the process of developing and testing the SGMP libraries, the idea of proxy SGMP was developed. The details of proxy SGMP will be discussed further in the document, but its implementation required some library modifications. Proxy SGMP added the PROXY_GET_REQUEST and PROXY_SET_REQUEST types and added another integer in the packet's *header* sequence, the IP address of the target machine in network byte order. These were easy modifications to make to the library due to the flexibility of its design.

In adding the IP address, a machine dependency was introduced to the library.

Hosts do not necessarily have the same byte ordering of integers as the network defines, so references to the host's network-to-host byte ordering routines have to be included in the library. As a result, the library now has a compile-time conditional dependency.

### 3.2.7 Summary Of The Portable Library

The portable library was successful in running on multiple platforms. It was originally developed under VAX/VMS using VAX C and the VAX debugger. When it became time to use the library with TCP/IP applications, development shifted to an IBM-RT running the ACIS operating system (based on BSD 4.2) and a VAX 8200 running Ultrix 2.0. After the first few applications were running and the library's practicability was determined, it was tested on an IBM PC/AT using the Microsoft C (version 4.0) compiler. The library was compiled and tested on SUN 3's running versions 3.5, an HP9000/350 running HP-UX version 6.0 (a System V variant with a BSD socket interface), a CDC Cyber 910 running IRIS-4D, and then back on VAX/VMS using the Wollongong TCP/IP implementation.

## 3.3 TOOLS DEVELOPED

The purpose of the library was to facilitate the implementation of the SGMP applications. Two main system-types were targeted to run these applications: BSD UNIX and MS-DOS. Berkeley UNIX is a widely available operating system that runs on a variety of machines and has TCP/IP built into the operating system. MS-DOS is available on a large number of low-cost IBM and IBM-compatible machines, but does not include a TCP/IP implementation. The Carnegie-Mellon

University (CMU) PC/IP implementation, which is based on the Massachusetts Institute of Technology PC/IP implementation, was chosen as it is available at no cost and requires no resources other than Microsoft C, Microsoft Macro Assembler and a PC platform to compile. It was thought that existing BSD machines could be used for network management, while the PC's would provide a low-cost alternative to those sites that did not have BSD machines available.

### 3.3.1 Tool Implementation Considerations

The programming interface to the TCP/IP implementations in BSD UNIX and in CMU's IBM PC implementation are entirely different. Code written for the tools was not portable between the PC platforms and the BSD platforms, although the general concepts were. As a result, a tool was usually developed first on a BSD system and then ported to the PC environment.

#### BSD-based Tool Implementations

The BSD-based tools were generally easier to write since BSD is a multitasking operating system with TCP/IP integrated into its kernel. The tools all follow the same general format. The first step is to check that the number of arguments is correct. Next, a time out signal handler is registered to be called if no packet is received by the time out interval. A UDP socket is then opened, a *socket_in* structure is completed for the target destination, the SGMP packet to be sent is created, and then the packet is sent with a *sendto()* call. Memory allocated in creating the packet is reclaimed, an alarm time out is set and a *recvfrom()* call is made to receive the response. If a time out is received, the application either exits or performs some retry mechanism. Otherwise, the packet is parsed, the response

is processed, and memory is recovered. If necessary, the application will loop to send another SGMP packet and continue until some implementation specific exit condition arises.

### MS-DOS-based Tool Implementations

Writing applications is more complex on the PC platform since it is not a multiprocessing system and the TCP/IP implementation is not integrated into the operating system. The arguments are checked and then a series of initialization routines are called to set up the TCP/IP and timer environment. Next, the SGMP packet is built and a timer allocated. Then, a UDP connection is opened by specifying the remote host and port and a buffer to be used when a packet is received from that host. A UDP packet is then allocated to hold the SGMP packet created earlier and the contents of the SGMP packet are copied into this packet. Next, a loop is entered to attempt to send the packet. The buffer pool in this implementation is limited and heavy traffic can fill all of the buffers, making transmission impossible until the some of the received packets have been processed. Therefore, the process yields its time slice to the other packet handlers until at least two buffers are in the free queue. Two are needed as an Address Resolution Protocol (ARP) packet may have to be sent first if the host is not in the ARP cache. After the packet has been sent, the process becomes the current task and the timer is set for a receive time out. The program then blocks this process until either the time out has occurred or a packet is received. This allows other process to run. When a packet is received, the program frees the memory previously allocated, parses the packet, process the packet, and loops, if necessary, to send another SGMP packet and continue as in the BSD implementation. The PC/IP implementation includes

a rudimentary process scheduler in order to perform the multitude of tasks that a TCP/IP process must do. Most of the complexity in the PC versions of the application is to accommodate this time-slice scheduler.

Once an initial application was written for a platform, it was easy to create more based on the same basic code. Most of the functions such as building, sending, receiving, and parsing the packets is the same between the implementations. This makes it easy to produce other tools by changing the variables being asked for or by changing how the received information is processed.

### 3.3.2 Generic Tools

The SGMP tools started out with very primitive programs to test the library and evolved into tools to answer specific network management questions.

#### GETID

GETID was the first SGMP tool written and is thus very simple. It generates an SGMP request for the ID variable *01 01 01* and displays the result it receives. This was meant primarily as a testing tool as the ID variable is usually the first variable implemented in a router SGMP agent. This allows the testing of the fundamental mechanisms of parsing and building in the remote entity before effort is put into populating the variable space.

#### GETANY

GETANY is a simple derivative of the GETID program. The primary difference is that the user specifies the variable to be retrieved rather than a canned variable

being requested. The variable is entered in a textual hex notation that reflects the naming of the variables in the RFC. To get the system ID, the user would enter *01 01 01* as the variable requested. To get the routing metric to the network 128.169.0.0, the user entered *01 04 01 02 04 80 A9 00 00.* If the variable was specified at the command line, multiple variables could be requested at one time. When the program received the response packet, it would print out the variables that matched the ones requested. If a variable was received that was not asked for, such as when the requested variable does not exist, the result for that variable is not printed out. In either case, the program then terminates.

### GETMANY

GETMANY is the most powerful tool of the simple tool set. It combines the requesting power of GETANY with a loop to allow the traversal of tables. The user inputs the *root* of the variable table that it wants to traverse, such as *01 04 01 02 04* to dump the routing metrics for all routes in the routing table or *01* to dump the entire SGMP variable tree. The program sends the *root* as a request to the agent to get the first variable in the sequence. It then checks that the variable returned matches the *root.* If they match, the variable is printed out and the program loops to make the next request. If they do not, the program is done and exits. In formulating the next request, the program sends back the fully qualified variable name it received from its previous request. That is, the variable name plus the trailing octet that uniquely identifies that variable. Since the GET_REQUEST operation is to return the variable that is the lexicographical antecedent of the value received, the agent sends back a GET_RESPONSE of the next variable in the table. The program continues through the variable tree until

it encounters a variable that differs from the *root* or it gets a *gmp_err_nix_name* is set in the response packet indicating that the end of the variable tree has been reached.

This is the most frequently used of the simple tools as it allows access to a single variable, a class of variables, or the entire variable tree just by specifying as much of the variable that is necessary to eliminate other, undesired, variables. One of the most common uses of this application was in a loop in a shell script that repeatedly traverses the variable trees in a router. This causes the parser and the builder to handle a large amount of traffic and any memory leaks become readily apparent.

**GETROUTE**

The GETROUTE program is a modification of the GETMANY program that specifically deals with routing table variables. It traverses the routing table's entries to get the next router for a route, the type of route, and the route's metric and prints them out in an easy to read form. This is the SGMP equivalent of the QUERY program, which dumps the routing table via the RIP protocol as described in the SGMPQUERY section below.

### 3.3.3 BSD UNIX Based Tools

UNIX is based on a philosophy of writing small tools that one can put together to perform larger functions. This philosophy was used to produce the programs HUDSON and JACKHAHN. This could also have been applied to re-write GETROUTE as a shell script that uses GETMANY and demonstrates the power

of this philosophy.

### JACKHAHN

The JACKHAHN program is actually a shell script based upon a program that returns the route metrics for a set of routes. Dr. Jack Hahn is the network administrator for SURANET and wanted a text-based tool that would tell him when routing metrics started to change from the steady state they had been before. The JACKHAHN shell script runs a program to dump the metrics for the routes currently in the router and saves them to a file. It then sleeps for some amount of time and runs again, comparing the current output to the saved output. Any differences in metrics result in the line being sent to the screen, so that the operators will receive notification as routes disappear and the metrics start to change.

### HUDSON

One common problem encountered in IP networks occurs when routers somehow learn invalid routes to networks. This can happen when an IP host is put on a network with the wrong IP address and it runs a ROUTED program that tells the routers on its network that it can get to the invalid IP network. The HUDSON program traces a route by querying successive routers for the route to a network. The user enters the network that HUDSON is trying to find. The program then asks the first router for the *next_gateway* variable for the network in question. Using this answer, it asks the new router pointed to by the *next_gateway* variable and does this successively until it finds the network or until it runs into a router for which it does not have configuration information about. By following the route, the user can determine the source of the bad information. This tool is also useful in

finding routing loops where routes to a network encounter the same router twice.

### Summary On BSD Based Tools

The tools were generally developed first on the IBM-RT system or on the VAX 8200 running Ultrix. Once the generic tools were running under the BSD system, the effort switched to getting IBM-PC versions running and developing the BSD-only tools. The generic tools ran under all the platforms described in the section on the portable library.

## 3.3.4 SGMPQUERY

The work on the libraries and the tools progressed rapidly. A PC-based graphical tool had been developed that showed connectivity on the basis of the metric a router had assigned for a given route. Unfortunately, there were no routers with SGMP agents in production at the time to facilitate the testing of the tool. This was the motivation to produce the program SGMPQUERY.

SGMPQUERY is a BSD UNIX based SGMP to Routing Information Protocol (RIP) gateway with the RIP code heavily borrowed from the QUERY program. RIP is the protocol developed at Berkeley for BSD UNIX and used in Proteon and other routers to allow hosts and routers to communicate routing data. The hop count is the metric for the route and the router with the lowest metric for a route is the first hop in the path to that network.

When a communications line goes down, the router can no longer learn routes from that interface. As a result, it gets routing information from its neighbor routers that replace the routes. If the neighbor was using a route through this

router to get to that particular network, the router will learn the route as costing one hop more than what the neighbor advertises. The neighbor will then re-learn the route from this router and add yet one more to the hop count. This continues until a route via another router is found or the hop count hits *infinity*, the point where the router decides the hop count is too high to be a real route. At the time, *infinity* was defined as 16 hops.

The RIP protocol has a special packet request that causes a router to dump its entire routing table to the requesting entity. SGMPQUERY made use of this request to extract the routing table out of the router it was answering for. It would then use this information to build an internal routing table that reflected the current table inside the router. It then used this information to respond to SGMP GET_REQUESTS in the router's stead.

SGMPQUERY accepts GET_REQUESTS for routing metric variables, determines if the routing table is up to date, and sends a GET_RESPONSE with the appropriate information. It was hoped that the router would not have to be burdened with a RIP request every time an SGMP request was received, so a timer to check the routing table's freshness was established. Whenever an SGMP request is received, the timer is checked to see if thirty seconds had elapsed. If so, it sends out a RIP request to get a new copy of the routing table. If the routing table update fails, SGMPQUERY would drop the SGMP request to simulate a non-responsive router. If the RIP update succeeds, an GET_RESPONSE is sent according to the information currently in the table. If the table is less than thirty seconds old, a response is immediately sent based on the routing tables the program currently has in memory. Thirty seconds was chosen as this is the interval that routers wait to send each other RIP routing updates.

The SGMPQUERY program is not a full SGMP agent implementation. It does not issue traps, send error packets back, or respond to SET_REQUESTS. It is further limited in that it can only respond to GET_REQUESTS for route metric variables and that the request packet can only contain one request. It does, however, perform the function that it returned the lexicographical antecedent variable of the one specification. Routing variables have a trailing *00* in the variable name. The routing metric variable for network 128.169.0.0 is fully specified as *01 04 01 02 04 80 A9 00 00 00.*

GET_REQUESTS are handled by returning the variable that is the lexicographical antecedent of the value that is sent to the SGMP agent. This means that a request for a specific route must send a variable name that is less than the actual name. This is the purpose of the trailing *00.* A specific route can be requested by leaving off the *00* and that route's variable will be the lexicographically antecedent variable. This has the additional advantage that a table can be traversed by sending to the agent the variable just received. The agent then responds by sending back the next variable in the table. By performing this function, SGMPQUERY is able to handle the SGMP requests that the PC-based graphical tool needed to monitor connectivity.

### 3.3.5 PROXY SGMP

While in the process of implementing and testing SGMP, the idea of proxy SGMP was developed. A proxy SGMP agent acts as an intermediary between the management station issuing the request and the SGMP entity the information is being requested from. The idea is to allow data caching so that several monitoring

stations could make requests without the monitored entity having to service all the requests. This has the benefits of reducing the amount of the entity's resources that are used for network management and, through strategic placement of proxy agents, reduces the amount of network management traffic that travels over lower bandwidth links. While the plan at first seemed achievable, an aspect of the SGMP variable space kept the first attempt to implement it from being successful.

A proxy SGMP agent gets a proxy Get or Set Request from a management station. The proxy request contains the IP address of the SGMP agent that the client wants information from. The proxy agent receives the proxy request, formulates an SGMP request and sends it to the agent. When the proxy agent receives the SGMP reply, it looks up the proxy request it received the reply for and sends an SGMP reply back to the station that issued the proxy request. If the proxy agent has a data cache of variables received from the monitored SGMP entity, it sends back to the requesting station a cached value, provided the entry is not out of date. This allows more monitoring stations, such as backup Network Operation Centers and for local, regional, and national centers, to monitor the same entity without the SGMP entity having to spend its time responding to SGMP queries from so many stations. It also allows the filtering of authentication as monitoring sites could be given session names to monitor a set of machines but the actual machines could use different session names, allowing for central control over access.

Proxy SGMP versions of the basic SGMP tools were developed. PGETID, PGETANY, PGETMANY and PGETROUTE are different from their non-proxy counterparts by the addition of an additional argument, the IP address of the target system, and changing the packet type the packet builder was to make from a GET_REQUEST to a PROXY_GET. This showed that it would be simple to

port any SGMP application to be a proxy SGMP application and take advantage of the benefits of proxy SGMP.

PSGMPD is a BSD UNIX based proxy SGMP agent that handles PROXY_GET requests. It maintains a variable tree for each of the routers it proxies for and populates the variable trees with the traffic that passes through the agent. The idea is to eventually associate timers with the variables and session names that proxy clients use to talk to the proxy agent. If a variable is requested that is not in the tree or is older than the time associated with the session name, an SGMP request is sent to the router. When the response is received from the router, the proxy agent's internal representation of the router's variable tree is updated and a response to the proxy request is sent back to the proxy client. The variable tree entries are independent of the variable they represent and thus can be applied to vendor-specific variables without having to have *a priori* knowledge of what the data means.

There was one fundamental flaw with this concept. SGMP requests are for the next lexicographical antecedent variable and the variable trees in a router are dynamic. For example, routes can appear and then go away and the routes which a router has is not predefined but discovered in the course of the router's operation. If the PSGMPD daemon has traffic pass through giving it the route to the IP address 192.2.4.0 and receives a subsequent request for the route to IP address 128.169.0.0, it has to perform a lexicographically *greater than* search and would hit the recently added 192.2.4.0 route and return that as the lexicographically antecedent variable.

A simple work-around was developed to try to recover from the problem. The PSGMPD program as it was actually written uses a special session name to indicate

a tickler session. This tickler session has all its proxy SGMP requests forwarded to its corresponding SGMP entity and the SGMP replies populate the variable trees in the proxy SGMP agent's memory. All other requests receive the data from the cached memory tree. The tickler session runs on a periodic basis and traverses the entire SGMP variable space, keeping the variable tree representation up to date.

This works, but many of the advantages of proxy SGMP are lost. One of the advantages of proxy SGMP is the reduction of the amount of traffic that an SGMP entity has to respond to. Few clients want to extract all of the variables from an entity, they are usually interested in only one sub-class or another. Unfortunately, there is no way to know which variables are going to be requested, so the tickler process has to ask for them all. Some of the variables change rapidly with time, such as packet counts. The actual change of other variables, such as routing metrics, is also of interest. Because the clients will need timely information, the tickler has to run often. In this implementation, the tickler process shell script that runs a proxy version of the GETMANY program named PGETMANY and asked for variable *01* every 30 seconds. The 30 seconds was decided upon based on the interval between RIP routing updates.

### 3.3.6 HOSTSGMPD

Another idea that was explored in the implementation of SGMP clients was that of using the SGMP to manage hosts. One of the design goals of the SGMP was "that the design be, as much as possible, independent of the architecture and mechanisms of particular hosts or particular gateways." [1] It was thought that the SGMP could possibly be extended to manage heterogeneous collections of hosts.

The RFC noted this as a possible future direction by specifying what the leading prefix for host variables would be, but did not address host management at all. Attempting to solve a current problem regarding account creation was motivation to explore this area.

The work done in producing PSGMPD covered nearly every aspect that an SGMP agent had to perform. The main difference was that instead of looking up values in a variable cache, the program had to look up the values from where ever they reside in the system. Also, the added complexity of caching requests to match responses that made up the bulk of PSGMPD was eliminated.

HOSTSGMPD is a BSD UNIX based host SGMP agent. It was built upon the PSGMPD program so that the main area to address was populating the variable space. It supports a few variables to experiment with the various aspects of writing an SGMP agent. The RFC did not specify any variables, so their assignment was arbitrary. Two of the variables supported are an ID variable (_HOST_version_id or *02 01 01*) and revision variable (_HOST_version_rev or *02 01 02*) to mirror the system variables the routers supported. There was also two test variables that did both Gets and Sets; one for integers (_HOST_impl_test or *02 FF 01*) and one for octet strings (_HOST_impl_system or *02 FF 02*). Lastly, there was a variable to do Gets and Sets to create accounts on the host (_HOST_impl_acct_create or *02 FF FE 01*). All of these variables are uniquely identified by a trailing *01*.

A generic system to support variables in the HOSTSGMPD program was developed to make the later addition of variables simple. The variables are maintained in a linked list in lexicographical order in structures that have entry points to be called if a GET_REQUEST is received and if a SET_REQUEST is received and a flag to indicate if the variable is read-only or read-write. When a request is re-

ceived, the authentication layer is parsed and processed to determine what access the session gets and to perform whatever transformation is necessary to the SGMP packet. If the request is a Get, a function is called to perform the Gets and to formulate the proper response. If the request is a Set, the variables are first sanity checked to make sure that all requests can be performed and then, if possible, the Sets are done. A GET_RESPONSE is then generated with the new values of the variables and returned to the client.

The Id and Version variables were easy to implement. They are just octet strings embedded in the program identifying the program's name and its version number. The integer test variable points to a static integer so that the value can be set and read over the course of the HOSTSGMPD program's execution. The octet string test integer was deliberately named *_HOST_impl_system.* The variable points to a static octet string, similar to the integer test variable, but also passes this string to the *system()* call when the variable is set. As a result, commands such as *shutdown -r now* can be performed to the amusement of all except those whose system has just been rebooted. It was once commented that the SGMP could be considered a proposal for a poor replacement for Telnet. This variable was not meant as such, but as a general demonstration as to the ease of implementation of variables. The idea was that as the variable space became more defined, it would be easy to perform Sets by having them perform the desired action through things such as system calls, a function that most systems support.

One case in point is the account-creation variable. On this BSD based agent, it receives an octet string that corresponds to a line in the /em /etc/passwd file. On a GET_REQUEST, the value of the last account string is returned. On a SET_REQUEST, the line is appended to the */etc/passwd* file. A *Yellow Pages* make

is then performed via a system call to update the yellow pages representation of the passwd file. The sanity checking on account name, user ID, and creating a home directory is performed in the program that issued the SGMP SET_REQUEST. It is unnecessary in this implementation, but these and other sanity checking functions could easily be incorporated in the set function.

# CHAPTER 4

# Findings and Summary

The success in creating the portable library and the applications described in the previous chapter show that the SGMP is implementable and that it can perform useful functions. This section of the thesis will address the questions raised in Chapter 1 and summarize the problems encountered in the creation of this implementation of the SGMP.

## 4.1 Findings

This thesis started with several questions about the Simple Gateway Monitoring Protocol. Among things questioned was whether the protocol was even practicable and what impact it would have on the managed systems. There were also questions about the impact of some of the design decisions of the protocol on the implementations.

The first set of questions dealt with the protocol as specified. It was asked whether the protocol could be implemented as specified. The presence of multiple independent implementations of the protocol stand as evidence that it can be implemented. The protocol was used to monitor the National Science Foundation backbone network (NSFNET), several NSFNET regional networks, and campus networks before the SNMP was adopted. This success stands as evidence that the protocol works and can perform a useful function.

The next question was what impact the protocol would have on the systems being monitored. HOSTSGMPD's binary is 25 kilobytes, PSGMPD is 32 kilobytes, and SGMPQUERY is 40 kilobytes of SUN 3 version 4.0.3 shared library code. While there will be memory allocation to build and parse the packets, it is obvious that code size is of minimal impact. Running HOSTSGMPD on a SUN 3/60 and using GETMANY to traverse it's variable space 1000 times took 50 CPU seconds. Since the HOSTSGMPD variable space is only five variables large and this invocation of GETMANY only asked for a single variable at a time, this represents 0.01 CPU seconds to receive, parse, extract, build, and send an SGMP message. This is a very crude test, but it shows that the protocol makes minimal impact on the CPU.

Another question raised was whether the SGMP would be extensible. As shown in this thesis, the protocol has been extended to allow the monitoring of hosts. In addition, the protocol was extended to allow a proxy mechanism to isolate network management traffic from the managed entities.

The use of a connectionless transport for the protocol was one of the most questioned design decisions. The HEMS/HEMP protocol required a connection-oriented transport to make sure the messages get through intact in a fire-fighting situation. There was a case, however, where a network manager was unable to reach his gateway via Telnet (which uses a connection-oriented transport), but was able to reach the gateway with the SGMP. It was felt that the reason for the SGMP client's success was that the communication is via transport that did not require the overhead of building up a connection in a hostile network environment.

The choice of using UDP did not come without some cost. The limited packet size meant that a single request could hold an approximate maximum of 15 vari-

ables, depending on the value size associated with the particular variables. Fortunately, the tables that lent themselves to the extraction of multiple variables per request were narrow enough that a row would fit in a packet. The use of UDP brought with it a few unexpected problems. The checksum in UDP packets is optional, and many implementations do not set the checksum on the UDP packets they transmit. Since the protocol itself has no data corruption checking other than that inherent with ASN.1 encoding, care had to be taken to insure that implementations used checksums. It was found that one implementor's checksum implementation had not been used except to send UDP packets that were multiples of two bytes long and did not correctly calculate checksums for the SGMP packets of odd length.

The last major question was what the impact of the use ASN.1 encoding in the protocol would be. ASN.1 provides for many complex types of data representation and was designed to be extensible for any possible data representation. As a result, the implementation of a full ASN.1 parser and builder would be quite large. It was feared that the ASN.1 requirement of the protocol would make it difficult to implement and taxing on resources to use. The designers of the protocol carefully limited the actual components of ASN.1 that would be needed in the protocol such that the impact was minimal. This is evident by the small memory and CPU time requirements needed to parse and build SGMP queries and responses discussed earlier. While building the portable parser and builder library was a challenge, the fact that the resultant code was only 1250 lines indicates that the task was not insurmountable.

## 4.2 Summary

### 4.2.1 The Portable Library

There were several important items to keep in mind when writing the portable library in the C language. One was the use of explicit *long* and *short* types and avoiding the use of *int* type variables. Another was to write code with minimal stack and memory usage. Third was to be careful to prevent memory leaks. The last, to use specific messages to identify the various error conditions that could arise in the libraries.

The use of explicit *long* and *short* types makes the code more portable between the PC platform where an *int* is 16 bits long and the other, larger, systems which use a 32 bit *int*. An additional problem was solved by this. The Microsoft C compiler (version 4.0) had a tendency to pad the stack to a long word boundary even when the parameter being passed was an *int* (a *short*). As a result, erroneous values would reach the routine. The use of *long* types worked around this bug in the compiler.

The PC is a rather confining platform to code for. The PC/IP package that was used required that the programs run under the small memory model. While it is possible to break outside the 64 kilobyte code and data segment limits of the small memory model, it was felt that keeping the library below this restriction was a good goal. This produced a library that can function on other memory-limited machines, such as small gateways and terminal servers. The result is a library that is about 11 kilobytes in SUN 3 or VAX binary form and 26 kilobytes in IBM/PC binary form.

One of the most disastrous conditions to possibly find in a library of this type is a memory leak. In the course of building and parsing the SGMP messages, a lot of memory is manipulated in data structures. If, in the course of building and parsing packets, all the memory used is not recovered, there will be memory loss on each packet sent. While this may not be critical in applications that build and parse only one packet, this is fatal for applications such as PSGMPD that handle thousands of such requests. Therefore, particular attention was taken to insure that the routines responsible for freeing memory would recover all memory allocated.

One item that was of immeasurable help in developing applications and testing this implementation with other implementations was the abundant use of error messages in the builder and the parser at the point where the error first occurs. These messages were often considered a burden in writing applications where the output of the program would never be used, but they helped pinpoint exactly where the packet being parsed or the application being written had gone wrong.

### 4.2.2 Tools

The GETMANY tool was the most valuable tool in developing this SGMP implementation. It served as an example for the other SGMP programs that were written that needed to repeatedly query a gateway for variables based on the previous answers. It was the main testing program when working with SGMP agent implementations. Using it to traverse an implementation's variable tree would exercise almost every aspect of a parser and builder. The error messages from the library's parser usually made it very easy to tell where the other implementation

had gone wrong. Putting a GETMANY in a shell program that loops repeatedly to span an agent's variable tree was an easy test for memory leaks in the agent's implementation. Its flexibility facilitates rapidly prototyping a tool in a shell script.

### 4.2.3 SGMPQUERY

SGMPQUERY was an early exposure to the problems that an SGMP agent would encounter. While the original purpose of the library was to build clients, it was hoped that it could also be used in all aspects of SGMP applications. The general applicability of the library was confirmed when SGMPQUERY was written without any changes to the library.

SGMPQUERY gave rise to a problem of network management that proxy SGMP was later developed to try to address: what is the load of network management on the network being managed? While demonstrating SGMP and SGMPQUERY at a National Science Foundation workshop for NSFNET regional network managers at Cornell, it was noticed that every time the gateway at the University of Tennesee, Knoxville, was monitored, Florida State University became unreachable. At the time the route from Florida State to Cornell was one metric away from infinity. The prototype applications running at the time sent their requests without any speed restrictions other than waiting for a response before sending its next request. The resulting traffic between Cornell and UTK was sufficient that the routing protocol used in the NSFNET backbone assigned a routing metric of one higher to the route, thus making Florida State unreachable by the routers.

It was learned from this experience that the speed at which future SGMP applications were going to communicate with gateways needed to be limited to prevent

the network from being saturated with network management traffic. This was not a problem with the protocol, but a problem with the application implementation in that it did not anticipate the speed that such a simple protocol could run and the bandwidth it could consume when running unchecked.

### 4.2.4 PSGMPD

Proxy SGMP was considered as a way to help prevent the saturation of networks with network management traffic as well as a way to reduce the amount of load on the monitored entity by cutting down on the number of requests it had to process. The original concept of proxy SGMP was to let a non-critical entity duplicate the proxy managed entities' variable space based on the traffic that passed through it. This did not work as previously explained in Chapter 3. At first, this was thought to be a problem with the proxy SGMP concept. As it later turned out, the concept worked fine but the implementation details were too literal in trying to mimic the concept.

A way to work around the flaw in PSGMPD would have been to use a request/response pairing cache rather than trying to simulate the gateway's variable tree. The variable space would be populated to represent the actual variable name used to formulate the request. When the response for a request is received, the response could be cached with the request so that future requests that exactly match the variable name that the request used and was within timer limitations for the session being used could then be given this cached response. This would result in a system that would not generate any more SGMP traffic to the gateways than was necessary to respond to actual requests. It would also have enough flexibility in

the variable name space that vendor-specific variables could pass through without the program needing to be modified. This was not done, as it would represent a major re-write of the program and was not thought of until after the SNMP had emerged, and the SGMP had been obsoleted.

### 4.2.5 HOSTSGMPD

Much was learned by experimenting with HOSTSGMPD. As would be expected, the trivial authentication of the unity function was unacceptable for commands that required Sets. Without a practicable authentication layer, variables that can be set, such as the *system* or the *create_account*, or even something as simple as an *interface status* (where an interface could be turned off) become a security risk. It is for this reason that the account creation system was not put into production. Another thing learned was that the program would have to be modified to properly support variables. The current program supported only a static variable name space. If items such as routing tables were implemented, the variables become dynamic and may come and go. The program required that all variables be known at compile time and could not handle routing tables. Other than that, it was apparent that the variable space that parallels gateways could be easily populated by using the kernel variable extraction methods of NETSTAT [33] and IFCONFIG [34] on BSD systems.

# CHAPTER 5

# Conclusions and Recommendations

This chapter addresses the practicability of the SGMP as a network management protocol and provides suggestions for further research in network management. The success of the SGMP was established through its use in the management of production networks during the period when this thesis was being written. Likewise, future directions for the SGMP have already been partly charted with the evolution of the SGMP into the SNMP and with the SNMP's acceptance as a standard for the short-term management of the Internet. Still, it benefits the reader to examine the lessons learned in the development of the SGMP so that the problems solved in the past will not reappear in the network management protocols of the future.

## 5.1 Conclusions

This thesis began by raising a number of questions about the SGMP's ability to be implemented and its usability, if implemented.

The first set of questions asked whether the SGMP could be implemented, and its practicality if it were implemented. The applications produced in this study stand as evidence of the SGMP's implementability. In addition, the applications that extract information from the remote agents show that it is a useful protocol. Lastly, the memory requirements listed in Chapter 4 for the library show that it

makes minimal impact on the systems that implement it.

The next topic was the choice of transport protocol for the network management protocol. Using request ID's and retransmission from the managing entity, manager could extract the information desired with the same efficiency as TCP. The complexity in maintaining this *virtual circuit* of information resides totally in the management station and thus relieves the managed entity of the additional complexity of implementing a *reliable* transport such as TCP. The only short-comings of using UDP as the transport was the limitation of the number of variables that can be addressed in one atomic request and that many UDP implementations do not support packet checksumming by default. The size limitation can be avoided through the judicious selection and ordering of the variable requests generated by the applications. UDP checksumming should have been required in the SGMP specification.

The final question was about the choice to use ASN.1 to represent the data. Again, the existence of the portable library shows that it can be done and its small size shows that it has a minimal impact on systems. The primary purpose of ASN.1 is as a language to permit data exchange between disparate architectures. The small additional complexity incurred by implementing the subset of ASN.1 in the SGMP was a small price to pay for the inherent interoperability that ASN.1 brings to the protocol.

## 5.2 Recommendations

The Simple Gateway Monitoring Protocol has evolved into the Simple Network Management Protocol, and for the purposes of future expansion of the protocol,

the SGMP is obsolete. However, since the SNMP is a derivative of the SGMP, ideas for future expansion apply equally to both protocols. For these reasons, this section will refer to recommendations for future directions for growth and study in the SNMP. They are equally applicable to the SGMP, but considering the SGMP's status, any effort expended in the exploration of these directions would be better suited if they used the SNMP.

A number of issues with the SNMP have arisen and are being addressed by working groups of the Internet Engineering Task Force for use in a production internet environment. There are other directions that need further experimentation and exploration but have not been receiving any attention. Lastly, there is work being done independently to expand the usability of these protocols in new directions and to give it life beyond the short term solution proposed by the Internet Activities Board.

Among the problems being addressed by the Internet Engineering Task Force (IETF) are more secure authentication schemes and extensions to the Management Information Base. The authentication working group has produced draft memos proposing three authentication schemes: trivial (the present one), host authentication, and an encrypted system [23, 35, 36]. It is hoped that these will address the authentication needs of the Internet and still provide interoperability between management domains. A new Management Information Base was recently published in RFC 1158 to define further variables to allow the protocols to extract and manipulate [37].

Other areas that are being researched include the development of a common Applications Programming Interface (API). A network management API would be a thin layer that translates from a common interface to the form needed by

the underlaying network management protocols, regardless of whether it is CMIP, the SNMP, the HEMS, CMOT, or some proprietary vendor protocol. This would isolate the applications from the network management protocols needed to extract the information. This will allow the development of common network management applications that can function across all of the networking environments.

This leads into the topic of tools for the network manager. Applications are needed to extract the data from the managed entities and present it to network managers in a manner they can deal with. One of the things that a network manager needs in a Network Operations Center (NOC) is a real-time display of the health of the network being managed. This can consist of a connectivity display, which is a quick visual gauge of health. Another need of the network manager is to know the level of utilization in the network. Historical reports of the load on links and nodes need to be available from a database established by querying the managed entities on a periodic basis. Lastly, the network manager needs to have a method to formulate custom requests to extract information from network entities that may not be available via the management station's normal queries.

Internets are organized in a hierarchical fashion. There are backbone networks connected to regional networks which in turn connect to local networks (which may, themselves, be an internet of local area networks). The responsibility for managing these networks is also spread in a hierarchical fashion. However, the events that a manager is interested in may be external to the network the manager is responsible for. To address this situation, either each management station must manage the entire Internet or there must be a way for network management stations to share the information they have gathered. This sharing would spread the burden of collecting and archiving the network management data across multiple stations

but still let the network managers deal with the entire network. An inter-NOC protocol could be developed for formulating requests for archived data and to transfer the data. It would require intensive work in determining exactly what information a NOC needs to gather and developing a protocol that is robust and fair enough to tie together management station implementations from competing vendors. The result of this would primarily be a distributed database of network information.

An area of interest that seems to be ignored is that of proxy network management in the sense of proxy in the SGMP. Proxy SNMP is actually a different concept from the original proxy presented in the SGMP. In the SGMP, the proxy agent is a multiplexing entity that allows multiple management stations to get slightly dated information about a remote managed node from the proxy agent's storehouse of network information. Proxy under the SNMP refers to the communication of a network entity that cannot speak the SNMP through an SNMP agent that can translate between the SNMP and whatever method the entity being proxied for can communicate. While the proxy name has been re-used, the original idea still has merit and should be further investigated.

As noted earlier in this thesis, the original implementation of proxy for the SGMP ran into some implementation specific problems that can be overcome. With more and more personnel being designated as *network managers*, both the number of requests for management information from managed entities and the load that corresponds to this traffic are increasing. It was the purpose of *proxy* under the SGMP to off-load some of this traffic to a proxy host that had more power, or its primary task was less important than that of the monitored entities. Through the judicious selection of the time sensitivity of specific variables and of

the placement of SGMP-style proxy nodes, the amount of administrative network management traffic can be reduced.

Finally, additional work that has been done independent of the IETF is an extension of the SNMP to run in the OSI networking environment. Marshall Rose has reported of an experimental implementation of the SNMP over Connectionless-mode Transport Service (CLTS — OSI's version of UDP) that has been used to extract variables about OSI network protocols [38, 39, 40]. This totally removes the SNMP from the TCP/IP environment and makes it a possible experimental alternative to the CMIS/CMIP as the network management protocol for the OSI networks of the future.

# BIBLIOGRAPHY

# BIBLIOGRAPHY


[1] James R. Davin, Jeffrey D. Case, Mark S. Fedor, and Martin L. Schoffstall. A Simple Gateway Monitoring Protocol. Request for Comments 1028, DDN Network Information Center, SRI International, November 1987.

[2] V. Cerf. The Catenet Model for Internetworking. Internet Engineering Note 48, DDN Network Information Center, SRI International, July 1978.

[3] D.L. Mills. Exterior Gateway Protocol Formal Specification. Request for Comments 904, DDN Network Information Center, SRI International, April 1984.

[4] J. Moy. OSPF Specification. Request for Comments 1131, DDN Network Information Center, SRI International, October 1989.

[5] C.L. Hedrick. Routing Information Protocol. Request for Comments 1058, DDN Network Information Center, SRI International, June 1988.

[6] R. Hinden. Host Monitoring Protocol. Request for Comments 869, DDN Network Information Center, SRI International, December 1983.

[7] Craig Partridge and Glenn Trewitt. The High-Level Entity Management System. Request for Comments 1021–1024, DDN Network Information Center, SRI International, October 1987.

[8] Jeffrey D. Case, Mark S. Fedor, Martin L. Schoffstall, and James R. Davin. A Simple Network Management Protocol. Request for Comments 1098, DDN Network Information Center, SRI International, April 1989.

[9] U. Warrier and L. Besaw. Common Management Information Services and Protocol over TCP/IP (CMOT). Request for Comments 1095, DDN Network Information Center, SRI International, April 1989.

[10] Mike Muuss. *ping*. University of California, Berkeley, May 1986.

[11] Berkeley University of California. query. University of California, Berkeley, 1980. Manual Pages.

[12] Van Jacobson. traceroute. Anonymous FTP to UUNET.UU.NET, February 1989. Manual Pages.

[13] Craig Partridge and Glenn Trewitt. The High-Level Entity Management System (HEMS). *IEEE Network*, 2(2), March 1988.

[14] Information Processing — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1). International Organization for Standardization, 1987. International Standard 8824.

[15] Information Processing — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). International Organization for Standardization, 1987. International Standard 8825.

[16] Message Handling Systems: Presentation Transfer Syntax and Notation. CCITT, October 1984. Recommendation X.409.

[17] James R. Davin, Jeffrey D. Case, Mark S. Fedor, and Martin L. Schoffstall. A Simple Network Management Protocol. IDEA 11, DDN Network Information Center, SRI International, March 1988.

[18] Vinton G. Cerf. Internet Activities Board. Request for Comments 1052, DDN Network Information Center, SRI International, September 1989.

[19] Vinton G. Cerf. IAB Recommendations for the Development of Internet Network Management Standards. Request for Comments 1052, DDN Network Information Center, SRI International, April 1988.

[20] Keith McCloghrie and Marshall T. Rose. Management Information Base Network Management of TCP/IP based internets. Request for Comments 1066, DDN Network Information Center, SRI International, August 1988.

[21] Marshall T. Rose and Keith McCloghrie. Structure and Identification of Management Information for TCP/IP based internets. Request for Comments 1065, DDN Network Information Center, SRI International, August 1988.

[22] W. Yeong, M. Schoffstall, and M. Fedor. A UNIX Implementation of the Simple Network Management Protocol. In *Proceedings of the Winter 1989 USENIX Conference*, January 1989.

[23] J. Galvin, K. McCloghrie, and J. Davin. Authentication and Privacy in the SNMP. Internet draft, DDN Network Information Center, SRI International, April 1990. found in file draft-ietf-snmpauth-authsnmp-01.txt.

[24] Information Processing Systems — Open Systems Interconnection —Common Management Information Service Definition. International Organization for Standardization, September 1989.

[25] Information Processing Systems — Open Systems Interconnection —Common Management Information Protocol Definition. International Organization for Standardization, September 1989.

[26] Information Processing Systems — Open Systems Interconnection — Basic Reference Model. International Organization for Standardization, October 1984.

[27] Marshall T. Rose and Dwight E. Cass. ISO Transport Services on top of the TCP. Request for Comments 1006, DDN Network Information Center, SRI International, May 1987.

[28] H. Lew and J. Robertson. TCP/IP Network Management with an Eye Toward OSI. *Data Communications*, 18(10), August 1989.

[29] Marshall T. Rose. ISO Presentation Services on top of TCP/IP-based internets. Request for Comments 1085, DDN Network Information Center, SRI International, December 1988.

[30] Marshall T. Rose. *The Open Book: A Practical Perspective on Open Systems Interconnection.* Prentice-Hall, 1989. ISBN 0–13–643–016–3 (to appear).

[31] Berkeley University of California. yacc. University of California, Berkeley, 1985. Manual Pages.

[32] Berkeley University of California. lex. University of California, Berkeley, 1985. Manual Pages.

[33] Berkeley University of California. netstat. University of California, Berkeley, 1983. Manual Pages.

[34] Berkeley University of California. ifconfig. University of California, Berkeley, 1983. Manual Pages.

[35] J. Davin, J. Galvin, and K. McCloghrie. Administration of SNMP Communities. Internet draft, DDN Network Information Center, SRI International, April 1990. found in file draft-ietf-snmpauth-communities-00.txt.

[36] K. McCloghrie, J. Davin, and J. Galvin. Experimental Definitions of Managed Objects for Administration of SNMP Communities. Internet draft, DDN Network Information Center, SRI International, April 1990. found in file draft-ietf-snmpauth-manageobject-00.txt.

[37] Marshall T. Rose. Management Information Base for Network Management of TCP/IP-based internets: MIB-II. Request for Comments 1158, DDN Network Information Center, SRI International, May 1990.

[38] Marshall T. Rose. draft document on CLNS MIB. Electronic Mail from Marshall T. Rose to the Simple Network Management Protocol Working Group, December 1989.

[39] Marshall T. Rose. SNMP over OSI. Request for Comments 1161, DDN Network Information Center, SRI International, June 1990.

[40] G. Satz. Connectionless Network Protocol and End System to Intermediate System Management Information Base. Request for Comments 1162, DDN Network Information Center, SRI International, June 1990.

# VITA

Kenneth William Key was born in Midland, Michigan on May 18th, 1962. He attended schools in Midland until moving to Knoxville, Tennessee in June of 1975. He graduated from Farragut High School in June, 1980 and immediately entered the University of Tennessee. In the course of his degree he worked as a Co-op student for Du Pont at the Savannah River Lab and Plant. He received a Bachelor of Science with honors in Chemical Engineering and a minor in Computer Science in December of 1985.

In January, 1986, he entered the graduate program in Computer Science. While a student, he worked as a graduate teaching assistant in the Computer Science Department, a teaching assistant for the Tennessee Governor's School of the Sciences, a Graduate Research Assistant for the University of Tennessee Computing Center, and as a Systems Programmer/Analyst for the University of Tennessee Computing Center. In addition, he authored a reference implementation of the Simple Gateway Monitoring Protocol and co-authored a reference implementation of the Simple Network Monitoring Protocol. Mr. Key will continue to work for the University of Tennessee after graduation.